Machine Learning I:

Theoretical Foundations

Prof. Marius Kloft

Contents

About the Lecturer

Prof. Dr. Marius Kloft

Bio

Since 2017 Marius Kloft has been a professor of computer science at TU Kaiserslautern, Germany. Previously, he was an adjunct faculty member of the University of Southern California (09/2018-03/2019), an assistant professor at HU Berlin (2014-2017) and a joint postdoctoral fellow (2012-2014) at the Courant Institute of Mathematical Sciences and Memorial Sloan-Kettering Cancer Center, New York, working with Mehryar Mohri, Corinna Cortes, and Gunnar Rätsch.

From 2007-2011, he was a PhD student in the machine learning program of TU Berlin, headed by Klaus-Robert Müller. He was co-advised by Gilles Blanchard and Peter L. Bartlett, whose learning theory group at UC Berkeley he visited from 10/2009 to 10/2010. In 2006, he received a master in mathematics from the University of Marburg with a thesis in algebraic geometry.

Research Interests

Marius Kloft is interested in theory and algorithms of statistical machine learning and its applications, especially in statistical genetics, mechanical engineering, and chemical process engineering. He has been working on, e.g., multiple kernel learning, transfer learning, anomaly detection, extreme classification, and adversarial learning. He co-organized workshops on these topics at NIPS 2010, 2013, 2014, 2017, ICML 2016, and Dagstuhl 2018.

His dissertation on Lp-norm multiple kernel learning was nominated by TU Berlin for the Doctoral Dissertation Award of the German Chapter of the ACM (GI). He received the Google Most Influential Papers Award and the DFG Emmy-Noether Career Award.

Requirements

This course requires a solid foundation in linear algebra and calculus at university level. For guidance on how to brush up your math for this course, we have set up a frequently updated page that links to various excellent resources[1]. The lecture will be accompanied by theoretical and programming exercises. The theory exercises can be solved using the concepts introduced in the lecture, but the programming exercises require basic familiarity with Python. If you are unsure about your Python knowledge, you can may want to have a look at the Google Python tutorial[2], for example. In Section 1 we briefly go through the basics of mathematics needed for machine learning.

Objectives of the textbook

After attending this lecture you should be able to

- understand the aim of machine learning
- know general machine learning algorithms
- understand how to apply a specific model to a specific problem
- understand the theoretical background needed for machine learning
- implement solutions to ML problems both on a high level using popular ML frameworks and on a low level using only linear algebra subroutines.

---

[1] https://ml.cs.uni-kl.de/guide.html
[2] https://developers.google.com/edu/python

Chapter 1: Mathematical Notation & Basics

In this chapter, we introduce some basic mathematical concepts and the notation we will use in this course. Machine learning requires solid mathematical knowledge in linear algebra and multivariate analysis.

1.1 Notation

In this section we will fix notation that we will use throughout the course.

- We denote vectors $\mathbf{v} \in \mathbb{R}^d$ by bold letters and scalars $s \in \mathbb{R}$ by regular letters:

$$\mathbf{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_d \end{pmatrix}. \tag{1.1}$$

- We denote matrices $A \in \mathbb{R}^{m \times n}$ by capital letters:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & \ddots & \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}. \tag{1.2}$$

- We define $\mathbf{0}$ (resp. $\mathbf{1}$) as the vector full of zeros (resp. full of ones) with appropriate dimension to the context:

$$\mathbf{0} := \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \mathbf{1} := \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}. \tag{1.3}$$

- If $\mathbf{v} \in \mathbb{R}^d$, then we define $\mathbf{v}^\top \in \mathbb{R}^{1 \times d}$ as its transpose:

$$\mathbf{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_d \end{pmatrix}, \quad \mathbf{v}^\top := \begin{pmatrix} v_1, \dots, v_d \end{pmatrix}. \tag{1.4}$$

  If $A \in \mathbb{R}^{m \times n}$, then we define $A^\top \in \mathbb{R}^{n \times m}$ as its transpose:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & \ddots & \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}, \quad A^\top := \begin{pmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & & \ddots & \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{pmatrix}. \tag{1.5}$$

- Recall that the scalar product of two vectors $\mathbf{v}, \mathbf{w} \in \mathbb{R}^d$ is defined by

$$\langle \mathbf{v}, \mathbf{w} \rangle := \mathbf{v}^\top \mathbf{w} = \sum_{i=1}^{d} v_i w_i. \tag{1.6}$$

- The (2-)norm of a vector $\mathbf{v} \in \mathbb{R}^d$ is defined as

$$\|\mathbf{v}\| := \sqrt{\langle \mathbf{v}, \mathbf{v} \rangle} = \sqrt{\mathbf{v}^\top \mathbf{v}} = \sqrt{\sum_{i=1}^{d} v_i^2}. \tag{1.7}$$

- Let $\mathbf{v}, \mathbf{w} \in \mathbb{R}^d$. Then we denote

$$\mathbf{v} \leq \mathbf{w} :\iff \forall i = 1 \ldots d : v_i \leq w_i. \tag{1.8}$$

- For a set S, we denote the cardinality of S (the number of elements in S) by $|S|$.

1.2 Scalar Projection

Definition 1.2.1 (Scalar Projection)**.** Let $\mathbf{v}, \mathbf{w} \in \mathbb{R}^d$ with $\mathbf{w} \neq \mathbf{0}$. The *scalar projection* of $\mathbf{v}$ onto $\mathbf{w}$ is defined as $\Pi_{\mathbf{w}}(\mathbf{v}) := \dfrac{\mathbf{w}^\top \mathbf{v}}{\|\mathbf{w}\|}$.

We observe that the scalar projection is a scalar, not a vector. This quantity also has a nice geometric interpretation[3] as illustrated in Fig. 1.1.

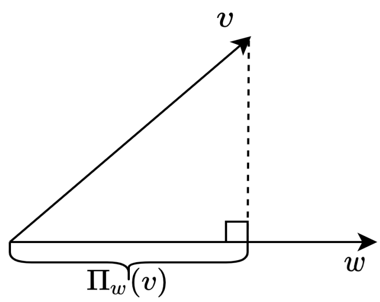

Figure 1.1: Geometrical illustration of the scalar projection: Given the vectors $\mathbf{v}$ and $\mathbf{w}$, here $\Pi_{\mathbf{w}}(\mathbf{v})$ is the scalar projection of $\mathbf{v}$ onto $\mathbf{w}$.

1.3 Hyperplanes

Definition 1.3.1 ((Affine-)linear Function)**.** An *(affine-)linear function* is a function $f : \mathbb{R}^d \to \mathbb{R}$ of the form $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$, where $\mathbf{w} \in \mathbb{R}^d \setminus \{0\}$ and $b \in \mathbb{R}$. For simplicity we call (affine-)linear functions just *linear functions*.

Example 1.3.2. The function

$$\begin{aligned} f &: \mathbb{R}^2 \to \mathbb{R} \\ \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} &\mapsto 3x_1 + 4x_2 - 7 \end{aligned}$$

is (affine-)linear as it is of the form $f(x) = \begin{pmatrix} 3 \\ 4 \end{pmatrix}^\top \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - 7$.

[3] If you are looking for a proof, see: https://www.youtube.com/watch?v=LyGKycYT2v0.

Definition 1.3.3 (Hyperplane)**.** A *hyperplane* is a subset $H \subset \mathbb{R}^d$ defined as $H := \left\{\mathbf{x} \in \mathbb{R}^d : f(\mathbf{x}) = 0\right\}$, where f is (affine-) linear.

Proposition 1.3.4. *Let H be a hyperplane defined by the affine-linear function $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$. The vector $\mathbf{w}$ is orthogonal to H, meaning that: $\forall \mathbf{x}_1, \mathbf{x}_2 \in H$ it holds $\mathbf{w}^\top (\mathbf{x}_1 - \mathbf{x}_2) = 0$.*

Proof. Let $\mathbf{x}_1, \mathbf{x}_2 \in H = \left\{\mathbf{x} \in \mathbb{R}^d : \mathbf{w}^\top \mathbf{x} + b = 0\right\}$. Then $\mathbf{w}^T \mathbf{x}_1 + b = 0$ and $\mathbf{w}^T \mathbf{x}_2 + b = 0$. Thus

$$\mathbf{w}^T \mathbf{x}_1 + b = \mathbf{w}^T \mathbf{x}_2 + b \iff \mathbf{w}^T (\mathbf{x}_1 - \mathbf{x}_2) = 0.$$

□

Definition 1.3.5 (Signed Distance)**.** The *signed distance* of a point x to H is given by

$$d(\mathbf{x}, H) := \operatorname{sign}\left(\mathbf{w}^\top \mathbf{x} + b\right) \min_{\tilde{x} \in H} \|x - \tilde{x}\| . \tag{1.9}$$

Observe how (1.9) implies that

$$|d(\mathbf{x}, H)| = \min_{\tilde{x} \in H} \|x - \tilde{x}\| .$$

As the name suggests, the signed distance will give us the distance from a point to the hyperplane. It will be positive if it is towards the normal and negative elsewise.

Proposition 1.3.6. *For the signed distance we have that:*

$$d(\mathbf{x}, H) = \frac{1}{\|\mathbf{w}\|} \left(\mathbf{w}^\top \mathbf{x} + b\right) . \tag{1.10}$$

Proof. Let $\tilde{\mathbf{x}}$ be an arbitrary element of H. By our previous proof, we know that $\mathbf{w}$ is orthogonal to our hyperplane. First we notice from Fig. 1.2 that

$$\forall \tilde{\mathbf{x}} \in H : \Pi_{\mathbf{w}}(\mathbf{x} - \tilde{\mathbf{x}}) = \operatorname{sign}\left(\mathbf{w}^\top \mathbf{x} + b\right) \min_{\tilde{\mathbf{x}} \in H} \|\mathbf{x} - \tilde{\mathbf{x}}\|,$$

where $\operatorname{sign}\left(\mathbf{w}^\top \mathbf{x} + b\right)$ indicates on which side of the hyperplane $\mathbf{x}$ lies. So

$$d(\mathbf{x}, H) = \Pi_w(\mathbf{x} - \tilde{\mathbf{x}}) \overset{\text{def.}}{=} \frac{\mathbf{w}^T (\mathbf{x} - \tilde{\mathbf{x}})}{\|\mathbf{w}\|} = \frac{\mathbf{w}^T \mathbf{x} - \mathbf{w}^T \tilde{\mathbf{x}}}{\|\mathbf{w}\|} \overset{(\star)}{=} \frac{\mathbf{w}^\top \mathbf{x} + b}{\|\mathbf{w}\|},$$

where in $(\star)$ we use: $\tilde{\mathbf{x}} \in H \Rightarrow \mathbf{w}^T \tilde{\mathbf{x}} + b = 0 \iff -\mathbf{w}^T \tilde{\mathbf{x}} = +b$. □

1.4 Eigenvalues

Definition 1.4.1 (Eigenvalues)**.** Let $A \in \mathbb{R}^{d \times d}$. $\lambda \in \mathbb{R}$ is called an *eigenvalue* of A if there is a vector $\mathbf{x} \in \mathbb{R}^d \setminus \{0\}$ such that $A\mathbf{x} = \lambda \mathbf{x}$. In that case $\mathbf{x}$ is an *eigenvector* corresponding to the eigenvalue λ. The set

$$\operatorname{Eig}(A, \lambda) := \left\{\mathbf{x} \in \mathbb{R}^d : A\mathbf{x} = \lambda \mathbf{x}\right\} \tag{1.11}$$

is called the *eigenspace* corresponding to the eigenvalue λ.

Intuitively an eigenvector preserves its direction under the linear transformation A but not necessarily its magnitude.

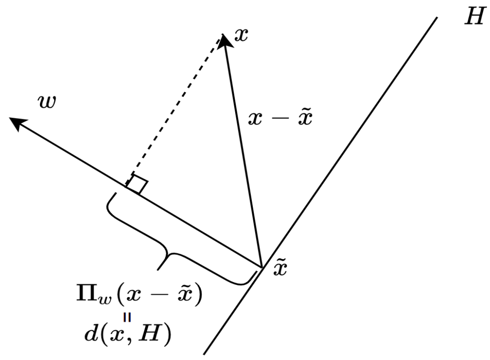

Figure 1.2: The hyperplane and the vectors are represented geometrically. Crucial for the proof of Proposition 1.3.6 is the observation that $d(\mathbf{x}, H) = \Pi_w(\mathbf{x} - \tilde{\mathbf{x}})$.

Example 1.4.2 (Eigenvalue Calculation)**.** Let

$$B = \begin{pmatrix} -6 & 3 \\ 4 & 5 \end{pmatrix}.$$

Let us try to find the eigenvalues of the matrix B. Let I denote the identity matrix. For $\lambda \in \mathbb{R}$ and $A \in \mathbb{R}^{d \times d}$ it holds:

$$\begin{aligned}
\lambda \text{ eigenvalue of } A &\iff \exists\, \mathbf{x} \neq \mathbf{0} \in \mathbb{R}^d \text{ with } A\mathbf{x} = \lambda\mathbf{x} \\
&\iff \exists\, \mathbf{x} \neq \mathbf{0} \in \mathbb{R}^d \text{ with } A\mathbf{x} = \lambda I\mathbf{x} \\
&\iff \exists\, \mathbf{x} \neq \mathbf{0} \in \mathbb{R}^d \text{ with } \lambda I\mathbf{x} - A\mathbf{x} = 0 \\
&\iff \dim \operatorname{Ker}(\lambda I - A) > \mathbf{0} \\
&\iff \dim \operatorname{Im}(\lambda I - A) < d \\
&\iff \lambda I - A \text{ not invertible} \\
&\iff \det(\lambda I - A) = \mathbf{0}.
\end{aligned}$$

Let us use this insight to calculate the eigenvalues of B:

$$\begin{aligned}
\det \begin{pmatrix} \lambda + 6 & 3 \\ 4 & \lambda - 5 \end{pmatrix} = 0 &\iff (\lambda + 6)(\lambda - 5) - 12 = 0 \\
&\iff \lambda^2 + \lambda - 42 = 0 \\
&\iff (\lambda + 7)(\lambda - 6) = 0.
\end{aligned}$$

Apparently the eigenvalues of A are -7 and 6. We can also find the corresponding eigenvectors by resubstituting these values back into $A\mathbf{x} = \lambda\mathbf{x}$.

1.5 Positive definite matrices

Definition 1.5.1 (Positive definiteness)**.** Let $A \in \mathbb{R}^{d\times d}$ be a symmetric matrix ($A^\top = A$). We call

$$\begin{aligned} A \text{ positive definite} : &\iff \forall \mathbf{x} \neq \mathbf{0} \in \mathbb{R}^d : \ \mathbf{x}^T A\mathbf{x} > 0 && (1.12)\\ A \text{ negative definite} : &\iff \forall \mathbf{x} \neq \mathbf{0} \in \mathbb{R}^d : \ \mathbf{x}^T A\mathbf{x} < 0 && (1.13)\\ A \text{ positive semi definite} : &\iff \forall \mathbf{x} \neq \mathbf{0} \in \mathbb{R}^d : \ \mathbf{x}^T A\mathbf{x} \geq 0 && (1.14)\\ A \text{ negative semi definite} : &\iff \forall \mathbf{x} \neq \mathbf{0} \in \mathbb{R}^d : \ \mathbf{x}^T A\mathbf{x} \leq 0. && (1.15) \end{aligned}$$

1.6 Gradient

Definition 1.6.1 (Gradient)**.** Let $f : \mathbb{R}^d \to \mathbb{R}$ be differentiable. We define the *gradient* by:

$$\nabla f = \operatorname{grad} f := \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right)^\top. \tag{1.16}$$

We observe that ∇f is again a function. Each component of the gradient tells us how fast our function changes in each direction. To see how fast the change is at a point $\mathbf{p}$ into direction $\mathbf{v}$, we multiply $\nabla f(\mathbf{p}) \cdot \mathbf{v}$. Observe that this scalar product is maximized if $\mathbf{v}$ is parallel to $\nabla f(\mathbf{p})$, which shows that ∇f points towards the direction of the steepest ascent.

1.7 Hessian Matrix

Definition 1.7.1 (Hessian Matrix)**.** Let $f : \mathbb{R}^d \to \mathbb{R}$. If all second partial derivatives of f exist and are continuous over the domain of the function, then the *Hessian matrix* (also simply called the *Hessian*) is defined by:

$$\mathrm{H}_f(x) := \left(\frac{\partial^2 f}{\partial x_i \partial x_j}(x)\right)_{i,j=1,\ldots,d} = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1}(x) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(x) & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(x) \\ \frac{\partial^2 f}{\partial x_2 \partial x_1}(x) & \frac{\partial^2 f}{\partial x_2 \partial x_2}(x) & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n}(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(x) & \frac{\partial^2 f}{\partial x_n \partial x_2}(x) & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n}(x) \end{pmatrix}. \tag{1.17}$$

Example 1.7.2 (Hessian Matrix)**.** Let $f : \mathbb{R}^2 \to \mathbb{R}, f(x, y) = x^3 + y^3 - 3xy$.
Let us try to calculate the Hessian matrix of f in a point (x, y). First we need to find the partial derivatives. We have:

$$\begin{aligned} \frac{\partial f}{\partial x}(x, y) &= 3x^2 - 3y \\ \frac{\partial f}{\partial y}(x, y) &= 3y^2 - 3x. \end{aligned}$$

We know that

$$\frac{\partial^2 f}{\partial x \partial x} = \frac{\partial}{\partial x}\left(\frac{\partial f}{\partial x}\right) = 6x$$
$$\frac{\partial^2 f}{\partial x \partial y} = \frac{\partial}{\partial x}\left(\frac{\partial f}{\partial y}\right) = -3$$
$$\frac{\partial^2 f}{\partial y \partial x} = \frac{\partial}{\partial y}\left(\frac{\partial f}{\partial x}\right) = -3$$
$$\frac{\partial^2 f}{\partial y \partial y} = \frac{\partial}{\partial y}\left(\frac{\partial f}{\partial y}\right) = 6y.$$

With this information, we can now state the Hessian matrix:

$$H = \begin{pmatrix} 6x & -3 \\ -3 & 6y \end{pmatrix}.$$

Theorem 1.7.3. *Let $H_f(x)$ be the Hessian (matrix) of a function f in a point $\mathbf{x}$, as defined in Definition 1.7.1. Then the following statements hold:*

- *For any $\mathbf{x}$, $H_f(\mathbf{x})$ is symmetric.*
- *If f is a convex function, then $H_f(\mathbf{x})$ is positive semi definite for any $\mathbf{x}$.*
- *If $H_f(\mathbf{x})$ is positive-definite at a critical point $\mathbf{x}$ (meaning $\nabla f(\mathbf{x}) = \mathbf{0}$), then f attains an isolated local minimum at $\mathbf{x}$.*
- *If $H_f(\mathbf{x})$ is negative-definite at a critical point $\mathbf{x}$, then f attains an isolated local maximum at $\mathbf{x}$.*

Chapter 2: Linear Classifiers & SVM

Classification is the problem of identifying to which of a set of categories a new observation belongs, on the basis of a training set of data elements whose category membership is known. In this chapter, we consider a simplification of classification, namely binary classification. In binary classification the set of categories has cardinality 2. In the context of binary classification, we consider linear classifiers which are fairly simple but work well surprisingly often.

2.1 Formal Setting

Let $\mathcal{X}$ (input space) and $\mathcal{Y}$ (label space) be some sets. A given set of the form

$$D := \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\} \subseteq \mathcal{X} \times \mathcal{Y}$$

is called training data, where

$$\mathbf{x_1}, \dots, \mathbf{x_n}$$

are our inputs and $y_1, \dots, y_n$ are the corresponding labels. We call

$$X := (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n)$$

the data matrix, whose columns consist of the input data points. Our goal is to write a function $f : \mathcal{X} \to \mathcal{Y}$ (prediction function), which correctly predicts the label of yet unseen data with the help of our training data. We call the elements of $\mathcal{Y}$ classes. If there are finitely many classes, we also call f a classifier. The algorithm used to find such an f, with the help of the training data is called *learning machine* or *learning algorithm*. This process is also called *training*. Unless otherwise stated, our setting will be $\mathcal{X} = \mathbb{R}^d$, $\mathcal{Y} = \{-1, 1\}$ and $D := \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\} \subseteq \mathcal{X} \times \mathcal{Y}$ as our training data. This is the *binary classification* setting.

2.2 Linear Classifier

Definition 2.2.1 (Linear Classifier)**.** A classifier of the form $f(\mathbf{x}) = \text{sign}(\mathbf{w}^\top \mathbf{x} + b)$ is called a *linear classifier.*

Example 2.2.2 (Nearest centroid classifier)**.** The idea of NCC is the following: Given our training data, look at the two clusters (groups of data points):

$$A := \{\mathbf{x}_i \ : (\mathbf{x}_i, y_i) \in D,\ y_i = -1\}, \quad B := \{\mathbf{x}_i \ : (\mathbf{x}_i, y_i) \in D,\ y_i = +1\}.$$

These clusters partition our training data into two groups. To find the label of a new data point, we check whether the new data point is closer to the centroid of cluster A or the centroid of cluster B and give it the label -1 or $+1$, respectively.

Algorithm:

- **Training:** Compute the centroids $\mathbf{c}_{-1} = \dfrac{1}{|A|} \sum\limits_{x \in A} \mathbf{x}$, $\mathbf{c}_{+1} = \dfrac{1}{|B|} \sum\limits_{x \in B} \mathbf{x}$.
- **Prediction:** Given a new data point $\mathbf{x_k}$, set $y_k := \arg\min_{l \in \{-1,+1\}} \|\mathbf{c}_l - \mathbf{x}\|$.

Theorem 2.2.3. *NCC is a linear classifier.*

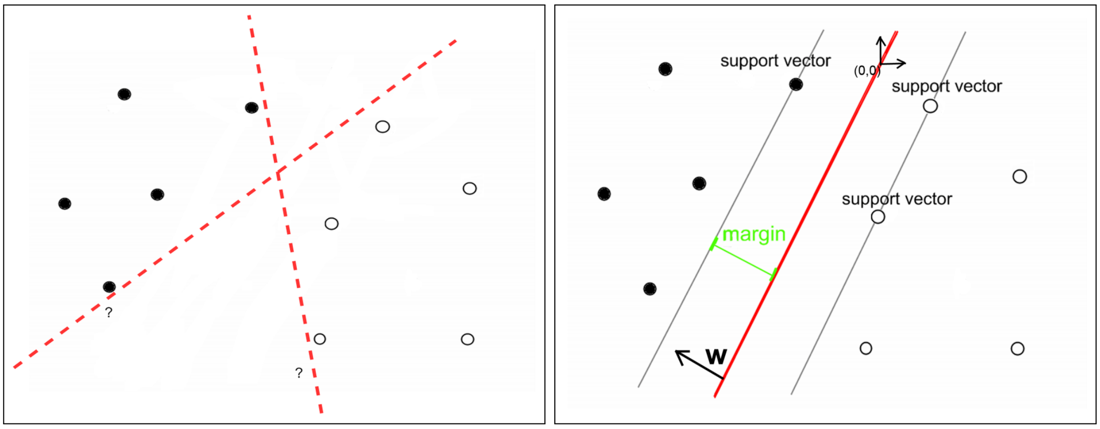

Figure 2.1: Visualization of linear classifiers. data points with label -1 are black and with label $+1$ are white. Left: Two potential classifiers. Right: The hard-margin SVM H and its margin hyperplanes H_+ and H_-.

Proof. We want to write the condition whether a new data point is closer to $\mathbf{c_{-1}}$ or to $\mathbf{c_{+1}}$ in the form $\text{sign}(\mathbf{w}^\top \mathbf{x} + b = 0)$. As NCC classifies points based on which centroid is closer, points that are equidistant to both centroids should be on the sought hyperplane.

So we want that $\mathbf{w}^\top \mathbf{x} + b = 0 \iff \|\mathbf{x} - \mathbf{c}_{-1}\| = \|\mathbf{x} - \mathbf{c}_{+1}\|$. By expanding the right side we get:

$$\begin{aligned}
\|\mathbf{x} - \mathbf{c}_{-1}\| = \|\mathbf{x} - \mathbf{c}_{+1}\| &\iff \|\mathbf{x} - \mathbf{c}_{-1}\|^2 = \|\mathbf{x} - \mathbf{c}_{+1}\|^2 \\
&\iff \|\mathbf{x}\|^2 - 2\mathbf{x}^\top \mathbf{c}_{-1} + \|\mathbf{c}_{-1}\|^2 = \|\mathbf{x}\|^2 - 2\mathbf{x}^\top \mathbf{c}_{+1} + \|\mathbf{c}_{+1}\|^2 \\
&\iff -2\mathbf{x}^\top \mathbf{c}_{-1} + 2\mathbf{x}^\top \mathbf{c}_{+1} + \|\mathbf{c}_{-1}\|^2 - \|\mathbf{c}_{+1}\|^2 = 0 \\
&\iff \underbrace{2\left(\mathbf{c}_{+1} - \mathbf{c}_{-1}\right)^\top}_{=:\mathbf{w}} \mathbf{x} + \underbrace{\|\mathbf{c}_{-1}\|^2 - \|\mathbf{c}_{+1}\|^2}_{=:b} = 0.
\end{aligned}$$

□

This shows that NCC is a linear classifier. Notice that $\mathbf{w} := 2\left(\mathbf{c}_{-1} - \mathbf{c}_{+1}\right)^\top$ and $b := \|\mathbf{c}_{+1}\|^2 - \|\mathbf{c}_{-1}\|^2$ would also yield a linear classifier, but the two hyperplanes would have swapped signed distance. To see which formulation matches with our setting, it suffices to plug in the centroids and see whether the labels match. The NCC is a first example of linear classifiers. There are more to come in the following.

2.3 Support Vector Machine (SVM)

We seek the best hyperplane separating our data. To this end, consider Fig. 2.1 (left). Intuitively the best (fairest) hyperplane would be the one that separates the data with the largest margin, visualized in Fig. 2.1 (right). We can formalize this thought mathematically: Assume there is a line that can separate our data. Let

$$H = \{\mathbf{x} \in \mathbb{R}^d | \mathbf{w}^\top \mathbf{x} + b = 0\},$$

with $\mathbf{w} \neq 0$, be a hyperplane. The margin γ is the size of the open space around the hyperplane that no points occupy. Its boundaries on either side can be described by

$$H_+ = \{\mathbf{x} \in \mathbb{R}^d | \mathbf{w}^\top \mathbf{x} + b = \gamma\}$$

and

$$H_- = \{\mathbf{x} \in \mathbb{R}^d | \mathbf{w}^\top \mathbf{x} + b = -\gamma\}.$$

We additionally require all points with label $+1$ to be "above" H_+ and all points with label -1 to be "below" H_-. Our goal is to maximize γ by choosing appropriate $\mathbf{w}$ and b. Recall the signed distance from Proposition 1.3.6:

$$d(\mathbf{x}, H) = \frac{1}{\|\mathbf{w}\|} \left(\mathbf{w}^\top \mathbf{x} + b\right).$$

We wish to not just choose *any* hyperplane, we wish to choose our hyperplane such that the classification is done correctly. Since we will use the signed distance for classification we need,

$$d(\mathbf{x_i}, H) \geq 0 \text{ if } y_i = +1$$

$$d(\mathbf{x_i}, H) \leq 0 \text{ if } y_i = -1.$$

We simplify these two constraints by the new constraint:

$$y_i \cdot d(\mathbf{x_i}, H) \geq 0. \tag{2.1}$$

Finally, the distance between any point and the hyperplane should be at least γ, giving the constraint:

$$y_i \cdot d(\mathbf{x_i}, H) \geq \gamma \iff y_i \left(\mathbf{w}^\top \mathbf{x}_i + b\right) \geq \|\mathbf{w}\|\gamma. \tag{2.2}$$

Lastly, we make the observation that if (2.2) is satisfied, (2.1) is trivially satisfied because $\gamma \geq 0$ ($\gamma \geq 0$ holds because γ=0 is a trivial lower bound of our maximization problem). This gives us our maximization problem:

$$\begin{aligned} &\max_{\gamma, b \in \mathbb{R}, \mathbf{w} \in \mathbb{R}^d \setminus \{0\}} \gamma \\ &\text{s.t. } y_i \left(\mathbf{w}^\top \mathbf{x}_i + b\right) \geq \|\mathbf{w}\|\gamma, \quad \forall i = 1, \ldots, n. \end{aligned} \tag{2.3}$$

Definition 2.3.1 (Hard-margin SVM)**.** The mathematical program (2.3) is called the *hard-margin SVM.*

We state a few problems with the hard-margin SVM. Firstly, data points are not always linearly separable (see Fig. 2.2). Also outlier points can potentially corrupt the SVM (see Fig. 2.3). To

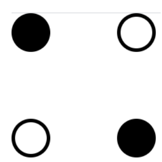

Figure 2.2: A set of data points that is not linearly separable

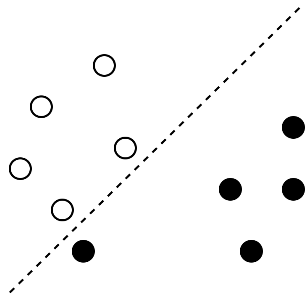

Figure 2.3: An outlier point forces us to choose a suboptimal hyperplane.

fix this issue, we closer inspect (2.3). A first idea could be to allow some more freedom in the constraints. Not every data point needs to have distance of γ to the hyperplane. If

$$\xi_i \geq 0 \quad \forall i = 1, \dots, n,$$

then we can formalize this by:

$$y_i \left(\mathbf{w}^\top \mathbf{x}_i + b \right) \geq \|\mathbf{w}\| \gamma - \xi_i, \quad \forall i = 1, \dots n.$$

By choosing ξ_i large enough, our problem becomes unbounded. To not let this happen, we need to somehow penalize our maximization function, every time we use ξ_i. One way could be:

$$\max_{\gamma, b \in \mathbb{R}, \mathbf{w} \in \mathbb{R}^d \setminus \{0\}, \xi_1, \dots, \xi_n \geq 0} \quad \gamma - C \sum_{i=1}^{n} \xi_i,$$

where C is a constant telling us how harsh to penalize. The higher C, the more we penalize violations. Finally we get:

$$\begin{aligned} \max_{\gamma, b \in \mathbb{R}, \mathbf{w} \in \mathbb{R}^d \setminus \{0\}, \xi_1, \dots, \xi_n \geq 0} \quad & \gamma - C \sum_{i=1}^{n} \xi_i \\ \text{s.t.} \quad & y_i \left(\mathbf{w}^\top \mathbf{x}_i + b \right) \geq \|\mathbf{w}\| \gamma - \xi_i \quad && \forall i = 1, \dots n \\ & \xi_i \geq 0 && \forall i = 1, \dots n. \end{aligned} \tag{2.4}$$

Definition 2.3.2 (Soft-margin SVM)**.** The mathematical program (2.4) is called the *soft-margin SVM.*

Denote by γ^*, $\mathbf{w}^*$, and $b*$ the optimal arguments of (2.4).

Definition 2.3.3 (Support Vector)**.** All vectors $\mathbf{x}_i$ with $y_i \cdot d\left(\mathbf{x}_i, H\left(\mathbf{w}^*, b^*\right)\right) \leq \gamma^*$ are called *support vectors.*

Observation 2.3.4. We notice that (2.4) allows the existence of support vectors. Also notice that the value of (2.4) only depends on these support vectors. Removing all other data points would not have an effect on the outcome of (2.4) because, if $\mathbf{x_i}$ is not a support vector, then $\xi_i = 0$. Now that we have the optimization problem (2.4), let us see in the following chapters how to solve it.

Chapter 3: Convex Optimization

3.1 Convex Functions

Definition 3.1.1 (Convex Set)**.** A set $\mathcal{X} \subset \mathbb{R}^d$ is called *convex* if and only if the line segment connecting any two points in $\mathcal{X}$ entirely lies within $\mathcal{X}$, that is,

$$\forall \mathbf{x}, \mathbf{x}' \in \mathcal{X}, \quad \forall \theta \in [0,1] : (1-\theta)\mathbf{x} + \theta \mathbf{x}' \in \mathcal{X}.$$

Example 3.1.2 (Hyperplanes are convex sets)**.** Let $H = \{\mathbf{x} \in \mathbb{R}^d : \mathbf{w}^\top \mathbf{x} + b = 0\}$ be a hyperplane. We will now show that hyperplanes are convex. Let $\mathbf{x_1}, \mathbf{x_2} \in H$ and $\theta \in [0,1]$. Then $\mathbf{w}^\top \mathbf{x_1} = -b$, $\mathbf{w}^\top \mathbf{x_2} = -b$, and

$$\mathbf{w}^\top((1-\theta)\mathbf{x}_1 + \theta \mathbf{x}_2) + b = (1-\theta)\mathbf{w}^\top \mathbf{x_1} + \theta \mathbf{w}^\top \mathbf{x_2} + b = (1-\theta)(-b) - \theta b + b = 0.$$

Definition 3.1.3 (Convex function)**.** A function $f : \mathbb{R}^d \to \mathbb{R}$ is *convex* if and only if the set above the graph is *convex* or equivalently,

$$\forall \mathbf{x}, \mathbf{x}' \in \mathbb{R}^d, \forall \theta \in [0,1] : f\left((1-\theta)\mathbf{x} + \theta \mathbf{x}_2\right) \leq (1-\theta) f\left(\mathbf{x}_1\right) + \theta f\left(\mathbf{x}_2\right).$$

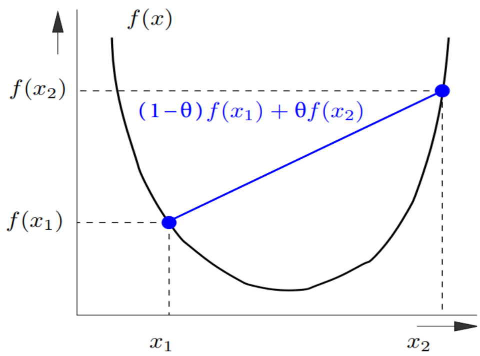

Figure 3.1: A convex function.

Definition 3.1.4 (Concave function)**.** A function f is *concave* if and only if $-f$ is convex.

Recall from Chapter 1 that f is convex if and only if the Hessian matrix $H_f(\mathbf{x})$ is positive semi-definite for all $\mathbf{x} \in \mathbb{R}^d$.

Theorem 3.1.5. *The following statements regarding a symmetric matrix $M \in \mathbb{R}^{d \times d}$ are equivalent:*

1. *M is positive semi-definite (we write $M \succeq 0$).*
2. *$\mathbf{x}^\top M \mathbf{x} \geq 0$ for all $\mathbf{x} \in \mathbb{R}^d$.*
3. *All eigenvalues of M are non-negative.*

Example 3.1.6 (Convex Function)**.** Define f as follows :

$$f : \begin{array}{c} \mathbb{R}^2 \to \mathbb{R} \\ (x_1, x_2) \mapsto x_1^2 + x_2^2 - 7. \end{array}$$

Note that

$$\nabla f \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2x_1 \\ 2x_2 \end{pmatrix} \text{ and } H_f(\mathbf{x}) = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}.$$

Observe that $H_f(\mathbf{x})$ is a diagonal matrix, and we can read the eigenvalues from the diagonal. As the eigenvalues are non negative it follows that $H_f(\mathbf{x})$ is positive semi-definite and that f is convex.

3.2 Convex Optimization Problems

Let $f_0, f_1, \ldots, f_n, g_1, \ldots, g_m : \mathcal{X} \to \mathbb{R}$.

Definition 3.2.1 (Optimization Problem)**.** An *optimization problem* (OP) is of the form:

$$\begin{array}{rll} \min_{\mathbf{x}\in\mathcal{X}} & f_0(\mathbf{x}) & \\ \text{s.t.} & f_i(\mathbf{x}) \leq 0, & i = 1, \ldots, n \\ & g_j(\mathbf{x}) = 0, & j = 1, \ldots, m. \end{array}$$

where we call:

- f_0 the *objective function.*
- $f_i \quad \forall i = 1 \ldots n$ the *inequality constraints.*
- $g_j \quad \forall j = 1 \ldots m$ the *equality constraints.*.

We observe that this special form is not really a restriction. As for a set S we have that

$$\max_{x \in S} x = -\min_{x \in S} -x.$$

The inequality constraints can be brought in this form by bringing everything to one side and multiplying by -1 if necessary.

Definition 3.2.2 (Convex Optimization Problem)**.** An OP is called convex if and only if the functions $f_0, f_1, \ldots, f_n$ are convex and $g_1, \ldots, g_m$ are linear.

Example 3.2.3 (Convex Optimization Problem)**.** Recall the hard-margin SVM (2.2).

$$\begin{array}{rl} \max\limits_{\gamma, b\in\mathbb{R}, \mathbf{w}\in\mathbb{R}^d\setminus\{0\}} & \gamma \\ \text{s.t.} & y_i \left(\mathbf{w}^\top \mathbf{x}_i + b\right) \geq \|\mathbf{w}\|\gamma \quad \forall i = 1 \ldots n. \end{array}$$

Let us check if this is a convex OP. By the definition above we need to check whether the following is convex for each i:

$$f(\gamma, b, \mathbf{w}) = \|\mathbf{w}\|\gamma \ - y_i \left(\mathbf{w}^\top \mathbf{x}_i + b\right).$$

We can show that even in the case $b = 0$ and $\mathbf{w} > 0 \in \mathbb{R}_+$, the function is not convex. Let us verify this for the simple case of $d = 1$. Then the objective function is just:

$$f(\gamma, w) = w\gamma - y_i(wx_i).$$

Let us calculate the gradient:

$$\nabla f \begin{pmatrix} \gamma \\ w \end{pmatrix} = \begin{pmatrix} w \\ \gamma - y_i x_i \end{pmatrix}.$$

Then the hessian matrix

$$H_f(\mathbf{x}) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},$$

is not positive semi-definite as $\det(H_f(x)) = -1$, which shows that the hard-margin SVM is not a convex OP.

3.3 Making the SVM convex

In the previous example, we saw that the hard-margin SVM is not a convex OP, but, as we will show, we can make it convex.

Theorem 3.3.1. *The linear hard-margin SVM, that is,*

$$\max_{\gamma, b\in\mathbb{R}, \mathbf{w}\in\mathbb{R}^d\setminus\{0\}} \gamma \text{ } s.t. \text{ } y_i\left(\mathbf{w}^\top\mathbf{x}_i + b\right) \geq \|\mathbf{w}\|\gamma,$$

can be equivalently rewritten in convex form as given below:

$$\begin{array}{ll} \min\limits_{b\in\mathbb{R}, \mathbf{w}\in\mathbb{R}^d} & \frac{1}{2}\|\mathbf{w}\|^2 \\ s.t. & 1 - y_i\left(\mathbf{w}^\top\mathbf{x}_i + b\right) \leq 0 \quad \forall i = 1, \ldots, n. \end{array}$$

Proof. We show the proof for data that is linearly separable. We also assume our data being separable, so $\gamma = 0$ will not be an optimal solution. So it suffices to find a convex OP for

$$\max_{\gamma>0, b\in\mathbb{R}, \mathbf{w}\in\mathbb{R}^d\setminus\{0\}} \gamma \text{ s.t. } y_i\left(\mathbf{w}^\top\mathbf{x}_i + b\right) \geq \|\mathbf{w}\|\gamma. \tag{3.1}$$

Our SVM will give us parameters $\mathbf{w}$ and b which we will use to define the classifier

$$f(\mathbf{x}) = \operatorname{sign}\left(\mathbf{w}^\top\mathbf{x} + b\right).$$

Notice these parameters are not unique as

$$\forall\lambda > 0 : \operatorname{sign}\left(\mathbf{w}^\top\mathbf{x} + b\right) = \operatorname{sign}(\underbrace{\lambda\mathbf{w}^T}_{=:\mathbf{w}_\lambda}\mathbf{x} + \underbrace{\lambda b}_{=:b_\lambda}). \tag{3.2}$$

Now observe that $\max\gamma$ has the same behavior as $\min\dfrac{1}{2\gamma^2}$. Decreasing/increasing γ has the same effect on both. So we can write (3.1) as

$$\min_{\gamma>0, b\in\mathbb{R}, \mathbf{w}\in\mathbb{R}^d\setminus\{0\}} \frac{1}{2\gamma^2} \text{ s.t. } y_i\left(\mathbf{w}^\top\mathbf{x}_i + b\right) \geq \|\mathbf{w}\|\gamma. \tag{3.3}$$

According to (3.2), we get that (3.3) is equivalent to

$$\min_{\gamma>0, b\in\mathbb{R}, \mathbf{w}\in\mathbb{R}^d\setminus\{0\}} \quad \frac{1}{2\gamma^2} \quad \text{s.t.} \quad y_i\left(\lambda\mathbf{w}^\top\mathbf{x}_i + \lambda b\right) \geq \lambda\|\mathbf{w}\|\gamma. \tag{3.4}$$

We can rename λb as b as both are any scalar in $\mathbb{R}$. So (3.4) is equivalent to

$$\min_{\gamma>0, b\in\mathbb{R}, \mathbf{w}\in\mathbb{R}^d\setminus\{0\}} \quad \frac{1}{2\gamma^2} \quad \text{s.t.} \quad y_i\left(\lambda\mathbf{w}^\top\mathbf{x}_i + b\right) \geq \lambda\|\mathbf{w}\|\gamma. \tag{3.5}$$

Choose $\lambda := \dfrac{1}{\|\mathbf{w}\|\,\gamma} > 0$ to write (3.5) in the form

$$\min_{\gamma>0, b\in\mathbb{R}, \mathbf{w}\in\mathbb{R}^d\setminus\{0\}} \frac{1}{2\gamma^2} \text{ s.t. } y_i \left(\frac{\mathbf{w}^\top \mathbf{x}_i}{\|\mathbf{w}\|\gamma} + b \right) \geq 1. \tag{3.6}$$

Substituting $\tilde{\mathbf{w}} := \dfrac{\mathbf{w}}{\|\mathbf{w}\|\gamma}$ and rewriting $\|\tilde{\mathbf{w}}\| = \left\| \dfrac{\mathbf{w}}{\|\mathbf{w}\|\gamma} \right\| = \dfrac{\|\mathbf{w}\|}{\|\mathbf{w}\|\gamma} = 1/\gamma$, we get:

$$\min_{b\in\mathbb{R}, \tilde{\mathbf{w}}\in\mathbb{R}^d\setminus\{0\}} \frac{1}{2}\|\tilde{\mathbf{w}}\|^2 \quad \text{s.t.} \quad y_i \left(\tilde{\mathbf{w}}^\top \mathbf{x}_i + b \right) \geq 1. \tag{3.7}$$

Assuming the set of data points contains at least one point from each class $(\exists i, j : (\mathbf{x}_i, 1), (\mathbf{x}_j, -1))$, we can now allow $\tilde{\mathbf{w}} = 0$, since $y_i \left(\tilde{\mathbf{w}}^\top \mathbf{x}_i + b \right) = y_i b \geq 1$ cannot be satisfied for $\tilde{\mathbf{w}} = 0$, because $\mathbf{w} = 0$ will never be in the set we optimize over. Finally we get (3.7) into,

$$\min_{b\in\mathbb{R}, \mathbf{w}\in\mathbb{R}^d} \frac{1}{2}\|\mathbf{w}\|^2 \quad \text{s.t.} \quad 1 - y_i \left(\mathbf{w}^\top \mathbf{x}_i + b \right) \leq 0. \tag{3.8}$$

□

It is easy to verify that (3.8) is indeed convex: The objective is convex, quadratic and the constraints linear.

Theorem 3.3.2. *We can formulate the linear soft-margin SVM, that is,*

$$\begin{aligned} \max_{\gamma, b\in\mathbb{R}, \mathbf{w}\in\mathbb{R}^d\setminus\{0\}, \xi_1,\ldots,\xi_n\geq 0} \quad & \gamma - C\sum_{i=1}^n \xi_i \\ \textit{s.t.} \quad & y_i \left(\mathbf{w}^\top \mathbf{x}_i + b \right) \geq \|\mathbf{w}\|\gamma - \xi_i, \quad && \forall i = 1,\ldots,n \\ & \xi_i \geq 0 && \forall i = 1,\ldots,n, \end{aligned}$$

as a convex OP of the form :

$$\begin{aligned} \min_{\mathbf{b}\in\mathbb{R}, \mathbf{w}\in\mathbb{R}^d, \boldsymbol{\xi}\in\mathbb{R}^n} \quad & \tfrac{1}{2}\|\mathbf{w}\|^2 + C\textstyle\sum_{i=1}^n \xi_i \\ \textit{s.t.} \quad & 1 - \xi_i - y_i \left(\mathbf{w}^\top \mathbf{x}_i + b \right) \leq 0, \quad -\xi_i \leq 0 \quad \forall i = 1,\ldots,n. \end{aligned} \tag{3.9}$$

3.4 Solving convex optimization problems

We saw that we can rewrite our soft/hard-margin SVM as convex optimization problems. Now it remains to find an algorithm to solve them. We will use the following useful theorem.

Theorem 3.4.1. *Every locally optimal point of a convex optimization problem is also globally optimal.*

Apparently it suffices to only search for a local optimum to find a global one in convex functions. The algorithm we will use is *gradient descent*. Its idea is to start at a random point and continuously go in the direction of steepest descent. The direction of steepest descent is the negative of the direction of steepest ascent, which is the direction of the gradient.

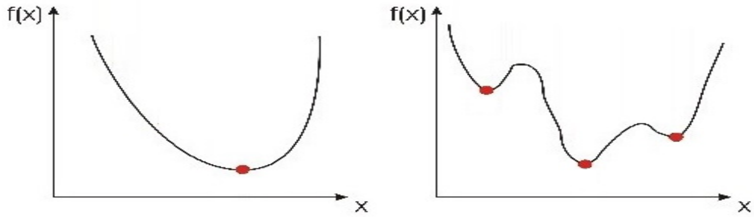

Figure 3.2: Left we see a convex function, to the right a non-convex function

Algorithm Gradient Descent

Input: A convex function f, point $\mathbf{x_1}$ (e.g chosen randomly)

```
function GRADDESCENT(f, x_1)
    for t ← 1 to T do
        x_{t+1} ← x_t − λ_t ∇f_0 (x_t)
    end for
    return x_T
end function
```

Definition 3.4.2 (Step Size)**.** We call λ_t the *step size* or *learning rate.*

Observe that finding a good step size is important. If the step size is too small, we need a higher T to reach a global minimum. If the step size is too big, we can overshoot the minimum or the algorithm might even not find a minimum at all. A typical choice would be $\lambda_t := \frac{1}{t}$.

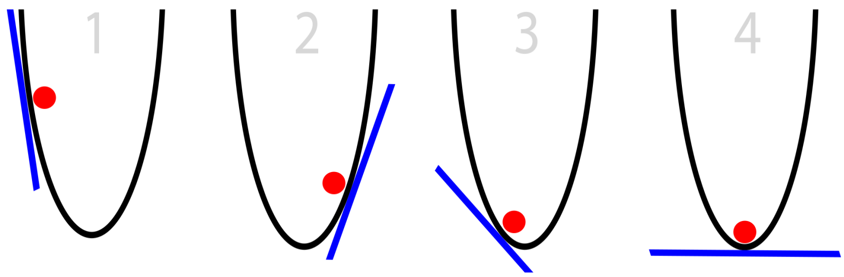

Figure 3.3: Shows the behavior of the gradient descent algorithm.

Theorem 3.4.3. *Let $f_0 : \mathbb{R}^d \to \mathbb{R}$ be an arbitrary (possibly non-convex) objective function. Then, under some assumptions,* [4] *we have:*

[4] The theorem assumes Lipschitz-continuous gradients with a uniformly bounded Lipschitz constant $\exists L : \|\nabla f(\mathbf{x}) - \nabla f(\tilde{\mathbf{x}})\| \leq L\|\mathbf{x} - \tilde{\mathbf{x}}\|$. Convergence is guaranteed for all learning rate schedules satisfying $\sum_{t=1}^{\infty} \lambda_t = \infty$ and $\lambda_t \xrightarrow[t\to\infty]{} 0$, but with varying rates of convergence. The favorable $O(1/t)$ rate is achieved using the minimization rule: $\lambda_t := \arg\min_\lambda f_0\left(\mathbf{x}_t - \lambda \nabla f_0\left(\mathbf{x}_t\right)\right)$. See [1].

- *Gradient descent converges towards a stationary point (maximum, minimum, saddle point), which is in practical purposes usually a minimum.*
- *For ideal choice of the learning rate, the convergence rate is at least as good as:*

$$f_0(\mathbf{x}_t) - f_0(\mathbf{x}^*_{local}) \leq O(1/t).$$

3.5 Applying gradient descent on the convex SVM

Let us recall the convex soft-margin SVM

$$\begin{array}{cl} \min\limits_{\mathbf{b}\in\mathbb{R},\mathbf{w}\in\mathbb{R}^d,\boldsymbol{\xi}\in\mathbb{R}^n} & \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^n \xi_i \\ \text{s.t.} & 1-\xi_i - y_i\left(\mathbf{w}^\top\mathbf{x}_i + b\right) \leq 0, \quad -\xi_i \leq 0 \quad \forall i = 1,\dots n. \end{array}$$

If we look at the gradient descent algorithm, we notice that the input is a convex function. But our OP has one objective function and two inequality constraints for each data point. A quick fix would be to include the two inequality constraints in the objective function. Let us look at the inequality constraints

$$1-\xi_i - y_i\left(\mathbf{w}^\top\mathbf{x}_i + b\right) \leq 0 \iff 1 - y_i\left(\mathbf{w}^\top\mathbf{x}_i + b\right) \leq \xi_i$$

and

$$-\xi_i \leq 0 \iff 0 \leq \xi_i.$$

Together we get that

$$\max\left(1 - y_i\left(\mathbf{w}^\top\mathbf{x}_i + b\right), 0\right) \leq \xi_i,$$

which we can safely plug in as an equality into our objective function as we are minimizing, resulting in the following Proposition:

Proposition 3.5.1. *The convex soft-margin SVM can be rewritten as*

$$\min_{b\in\mathbb{R},\mathbf{w}\in\mathbb{R}^d} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^n \max\left(0, 1 - y_i\left(\mathbf{w}^\top\mathbf{x}_i + b\right)\right).$$

Our new unconstrained objective becomes

$$f_0(\mathbf{w}, b) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^n \max\left(0, 1 - y_i\left(\mathbf{w}^\top\mathbf{x}_i + b\right)\right).$$

Let us look at the part

$$\max\left(1 - y_i\left(\mathbf{w}^\top\mathbf{x}_i + b\right), 0\right).$$

This part of the function is not differentiable. We can see this by replacing $y_i\left(\mathbf{w}^\top\mathbf{x}_i + b\right)$ (a linear growth function) by z resulting in the so called *hinge function*

$$\max(1-z, 0),$$

which has two tangent lines at $z = 1$, one with slope -1 and the other with slope 0. A potential solution would be to just choose any of the derivatives at this point, so we consider the subgradient:

$$\nabla \max\left(1 - y_i\left(\mathbf{w}^\top\mathbf{x}_i + b\right), 0\right) := \begin{cases} \nabla(1 - y_i\left(\mathbf{w}^\top\mathbf{x}_i + b\right)) & \text{if } y_i\left(\mathbf{w}^\top\mathbf{x}_i + b\right) < 1 \\ 0 & \text{else.} \end{cases}.$$

Finally we use the subgradient descent algorithm, which is the normal gradient descent algorithm but using the subgradient in place of the gradient. An alternative solution could be to replace this function with something differentiable exhibiting similar behavior. Let us look at the *logistic function*

$$l(z) := \ln(1 + \exp(-z)).$$

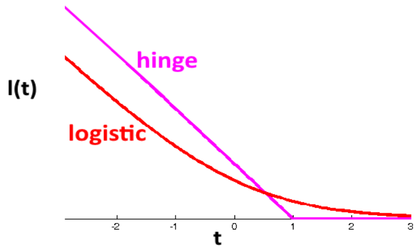

Figure 3.4: We observe that the logistic function is a differentiable approximation to the hinge function.

Definition 3.5.2 (Logistic Regression)**.** Replacing, in the unconstrained convex soft-margin SVM (3.9), the hinge function by the logistic function, we obtain the *logistic regression*:

$$f_0(\mathbf{w}, b) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \ln(1 + \exp(-y_i(\mathbf{w}^\top \mathbf{x_i} + b))). \tag{3.10}$$

As the logistic regression is differentiable, we can use the gradient descent algorithm. Let us discuss a final problem. Notice we need to iterate through all data points to calculate the (sub)gradient. That would mean n iterations, where n can be large for big data. A solution would be to try and approximate the value of the function, leading us to the following algorithm:

Algorithm Stochastic Subgradient Descent (SGD) SVM

Input: A starting point $(\mathbf{w}, b)_1$, batch size B

```
for t ← 1 to T do
    Randomly select B data points
    Denote their indices by I ⊆ {1 ... n}
    (w, b)_{t+1} ← (w, b)_t − λ_t ∇ ( ½‖w_t‖² + (Cn/B) Σ_{i∈I} max(0, 1 − y_i (w_t^⊤ x_i + b_t)) )
end for
return x_T
```

$$(\mathbf{w}, b)_{t+1} \leftarrow (\mathbf{w}, b)_t - \lambda_t \nabla \left(\tfrac{1}{2}\|\mathbf{w_t}\|^2 + \tfrac{Cn}{B} \sum_{i \in I} \max\left(0, 1 - y_i\left(\mathbf{w_t}^\top \mathbf{x}_i + b_t\right)\right)\right)$$

Observation 3.5.3. In SGD we approximate

$$C\sum_{i=1}^{n} \max\left(0, 1 - y_i\left(\mathbf{w}^\top \mathbf{x}_i + b\right)\right)$$

by

$$\frac{Cn}{B}\sum_{i \in I} \max\left(0, 1 - y_i\left(\mathbf{w}^\top \mathbf{x}_i + b\right)\right).$$

The scalar $\frac{n}{B}$ is necessary to rescale the value up as we only calculated the value for B points. The value of $B \in [1 \ldots N]$ called the *batch size* needs to be chosen beforehand. We can use the exact same algorithm also for logistic regression, by just substituting it in step 4 of the algorithm.

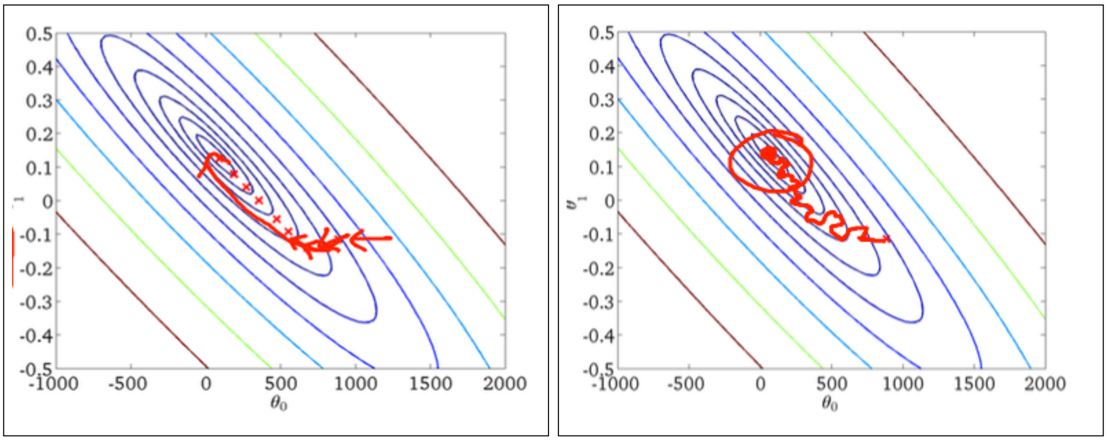

Figure 3.5: Visualization of the gradient descent algorithms. To the left we have the classical gradient descent algorithm and to the right the stochastic gradient descent. Notice how on the right we are not always perfectly going towards the steepest descent as the gradient is only approximated.

Theorem 3.5.4. *Consider SGD using the learning rate $\lambda_t := 1/t$. Then, under mild assumptions, SGD converges with high probability to a stationary point which is usually a minimum with rate:*

$$f_0\left(\mathbf{x}_t\right) - f_0\left(\mathbf{x}_{local}^*\right) \leq O(1/t).$$

Proof. See [1]. □

Chapter 4: Kernel SVM

In the last chapters we studied how to separate data. We mostly ignored the case where the data was not linearly separable, but we will come back to it now.

4.1 Kernel Methods

Kernel Methods are a paradigm to efficiently convert linear learning machines into non-linear ones. At a high level, it works the following way:

1. Define, in a clever way, a non-linear map
$$\phi : \mathbb{R}^d \to \mathbb{R}^D,$$
where $\mathbb{R}^D$ is a very high dimensional space ($D >> d$).
2. Map the inputs into that space,
$$\mathbf{x_i} \mapsto \phi(\mathbf{x_i}), \quad i = 1, \dots, n.$$
3. Separate the data linearly in that space,
$$f(x) = \text{sign}(\langle \mathbf{w}, \phi(x) \rangle + b).$$
4. This linear separation in the high dimensional space, corresponds to a non-linear separation in the input space.

Example 4.1.1. Consider the map :

$$\begin{aligned} \phi :&\mathbb{R}^2 \to \mathbb{R}^3 \\ &(x_1, x_2) \mapsto (x_1^2, \sqrt{2}x_1x_2, x_2^2). \end{aligned}$$

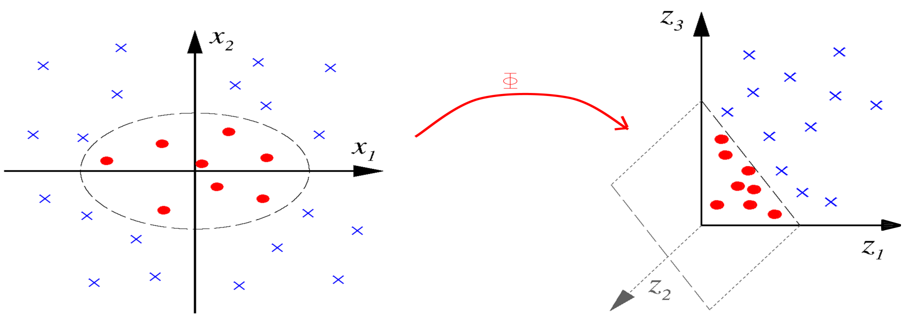

Figure 4.1: Visualization of the mapping ϕ.

Observation 4.1.2. Notice how in Figure 4.1 a linear separation in $\mathbb{R}^3$ corresponds to a non-linear separation in $\mathbb{R}^2$: For instance, for the hyperplane $\mathbf{w}^T\mathbf{x} + b$ defined by:

$$\mathbf{w} := \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \quad \text{and} \quad b := -1,$$

the corresponding separation for $\mathbf{x} = (x_1, x_2)$ in $\mathbb{R}^2$ would be

$$\mathbf{w}^\top \phi(\mathbf{x}) + b = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}^\top \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix} + -1 = x_1^2 + x_2^2 - 1,$$

which would be the unit circle. The image space is usually too high-dimensional to perform operations such as scalar products in it. In our example, we can do the following. Let $\mathbf{x} = (x_1, x_2)$ and $\tilde{\mathbf{x}} = (\tilde{x_1}, \tilde{x_2})$. Let us try to compute their inner product in the image space:

$$\begin{aligned} \phi(\mathbf{x})^\top \phi(\tilde{\mathbf{x}}) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}^\top \begin{pmatrix} \tilde{x_1}^2 \\ \sqrt{2}\tilde{x_1}\tilde{x_2} \\ \tilde{x_2}^2 \end{pmatrix} &= x_1^2\tilde{x_1}^2 + \sqrt{2}\tilde{x_1}\tilde{x_2}\sqrt{2}x_1x_2 + x_2^2\tilde{x_2}^2 \\ &= (x_1\tilde{x_1})^2 + 2(x_1\tilde{x_1}x_2\tilde{x_2}) + (x_2\tilde{x_2})^2 \\ &= (\mathbf{x}^\top\tilde{\mathbf{x}})^2. \end{aligned}$$

Apparently we can compute the scalar product in the image space by only computing scalar products in $\mathbb{R}^2$. This is an example of the kernel trick and $k(\mathbf{x}, \tilde{\mathbf{x}}) := \langle \mathbf{x}, \tilde{\mathbf{x}} \rangle^2$ is an example of a kernel.

Definition 4.1.3 (Kernel)**.** A function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is called *kernel function* (or simply *kernel*) if all of the following holds:

1. it is symmetric, that is: $\forall \mathbf{x}, \mathbf{x}' : k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x}', \mathbf{x})$
2. there exists a map $\phi : \mathbb{R}^d \to \mathcal{H}$ (called kernel feature map into some high-dimensional kernel feature space $\mathcal{H}$ (e.g., $\mathcal{H} = \mathbb{R}^l$ or $\mathcal{H} = \mathbb{R}^\infty$) such that:

$$\forall \mathbf{x}, \tilde{\mathbf{x}} \in \mathbb{R}^d : \quad k(\mathbf{x}, \tilde{\mathbf{x}}) = \langle \phi(\mathbf{x}), \phi(\tilde{\mathbf{x}}) \rangle.$$

We can now formalize the Kernel Trick.

Kernel Trick

1. Formulate the (linear) learning machine (training and prediction) solely in terms of inner products.
2. Replace the inner products $\langle \mathbf{x_i}, \mathbf{x_j} \rangle$ by the kernel $k(\mathbf{x_i}, \mathbf{x_j})$.

Example 4.1.4 (Linear Kernel)**.** The *linear kernel* defined as

$$k(\mathbf{x}, \tilde{\mathbf{x}}) := \langle \mathbf{x}, \tilde{\mathbf{x}} \rangle.$$

is a kernel.

Proof. Choose the identity map $\phi = \mathrm{id}$ Then for all $\mathbf{x}, \tilde{\mathbf{x}} \in \mathbb{R}^d$ we have:

$$k(\mathbf{x}, \tilde{\mathbf{x}}) = \langle \mathbf{x}, \tilde{\mathbf{x}} \rangle = \langle \mathrm{id}(\mathbf{x}), \mathrm{id}(\tilde{\mathbf{x}}) \rangle = \langle \phi(\mathbf{x}), \phi(\tilde{\mathbf{x}}) \rangle.$$

Symmetry follows by the symmetry of the scalar product. □

Example 4.1.5 (Polynomial Kernel)**.** The kernel called *polynomial kernel* of degree $m \in \mathbb{N}$ is defined as

$$k(\mathbf{x}, \tilde{\mathbf{x}}) := (\langle \mathbf{x}, \tilde{\mathbf{x}} \rangle + c)^m$$

where $c \geq 0$ is a parameter.

We finally define one of the most important kernels in practice.

Example 4.1.6 (Gaussian RBF Kernel)**.** The *Gaussian RBF kernel* is a kernel defined as

$$k(\mathbf{x}, \tilde{\mathbf{x}}) := \exp\left(-\frac{1}{2\sigma^2}\|\mathbf{x} - \tilde{\mathbf{x}}\|^2\right).$$

We call the parameter $\sigma^2 > 0$ kernel width (or bandwith).

Definition 4.1.7 (Kernel Matrix)**.** Let $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ be the input data, and let $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ be a kernel function. Then the matrix

$$K := \begin{pmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \ldots & k(\mathbf{x}_1, \mathbf{x}_n) \\ \vdots & & \vdots \\ k(\mathbf{x}_n, \mathbf{x}_1) & \ldots & k(\mathbf{x}_n, \mathbf{x}_n) \end{pmatrix} \in \mathbb{R}^{n \times n}$$

is called kernel matrix.

Theorem 4.1.8 (Closure Properties)**.** *The following properties hold for given functions k, k_1 and k_2.*

1. *If k is a kernel and $c \in \mathbb{R}_+$, then ck is a kernel.*
2. *If k_1 and k_2 are kernels, then $k_1 + k_2$ is a kernel.*
3. *If k_1 and k_2 are kernels, then $k_1 \cdot k_2$ is a kernel*

Theorem 4.1.9. *A function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathcal{H}$ is a kernel if and only if for any $n \in \mathbb{N}$ and any input points $\mathbf{x}_1, \ldots, \mathbf{x}_n$ the matrix K is positive semi-definite (meaning $\forall \mathbf{v} \in \mathbb{R}^n : \mathbf{v}^\top K \mathbf{v} \geq 0$)*

As seen in the following, Theorem 4.1.9 is quite useful for proving functions to be kernels.

Example 4.1.10 (Closure of kernels under addition)**.** Let $\mathbf{x_1}, \ldots, \mathbf{x_n} \in \mathbb{R}^d$ be arbitrary data points, k_1, k_2 be two kernels and K_1, K_2 the corresponding kernel matrices. Define $k := k_1 + k_2$. The kernel matrix K of k will look like the following:

$$\begin{aligned} K &= \begin{pmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \ldots & k(\mathbf{x}_1, \mathbf{x}_n) \\ \vdots & & \vdots \\ k(\mathbf{x}_n, \mathbf{x}_1) & \ldots & k(\mathbf{x}_n, \mathbf{x}_n) \end{pmatrix} \\ &= \begin{pmatrix} k_1(\mathbf{x}_1, \mathbf{x}_1) + k_2(\mathbf{x}_1, \mathbf{x}_1) & \ldots & k_1(\mathbf{x}_1, \mathbf{x}_n) + k_2(\mathbf{x}_1, \mathbf{x}_n) \\ \vdots & & \vdots \\ k_1(\mathbf{x}_n, \mathbf{x}_1) + k_2(\mathbf{x}_n, \mathbf{x}_1) & \ldots & k_1(\mathbf{x}_n, \mathbf{x}_n) + k_2(\mathbf{x}_n, \mathbf{x}_n) \end{pmatrix} \\ &= K_1 + K_2. \end{aligned}$$

We know K_1, K_2 are positive semi-definite. We now prove K being positive semi-definite. It then follows by Theorem 4.1.9 that k is a kernel. Let $\mathbf{v} \in \mathbb{R}^d$. We have:

$$\mathbf{v}^\top K \mathbf{v} = \mathbf{v}^\top (K_1 + K_2) \mathbf{v} = \mathbf{v}^\top (K_1 \mathbf{v} + K_2 \mathbf{v}) = \mathbf{v}^\top K_1 \mathbf{v} + \mathbf{v}^\top K_2 \mathbf{v} \geq 0.$$

Thus K is positive semi-definite and k is a kernel.

4.2 Applying the Kernel Trick to the SVM

Let us recall the unconstrained version of our soft-margin svm (3.9).

$$\min_{b\in\mathbb{R},\mathbf{w}\in\mathbb{R}^d} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \max\left(0, 1-y_i\left(\mathbf{w}^\top \mathbf{x}_i + b\right)\right).$$

Let $\mathbf{w}^*$ be the optimal solution. We use the fact that $\mathbf{w}^* \in \operatorname{span}(\{\mathbf{x_1}, \ldots \mathbf{x_n}\})$, which is a special case of the general representer theorem, which we will prove later on. So there exists $\alpha \in \mathbb{R}^n$ such that

$$\mathbf{w}^* = \sum_{j=1}^{n} \alpha_j \phi(\mathbf{x_j}). \tag{4.1}$$

So we can we also find the optimal solution by optimizing over the possible α. By plugging in (4.1) into (3.9), we get

$$\min_{b\in\mathbb{R},\alpha\in\mathbb{R}^n} \frac{1}{2}\|\sum_{j=1}^{n}\alpha_j\phi(\mathbf{x_j})\|^2 + C\sum_{i=1}^{n}\max\left(0, 1-y_i\left(\left(\sum_{j=1}^{n}\alpha_j\phi(\mathbf{x_j})\right)^\top \phi(\mathbf{x}_i)+b\right)\right). \tag{4.2}$$

By inserting the definition of the norm and then multiplying out we equivalently get

$$\min_{b\in\mathbb{R},\alpha\in\mathbb{R}^n} \frac{1}{2}\left(\sum_{i=1}^{n}\alpha_i\phi(\mathbf{x_i})\right)^\top\left(\sum_{j=1}^{n}\alpha_j\phi(\mathbf{x_j})\right) + C\sum_{i=1}^{n}\max\left(0, 1-y_i\left(\sum_{j=1}^{n}\alpha_j\phi(\mathbf{x_j})^\top\phi(\mathbf{x}_i)+b\right)\right), \tag{4.3}$$

which, by multiplying out and reordering in the last sum, is equivalent to

$$\min_{b\in\mathbb{R},\alpha\in\mathbb{R}^n} \frac{1}{2}\left(\sum_{i,j=1}^{n}\alpha_i\alpha_j\phi(\mathbf{x_i})^\top\phi(\mathbf{x_j})\right) + C\sum_{i=1}^{n}\max\left(0, 1-y_i\left(\sum_{j=1}^{n}\alpha_j\phi(\mathbf{x_i})^\top\phi(\mathbf{x}_j)+b\right)\right). \tag{4.4}$$

As the SVM is used in the higher-dimensional space, we want to make use of our kernel function. We then replace all occurrences of $\phi(\mathbf{x_i})^\top\phi(\mathbf{x}_j)$ by $k\left(\mathbf{x}_i, \mathbf{x}_j\right)$ to finally get:

$$\min_{b\in\mathbb{R},\boldsymbol{\alpha}\in\mathbb{R}^n} \frac{1}{2}\sum_{i,j=1}^{n}\alpha_i\alpha_j k\left(\mathbf{x}_i,\mathbf{x}_j\right) + C\sum_{i=1}^{n}\max\left(0, 1-y_i\left(\sum_{j=1}^{n}\alpha_j k\left(\mathbf{x}_i,\mathbf{x}_j\right)+b\right)\right). \tag{4.5}$$

We finally get the prediction function:

$$\begin{aligned} f(\mathbf{x}) =& \operatorname{sign}\left(\mathbf{w}^{*\top}\phi\left(\mathbf{x}\right)+b\right) \\ =& \operatorname{sign}\left(\left(\sum_{j=1}^{n}\alpha_j\phi(\mathbf{x_j})\right)^\top\phi(\mathbf{x})+b\right) \\ =& \operatorname{sign}\left(\sum_{j=1}^{n}\alpha_j k\left(\mathbf{x},\mathbf{x}_j\right)+b\right). \end{aligned}$$

Notice that we actually do not need the feature map ϕ. Now that we have the training and prediction algorithm, it remains to solve the optimization problem (**??**). For this we use a similar approach to the stochastic gradient method from Chapter 3.

Algorithm Doubly SGD algorithm for kernel SVM

Input: A starting point $(b.\alpha)_1$, batch size B

for $t \leftarrow 1$ to T **do**
  Randomly select B data points
  Denote their indexes by $J \subset \{1, \ldots, n\}$
  $(b, \alpha)_{t+1} := (b, \alpha)_t - \lambda_t \nabla_{(b,\alpha)} \left(\frac{n^2}{2B^2} \sum_{i,j \in J} \alpha_i \alpha_j k(\mathbf{x}_i, \mathbf{x}_j) + \frac{Cn}{B} \sum_{i \in J} \max\left(0, 1 - y_i \left(\frac{n}{B} \sum_{j \in J} \alpha_j k(\mathbf{x}_i, \mathbf{x}_j) + b \right)\right)\right)$
end for
return $(b, \alpha)_T$

Observation 4.2.1. This is basically the same algorithm as SGD. Notice that we approximate

$$\sum_{i,j=1}^{n} \alpha_i \alpha_j k(\mathbf{x}_i, \mathbf{x}_j)$$

by

$$\frac{n^2}{2B^2} \sum_{i,j \in J} \alpha_i \alpha_j k(\mathbf{x}_i, \mathbf{x}_j),$$

as we only calculated the sum for $2B^2$ items, so we need to scale it up. Similar approximations are done in the rest of the algorithm. Just like in SGD, we are giving up a bit of preciseness to be faster.

In the construction of the Kernel SVM, we used the fact that $\mathbf{w}^* \in \operatorname{span}(\{\mathbf{x_1}, \ldots \mathbf{x_n}\})$. We will now prove this using a more general statement.

Theorem 4.2.2 (General Representer Theorem). *Let $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ be a kernel over a Hilbert space $\mathcal{H}$. Let $L : \mathbb{R}^n \to \mathbb{R}$ be any function. Then any solution*

$$\mathbf{w}^* \in \arg\min_{\mathbf{w} \in \mathcal{H}} \frac{1}{2} \|\mathbf{w}\|^2 + L(\langle \mathbf{w}, \phi(\mathbf{x}_1) \rangle, \ldots, \langle \mathbf{w}, \phi(\mathbf{x}_n) \rangle)$$

satisfies:

$$\text{there exist } \alpha_1, \ldots, \alpha_n \text{ such that } \mathbf{w}^* = \sum_{i=1}^{n} \alpha_i \phi(\mathbf{x}_i).$$

Proof. Assume the contrary, that is, there exists an optimal solution with:

$$\mathbf{w}^* = \mathbf{w}_\parallel + \underbrace{\mathbf{w}_\perp}_{\neq 0}$$

with

$$\mathbf{w}_\parallel \in \operatorname{span}(\phi(x_1), \ldots, \phi(x_n)),$$

and

$$\mathbf{w}_\perp \notin \operatorname{span}\left(\phi\left(x_1\right), \ldots, \phi\left(x_n\right)\right), \text{ i.e., } \forall i : \left\langle \mathbf{w}_\perp, \phi\left(\mathbf{x}_i\right)\right\rangle = 0.$$

Then, we have, for all $i = 1, \ldots, n$:

$$\begin{aligned}
\left\langle \mathbf{w}^*, \phi\left(\mathbf{x}_i\right)\right\rangle &= \left\langle \mathbf{w}_\| + \mathbf{w}_\perp, \phi\left(\mathbf{x}_i\right)\right\rangle \\
&= \left\langle \mathbf{w}_\|, \phi\left(\mathbf{x}_i\right)\right\rangle + \underbrace{\left\langle \mathbf{w}_\perp, \phi\left(\mathbf{x}_i\right)\right\rangle}_{=0} \\
&= \left\langle \mathbf{w}_\|, \phi\left(\mathbf{x}_i\right)\right\rangle .
\end{aligned}$$

Furthermore, we have:

$$\left\|\mathbf{w}^*\right\|^2 = \left\|\mathbf{w}_\|\right\|^2 + \underbrace{\left\|\mathbf{w}_\perp\right\|^2}_{>0} > \left\|\mathbf{w}_\|\right\|^2 .$$

Thus the objective function is smaller in the point $\mathbf{w}_\perp$ than it is in the optimum $\mathbf{w}^*$. This contradicts the optimality of $\mathbf{w}^*$. $\square$

Chapter 5: Neural Networks

In the previous chapters, we saw the advantages of linear and kernel learning methods. Kernel methods work well in separating non-linearly separable data, assuming we have a good feature map ϕ. We will now turn to neural networks, whose main advantage will be that they can also learn this feature map. As a motivation let us look at logistic regression in a high-dimensional space:

$$\min_{b\in\mathbb{R},\boldsymbol{w}\in\mathbb{R}^d,\phi} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \ln\left(1+\exp\left(-y_i\left(\mathbf{w}^\top\phi\left(\mathbf{x}_i\right)+b\right)\right)\right).$$

To find this minimum, we also optimize over possible mappings ϕ. The main problem is, there are too many mappings ϕ to consider. As a motivation, we can firstly look at how our brain does this.

5.1 The brain graph

In simple terms, we can consider the brain to be a graph. We call a node *neuron* and an edge *synapse*. Another word for graph is called network, as the nodes are called neurons, we call the brain graph a *neural network*. Let us in very simple terms look at what the brain does, when it

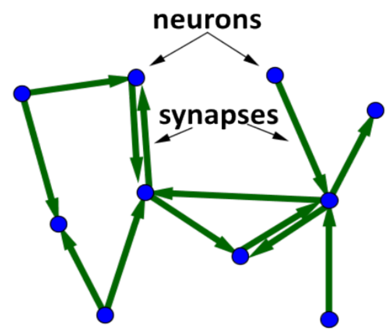

Figure 5.1: Visualization of the brain graph.

sees something.

1. Some neurons will light up (are activated).
2. Activated neuron i will shoot an electrical signal v (*spike*) to neuron j whith strength proportional to W_{ij} (the weight of the synapse).
3. If the in-going electrical signals of neuron j (called the *potential* of j) exceeds a threshold, it will also become activated and thus is able to shoot electrical signals v to its neighboring neurons.

McCulloch and Pitts model
Denote by u the neuron's potential and by v the emitted spike. Then:

$$v = \sigma(u),$$

where σ is the *sigmoid function*: $\sigma(u) = \frac{1}{1+e^{-u}}$.

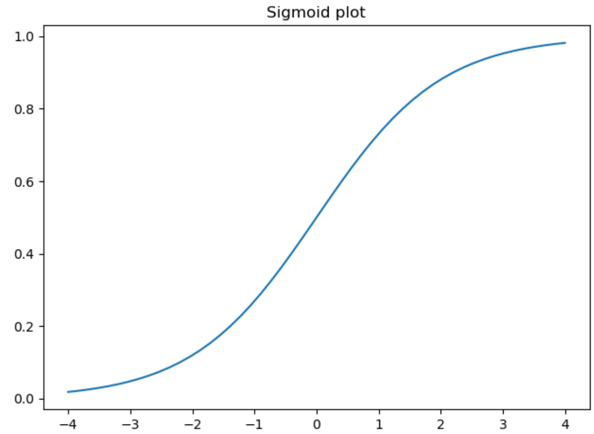

Figure 5.2: The sigmoid function fits the values u between 0 and 1

Today's artificial neural networks use the *ReLU activation* function instead of the sigmoid function

$$\sigma(u) := \max(0, u).$$

5.2 Artifical neural networks (ANN)

Let us formalize, the propagation of neuron activation. Neuron j is connected to other neurons with strength W_{ij} who emit spike v_j. The potential will be

$$u_j = \sum_i w_{ij} v_i$$

and its activation

$$v_j = \sigma(u_j) = \sigma\left(\sum_i w_{ij} v_i\right).$$

Let W be the adjacency matrix of the neural network, $\mathbf{u} = (u_1, \ldots, u_d)^\top$ and $\mathbf{v} = \sigma(\mathbf{u})$ be the activation of all neurons as a vector. Notice that by transposing W, the potential can be calculated as

$$\mathbf{u} = W^\top \mathbf{v}$$

and the activation as

$$\mathbf{v} = \sigma(\mathbf{u}) = \sigma\left(W^\top \mathbf{v}\right).$$

Notice that our equation is dependent on $\mathbf{v}$ on the left and right side. To not allow a cyclic dependency, we restrict our neural network to be an acyclic graph.

5.3 Feed-forward ANN

The acyclic construction from above will be called *feed-forward ANN* or *multi-layer perceptron.*

We use Fig. 5.3 to define some notions. Shown in Fig. 5.3, we have a feed forward network, with an input layer (1) of 5 nodes, two hidden layers (2,3) of 8 and 6 nodes and an output layer

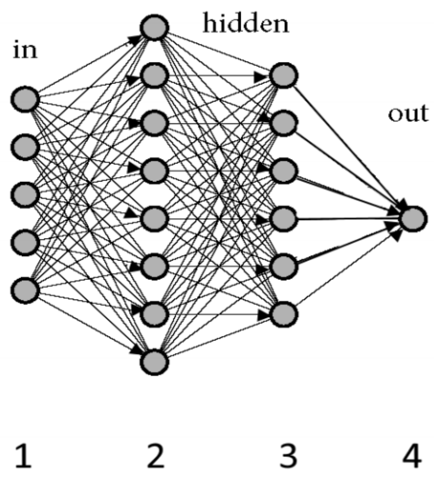

Figure 5.3: Example of a feed forward ANN.

(4) with 1 node. Let $\mathbf{v_l}$ be the vector of activation of neurons in the l-th layer and W_l be the submatrix of strengths of connections between $l-1$-th and l-th layer. Recall that

$$\mathbf{v} = \sigma(\mathbf{u}) = \sigma\left(W^\top \mathbf{v}\right).$$

In a feed forward network, we can calculate v_{l+1} by

$$\mathbf{v}_{l+1} = \sigma(\underbrace{W_{l+1}^\top \mathbf{v}_l}_{=\mathbf{u}_{l+1}}).$$

Let us use this observation. Let 0 and $\mathbf{x} := \mathbf{v_0}$ be our starting layer and L be the number of hidden layers sorted from left to right with $L+1$ being the output layer. We can calculate $\mathbf{v_l}$ as :

$$\phi_W(\mathbf{x}) := \mathbf{v_l} = \sigma\underbrace{\left(W_L^\top \sigma(\ldots \underbrace{\underbrace{\sigma(W_1^\top \underbrace{\mathbf{x}}_{=\mathbf{v}_0}}_{=\mathbf{u}_1})}_{=\mathbf{v1}} \ldots)\right)}_{=\mathbf{v}_l}$$

and the potential of the nodes in the last layer as

$$\mathbf{u_{l+1}} = W_{L+1}^\top \phi_w(\mathbf{x}).$$

So the feature map we just created will have as output the potential u_{L+1}, which will be a scalar, if we have one output node. To wrap it all up, we will get the following optimization problem:

$$\min_{\mathbf{w},W} \frac{1}{2}\|\mathbf{w}\|^2 + \frac{1}{2}\sum_{l=1}^{L}\|W_l\|_{\text{Fro}}^2 + C\sum_{i=1}^{n}\log\left(1+\exp\left(-y_i\mathbf{w}^\top\phi_W\left(\mathbf{x}_i\right)\right)\right), \tag{5.1}$$

where

- $W := (W_1, \ldots, W_L)$,
- $\mathbf{w} = W_{L+1}$,
- $\|W_k\|_{\text{Fro}}^2 = \sum_{ij} w_{kij}^2$,
- $\phi_W(\mathbf{x}_i) := \sigma\left(W_L^\top \sigma\left(\ldots \sigma\left(W_1^\top \mathbf{x}_i\right) \ldots\right)\right)$,

which is just logistic regression with feature map ϕ_W and the regularizer

$$\sum_{l=1}^{L} \|W_l\|_{\text{Fro}}^2,$$

which is a generalization of a vector norm to a matrix norm. Observe that the hyperplane bias parameter b is left out. We can do this as, if needed, the machine can put the bias inside the feature map ϕ_W.

5.4 Training ANN

Now we turn to solving the Optimization problem (**??**). Our approach will again be stochastic gradient descent. We firstly need to compute some gradients. For simplicity let us call

$$F(\mathbf{w}, W) := \frac{1}{2}\|\mathbf{w}\|^2 + \frac{1}{2}\sum_{l=1}^{L} \|W_l\|_{\text{Fro}}^2 + C\sum_{i=1}^{n} \log\left(1 + \exp\left(-y_i \mathbf{w}^\top \phi_W(\mathbf{x}_i)\right)\right).$$

Let us first compute the gradient with respect to $\mathbf{w}$. By linearity, we have:

$$\nabla_{\mathbf{w}} F(\mathbf{w}, W) = \underbrace{\nabla_{\mathbf{w}}\left(\frac{1}{2}\|\mathbf{w}\|^2\right)}_{=\mathbf{w}} + \underbrace{\nabla_{\mathbf{w}}\left(\frac{1}{2}\sum_{l=1}^{L} \|W_l\|_{\text{Fro}}^2\right)}_{=0} + C\sum_{i=1}^{n} \nabla_{\mathbf{w}}\left(\log\left(1 + \exp\left(-y_i \mathbf{w}^\top \phi_W(\mathbf{x}_i)\right)\right)\right).$$

If we denote

$$g(x) := \log(1 + \exp(x)) \text{ with } g'(x) = \frac{\exp(x)}{1 + \exp(x)} = \frac{1}{1 + \exp(-x)}$$

and

$$f_i(\mathbf{w}) = -y_i \mathbf{w}^T \phi_W(\mathbf{x_i}) \text{ with } \nabla_{\mathbf{w}} f_i(\mathbf{w}) = -y_i \phi_W(\mathbf{x_i}),$$

we get:

$$\begin{aligned}
\nabla_{\mathbf{w}}\left(\log\left(1 + \exp\left(-y_i \mathbf{w}^\top \phi_W(\mathbf{x}_i)\right)\right)\right) &= \nabla_{\mathbf{w}}\left(g(-y_i \mathbf{w}^\top \phi_W(\mathbf{x}_i))\right) \\
&= \nabla_{\mathbf{w}}(g(f_i(\mathbf{w})) \\
&= \nabla g\left(f_i(\mathbf{w})\right) \nabla f_i(\mathbf{w}) \\
&= \frac{1}{1 + \exp(y_i \mathbf{w}^T \phi_W(\mathbf{x_i}))} \cdot -y_i \phi_W(\mathbf{x_i}).
\end{aligned}$$

In conclusion, we get

$$\nabla_{\mathbf{w}} F(\mathbf{w}, W) = \mathbf{w} - C\sum_{i=1}^{n} \frac{y_i \phi_W(\mathbf{x}_i)}{1 + \exp\left(y_i \mathbf{w}^\top \phi_W(\mathbf{x}_i)\right)}.$$

Now it remains to find the gradient with respect to $W = (W_1, \dots, W_L)$. Let us take the gradient with respect to each W_l. Again by linearity, we have:

$$\begin{aligned}
\nabla_{W_l} F(\mathbf{w}, W) &= \underbrace{\nabla_{W_l}\left(\frac{1}{2}\|\mathbf{w}\|^2\right)}_{=0} + \underbrace{\nabla_{W_l}\left(\frac{1}{2}\sum_{t=1}^{L}\|W_t\|_{\text{Fro}}^2\right)}_{=W_l} \\
&\quad + C\sum_{i=1}^{n}\nabla_{W_l}\left(\log\left(1+\exp\left(-y_i\mathbf{w}^\top\phi_W\left(\mathbf{x}_i\right)\right)\right)\right) \\
&= W_l - C\sum_{i=1}^{n}\frac{y_i\mathbf{w}^\top\nabla_{W_l}\phi_W\left(\mathbf{x}_i\right)}{1+\exp\left(-y_i\mathbf{w}^\top\phi_W\left(\mathbf{x}_i\right)\right)}.
\end{aligned}$$

It remains to calculate

$$\nabla_{W_l}\phi_W\left(\mathbf{x}_i\right).$$

For a better overview, we calculate $\nabla_{W_{ijl}}\phi_W(\mathbf{x_i})$ with W_{ijl} being the i,j-th entry of the matrix W_l By definition of $\phi_W(\mathbf{x})$, we have:

$$\nabla_{W_{ijl}}\phi_W(\mathbf{x}) := \nabla_{W_{ijl}}\underbrace{\sigma\left(W_L^\top\sigma(\dots\underbrace{\sigma(\underbrace{W_1^\top\underbrace{\mathbf{x}}_{=\mathbf{v}_0}}_{=\mathbf{u}_1})}_{=\mathbf{v_1}}\dots)\right)}_{=\mathbf{v}_l}.$$

We apparently need the derivative of a nested function, for which we will use the chain rule, going up to $\mathbf{u}_l$ for the matrix W_l:

$$\nabla_{w_{ijl}}\phi_W(\mathbf{x}) = \frac{\partial\mathbf{v}_L}{\partial w_{ijl}} = \frac{\partial\mathbf{v}_L}{\partial\mathbf{u}_L}\cdot\frac{\partial\mathbf{u}_L}{\partial\mathbf{v}_{L-1}}\cdot\frac{\partial\mathbf{v}_{L-1}}{\partial\mathbf{u}_{L-1}}\cdots\frac{\partial\mathbf{u}_{l+1}}{\partial\mathbf{v}_l}\cdot\frac{\partial\mathbf{v}_l}{\partial\mathbf{u}_l}\cdot\frac{\partial\mathbf{u}_l}{\partial w_{ijl}}.$$

So we need to calculate three derivatives

- $\frac{\partial\mathbf{v}_l}{\partial\mathrm{u}_l}$
- $\frac{\partial\mathbf{u}_l}{\partial\mathbf{v}_{l-1}}$
- $\frac{\partial\mathbf{u}_l}{\partial w_{ijl}}$.

Let us first define the following auxiliary function

$$\Theta : \mathbb{R} \to \mathbb{R}$$

$$\Theta(c) := \begin{cases} 0 & c \leq 0 \\ 1 & \text{otherwise} \end{cases},$$

where Θ of a vector is applied element-wise:

$$\Theta(\mathbf{x}) := \begin{pmatrix} \Theta\left(x_1\right) \\ \vdots \\ \Theta\left(x_d\right) \end{pmatrix}.$$

Now, we go back to calculating the sought derivatives. For the activation function, we use the ReLU $\sigma(\mathbf{v}) = \max(0, \mathbf{u})$. We have:

$$\frac{\partial \mathbf{v}_l}{\partial \mathbf{u}_l} = \frac{\partial \sigma(\mathbf{u}_l)}{\partial \mathbf{u}_l} = \frac{\partial \max(0, \mathbf{u}_l)}{\partial \mathbf{u}_l} = \Theta(\mathbf{u}_l).$$

Moving on to the next derivative, we have:

$$\frac{\partial \mathbf{u}_l}{\partial \mathbf{v}_{l-1}} = \frac{\partial \left(W_l^\top \mathbf{v}_{l-1}\right)}{\partial \mathbf{v}_{l-1}} = W_l^\top.$$

Let us briefly recall the column-wise matrix multiplication

$$A\mathbf{x} = A_{\cdot,1}x_1 + A_{\cdot,2}x_2 + \cdots + A_{\cdot,n}x_n,$$

where $A_{\cdot,i}$ stands for the i-th column in the matrix A. We will use this to calculate our final derivative. Noticing that transposition interchanges indices, we get:

$$\begin{aligned}
\frac{\partial \mathbf{u}_l}{\partial W_{ijl}} &= \frac{\partial \left(W_l^\top \mathbf{v}_{l-1}\right)}{\partial W_{ijl}} = \frac{\partial \left(W_{\cdot,1,l}^\top \mathbf{v}_{1,l-1} + W_{\cdot,2,l}^\top \mathbf{v}_{2,l-1} + \cdots + W_{\cdot,n}^\top \mathbf{v}_{n,l-1}\right)}{\partial W_{ijl}} \\
&= \frac{\partial \left(W_{\cdot,1,l}^\top \mathbf{v}_{1,l-1}\right)}{\partial W_{ijl}} + \frac{\partial \left(W_{\cdot,2,l}^\top \mathbf{v}_{2,l-1}\right)}{\partial W_{ijl}} + \cdots + \frac{\partial \left(W_{\cdot,n,l}^\top \mathbf{v}_{n,l-1}\right)}{\partial W_{ijl}} \\
&= 0 + 0 + \cdots + \frac{\partial \left(W_{\cdot,i,l}^\top \mathbf{v}_{i,l-1}\right)}{\partial W_{ijl}} + 0 + \cdots + 0 \\
&= \mathbf{v}_{i,l-1}\mathbf{e}_j.
\end{aligned}$$

We can now look back to the chain-rule formulation of $\nabla_{w_{ijl}}\phi_W(\mathbf{x})$. With our calculations so far, it translates to

$$\begin{aligned}
\nabla_{w_{ijl}}\phi_W(\mathbf{x}) &= \frac{\partial \mathbf{v}_L}{\partial \mathbf{u}_L} \cdot \frac{\partial \mathbf{u}_L}{\partial \mathbf{v}_{L-1}} \cdot \frac{\partial \mathbf{v}_{L-1}}{\partial \mathbf{u}_{L-1}} \cdots \frac{\partial \mathbf{u}_{l+1}}{\partial \mathbf{v}_l} \cdot \frac{\partial \mathbf{v}_l}{\partial \mathbf{u}_l} \cdot \frac{\partial \mathbf{u}_l}{\partial w_{ijl}} \\
&= \Theta(\mathbf{u}_L)\, W_L^\top \Theta(\mathbf{u}_{L-1}) \cdots W_{l+1}^\top \Theta(\mathbf{u}_l)\, v_{i,l-1}\mathbf{e}_j.
\end{aligned}$$

Notice that we need to compute each $\mathbf{u}_l$ to use $\Theta(\mathbf{u}_l)$. We can achieve this with the following algorithm.

Algorithm Forward Propagation

```
1: Initialize v_0 = x
2: for l ← 1 to (L − 1) do
3:     u_l := W_l^⊤ v_{l−1}
4:     v_l := σ(u_l)
5: end for
```

Now, we can finally compute our gradient $\nabla_{w_{ijl}} \phi_W(\mathbf{x})$ for all i, j, l with the following algorithm. The algorithm will just calculate our chain-rule formulation.

Algorithm Back Propagation

Initialize $\boldsymbol{\delta}_L := \Theta(\mathbf{u}_L)$
$\forall i, j : \nabla_{w_{ijL}} \phi_W(\mathbf{x}) := \boldsymbol{\delta_L} v_{i,L-1} \mathbf{e}_j$
for $l \leftarrow (L-1)$ to 1 **do**
    $\boldsymbol{\delta}_l := \boldsymbol{\delta}_{l+1} W_{l+1}^{\top} \Theta(\mathbf{u}_l)$
    $\forall i, j : \nabla_{w_{ijl}} \phi_W(\mathbf{x}) := \boldsymbol{\delta_l} \mathbf{v_{i,l-1}} \mathbf{e}_j$
end for

5.5 Convolutional neural networks (CNN)

CNN are a type of artificial neural network. They are mostly used in image and speech recognition. In the following, we will use the example of image classification. In image processing, it makes sense to process portions of an image (called *patches*). An apple could be anywhere in the image, and we need to make sure to identify it regardless of its position. To focus on 'interesting parts' of the image (e.g. the apple stalk), we apply so called *filters*. The high-level idea is to identify parts of an object and then use their position in relation to each other to determine if the object is present. However in practice it is difficult to affirm whether CNN actually follow this basic idea.

CNN incorporate spatial information and weight sharing. Weight sharing is the idea of having multiple weights being equal. To understand spatial information consider an image of an apple. Now randomly permute the pixels of this image. The apple will no longer be recognizable, even though the set of pixel values is still the same. In images, pixels close together have more interaction than pixels far apart, which is what we try to capture in a network. A *convolutional filter* is a filter based on convolutions. Examples include the Gaussian filter(blurring) and Laplace filter (edge detection). You can think of the filter as being the weights of the edges in a neural network. The key point of CNN is to learn the convolutional filter which helps us recognize important parts of the image. Notice that a picture can have many pixels and thus a CNN can get quite large, so we need techniques to reduce their size. *Pooling* is a common choice to reduce a CNN's size. Pooling shrinks many neighboring pixels into one. The typical pooling method is max pooling. However, other pooling methods are also used in specific settings. In the following we give two examples.

- *Max Pooling:* The newly formed pixel will be the maximum of all pixel values in the patch.
- *Average Pooling:* The newly formed pixel will be the average of all pixel values in the patch.

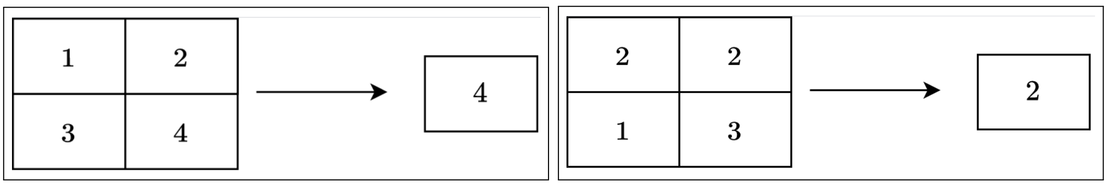

Figure 5.4: Examples of pooling: To the left, we see max pooling, to the right average pooling.

5.6 CNN architecture

We will explain how a CNN layer is set up. First we consider an input, which has a *width*, *height*, and *depth*. The depth is usually referred to as the channel dimension. For typical images this dimension contains different color channels like RGB channels. We will denote width by w, height by h, and the channels by c. Then the input I is in $\mathbb{R}^{h\times w\times c}$. Now a CNN layer is made up of multiple filters, also called kernels. Let f be the number of such filters and let g be their size; i.e. one filter is of size $g \times g \times c$, typically only referred to as $g \times g$. Each filter is applied using a stride, which we call s, and a padding, which we call p. Now to formalize what the CNN layer actually does, we define the size of the output of the layer and then specify how each component of the output can be computed. Afterwards, we will give intuitions into how the output can be interpreted. The spatial dimension of the output can be computed by the following equation:

$$o(i, g, p, s) = \lfloor \frac{i - g + 2p}{s} + 1 \rfloor, \tag{5.2}$$

where i refers to the input size, and o refers to the output size. Now we can compute the output width $ow = o(w, g, p, s)$ and the output height $oh = o(h, g, p, s)$. Thus the output O is finally in $\mathbb{R}^{oh\times ow\times f}$. We typically refer to both the input and the output as images, even though the output can typically not be visualized in the same way the input can. Finally each component of the output can be computed as

$$O_{i,j,k} := \sigma\left(\sum_{l\in\{1,\dots,g\}}\ \sum_{m\in\{1,\dots,g\}}\ \sum_{n\in\{1,\dots,c\}} I_{si+l-p,sj+m-p,n} W^k_{g-l,g-m,c-n}\right), \tag{5.3}$$

where $W^k \in \mathbb{R}^{g\times g\times c}$ are the weights of the k-th filter. Note that in this equation i, j, k, l, m, and n are used as indices and do not refer to the variables defined above. However s, p, and g are the variables as defined above. The indices for W^k might be confusing. This is due to the equation being a convolution. They can also be thought of as $W^k_{l,m,n}$ for simplicity's sake. Now we can talk about the intuitions of what the CNN layer actually computes. Each channel of the output refers to an image of where the feature that is defined by the filter is observable. If, for example, the first filter is an edge detector, i.e. the Laplace filter, then the first slice of O, i.e. $O_{\cdot,\cdot,1}$ would be a black and white image of all edges, or simply the Laplace-filter applied to the original image. However, a slight complication has to be taken into account: a filter does not only have width and height but also depth. Thus this Laplace filter analogy only works, if we either consider only one color channel of the input, or if we consider an input with only one channel, since the Laplace filter is only defined in two dimensions, not three.

One component of the output can be thought of as a neuron. As such it is possible to think of a CNN layer as a modified NN layer. Let $vec(M)$ be the vectorization of matrix M and W be the weight matrix of the layer, i.e. $Wvec(I) = vec(O)$. Since each component in one channel was produced using the same filter, these components share their weights. If $vec(O)_i$ is in the same channel as $vec(O)_j$, then $W_{i,\cdot} = W_{j,\cdot}$. This is weight sharing in mathematical terms: it is effectively a constraint in a typical NN. Also W is quite sparse, as two input components in the same channel with a distance greater than g will never be present in the same sum of equation 5.3. One line in this matrix contains at most g^2c (the weight parameters of the filter) many non-zero values, while its actual size is $hwc >> g^2c$. A typical example for this would be the MNIST dataset, where one image has size $28 \times 28 \times 1$, whereas a filter will typically have size 3×3 or 5×5, $5^2 = 25 < 784 = 28^2$. This is how the parameter count in a CNN is kept quite low.

5.7 Deep Learning

An ANN having 8 or more layers is called *deep neural network*. Deep CNN are state of the art in image classification. We now have a look at the AlexNet, a CNN that competed in the ImageNet Large Scale Visual Recognition Challenge on September 30, 2012. AlexNet was trained by GPU's as they are faster in matrix and vector operations.

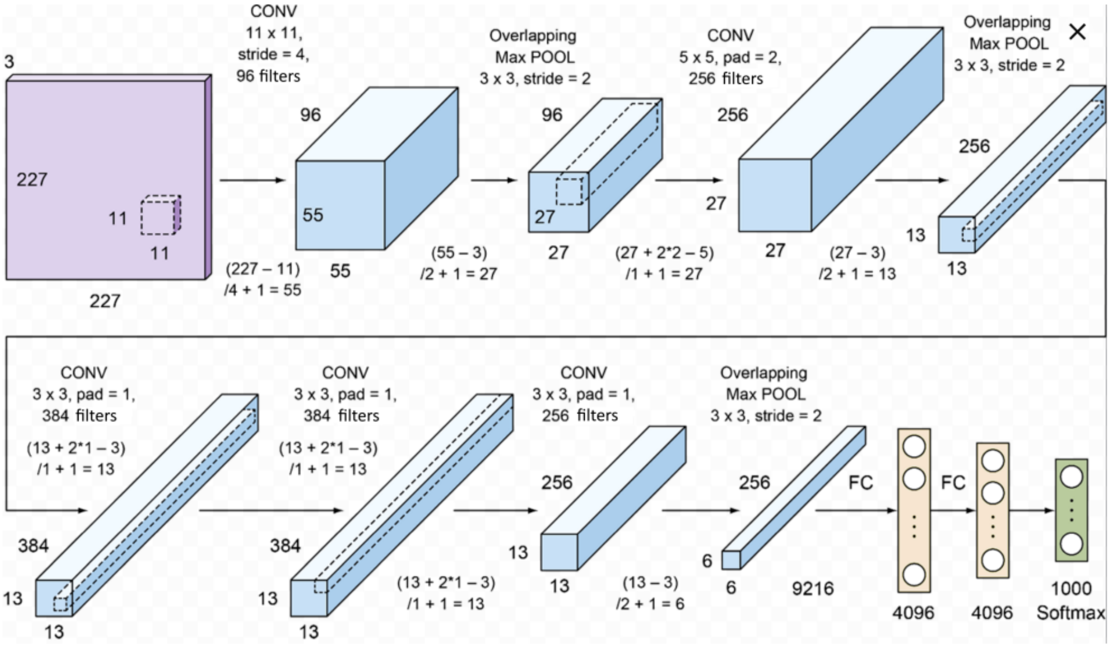

Figure 5.5: The AlexNet consists of 8 layers. It used max pooling and the ReLu activation function. In the image you can see the size of the patches, the number of filters used, and the size of the stride. In the end, we end up with fully connected layers, leading us to the output.

Chapter 6: Overfitting & Regularization

6.1 Overfitting & Underfitting

In the previous chapters, we learned about prediction functions but never talked about how well they describe our data. Our prediction function might be too simple, leading to inaccurate predictions. Also our prediction function might be too complex and adapting too strongly to the data. In Machine learning, we want to find a middle ground: the prediction function should not be too complex nor too simple.

Figure 6.1: Various SVM prediction functions using various Gaussian kernel width. We see that the middle one with moderate width describes our data best.

Definition 6.1.1 (Underfitting)**.** Learning a too simple classifier (not complex enough to describe the training data well) is called *underfitting*.

Definition 6.1.2 (Overfitting)**.** Learning a too complex classifier that fits the training data too well (does not "generalize" to new data) is called *overfitting*.

We can observe the underfitting and overfitting behavior in other learning machines as well. Let us consider the k-nearest neighbor algorithm:

- Choosing k too small, we consider too few points to make generalized prediction, thus leading to overfitting.
- Choosing k too large, we include points that might be too far away to be locally useful, thus leading to underfitting.

Our high level idea to avoid overfitting and underfitting, is:

1. Choose a sufficiently complex classifier to avoid underfitting.
2. Use a technique called *regularization* to avoid overfitting.

6.2 Unifying Loss View

We would like to take a more abstract view on the learning machines we studied so far (SVM, LR, ANN). Let us take a look at the following optimization problem:

$$\min_{[W,]b,\mathbf{w}} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \ell\left(y_i\left(\langle \mathbf{w}, \phi\left(\mathbf{x}_i\right)\rangle + b\right)\right) \left[+\frac{1}{2}\sum_{l=1}^{L}\|W_l\|_{\mathrm{Fro}}^2\right]. \tag{UE}$$

The above general formulation lets us recover several learning problems introduced earlier in this course, as we show in the following:

1. Let $\ell(t) := \max\{0, 1-t\}$ and $\phi := \mathrm{id}$. By leaving out the bracketed values, we obtain our standard soft-margin SVM (3.9):

$$\min_{b,\mathbf{w}} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \max\left(0, 1 - y_i\left(\langle \mathbf{w}, \mathbf{x}_i\rangle + b\right)\right). \tag{SVM}$$

2. Let $\ell(t) := \max\{0, 1-t\}$ and $\phi := \phi_k$. By leaving out the bracketed values, we obtain our kernel SVM (4.5):

$$\min_{b\in\mathbb{R},\mathbf{w}\in\mathbb{R}^d} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \max\left(0, 1 - y_i\left(\langle \mathbf{w}, \phi_k\left(\mathbf{x}_i\right)\rangle + b\right)\right). \tag{Kernel SVM}$$

3. Let $\ell(t) := \ln((1+\exp(-t))$ and $\phi := \mathrm{id}$. By leaving out the bracketed values, we obtain logistic regression (3.10):

$$\min_{b\in\mathbb{R},\mathbf{w}\in\mathbb{R}^d} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \ln\left(1 + \exp\left(-y_i\left(\langle \mathbf{w}, \mathbf{x}_i\rangle + b\right)\right)\right). \tag{LR}$$

4. Let $\ell(t) := \ln((1+\exp(-t))$ and $\phi := \phi_k$. By leaving out the bracketed values, we obtain kernelized logistic regression:

$$\min_{b\in\mathbb{R},\mathbf{w}\in\mathbb{R}^d} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \ln\left(1 + \exp\left(-y_i\left(\langle \mathbf{w}, \phi_k\left(\mathbf{x}_i\right)\rangle + b\right)\right)\right). \tag{Kernel LR}$$

5. Let $\ell(t) := \ln((1+\exp(-t))$, $\phi := \phi_W$ and taking into account the bracketed values, we obtain our ANN (5.1):

$$\min_{b,\mathbf{w},W} \frac{1}{2}\|\mathbf{w}\|^2 + \frac{1}{2}\sum_{l=1}^{L}\|W_l\|_{\mathrm{Fro}}^2 + C\sum_{i=1}^{n} \ln\left(1 + \exp\left(-y_i\left(\mathbf{w}^\top\phi_W\left(\mathbf{x}_i\right) + b\right)\right)\right). \tag{ANN}$$

map ϕ / loss ℓ	id	ϕ_k	ϕ_W
hinge	linear SVM	kernel SVM	[5]
logistic	linear LR	kernel LR	ANN

Table 1: The 5 big learning machines summarized in a table.

We summarize the above results in Table 1. Note that the unifying equation (UE) contains, for every training example $(\mathbf{x}_i, y_i)$, a term

$$\ell(\underbrace{y_i\langle \mathbf{w}, \phi(\mathbf{x})\rangle + b}_{=:t_i}),$$

which we call the *loss* of the i-th example.

Figure 6.2: Large t leads to lower $\ell(t)$ values.

We try to minimize:

$$\sum_{i=1}^{n} \ell\left(y_i\left(\langle \mathbf{w}, \phi\left(\mathbf{x}_i\right)\rangle + b\right)\right),$$

which promotes large t_i values. This is intuitive as $t_i > 0$ means classifying the i-th example correctly. In theory, we are allowed to choose arbitrary loss functions. Let us have a look at the 0-1 loss function.

Definition 6.2.1 (0-1 Loss)**.** The function $\ell(t) := \text{sign}(-t)$ is called *0-1 loss*.

Observation 6.2.2. We consider

$$\ell(t_i) := -\text{sign}(y_i\left(\langle \mathbf{w}, \phi\left(\mathbf{x}_i\right)\rangle + b\right)).$$

As discussed in Chapter 1, we have $\ell(t_i) = 1$ when the label is misclassified ($t_i < 0$) and $\ell(t_i) = 0$ otherwise. Thus the cumulative $0-1$ loss,

$$\sum_{i=1}^{n} \ell\left(y_i\left(\langle \mathbf{w}, \phi\left(\mathbf{x}_i\right)\rangle + b\right)\right),$$

measures the number of training errors. Sadly, optimizing over this discrete problem is NP-hard, so we need to consider the convex alternatives such as the hinge loss and the logistic loss.

[5] The hinge loss is theoretically possible but uncommon in neural networks.

Figure 6.3: The hinge loss and 0-1 loss in comparison.

6.3 Regularization

Let us recall the unifying equation with parameters $\theta := (b, \mathbf{w}, [, W])$ and with the bracketed terms only for ANNs:

$$\min_{[W,]b,\mathbf{w}} \underbrace{\frac{1}{2}\|\mathbf{w}\|^2 \left[+\frac{1}{2}\sum_{l=1}^{L}\|W_l\|^2\right]}_{\text{regularizer } R(\theta)} + \underbrace{C}_{\text{regularization constant}} \left(\underbrace{\sum_{i=1}^{n} \ell\left(y_i\left(\langle \mathbf{w}, \phi\left(\mathbf{x}_i\right)\rangle + b\right)\right)}_{\text{loss } L(\theta)}\right).$$

We want to discuss the influence of the regularization constant C. Increasing the value of C means higher importance of the loss term $L(\theta)$ and lower importance of the regularizer $R(\theta)$. The regularizer $R(\theta)$ will impose a penalty on the complexity of our chosen prediction function. Regularization is a general concept restricting the choice of parameters. We observe that:

High regularization constant $C \implies$ low regularization $\implies$ high overfitting

Low regularization constant $C \implies$ high regularization $\implies$ low overfitting.

Theorem 6.3.1.

$$\min_{\theta} \underbrace{R(\theta)}_{\text{regularizer}} + \underbrace{C}_{\text{regularization constant}} \underbrace{L(\theta)}_{\text{loss}},$$

is equivalent to

$$\begin{array}{ll} \min_\theta & L(\theta) \\ \text{s.t.} & R(\theta) \leq \tilde{C} \end{array}$$

for an appropriate choice of $\tilde{C}$.

Proof. By the Lagrangian duality theorem (L),

$$\begin{aligned}\min_\theta L(\theta) \quad \text{s.t.} \quad R(\theta) \le \tilde{C} &\overset{(L)}{=} \max_{\lambda\ge 0}\min_\theta L(\theta) + \lambda(R(\theta) - \tilde{C}) \\ &\overset{(L)}{=} \min_\theta \max_{\lambda\ge 0} L(\theta) + \lambda(R(\theta) - \tilde{C}) \\ &= \min_\theta L(\theta) + \lambda^*(R(\theta) - \tilde{C}) \\ &= \min_\theta L(\theta) + \underbrace{\lambda^*}_{=:C} R(\theta),\end{aligned}$$

where (θ^*, λ^*) is the optimal value the above min-max problem. □

The proof is important because it tells us how to generate a regularizer from a hard constraint. In other words, the regularizer origin is a constraint. By changing $\tilde{C}$, we can control the size of the set

$$\{\theta : R(\theta) \le \tilde{C}\}.$$

The larger the set, the more likely our learning machine will pick a function that describes our training data too well, so we risk that our prediction function overfits.

6.4 Regularization in Deep Learning

Deep neural networks have a high risk of overfitting due to their complexity. We discuss some special strategies to avoid overfitting in the setting of neural networks.

Consider a large neural network involving many layers and neurons. Clearly, a large neural network can adapt to our training data much better than a small network, leading to the potential of overfitting. Below we give several regularization techniques for neural networks.

Strategy 1: Early Stopping

- Split the training data into two sets:
    1. New smaller training set
    2. Validation set
- Monitor the validation set errors every couple of mini batch iterations
- Stop SGD when the validation error goes up.

Strategy 2: Batch normalization
Batch normalization is a strategy in which we attempt to normalize the input of the activation function. The gradient of an activation function is typically most informative when its input has mean 0 and variance 1. However this is dependent on the activation function. Some examples:

- ReLU: if its input is very large (very small) its gradient is 1 (0). As such ReLU does not seem to be relevant unless its input is both large and small within a batch, in which case its mean will be around 0.
- All sigmoidal functions (i.e. the sigmoid, tanh, err) have vanishing gradients, i.e. for very small or very large inputs their gradient are close to 0. As such, we want their input to have mean 0 and a small variance.

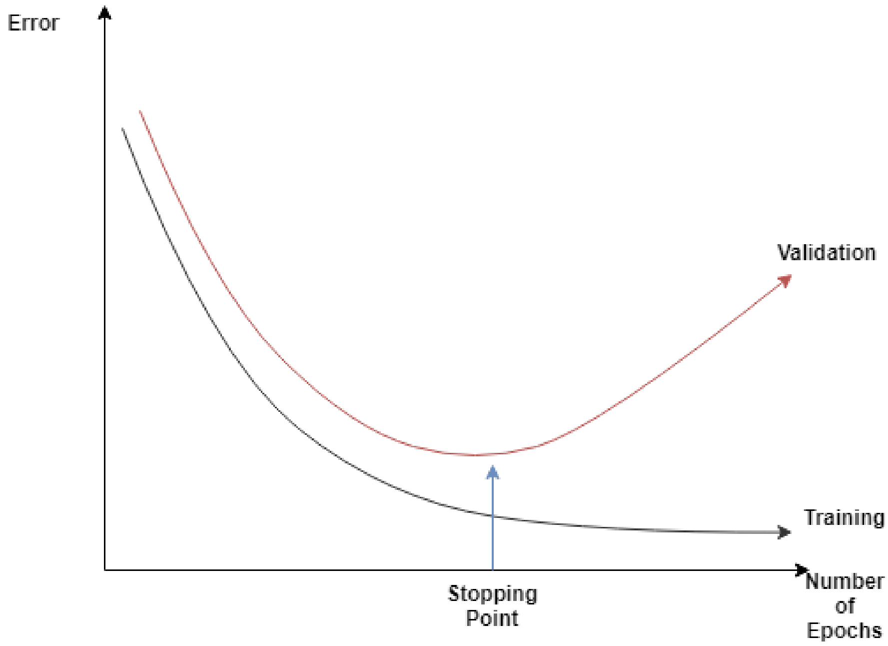

Figure 6.4: Initially, as the number of iterations increases, the neural network describes our control data (validation set) better, so the validation error decreases. Once the network starts overfitting, the validation error will start increasing. This is our ideal stopping point.

With this in mind, we can learn parameters allowing us to normalize the input of an activation function. Batch normalization first normalizes the input by learning the mean μ and variance σ and subtracting this learned mean from the input and dividing by the variance, normalizing the input to mean 0 and variance 1. It additionally allows learning parameters γ and β, which rescale and shift the normalized input. After batch normalization, the input to the activation can have any learned mean and variance.

Algorithm Batch Normalization

Denote, for a neuron f, its activation values on a mini-batch by $\mathbf{f} = (f_1, \dots f_B)$. Then, for each mini-batch in the SGD optimization and every neuron in the ANN, do:

1: Center each neuron activation: $\forall i = 1, \dots, B : f_i \leftarrow f_i - \mu_f$

2: Normalize the spread: $f_i \leftarrow f_i / \sigma_f$

3: Overwrite the neuron's activation by the learned parameters (γ, β) with $\gamma\mathbf{f} + \beta$

Definition 6.4.1 (Ensemble Methods)**.** *Ensemble methods* are methods that aggregate the prediction from multiple predictors.

Example 6.4.2 (Ensemble Method)**.** A typical ensemble method would be:

1. Train k machine learning algorithms, resulting in prediction functions $f_1, \dots, f_k$
2. Predict by majority vote: $f(\mathbf{x}) = +1$ if $|i : f_i(\mathbf{x}) = +1| \geq |i : f_i(\mathbf{x}) = -1|$ and $f(\mathbf{x}) = -1$ elsewise.

The drawback to ensemble methods is that training multiple deep ANN is very costly. We try to bypass this problem by using the following trick.

Strategy 3: Dropout Regularization
We simulate many ANNs by randomly hiding some neurons from our original net in each SGD step. This process is formally called *dropout regularization.* Dropout regularization randomly hides (makes unusable) in each mini-batch SGD iteration a fixed percentage of randomly selected neurons of the network. The intuition is that the network can not focus on one feature, as this feature might not be present during optimization, as it might be hidden. As such the network learns more general representations for concepts.

Strategy 4: Data Augmentation
The optimal setting for optimization is online learning. In this setting there is an infinite amount of data and in each optimization step it uses new previously unseen data. Obviously online learning is not very realistic, save for a very few problems. However, *data augmentation* creates "new training data" from old data. For example, an image of a cat could be mirrored, translated, noised, rescaled, zoomed in, zoomed out, etc. This leads to more informative training data. It promotes generalization, as we are adding variety to our training data.

Strategy 5: Transfer Learning
Assuming we have a small training dataset, we can do the following:

1. Pre-training: Download an ANN from the web that was pre-trained on huge amounts of images, or alternatively train an ANN on some huge dataset.
2. Fine-tuning: Run SGD optimization on our small dataset but using the ANN from the previous step, effectively "fine-tuning" it to the small dataset.

This regularization strategy is known as *transfer learning*, that is, transferring information from a related problem onto a learning problem.

Chapter 7: Regression

Up until now, we only had a look at classification problems. For a prediction function f, we had that $f(x) \in \{-1, +1\}$. Real word problems are not only of the classification kind. Suppose we are given houses as training data, and we want to predict the price of a house. A binary classification doesn't suffice, we need a real-valued prediction function.

The formal setting will be the following:

- Given training data $(\mathbf{x_i}, (y_i))$, with $\mathbf{x_i} \in \mathbb{R}^d, y_i \in \mathbb{R}$.
- Find $f : \mathbb{R}^d \to \mathbb{R}$ with $f(\mathbf{x}) \approx y$ for new data $\mathbf{x}, y$.

In the following, we will look at an old method of solving these kind of problems, called *regression.*

7.1 Linear Regression

The idea of *linear regression* is quite simple. Suppose we are given some data points. A first idea would be to approximate these values linearly, by a hyperplane

$$\mathbf{w}^\top \mathbf{x}.$$

For now, we will intentionally ignore the displacement parameter $+b$ as we will later see that we do not need it. The line would be good if it reduces the error. One way to measure the error of data point i would be the squared distance

$$(y_i - \mathbf{w}^\top \mathbf{x_i})^2.$$

Observe that it is an intuitive choice of error, it is always positive and it is higher if the value predicted by the hyperplane and y_i are far apart. Finally, we want to minimize the error for all data points, so we just sum over them, leading us to the following optimization problem.

Definition 7.1.1 (Least square regression (Legendre))**.**

$$\mathbf{w}_{\mathrm{LS}} := \operatorname*{arg\,min}_{\mathbf{w} \in \mathbb{R}^d} \sum_{i=1}^{n} \left(y_i - \mathbf{w}^\top \mathbf{x}_i\right)^2 = \operatorname*{arg\,min}_{\mathbf{w} \in \mathbb{R}^d} \left\|\mathbf{y} - X^\top \mathbf{w}\right\|^2.$$

We will also define a loss function, exhibiting our choice of error:

Definition 7.1.2 (Least-squares loss)**.** The function $\ell(t, y) := (t - y)^2$ is called *least-squares loss.* Our idea so far is good but is prone to overfitting, because so far our model is only loss concentrated. To make our model less prone to overfitting, we add our known regularizer, to end up with:

Definition 7.1.3 (Ridge regression)**.**

$$\mathbf{w}_{\mathrm{RR}} := \operatorname*{arg\,min}_{\mathbf{w} \in \mathbb{R}^d} \underbrace{\frac{1}{2}\|\mathbf{w}\|^2 + C\left\|\mathbf{y} - X^\top \mathbf{w}\right\|^2}_{=:\mathcal{L}(\mathbf{w})}.$$

We can now talk about how to find our optimum. One way would be to use SGD but it turns out that our problem is much simpler. Ridge regression is an unconstrained optimization problem.

Thus, we find the global optima by differentiation. Let us calculate:

$$\begin{aligned}
\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) =& \nabla_{\mathbf{w}}\left(\frac{1}{2}\|\mathbf{w}\|^2 + C\left\|\mathbf{y} - X^\top\mathbf{w}\right\|^2\right)\\
=& \nabla_{\mathbf{w}}\left(\frac{1}{2}\mathbf{w}^\top\mathbf{w} + C\left(\mathbf{y} - X^\top\mathbf{w}\right)^\top\left(\mathbf{y} - X^\top\mathbf{w}\right)\right)\\
=& \nabla_{\mathbf{w}}\left(\frac{1}{2}\mathbf{w}^\top\mathbf{w} + C\left(\mathbf{y}^\top - \mathbf{w}^\top X\right)\left(\mathbf{y} - X^\top\mathbf{w}\right)\right)\\
=& \nabla_{\mathbf{w}}\left(\frac{1}{2}\mathbf{w}^\top\mathbf{w} + C\left(\mathbf{y}^\top\mathbf{y} - \mathbf{y}^\top X^\top\mathbf{w} - \mathbf{w}^\top X\mathbf{y} + \mathbf{w}^\top XX^\top\mathbf{w}\right)\right)\\
=& \nabla_{\mathbf{w}}\left(\frac{1}{2}\mathbf{w}^\top\mathbf{w} + C\mathbf{y}^\top\mathbf{y} - 2C\mathbf{w}^\top X\mathbf{y} + C\mathbf{w}^\top XX^\top\mathbf{w}\right)\\
=& \mathbf{w} - 2CX\mathbf{y} + 2CXX^\top\mathbf{w}.
\end{aligned}$$

Finally, we set $\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) = 0$ to find our optimum.

$$\begin{aligned}
\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) = 0 &\iff \mathbf{w} - 2CX\mathbf{y} + 2CXX^\top\mathbf{w} = 0\\
&\iff -2CX\mathbf{y} + (I + 2CXX^\top)\mathbf{w} = 0\\
&\iff (I + 2CCXX^\top)\mathbf{w} = 2CX\mathbf{y}\\
&\iff \mathbf{w} = (I + 2CXX^\top)^{-1}2CX\mathbf{y}\\
&\iff \mathbf{w} = 2C(I + 2CXX^\top)^{-1}Xy\\
&\iff \mathbf{w} = \left(\frac{1}{2C}I + XX^\top\right)^{-1}Xy.
\end{aligned}$$

Theorem 7.1.4.

$$\mathbf{w}_{\mathrm{RR}} = \left(XX^\top + \frac{1}{2C}I\right)^{-1}Xy.$$

We have considered a linear model without bias b. However, we can easily incorporate a bias into any linear learning machine (regression, SVM, etc.) by the following trick: Let $\mathbf{x_1}, \dots \mathbf{x_n}$ be our training data and define

$$\tilde{\mathbf{x}}_i := \begin{pmatrix} \mathbf{x}_i \\ 1 \end{pmatrix},$$

and change our training data matrix into

$$\tilde{X} := (\tilde{\mathbf{x}}_1, \dots, \tilde{\mathbf{x}}_n) = \begin{pmatrix} X \\ \mathbf{1}^\top \end{pmatrix}.$$

We can conclude that every solution :

$$\begin{aligned}
\mathbf{w}^* :=& \arg\min_{\mathbf{w}\in\mathbb{R}^{d+1}}\frac{1}{2}\|\tilde{\mathbf{w}}\|^2 + C\left\|\mathbf{y} - \tilde{X}^\top\tilde{\mathbf{w}}\right\|^2\\
=& \arg\min_{\mathbf{w}\in\mathbb{R}^d, b\in\mathbb{R}}\frac{1}{2}\left(\|\mathbf{w}\|^2 + b^2\right) + C\left\|\mathbf{y} - X^\top\mathbf{w} - b\mathbf{1}\right\|^2.
\end{aligned}$$

7.2 Leave one out cross validation (LOOCV)

Let us think about how we can choose our regularization constant C. An idea would be to separate our training data set into a smaller training data set and a test set. We would then measure the average error and choose our best regularization constant. To make things fair, we would also alternate through all possible test sets and training sets. Let us make this idea more formal in the following algorithm.

Algorithm ***k*-fold Cross Validation**

split data into $k \overset{\text{e.g.}}{=} 10$ equally-sized chunks (called "folds")
for $t \leftarrow 1$ to k and $C \overset{\text{e.g.}}{\in} \{0.01, 0.1, 1, 10, 100\}$ **do**
    use ith fold as test set and union of all others as training set
    train learner on training set (using C) and test on test set
end for
output learner with lowest average error

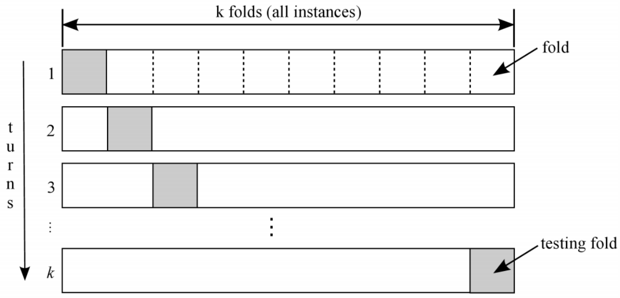

Figure 7.1: Visualization of the k-fold Algorithm.

Back in the classification setting, the meaning of error was pretty clear. Let us see how we can define the meaning of error in the regression setting.

Definition 7.2.1 (Root mean square error)**.** Let $\{(\mathbf{x}_1, y_1), \ldots (\mathbf{x}_n, y_n)\}$ be a test set, and let f be a learned regression function. The *root mean squared error* (RMSE) of f is:

$$\text{RMSE}(f) := \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(f\left(\mathbf{x}_i\right) - y_i\right)^2}.$$

Sometimes our dataset is very small, in such a dataset itwould be intuitive to choose $k = n$.

Definition 7.2.2 (Leave one out cross validation (LOOCV))**.** The special case of $k = n$ in the k-fold Cross Validation is called, *leave one out cross validation.*

We can use LOOCV also for deciding other parameters like kernel width or learning rate in ANN. Sadly the runtime of LOOCV is relatively high.

- In Ridge Regression, we loop over all data points with runtime $O(n)$.

- Each time we train with $n-1$ data points, which needs to invert a matrix with runtime $O(d^3)$.
- Resulting in total runtime of $O(nd^3)$.

Turns out, we can be faster. Let us look at the particular setting of LOOCV. Recall:

$$\mathbf{w}_{\mathrm{RR}} = (\underbrace{XX^\top}_{=\sum_{i=1}^n \mathbf{x}_i\mathbf{x}_i^\top} + \frac{1}{2C}l)^{-1} \underbrace{Xy}_{=\sum_{i=1}^n \mathbf{x}_i y_i} .$$

Let $\mathbf{w_i}$ be the RR solution when the i-th data point is left out during training. We have:

$$\mathbf{w}_i = \left(XX^\top - \mathbf{x}_i\mathbf{x}_i^\top + \frac{1}{2C}I\right)^{-1} (Xy - \mathbf{x}_i y_i) .$$

The error in LOOCV is defined as :

$$\mathrm{RSME}_{\mathrm{loocv}} = \sqrt{\frac{1}{n}\sum_{i=1}^n \left(\mathbf{x}_i^\top \mathbf{w}_i - y_i\right)^2}.$$

We will now see, that we actually do not need to invert n times. We will make use the following theorem:

Theorem 7.2.3 (Sherman-Morrison Formula)**.** *Let $A \in \mathbb{R}^{d\times d}$ be an invertible matrix, and let $\mathbf{u} \in \mathbb{R}^d$. If $\mathbf{u}^\top A^{-1}\mathbf{u} \neq 1$, then:*

$$\left(A - \mathbf{u}\mathbf{u}^\top\right)^{-1} = A^{-1} + \frac{A^{-1}\mathbf{u}\mathbf{u}^\top A^{-1}}{1 - \mathbf{u}^\top A^{-1}\mathbf{u}}$$

Firstly let us rewrite:

$$\mathbf{w}_i = \left(\underbrace{XX^\top + \frac{1}{2C}I}_{:=A} - \mathbf{x}_i\mathbf{x}_i^\top\right)^{-1} (Xy - \mathbf{x}_i y_i)$$

By using the theorem, we get :

$$\begin{aligned}\mathrm{RSME}_{\mathrm{loocv}} &= \sqrt{\sum_{i=1}^n \left(\mathbf{x}_i^\top \mathbf{w}_i - y_i\right)^2} \\ &= \sqrt{\sum_{i=1}^n \left(\mathbf{x}_i^\top \left(A^{-1} + \frac{A^{-1}\mathbf{x}_i\mathbf{x}_i^\top A^{-1}}{1 - \mathbf{x}_i^\top A^{-1}\mathbf{x}_i}\right)(Xy - \mathbf{x}_i y_i) - y_i\right)^2}\end{aligned}$$

This shows we only need $O(d^3)$ iterations to compute LOOCV error. But we can further simplify the equation.

Theorem 7.2.4. *The LOOCV-RMSE of ridge regression can be computed in $O\left(d^3\right)$ through:*

$$\mathrm{RSME}_{\mathrm{loocv}} = \sqrt{\sum_{i=1}^n \left(\frac{\mathbf{x}_i^\top \mathbf{w}_{\mathrm{RR}} - y_i}{1 - \mathbf{x}_i^\top A^{-1}\mathbf{x}_i}\right)^2},$$

where $A := XX^\top + \frac{1}{2C}I$.

Proof. Recalling $\mathbf{w}_{\text{RR}} = A^{-1}Xy$ and denoting $\beta_i := \mathbf{x}_i^\top A^{-1}\mathbf{x}_i$, we get:

$$
\begin{aligned}
\text{RSME}_{\text{loocv}} &= \sqrt{\sum_{i=1}^{n}\left(\mathbf{x}_i^\top\left(A^{-1}+\frac{A^{-1}\mathbf{x}_i\mathbf{x}_i^\top A^{-1}}{1-\mathbf{x}_i^\top A^{-1}\mathbf{x}_i}\right)(Xy-\mathbf{x}_iy_i)-y_i\right)^2} \\
&= \sqrt{\sum_{i=1}^{n}\left(\mathbf{x}_i^\top\mathbf{w}_{RR}+\frac{\beta_i\mathbf{x}_i^\top\mathbf{w}_{RR}}{1-\beta_i}-\beta_iy_i-\frac{\beta_i^2}{1-\beta_i}y_i-y_i\right)^2} \\
&= \sqrt{\sum_{i=1}^{n}\left(\left(1+\frac{\beta_i}{1-\beta_i}\right)\mathbf{x}_i^\top\mathbf{w}_{\text{RR}}-\left(\beta_i+\frac{\beta_i^2}{1-\beta_i}+1\right)y_i\right)^2} \\
&= \sqrt{\sum_{i=1}^{n}\left(\frac{\mathbf{x}_i^\top\mathbf{w}_{\text{RR}}-y_i}{1-\beta_i}\right)^2} = \sqrt{\sum_{i=1}^{n}\left(\frac{\mathbf{x}_i^\top\mathbf{w}_{\text{RR}}-y_i}{1-\mathbf{x}_i^\top A^{-1}\mathbf{x}_i}\right)^2}.
\end{aligned}
$$

□

7.3 Non-linear Regression

As one might think, a linear function is not always the best prediction function. We will see now how to employ non-linearity into regression.

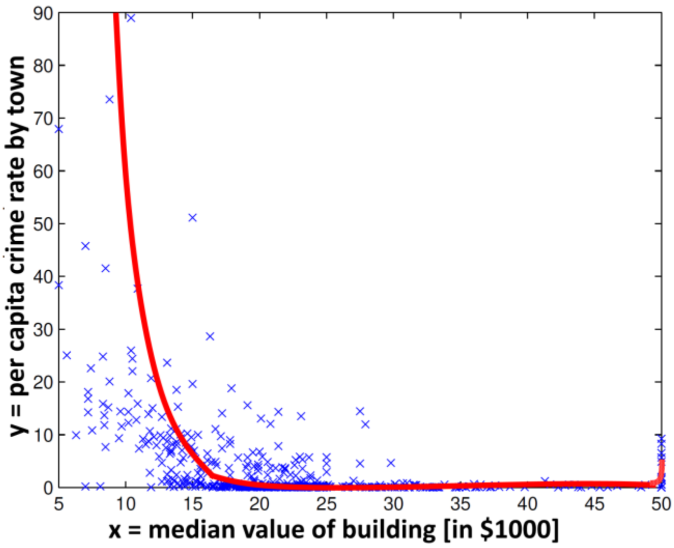

Figure 7.2: The red non-linear function, approximates this dataset much better than any linear function would.

But firstly let us try a quick fix, with which we could still use linear regression. Consider applying a log transformation on the label $y = log(y)$. We have a look at a generalization of the above

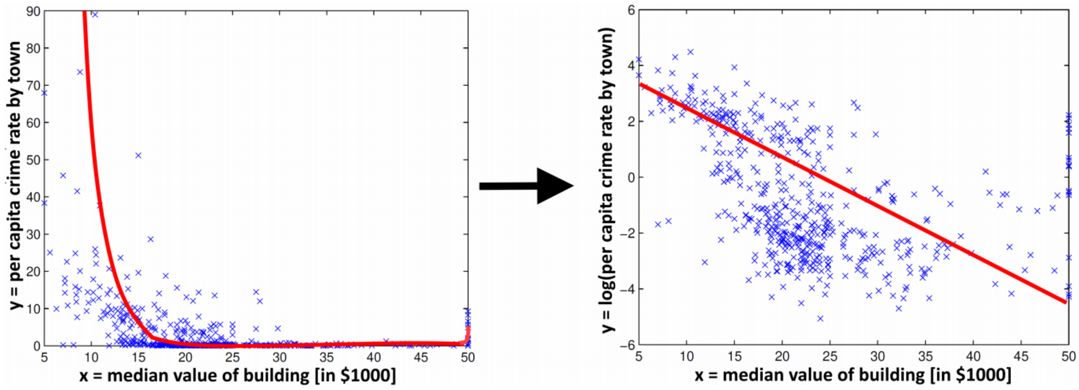

Figure 7.3: After the log transformation, a linear regression line is a reasonable approach.

method.

Definition 7.3.1 (Box Cox Transformation)**.** The *Box-Cox transformation* (or 'power transformation') with parameter λ is:

$$x_{\text{new}} \leftarrow \begin{cases} \log\left(x_{\text{old}}\right) & \text{if } \lambda = 0 \\ \frac{x_{01}^{\lambda}-1}{\lambda} & \text{if } \lambda \neq 0 \end{cases}$$

The parameter λ has intuitive interpretations, as follows:

- $\lambda > 1:$ data is stretched(e.g $\cdot\, \lambda = 2:$ quadratically).
- $0 < \lambda < 1:$ data is concentrated (e.g. $\lambda = 0.5:$ square root.)
- $\lambda = 0:$ log transform.
- $\lambda < 0:$ analogously, with the order of data reversed.

In general, before using any other method, one would try out some transformations on the dataset. If this still does not help, we can use kernel ridge regression.

Definition 7.3.2 (Kernel Ridge Regression)**.** Let k be a kernel with associated feature map $\phi : \mathbb{R}^d \to \mathcal{H}$, i.e., $k(\mathbf{x}, \tilde{\mathbf{x}}) = \langle \phi(\mathbf{x}), \phi(\tilde{\mathbf{x}}) \rangle$. Then *kernel ridge regression* (KRR) is defined as:

$$\mathbf{w}_{KRR} := \underset{\mathbf{w} \in \mathcal{H}}{\arg\min} \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^{n} \|y_i - \langle \mathbf{w}, \phi(x_i) \rangle\|^2 .$$

Let us try use the kernel trick onto KRR. Recall the representer theorem statement, the optimal solution is in the span of the dataset:

$$\exists \boldsymbol{\alpha} \in \mathbb{R}^n : \quad \mathbf{w}_{KRR} = \sum_{i=1}^{n} \alpha_i \phi(\mathbf{x}_i) = \phi(X) \boldsymbol{\alpha},$$

with

$$\phi(X) := \left(\phi(\mathbf{x}_1), \ldots, \phi(\mathbf{x}_n)\right).$$

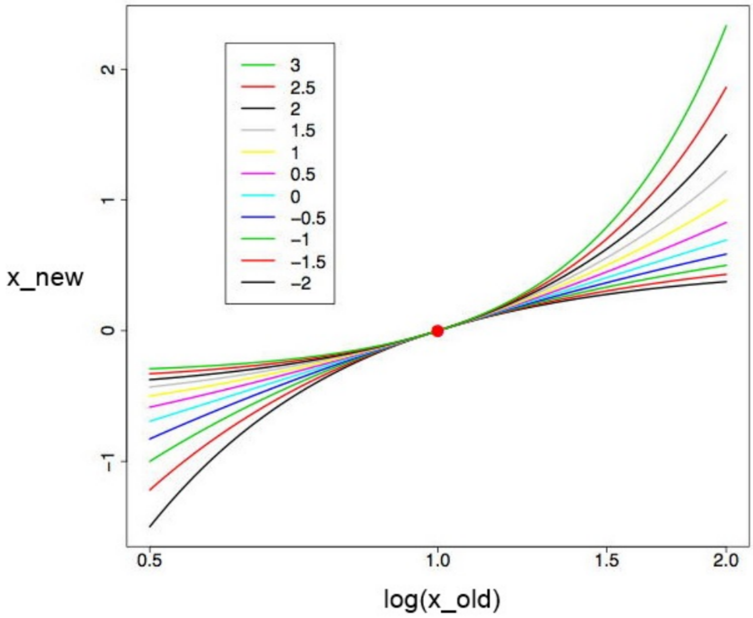

Figure 7.4: Visualization of the Box-Cox transformation. Here we see the parameters λ and their influence.

Recall the definition of the kernel matrix K, we have :

$$\begin{aligned}
&\min_{\mathbf{w}\in\mathbb{R}^d} \frac{1}{2}\|\mathbf{w}\|^2 + C\left\|\mathbf{y}-\phi(X)^\top\mathbf{w}\right\|^2 \\
&= \min_{\alpha\in\mathbb{R}^n} \frac{1}{2} \underbrace{\|\phi(X)\alpha\|^2}_{\boldsymbol{\alpha}^\top \underbrace{\phi(X)^\top\phi(X)}_{=K}\boldsymbol{\alpha}} + C\|\mathbf{y}-\underbrace{\phi(X)^\top\phi(X)}_{=K}\alpha\|^2 \\
&= \min_{\boldsymbol{\alpha}\in\mathbb{R}^n} \frac{1}{2}\boldsymbol{\alpha}^\top K\boldsymbol{\alpha} + C\|\mathbf{y}-K\boldsymbol{\alpha}\|^2.
\end{aligned}$$

By differentiation we can also find the optimal solution, leading us to the following theorem.

Theorem 7.3.3. *The solution of KRR is given by*

$$f(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i k\left(\mathbf{x}_i, \mathbf{x}\right) \qquad \textit{with} \qquad \alpha = \left(K + \frac{1}{2C} I_{n\times n}\right)^{-1} y$$

.

Proof. Left as an exercise. □

The solution can thus be computed in $O\left(n^3\right)$. Usually we use KRR together with a Gaussian kernel, but sometimes it can make sense to use a linear kernel. By definition, linear kernel regression is the same as ridge regression. If $n < d$, it makes sense to use a linear kernel instead of ridge regression, as the matrix we need to invert is of size $n \times n$ as seen above, and the matrix inversion is in $O(n^3)$ time instead of $O(d^3)$. We also use regression in deep learning, for tasks like deciding the price of a house, given a picture. We only need to change the loss to least squares.

Definition 7.3.4 (Deep Regression)**.** Deep regression predicts $f(\mathbf{x}) = \mathbf{w}_*^\top \phi_{W_*}(\mathbf{x})$, where:

$$(\mathbf{w}_*, W_*) := \min_{\mathbf{w},W} \frac{1}{2}\|\mathbf{w}\|^2 + \sum_{l=1}^{L} \|W_l\|_{\text{Fro}}^2 + C\sum_{i=1}^{n} \left(y_i - \mathbf{w}^\top \phi_W(\mathbf{x}_i)\right)^2$$

7.4 Unifying View

Recall our unifying equation:

$$\min_{[W,]b,\mathbf{w}} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \ell\left(y_i\left(\langle \mathbf{w}, \phi(\mathbf{x}_i)\rangle + b\right)\right)\left[+\frac{1}{2}\sum_{l=1}^{L}\|W_l\|_{\text{Fro}}^2\right]. \tag{UE}$$

We would like to also regression into this equation, making it further more general. In order to do so, we define new notation for our loss function:

$$l(t,y) := \begin{cases} (t-y)^2 & \text{for regression} \\ \ell(yt) & \text{for classification} \end{cases}$$

We can now get all of our known loss functions, by plugging in:

- Use $l(t,y) := \max(0, 1-yt)$ for SVM ("hinge loss").
- Use $l(t,y) := \ln(1+\exp(-yt))$ for LR and ANN("logistic loss").
- Use $l(t,y) := (t-y)^2$ for regression.

Similarly, by plugging the following kernel function in each loss functions from above, into UE, we get respectively:

- $\phi :=$ id for linear SVM, linear LR, and RR.
- $\phi := \phi_k$ for kernel SVM, kernel LR, and KRR.
- $\phi := \phi_W$ for ANN and DR.

Definition 7.4.1 (Support Vector Regression (SVR))**.** Using a loss function called ϵ *insensitive loss:*

$$\ell(t,y) := \max(0, |y-t| - \epsilon),$$

we can define the *support vector regression*, whose aim it is to find a line predicting $f(x_i)$, such that the predicted value does not deviate more than ϵ to y_i. This error is captured by the distance of the prediction line and y_i as

$$\mathbf{w}_{\text{SVR}}^* := \arg\min_{\mathbf{w}\in\mathcal{H}} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \max\left(0, |y_i - \langle \mathbf{w}, \phi(\mathbf{x}_i)\rangle| - \epsilon\right).$$

Chapter 8: Clustering

So far, we have only looked at so called *supervised learning* settings. In these scenarios we are given

$$\begin{aligned} \text{Inputs:} \quad & \mathbf{x}_1, ..., \mathbf{x}_n \in \mathbb{R}^d \\ \text{Labels:} \quad & y_1, ..., y_n \\ & y_i \in \begin{cases} \{-1, +1\} & \text{for binary classification} \\ \mathbb{R} & \text{for regression} \end{cases} \end{aligned}$$

Now, we will take a look at what happens if no labels are given, i.e if $y_1, ..., y_n$ are unknown. This setting is called *unsupervised learning* and we will start by looking at so called *clustering* problems.

8.1 Linear Clustering

Definition 8.1.1 (Clustering)**.** Clustering is the process of organizing objects into groups -called clusters- whose members are similar in some way and objects in different clusters are non-similar.

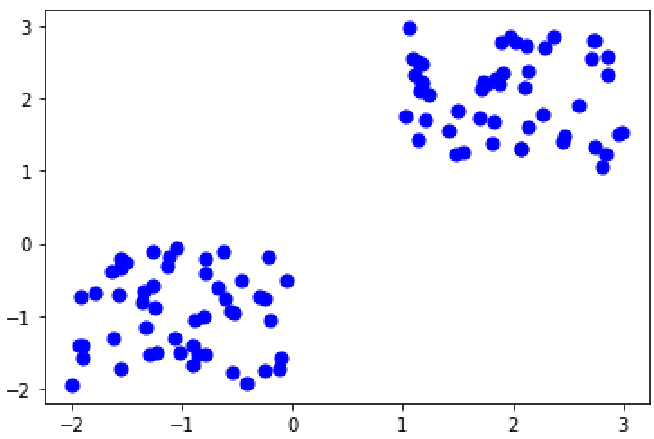

Figure 8.1: Two sets of points in $\mathbb{R}^2$ that could be clustered.

We are now looking for an algorithm that is capable of correctly clustering our inputs $\mathbf{x}_1, ..., \mathbf{x}_n$ in a way that allows us to make meaningful interpretations of the resulting clusters.

Example 8.1.2. In 8.1 the two axes could represent the weight and tail length of two types of mice (transformed such as to fit neatly into the plot), which we want to distinguish from each other. In that case good algorithm should yield the following clusters:

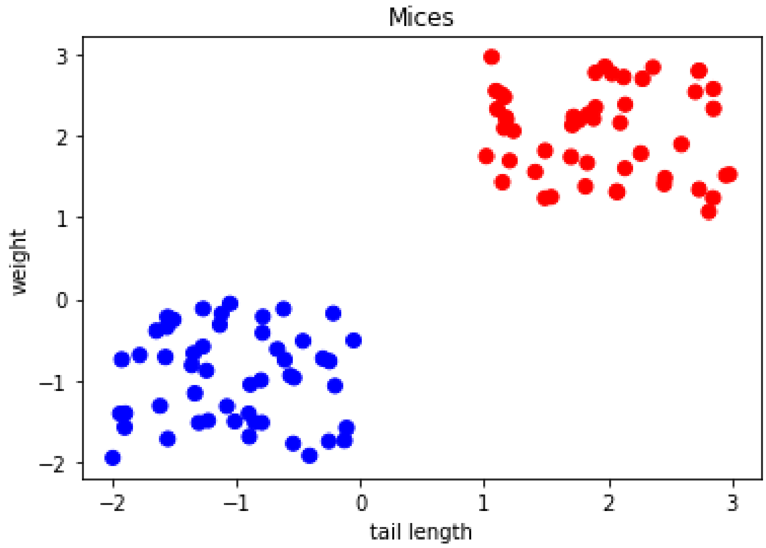

Figure 8.2: Clustered mices.

In the above scenario the clustering was easy to eyeball but since we do not know what the ground truth labels are, we can not be sure that this is the optimal way to cluster our mice. Therefore the results usually need to be inspected manually afterwards.

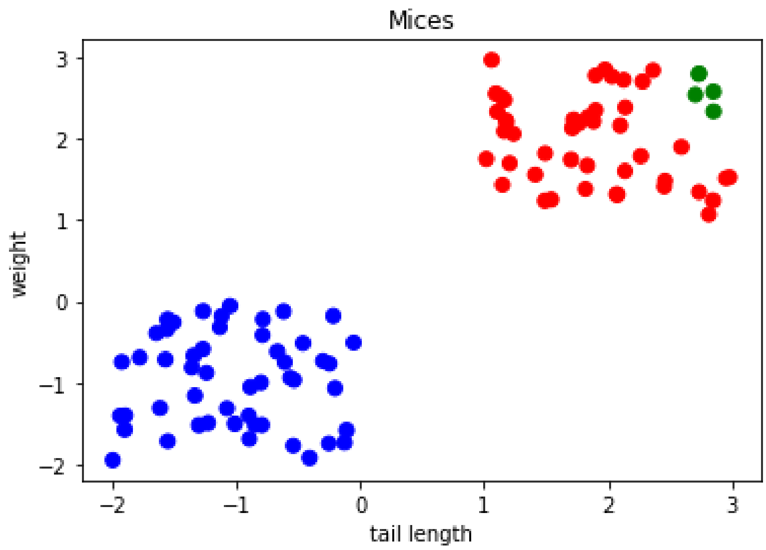

Figure 8.3: What an even better clustering could look like.

8.2 K-means

Now, we will look at arguably the most popular clustering algorithm.

Algorithm K-means

Input: $k \in \mathbb{N}$ and $\mathbf{X} = \mathbf{x}_1, ..., \mathbf{x}_n \in \mathbb{R}^d$

1: **function** K-MEANS($k, \mathbf{X}$)
2: Initialize cluster centers $\mathbf{c}_1, ..., \mathbf{c}_k$ (e.g., randomly drawn inputs)
3: **repeat**
4: **for** i = 1 : n **do**
5: Label the input $\mathbf{x}_i$ as belonging to the nearest cluster

$$y_i = \underset{j=1,...,k}{\arg\min} \|\mathbf{x}_i - \mathbf{c}_j\|^2 .$$

6: **end for**
7: **for** j = 1 : k **do**
8: Compute cluster center $\mathbf{c}_j$ as the mean of all inputs of the j-th cluster,

$$\mathbf{c}_j := \mathrm{mean}(\{\mathbf{x}_i : y_i = j\}).$$

9: **end for**
10: **until** Clusters do not change between subsequent iterations.
11: **return** Cluster centers $\mathbf{c}_1, ..., \mathbf{c}_k$
12: **end function**

The k-means algorithm takes two inputs, k the number of clusters we want to create and a set of inputs $\mathbf{x}_1, ..., \mathbf{x}_n \in \mathbb{R}$ which will usually be passed in form of a matrix $X \in \mathbb{R}^{n \times d}$ where d is the number of features each input $\mathbf{x}_i$ has ($d = 2$ in our mouse example). We then initialize the cluster centers $\mathbf{c}_1, ..., \mathbf{c}_k$ by randomly picking k of our input data points $\mathbf{x_1}, ...\mathbf{x_n}$. Afterwards, we repeat the procedure described in lines 4-9 until our clusters do not change anymore. In the following we have applied the algorithm to our mouse example the white crosses represent the centroids. The colored regions indicate to which cluster a new input $\mathbf{x}_{n'}$ would be assigned.

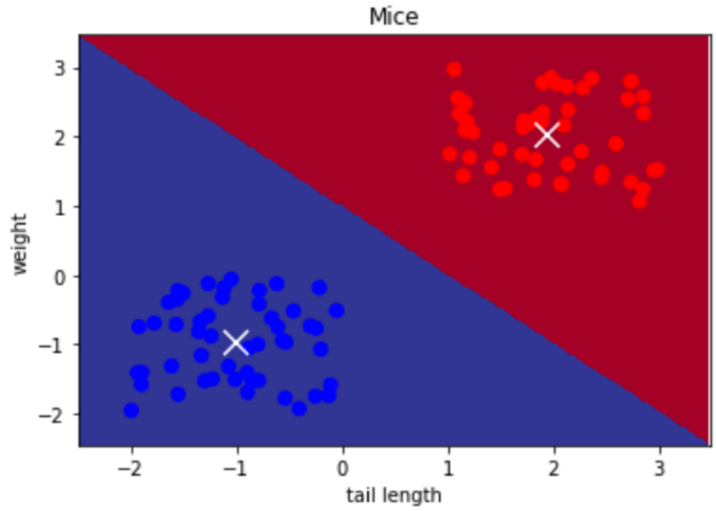

Figure 8.4: K-means clustering applied to our mouse example

Example 8.2.1 (Limitations)**.** We will now see that k-means is limited to finding linear boundaries between classes. That means it will partition the feature space into polygons. This partition is also called *Voronoi diagram*. The polygons consist of a centroid and all points of the space that are closer to this particular centroid than to any other.

Theorem 8.2.2. *K-means finds piecewise linear cluster boundaries*

Proof. We first look at case $k = 2$. Since we only have two clusters we can partition our inputs $\mathbf{x}_1, ..., \mathbf{x}_n$ by assigning the label $y_i = -1$ to every input that lies in cluster one and $y_i = 1$ to every input that lies in cluster two. Therefore we get the partition

$$A := \{x_i : (x_i, y_i) \in M, y_i = -1\} \quad B := \{x_i : (x_i, y_i) \in M, y_i = +1\}.$$

From line 8 of the algorithm we know that the centroids can be computed as

$$\mathbf{c}_{-1} = \frac{1}{|A|} \sum_{x \in A} \mathbf{x}, \quad \mathbf{c}_{+1} = \frac{1}{|B|} \sum_{x \in B} \mathbf{x}.$$

Look back at 2.2 Linear Classifier, where we presented the nearest centroid classifier. Our formulation here is equivalent to it. What happened is that for $k = 2$ k-means outputs the centroids for a NCC which we can then use to predict new points. This also means that the cluster boundaries have to be linear.

If $k > 2$ we look at pairwise linear boundaries between clusters and use the fact that the intersection of linear boundaries creates polygons which are piece-wise linear. $\square$

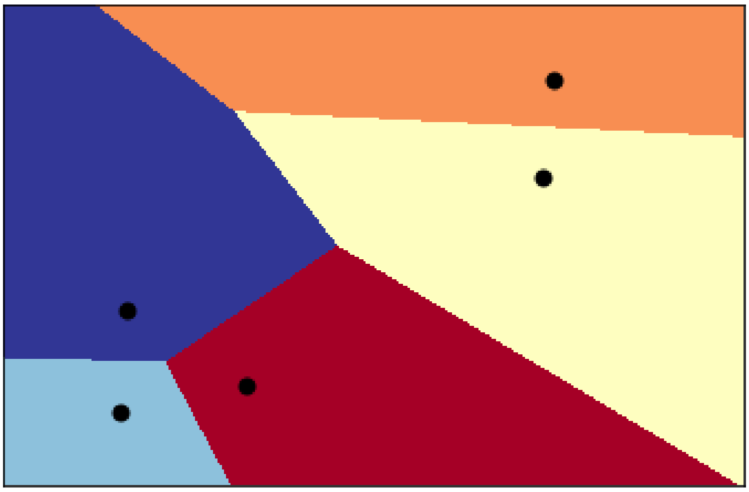

Figure 8.5: Voronoi Diagram of our mice example for $k = 5$

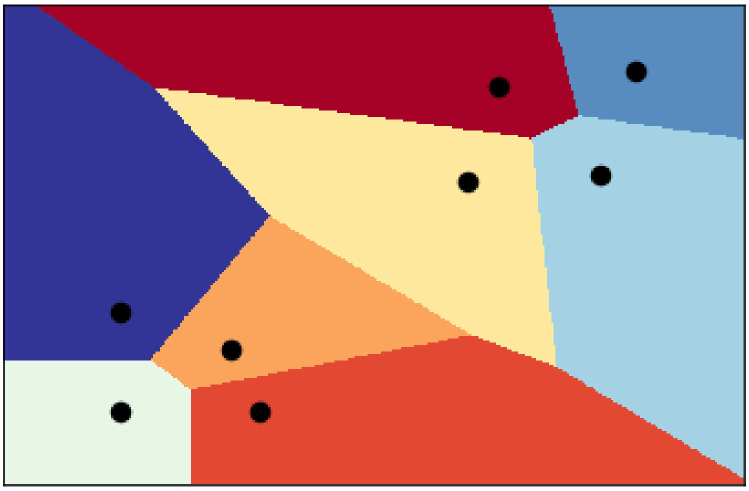

Figure 8.6: Voronoi Diagram of our mice example for $k = 8$

Next up. we will see how to deal with clustering tasks where the optimal decision boundary is not linear anymore.

8.3 Non-linear Clustering

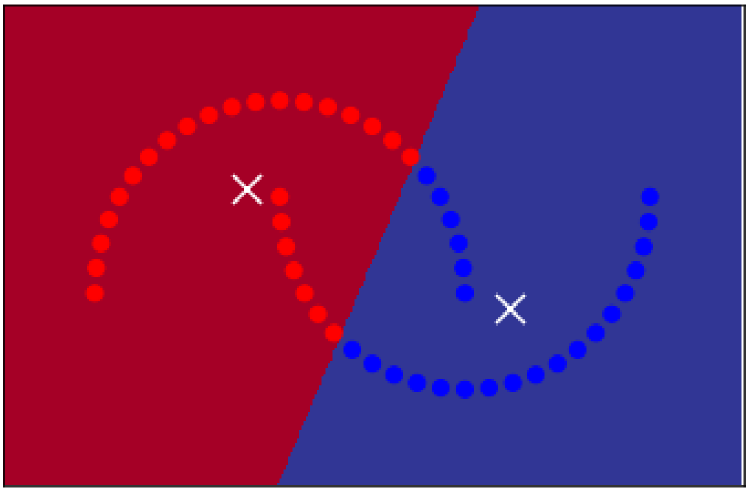

Figure 8.7: K-means can not cluster this data effectively.

8.3.1 Kernel k-means

Just like the SVM the k-means clustering algorithm can be kernelized. Therefore we have to change two parts of our algorithm. First of all, we will compute the norm $\|\mathbf{x}_i - \mathbf{c}_i\|^2$ line 5 via a kernel function and secondly we will not compute the mean in line 8 explicitly but rather find an implicit representation for it. We will now see how this works in detail. Let $\phi : \mathbb{R}^d \to \mathcal{H}$ be a kernel feature map for a kernel $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ and

$$l_j := \{i \in \{1, \ldots, n\} : y_i = j\} \text{ for all } j = 1, \ldots, k.$$

Then

$$\mathbf{c}_j := \frac{1}{|I_j|} \sum_{i \in I_j} \phi(\mathbf{x}_i) \, .$$

That is, we map all inputs $\mathbf{x}_1, ..., \mathbf{x}_n$ into a higher-dimensional space via ϕ where they can be linearly separated and then apply k-means in that space.

We can then completely express the norm $\left\|\phi\left(\mathbf{x}_i\right)-\mathbf{c}_j\right\|^2$ in terms of kernel functions as

$$\left\|\phi\left(\mathbf{x}_i\right)-\mathbf{c}_j\right\|^2=\underbrace{\left\|\phi\left(\mathbf{x}_i\right)\right\|^2}_{=k\left(\mathbf{x}_i, \mathbf{x}_i\right)}-2\left\langle\phi\left(\mathbf{x}_i\right), \mathbf{c}_j\right\rangle+\left\|\mathbf{c}_j\right\|^2, \tag{8.1}$$

where

$$\begin{aligned}
\left\|\mathbf{c}_j\right\|^2 &=\left\langle\frac{1}{\left|I_j\right|} \sum_{i \in I_j} \phi\left(\mathbf{x}_i\right), \frac{1}{\left|I_j\right|} \sum_{i' \in I_j} \phi\left(\mathbf{x}_{i'}\right)\right\rangle & (8.2)\\
&=\frac{1}{\left|I_j\right|^2}\left\langle\sum_{i \in I_j} \phi\left(\mathbf{x}_i\right), \sum_{i' \in I_j} \phi\left(\mathbf{x}_{i'}\right)\right\rangle & (8.3)\\
&=\frac{1}{\left|I_j\right|^2} \sum_{i, i' \in l_j} k\left(\mathbf{x}_i, \mathbf{x}_{i'}\right) & (8.4)
\end{aligned}$$

Here we used the kernel property $k(\mathbf{x}, \tilde{\mathbf{x}})=\langle\phi(\mathbf{x}), \phi(\tilde{\mathbf{x}})\rangle$ and the properties of dot products in general in order to move the sum and $\frac{1}{|I_j|^2}$ out of the dot product. Through an analogous calculation we get

$$\left\langle\phi\left(\mathbf{x}_i\right), \mathbf{c}_j\right\rangle=\frac{1}{\left|I_j\right|} \sum_{i' \in I_j} k\left(\mathbf{x}_i, \mathbf{x}_{i'}\right).$$

Therefore (8.1) can be completely expressed in terms of kernel functions.

$$\left\|\phi\left(\mathbf{x}_i\right)-\mathbf{c}_j\right\|^2=k\left(\mathbf{x}_i, \mathbf{x}_i\right)-\frac{2}{\left|I_j\right|} \sum_{i' \in I_j} k\left(\mathbf{x}_i, \mathbf{x}_{i'}\right)+\frac{1}{\left|I_j\right|^2} \sum_{i, i' \in I_j} k\left(\mathbf{x}_i, \mathbf{x}_{i'}\right) \tag{8.5}$$

Since we usually can not compute $\phi(\cdot)$ explicitly we have to find a different way to assign $\mathbf{c}_j$ in the next step e.g. line 8.
The trick we are applying, is to assign points to a cluster C_j directly (notice the capital non bold letter). Since we showed above that we do not need to know the cluster center $\mathbf{c}_j$, we can just skip its calculation in line 8 all together. What counts for the classification is not a point $\mathbf{c}_j$ but rather the indices in I_j, which then determine the value of the R.H.S of equation (8.4). We can say that I_j is basically an approximation for $\mathbf{c}_j$ which encodes the same information. That means which points should be clustered together. In case of I_j this works by explicitly giving their indices and in case of $\mathbf{c}_j$, by being able to calculate the distance from every input $\mathbf{x}_1, ..., \mathbf{x}_n$ and then choosing the centroid $\mathbf{c}_j$ which is nearest.

Algorithm Kernel k-means

Input: $k \in \mathbb{N}$ and $\mathbf{X} = \mathbf{x}_1, ..., \mathbf{x}_n \in \mathbb{R}^d$, a kernel function $\mathbf{k}'$

1: **function** K-MEANS($k, \mathbf{X}$)
2: Randomly assign every $\mathbf{x}_i$ to one of the k clusters C_j, $j = 1, ..., k$.

3: **repeat**
4: **for** i = 1 : n **do**
5: Label the input $\mathbf{x}_i$ as belonging to the nearest cluster C_j.

$$y_i = \arg\min_{j=1,\dots,k} \|\phi(\mathbf{x}_i) - \mathbf{c}_j\|^2 .$$

Thanks to (8.4) we do not have to know how $\mathbf{c}_j$ looks.

6: **end for**
7: **for** j = 1 : k **do**
8: Update clusters as

$$C_j = \{x_i : y_i = j\}.$$

9: **end for**
10: **until** Clusters do not change between subsequent iterations.

11: **return** Final clusters $C_1, ..., C_k$.
12: **end function**

As we see the only part that really changes in comparison to k-means is line 8 where we computed each cluster as the set of inputs $\mathbf{x}_i$ whose label y_i is j. With this algorithm we can indeed improve our results drastically in some cases. Take a look at figure 8.7 again and then at figure 8.8.

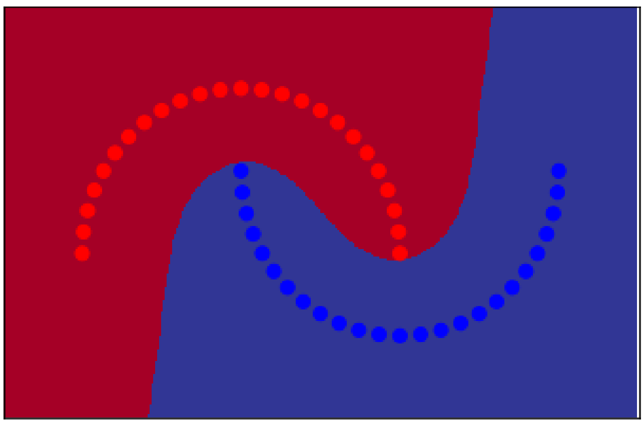

Figure 8.8: Kernel k-means can improve clustering by a lot.

So far we had to choose the number of clusters k ourselves. In the next part we will deal with an algorithm that somewhat automates this process.

8.4 Hierarchical Clustering

Hierarchical clustering is an approach that generates bigger clusters from smaller clusters, the hierarchy going from small to big. Bigger clusters are formed out of smaller clusters with the biggest possible cluster being the entire set of inputs $\{\mathbf{x}_i, ..., \mathbf{x}_n\}$, and the smallest possible cluster the inputs $\mathbf{x}_i$ themselves.

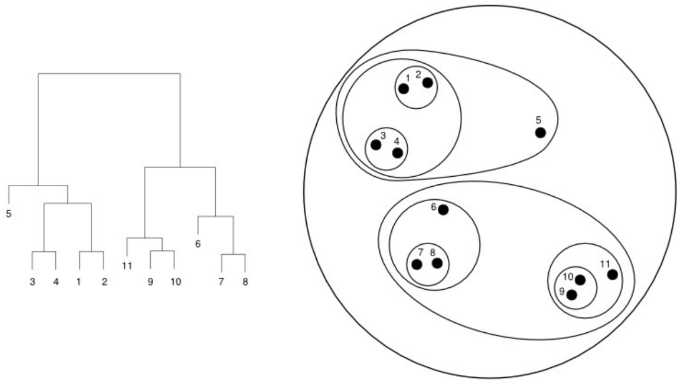

Figure 8.9: Example of hierarchical clustering of a set.

In 8.9 we see that the first clusters are assigned as $C_1 = \{1, 2\}, C_2 = \{3, 4\}, C_3 = \{7, 8\}, C_4 = \{9, 10\}, C_5 = \{5\}, C_6 = \{6\}, C_7 = \{11\}$. Notice how this corresponds to the leaves of the tree on the left side of figure 8.

Algorithm Hierarchical clustering

Input: $\mathbf{X} = \mathbf{x}_1, ..., \mathbf{x}_n \in \mathbb{R}^d$

function HIERARCHICAL CLUSTERING($\mathbf{X}$)
    Assign each input $\mathbf{x}_i$ into a cluster.

    repeat
        Link the two clusters with *minimal distance.*
    until Only one cluster is left.

    return Tree describing cluster hierarchy.
end function

The cursively written "*assign*" in line 2 and "*minimal distance*" in line 4 merit further attention. For the assign step, we can define our own criteria for how big our initial clusters should be and by which criterion they should be assigned. We might be inclined to not just assign every point to its own cluster, since the whole point of clustering is to associate points with others. A simple approach would be to use k-means for some k to initialize our first clusters.

Example 8.4.1. As we can see in 8.9 it would have been possible to put point 5 and 6 into a cluster, however they are rather far apart, so there might have been a limit of how far points can be from each other before becoming ineligible to be clustered together.

In order to calculate the minimal distance in line 4 there are three common options. Let $S_j \subseteq \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ be the set of inputs contained in the jth cluster

$$\text{Simple linkage: } d(i,j) := \min_{\mathbf{x}\in\mathcal{S}_i, \tilde{\mathbf{x}}\in S_j} \|\mathbf{x} - \tilde{\mathbf{x}}\|^2$$
$$\text{Average linkage: } \mathrm{d}(i,j) := \operatorname*{mean}_{\mathbf{x}\in S_i, \tilde{\mathbf{x}}\in S_j} \|\mathbf{x} - \tilde{\mathbf{x}}\|^2$$
$$\text{Complete linkage: } d(i,j) := \max_{\mathbf{x}\in S_i, \tilde{\mathbf{x}}\in S_j} \|\mathbf{x} - \tilde{\mathbf{x}}\|^2$$

Notice that all these expressions can be kernelized again. Which linkage we choose depends on our application. If we wanted to prevent "big" distances between points of two clusters, we would choose complete linkage for instance.

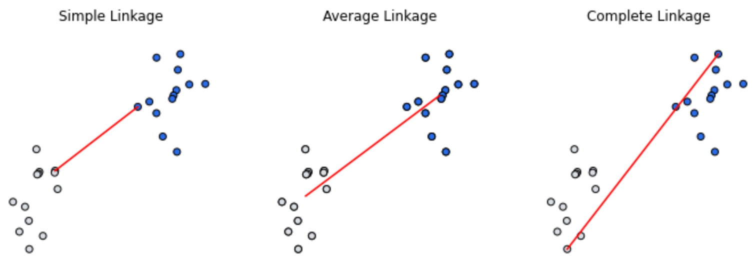

Figure 8.10: Geometric intuition behind the linkage types.

Chapter 9: Dimensionality Reduction

Definition 9.0.1 (Dimensionality Reduction)**.** With dimensionality reduction, we want to represent the data $\mathbf{x}_i, ..., \mathbf{x}_n$in a lower dimensional space $\mathbb{R}^k$ with $k < d$ with as little loss of relevant information as possible.

There are three main reasons why this is important: First off all data can be visualized in two or three dimensions, also less dimensions mean less data to be stored and worked with, which is more resourceful and leads to faster computation. Lastly, less dimensions decrease the risk of overfitting since it decreases the complexity of the data.

9.1 Linear dimensionality reduction

We are given $\mathbf{x}_i, ..., \mathbf{x}_n \in \mathbb{R}$. We assume w.l.o.g that

$$\hat{\boldsymbol{\mu}} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i = 0.$$

That means that the data has been centered in a pre-processing step. We then have to find a $k-$dimensional subspace $\mathcal{U}_W = \operatorname{span}(\mathbf{w}_1, \dots, \mathbf{w}_k)$ where $W = (\mathbf{w}_1, \dots, \mathbf{w}_k) \in \mathbb{R}^{d \times k}$ is an orthonormal basis, such that projected onto that space via the projection

$$\Pi_{\mathcal{U}_W}(\mathbf{x}_i) := \arg\min_{\mathbf{x} \in \mathcal{U}_W} \|\mathbf{x} - \mathbf{x}_i\|^2, \quad i = 1, \dots, n$$

is as "close" to the original data as possible. As a measure for closeness, we use the *average squared error* and pick the subspace $\mathcal{U}_{W^*}$ with

$$W^* = \arg\min_{W \in \mathbb{R}^{d\times k}} \quad \frac{1}{n}\sum_{i=1}^{n} \|\mathbf{x}_i - \Pi_{\mathcal{U}_W}(\mathbf{x}_i)\|^2.$$

Next, we will compute $\Pi_{\mathcal{U}_W}(\mathbf{x}_i)$ explicitly. Recall that orthonormality means $\langle \mathbf{w}_i, \mathbf{w}_j \rangle = 0$ for all $i \neq j$ and $\|\mathbf{w}_j\| = 1$. In the simplest case $k = 1$, we have $\mathcal{U}_W = \operatorname{span}(\mathbf{w_1})$. Then

$$\Pi_{\mathcal{U}_W}(\mathbf{x}_i) \overset{\text{def}}{=} \arg\min_{\mathbf{x} \in \mathcal{U}_W} \|\mathbf{x} - \mathbf{x}_i\|^2 = \underset{\mathbf{x} \in \mathbb{R}^d : \exists \lambda \in \mathbb{R} \text{ with } \mathbf{x} = \lambda \mathbf{w}_1}{\arg\min} \|\mathbf{x} - \mathbf{x}_i\|^2.$$

We then calculate the derivative of $f(\lambda) := \|\lambda \mathbf{w}_1 - \mathbf{x}_i\|^2$ and set it to zero

$$\begin{aligned} f'(\lambda) &= \frac{d}{d\lambda}(\lambda \mathbf{w}_1 - \mathbf{x}_i)^T(\lambda \mathbf{w}_1 - \mathbf{x}_i) \\ &= \frac{d}{d\lambda}(\lambda \mathbf{w}_1)^T(\lambda \mathbf{w}_1) - 2\frac{d}{d\lambda}\lambda \mathbf{w}_1^T \mathbf{x}_i \\ &= 2\lambda \mathbf{w}_1^T \mathbf{w}_1 - 2\mathbf{w}_1^T \mathbf{x}_i \end{aligned}$$

Now since $\mathbf{w}_1^T \mathbf{w}_1 = \|\mathbf{w}_1\|^2 = 1$, we get $\lambda^* = \mathbf{w}_1^T \mathbf{x}_i$ as optimal value. Thus

$$\Pi_{\mathcal{U}_W}(\mathbf{x}_i) = \lambda^* \mathbf{w}_1 = \mathbf{w}_1 \lambda^* = \mathbf{w}_1 \mathbf{w}_1^\top \mathbf{x}_i.$$

In the general case $k \geq 1$, we have $\mathcal{U}_W := \operatorname{span}(\mathbf{w}_1 \dots, \mathbf{w}_k)$ with $\mathbf{w}_i \perp \mathbf{w}_j$ for all $i \neq j$ and $\|\mathbf{w}_1\| = \dots = \|\mathbf{w}_k\| = 1$. Hence:

$$\begin{aligned} \Pi_{\mathcal{U}_W}(\mathbf{x}_i) &\overset{\text{def.}}{=} \underset{\mathbf{x} \in \mathcal{U}_W}{\arg\min} \|\mathbf{x} - \mathbf{x}_i\|^2 \\ &= \underset{\substack{\mathbf{x} \in \mathbb{R}^d : \exists \boldsymbol{\lambda} \in \mathbb{R}^k \\ \text{with } \mathbf{x} = \sum_{j=1}^k \lambda_j \mathbf{w}_j}}{\arg\min} \|\mathbf{x} - \mathbf{x}_i\|^2. \end{aligned}$$

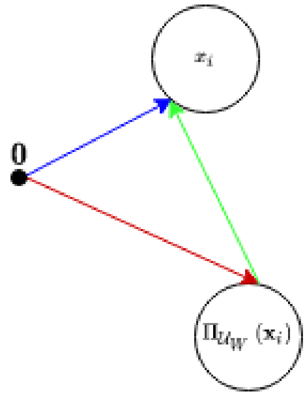

Figure 9.1: An illustration of the data point $\mathbf{x_i}$ its orthogonal projection and the origin $\mathbf{0}$.

As for $k = 1$ we can then set the derivative of $f(\lambda) := \left\| \sum_{j=1}^{k} \lambda_j \mathbf{w}_j - \mathbf{x}_i \right\|^2$ to zero, which revelas the optimum $\lambda_j^* = \mathbf{w}_j^\top \mathbf{x}_i$ for all $j = 1, \ldots, k$. Therefore:

$$\Pi_{\mathcal{U}_W}(\mathbf{x}_i) = \sum_{j=1}^{k} \lambda_j^* \mathbf{w}_j = \sum_{j=1}^{k} \mathbf{w}_j \lambda_j^* = \sum_{j=1}^{k} \mathbf{w}_j \mathbf{w}_j^\top \mathbf{x}_i = WW^\top \mathbf{x}_i.$$

Then the projected data can be written as :

$$\begin{aligned} \hat{X} &:= (\Pi_{\mathcal{U}}(\mathbf{x}_1), \ldots, \Pi_{\mathcal{U}_W}(\mathbf{x}_n)) \\ &= \left(WW^\top \mathbf{x}_1, \ldots, WW^\top \mathbf{x}_n\right) \\ &= WW^\top X. \end{aligned}$$

If we look at this with respect to the basis $\mathbf{w_1}, \ldots \mathbf{w_k}$ the coordinates of projection of point $\mathbf{x_i}$ would clearly just be

$$\widetilde{\mathbf{x}}_i := \begin{pmatrix} \mathbf{w}_1^\top \mathbf{x}_i \\ \vdots \\ \mathbf{w}_k^\top \mathbf{x}_i \end{pmatrix} = W^\top \mathbf{x}_i.$$

Repeating this for each data point would lead to the following matrix :

$$\tilde{X} := (\tilde{\mathbf{x}}_1, \ldots, \widetilde{\mathbf{x}}_n) = \left(W^\top \mathbf{x}_1, \ldots, W^\top \mathbf{x}_n\right) = W^\top X.$$

Now that we know how the projection looks like, our *PCA* problem becomes the following optimization problem to a find matrix $W \in \mathbb{R}^{d \times k}$, which solves

$$\begin{aligned} &\arg \min_{W \in \mathbb{R}^{d \times k}} \sum_{i=1}^{n} \left\| x_i - WW^\top \mathbf{x}_i \right\|^2 \\ &\text{s.t.} \quad \mathbf{w}_i \perp \mathbf{w}_j \text{ for all } i \neq j \text{ and } \|\mathbf{w}_1\| = \cdots = \|\mathbf{w}_k\| = 1. \end{aligned}$$

Let us now look at how to solve this optimization problem. Firstly recall that the projection is orthogonal. We will have the situation as seen in 9.1.

By the pythagorean theorem we get that the following equality holds

$$\|\Pi_{\mathcal{U}_W}(\mathbf{x}_i)\|^2 + \|\mathbf{x}_i - \Pi_{\mathcal{U}_W}(\mathbf{x}_i)\|^2 = \|\mathbf{x}_i\|^2$$

Observe the fact that minimizing $\left\|\mathbf{x}_i - \Pi_{\mathcal{U}_W}\left(\mathbf{x}_i\right)\right\|^2$ would lead to a maximization of $\left\|\Pi_{\mathcal{U}_W}\left(\mathbf{x}_i\right)\right\|^2$. So we can safely exchange this in our optimization problem, furthermore we have that

$$\begin{aligned}
\sum_{j=1}^{n}\left\|\Pi_{\mathcal{U}_w}\left(\mathbf{x}_i\right)\right\|^2 &= \sum_{i=1}^{n} \mathbf{x}_i^\top \underbrace{W^\top W}_{=I} \underbrace{W W^\top}_{=\sum_{j=1}^{k} \mathbf{w}_j \mathbf{w}_j^\top} \mathbf{x}_i \\
&= \sum_{j=1}^{k} \mathbf{w}_j^\top \sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^\top \mathbf{w}_j \\
&= \sum_{j=1}^{k} \mathbf{w}_j^\top X X^\top \mathbf{w}_j .
\end{aligned}$$

These observations lead us to the following theorem:

Theorem 9.1.1. *The PCA optimization problem can be equivalently written as*

$$\begin{aligned}
W_* := \arg \max_{W \in \mathbb{R}^{d\times k}} & \sum_{j=1}^{k} \mathbf{w}_j^\top X X^\top \mathbf{w}_j . \\
\textit{s.t.} \quad & \mathbf{w}_i \perp \mathbf{w}_j \textit{ for all } i \neq j \textit{ and } \|\mathbf{w}_1\| = \cdots = \|\mathbf{w}_k\| = 1
\end{aligned}$$

Definition 9.1.2 (Scatter matrix)**.** $S_n := XX^\top$ is called the *scatter matrix.*

Theorem 9.1.3. *The optimal PCA solution $W_* = (\mathbf{w}_1^*, \ldots, \mathbf{w}_k^*)$ is given by the k largest eigenvectors of the (centered) scatter matrix S_n.*

Proof. Left as an exercise. □

Theorem 9.1.3 gives us an algorithm to solve our PCA optimization problem.

Algorithm PCA Linear Dimensionality Reduction

Input: parameter k, inputs $X = (\mathbf{x}_1, \ldots, \mathbf{x}_n) \in \mathbb{R}^{d\times n}$

1: Compute sample mean $\hat{\boldsymbol{\mu}} := \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i$.
2: Center each input: $\mathbf{x}_i \leftarrow \mathbf{x}_i - \hat{\boldsymbol{\mu}}$ and update X.
3: Compute scatter matrix $S_n := XX^\top$.
4: Compute k largest eigenvalues of S_n with eigenvectors $W = (\mathbf{w}_1, \ldots, \mathbf{w}_k)$.
5: **return** dim.-reduced data: $\widetilde{X} = W^\top X \in \mathbb{R}^{k\times n}$ and $\hat{X} = WW^\top X \in \mathbb{R}^{d\times n}$.

9.2 Kernel PCA

The linear dimensionality reduction will struggle on a few inputs because of its inherent linearity. As seen in previous chapters in similar situations, we can apply kernelization. Firstly apply the general representer theorem, that is for each basis vector $\mathbf{w_j}$ of $W_* = (\mathbf{w}_1^*, \ldots, \mathbf{w}_k^*)$ there exist coefficients $\boldsymbol{\alpha}_j = (\alpha_{1j}, \ldots, \alpha_{nj})^\top \in \mathbb{R}^n$ so that we can write

$$\mathbf{w}_j^* = \sum_{i=1}^{n} \alpha_{ij} \phi\left(\mathbf{x}_i\right).$$

Let us denote $\phi(X) := (\phi(\mathbf{x}_1), \ldots, \phi(\mathbf{x}_1))$ and $\boldsymbol{\alpha} = (\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_k)$ we get that

$$\mathbf{w}_{\mathbf{j}}^* = \phi(X)\boldsymbol{\alpha}_j,$$

and in general the matrix can be written as a matrix multiplication more compactly as

$$W_* = \phi(X)\boldsymbol{\alpha}.$$

Finally the data in the dimensionality reduced basis will be

$$\widetilde{X} := W_*^\top \phi(X) = (\phi(X)\boldsymbol{\alpha})^\top \phi(X) = \boldsymbol{\alpha}^\top \phi(X)^\top \phi(X) = \boldsymbol{\alpha}^\top K.$$

Let us use this insight to kernelize our PCA optimization problem. We have that $\mathbf{w}_j = \phi(X)\boldsymbol{\alpha}_j$ and thus :

$$\begin{aligned}\max_{W \in \mathbb{R}^{d\times k}} \sum_{j=1}^k \mathbf{w}_j^\top \phi(X)\phi(X)^\top \mathbf{w}_j &= \max_{\boldsymbol{\alpha}\in\mathbb{R}^{n\times k}} \sum_{j=1}^k \boldsymbol{\alpha}_j^\top \underbrace{\phi(X)^\top\phi(X)\phi(X)^\top\phi(X)}_{=K^2} \boldsymbol{\alpha}_j. \\ &= \max_{\boldsymbol{\alpha}\in\mathbb{R}^{n\times k}} \boldsymbol{\alpha}^T K^2 \boldsymbol{\alpha}.\end{aligned}$$

Let us now just add the normality and orthogonality constraints to get the final version of kernel PCA

$$\begin{aligned}
\begin{array}{l}\max \boldsymbol{\alpha}^\top K^2 \boldsymbol{\alpha} \\ \text{s.t. } \forall j : ||\mathbf{w}_{\mathbf{j}}^*|| = 1, \forall i,j : {\mathbf{w}_{\mathbf{i}}^*}^\top \mathbf{w}_{\mathbf{j}}^* = 0\end{array} &\iff \begin{array}{l}\max \boldsymbol{\alpha}^\top K^2 \boldsymbol{\alpha} \\ \text{s.t. } \forall j : (\phi(X)\boldsymbol{\alpha}_j)^\top (\phi(X)\boldsymbol{\alpha}_j) = 1, \forall i,j : {\mathbf{w}_{\mathbf{i}}^*}^\top \mathbf{w}_{\mathbf{j}}^* = 0\end{array} \\
&\iff \begin{array}{l}\max \boldsymbol{\alpha}^\top K^2 \boldsymbol{\alpha} \\ \text{s.t. } \forall j : \boldsymbol{\alpha}_j^\top K \boldsymbol{\alpha}_j = 1, \forall i,j : {\mathbf{w}_{\mathbf{i}}^*}^\top \mathbf{w}_{\mathbf{j}}^* = 0\end{array} \\
&\iff \begin{array}{l}\max \boldsymbol{\alpha}^\top K^2 \boldsymbol{\alpha} \\ \text{s.t. } \boldsymbol{\alpha}^\top K \boldsymbol{\alpha} = I.\end{array}
\end{aligned}$$

Theorem 9.2.1. *The objective function for kernel PCA (kPCA) is*

$$\begin{aligned}&\max_{\boldsymbol{\alpha}\in\mathbb{R}^{n\times k}} \boldsymbol{\alpha}^\top K^2 \boldsymbol{\alpha} \\ &\text{s.t. } \boldsymbol{\alpha}^\top K \boldsymbol{\alpha} = I.\end{aligned}$$

Definition 9.2.2 (Centered Kernel Matrix)**.** The centered kernel matrix is the kernel matrix computed on the data that has been centered in feature space.

Theorem 9.2.3. *The centered kernel matrix $\widetilde{K}$ can be computed from the (uncentered) kernel matrix K by:*

$$\widetilde{K} = \left(I - \frac{11^\top}{n}\right) K \left(I - \frac{11^\top}{n}\right).$$

Theorem 9.2.4. *The kPCA solution $\boldsymbol{\alpha}_* = (\boldsymbol{\alpha}_1^*, \ldots, \boldsymbol{\alpha}_k^*)$ is given by the k largest eigenvectors of the (centered) kernel matrix K.*

Proof. Left as an exercise. □

Both last theorems pave the way for the following algorithm to solve the kPCA optimization problem.

Algorithm kPCA Kernelized Dimensionality Reduction

Input: parameter k, kernel matrix $K \in \mathbb{R}^{n \times n}$

1: Center the kernel matrix: $K \leftarrow \left(I - \frac{11^\top}{n}\right) K \left(I - \frac{11^\top}{n}\right)$.
2: Compute k largest eigenvectors $\boldsymbol{\alpha} = (\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_k)$ of K.
3: Compute: $\widetilde{X} := \boldsymbol{\alpha}^\top K$.
4: **return** dim.-reduced data: $\widetilde{X} \in \mathbb{R}^{k \times n}$

As in the case of kernelized regression, linear kernels are useful if $d > n$, as the matrix we are looking the eigenvectors for, is smaller.

9.3 Autoencoders

Dimensionality reduction can also be achieved by means of neural networks, leading us to the concept of autoencoders. Let

$$f_W : \mathbb{R}^d \to \mathbb{R}^k$$

be a neural network called encoder and

$$g_{\tilde{W}} : \mathbb{R}^k \to \mathbb{R}^d,$$

be a neural netword called decoder. An autoencoder is defined as

$$\min_{W, \tilde{W}} \sum_{i=1}^{n} \left\| \mathbf{x}_i - g_{\tilde{W}} \left(f_W \left(\mathbf{x}_i \right) \right) \right\|^2 .$$

At first glance this might not make sense, we are trying to minimize the loss between the original data point $\mathbf{x_i}$ and the decoded encoded version of $\mathbf{x_i}$. This can easily be achieved by not changing the data point at all, but we will not allow this behavior. A simple way to prohibit this is to enforce some bottleneck on a layer of neurons in our neural network as depicted in 9.2. On a high level, if done right, the output layer will resemble the input layer as close as possible but with missing details.In the subcase of linear encoders and decoders, the linear encoder will look like $f_W(\mathbf{x}) := W^\top \mathbf{x}$ and the linear decoder will look like $g_{\tilde{W}}(\mathbf{x}) := \tilde{W}^\top \mathbf{x}$. In the same fashion as in PCA, the data in our basis would be

$$\hat{X} := g_{\tilde{W}} \left(f_W(X) \right) \in \mathbb{R}^{d \times n},$$

and the data in the encoded basis would be

$$\widetilde{X} := f_W(X) \in \mathbb{R}^{k \times n}.$$

For the optimal choice of $\tilde{W}$, $\tilde{W} := W^\top$ we have that

$$\widetilde{X} = f_W(X) = W^\top X,$$

and that

$$\hat{X} = g_{\tilde{W}} \left(f_W(X) \right) = W W^\top X.$$

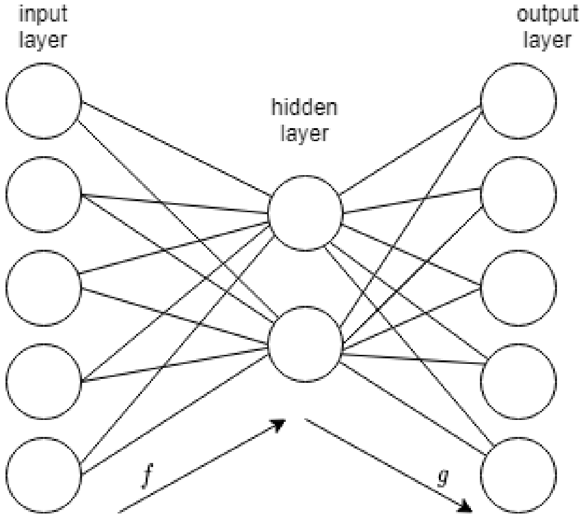

Figure 9.2: An illustration of an autoencoder neural network

Observation 9.3.1. The solution of the linear autoencoder corresponds to the same solution as PCA. One thing to keep in mind is that the autoencoder does not constrain W to be orthonormal. The real power of autoencoders come from its non-linearity. For more on autoencoders have a look at [6]. On a final note observe that for $I(t) := t$ and $f(\mathbf{x}) = \left\|\phi(\mathbf{x}) - \mathbf{w}\mathbf{w}^\top\phi(\mathbf{x})\right\|^2$ we can fit dimensionality reduction in our known Unifying View

$$\min_{[W,]b,\mathbf{w}} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} I\left(f\left(\mathbf{x}_i\right)[, y_i]\right) + \left[+\frac{1}{2}\sum_{l=1}^{L}\|W_l\|_{\text{Fro}}^2\right].$$

Chapter 10: Decision Trees & Random Forests

In this chapter we will consider tree based learning. This is decision trees and based on that bagging and random forests. Decision tree learning is one of the oldest non-linear machine learning methods. Moreover, in [6] M. Fernandez-Delgado et al. evaluated 179 classifiers arising from 17 families (neural networks, support vector machines, nearest-neighbors, random forests, ...). They found that the classifiers most likely to be the best are random forests and SVM with gaussian kernel. That means if you have a standard data set with some numerical features, then it is very likely that random forests or SVM with gaussian kernels will be the best method for this data set. This result does not hold for image classification for example, where you need of course a deep neural network.

Here we only regard numerical datasets. Those are given by d-dimensional data points $\mathbf{x} \in \mathbb{R}^d$ each equipped with a label $y \in \mathbb{R}$ (for Regression problems) or $y \in \{-1, +1\}$ (for Classification problems). As before we have some training data with that we construct our decision tree or random forest. Those tools shall predict labels for new data points. So let us start.

10.1 Decision Trees

Definition 10.1.1 (Decision Trees)**.** A decision tree is a decision support tool from operations research that uses a tree-like graph or model of decisions and their possible consequences.

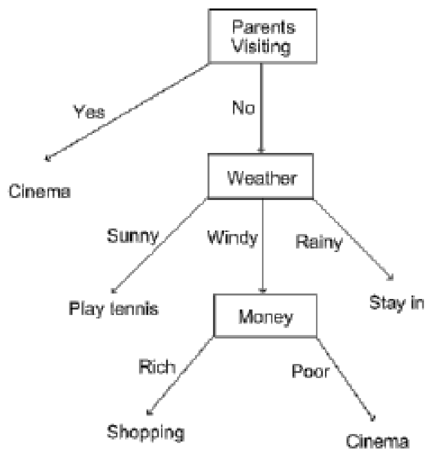

Figure 10.1: A decision tree for what to do at a normal day.

A non-ML example for such a decision supporting tree can be found in Figure 10.1. An example for ML can be found in Figure 10.2. Here we have 2-dimensional data points $\mathbf{x} = (x_1, x_2)^T \in \mathbb{R}^2$. For each given data point we follow the tree down to a terminal node and take this label as predicted label for $\mathbf{x}$. The induced decision region of this decision tree are shown in Figure 10.3. To each given data point (x_1, x_2) we assign the appropriate label from point (x_1, x_2) in the figure.

[6][M. Fernandez-Delgado, E. Cernadas and S. Barro, 2014: Do we Need Hundreds of Classifiers to Solve Real World Classification Problems?]

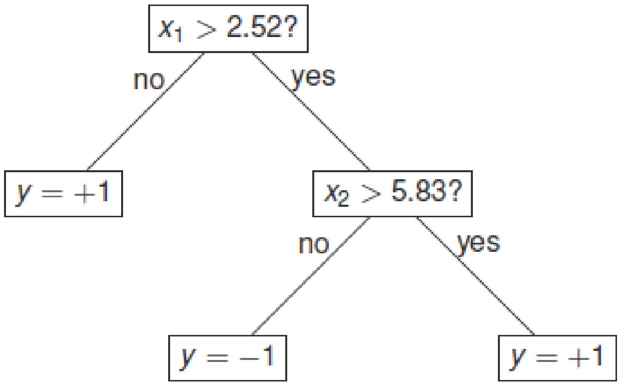

Figure 10.2: A decision tree for predicting labels of 2-dimensional data points $\mathbf{x} = (x_1, x_2) \in \mathbb{R}^2$.

We see that the first node '$x_1 > 2.52$' splits according to the x_1-coordinate at 2.52. To the left region $+1$ is assigned and if we are in the right region the decision tree performs another check, the node $x_2 > 5.83$. Note that such a visualisation can be only done for 2-dimensional data.
So we know how to apply a given tree to new data points. In the following we consider the question how to construct such a decision tree that predicts well on a given dataset.

In order to solve this we do the following. Assume we have d-dimensional data points. Then we first construct a complete binary tree, that is with depth d and 2^d terminal nodes. Afterwards we delete some of the internal nodes, that produce no substantial different. Firstly we explain how to grow the tree, afterwards how to reduce it again using pruning.

For growing the tree three question arises.

1. For each internal node, which feature to take?
2. For each internal node, which threshold to take?
3. Which labels to use at the terminal nodes?

In other words: For a d-dimensional dataset an internal node is of the structure $x_j > t$. We have to specify which feature $j \in \{1, \dots, d\}$ (Question 1.) and which threshold t (2.) to take. A terminal node has the structure $y = l$ for l a label and we have to choose a label l (3.).

Question 1. and 2. correspond and we answer them together. But first some notation. As before we consider the node '$x_j > t$' with j the feature that we split upon, t the split threshold and $R_1(j,t), R_2(j,t) \subset \mathbb{R}^d$ denotes the two regions under the internal node $x_j > t$. That is $R_1(j,t) = \{\mathbf{x} \in \mathbb{R}^d \mid x_j < t\}$ and $R_2(j,t) = \{\mathbf{x} \in \mathbb{R}^d \mid x_j \geq t\}$, see also Figure 10.4. In other words R_1 and R_2 are the two regions induced by splitting the j-th feature using the threshold t, where R_1 corresponds to everything under the left hand subtree and R_2 to everything under the right hand subtree. We look at the questions for the regression and the classification case.

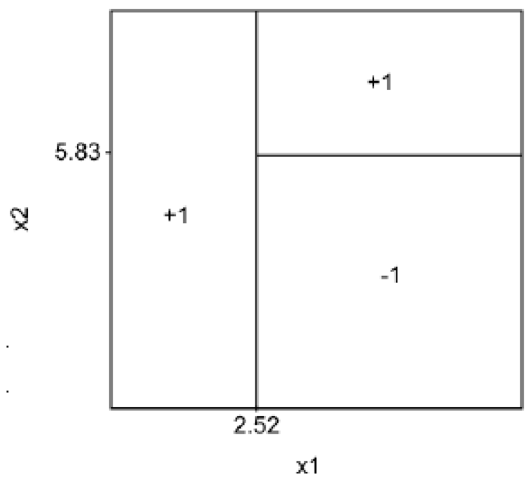

Figure 10.3: A visualisation of the induced decision regions of the decision tree in Figure 10.2.

For the regression case we have labels $l \in \mathbb{R}$ and we choose the feature j and threshold t by:

$$\arg\min_{j,t} \sum_{k=1}^{2} \min_{c \in \mathbb{R}} \sum_{i:\mathbf{x}^i \in R_k(j,t)} \left(y^i - c\right)^2$$

That means, we choose j and t such that for each region $R_1(j,t)$ and $R_2(j,t)$, respectively, we can find an appropriate value c such that the mean squared error of c and the labels y^i of the points $\mathbf{x}^i$ in $R_1(j,t)$ and $R_2(j,t)$, respectively, is minimal. One can show that in the optimum c is the mean label of points in that region $R_1(j,t)$ or $R_2(j,t)$ respectively. We define $\hat{y}_{j,t,k}$ to be this value. Formalised this means, that for a region $R(j,t)$ the optimal c is given by $c = \text{mean}\left(\left\{y^i : \mathbf{x}^i \in R_k(j,t)\right\}\right) =: \hat{y}_{j,t,k}$. So in other words we choose the feature j and treshold t such that the average squared distance of labels of the points under a region to the mean label in that region is minimal.
To assign a label at a terminal node (question 3.) just take the mean label of points in the region under this terminal. So if this terminal node is the left child of the internal node given by j and t we take $\hat{y}_{j,t,1}$ for the label.

For the Classification case we have labels $l \in \{-1, +1\}$ and we want the predicted labels to be in $\{-1, +1\}$, too. Therefore taking the mean label of points does not work anymore. So for this case we use a little bit different strategy.
Define

$$p_{j,t,k} = \frac{|\{i \mid \mathbf{x}^i \in R_k(j,t) \wedge y^i = +1\}|}{|\{i \mid \mathbf{x}^i \in R_k(j,t)\}|}$$

the fraction of instances in region $R_k(j,t)$ that are labeled $+1$. Hence, $1 - p_{j,t,k}$ is the fraction of instances in that region that are labeled -1. Now we choose the feature j and threshold t by

$$\arg\min_{j,t} \sum_{k=1}^{2} g\left(p_{j,t,k}\right)$$

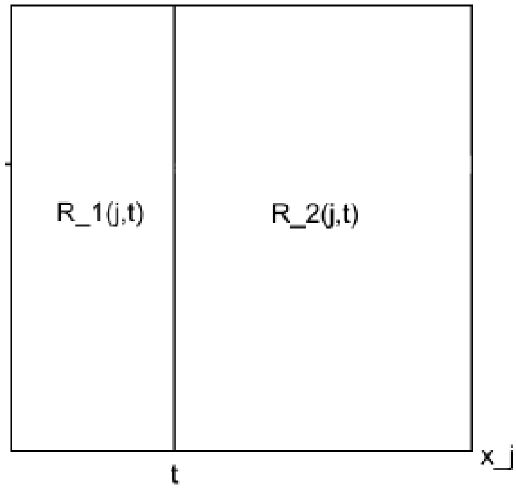

Figure 10.4: The two regions $R_1(j,t)$ and $R_2(j,t)$ for the internal node given by j and t, that is for the node '$x_j > t$'.

where g is the *Gini impurity measure* given by $g(p) := 2p(1-p)$, see also Figure 10.5. Observe that the minimum of g is attained for very small or large values of p. Therefore j and t are chosen such that $p_{j,t,k}$ is very small or large but preferably not near to 0.5. $p = 0.5$ would mean that we have as many points with positive labels as points with negative labels. So this means our algorithm searches as much disparity as possible and that is exactly what we want.
To assign a label at a terminal node (question 3.) just do a majority vote over the labels of the instances (that are the points $\mathbf{x} \in \mathbb{R}^d$ in the training data) falling into the region under the terminal node. If for example in the region there are two points with positive label and one with a negative one, this terminal node gets the label $+1$.
To reduce the size of the tree we use *pruning*.

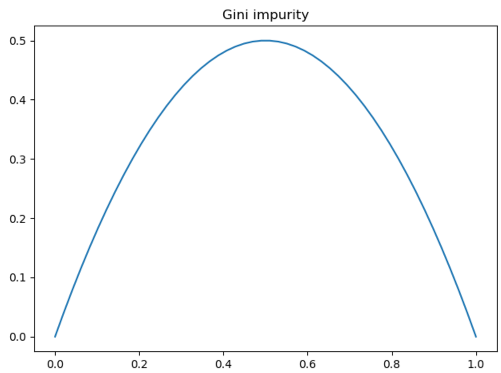

Figure 10.5: The Gini impurity measure g.

Definition 10.1.2 (Pruning)**.** Removing internal nodes that produce no substantial decrease in prediction accuracy as measured on a hold-out data set.

That is we train the tree on a training data set until it is fully grown, then we start pruning the tree on another data set, called hold-out data set. Thereby we successively remove nodes of the tree that do not decrease the prediction accuracy very much. For further reading see [7] Chapter 9, e.g. **cost-complexity pruning**.
We want to mention that pruning is especially important to remedy overfitting. A large, fully grown tree with 2^d terminal nodes is prone to overfit!

10.2 Random Forests

One major problem of the decision trees is their high variance: Often a small change in data can result in a very different series of splits. For example consider a feature that we split very late. If we split this feature at the beginning, we get a completely different tree. This is bad for reproducability but we can use this to gain advantage. The technique bagging averages many trees to reduce this variance. That means we repeat our method, decision tree, several times each on a slightly different data set. Then we combine the different results. In detail this means:

1. We draw each data set randomly with replacement from the training data such that it has the same size as the original training set. (i.e. one data point could occure more than once)
2. Grow a tree on each data set.
3. In the case of regression we average the resulting prediction functions and for the classification case we perform majority vote.

The resulting ensemble of trees is called a *forest*. In principle, this trick can be applied to any arbitrary machine algorithm (e.g., SVM, KRR, neural network), but in practice it works best with (small) decision trees because they have a very high variance. With this we can define random forests.

Definition 10.2.1 (Random Forests)**.** Random forests are pretty much the same as bagging of decision trees except for one difference: When growing a tree of the forest, for each split of an internal node in the tree, we randomly select a subset of $m < d$ many features and choose the optimal split feature j only among those m features.

In other words, for each internal node we use a new random draw of m features, call the set of those features M, and instead of $\arg\min_{j:t}$ we consider $\arg\min_{j \in M, t}$. Typically this subset of features is very small. If for example $d = 1000$ then a common choice for m is 40. Finally, we again look at the spam data and compare the different methods in Figure 10.6. Gradient Boosting is strongly related to random forests.

[7] [T. Hastie, R. Tibshirani and J. Friedman, The elements of statistical learning, 2nd edition, Springer series, 2009]

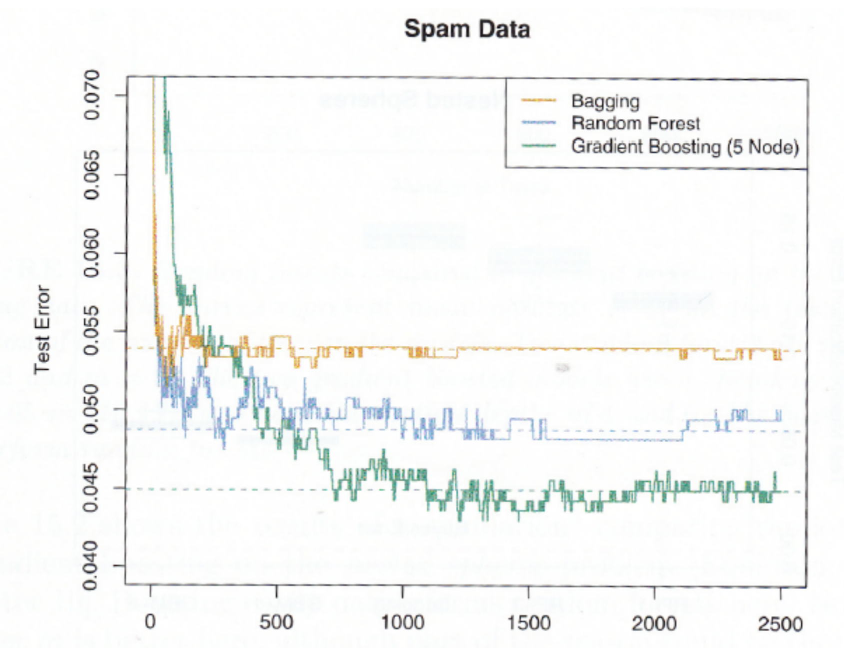

Figure 10.6: The performance of the algorithms. From up to down: Bagging, Random Forests and Gradient Boosting. The x axis gives the number of trees.

References

[1] Léon Bottou, Frank E Curtis and Jorge Nocedal. 'Optimization methods for large-scale machine learning'. In: *Siam Review* 60.2 (2018), pp. 223–311.

[2] Stephen Boyd, Stephen P Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[3] Leo Breiman. 'Random forests'. In: *Machine learning* 45.1 (2001), pp. 5–32.

[4] Corinna Cortes and Vladimir Vapnik. 'Support-vector networks'. In: *Machine learning* 20.3 (1995), pp. 273–297.

[5] Nello Cristianini, John Shawe-Taylor et al. *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000.

[6] Ian Goodfellow et al. *Deep learning*. Vol. 1. 2. MIT press Cambridge, 2016.

[7] Trevor Hastie, Robert Tibshirani and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009.

[8] Anil K Jain, M Narasimha Murty and Patrick J Flynn. 'Data clustering: a review'. In: *ACM computing surveys (CSUR)* 31.3 (1999), pp. 264–323.

[9] Alex J Smola and Bernhard Schölkopf. 'A tutorial on support vector regression'. In: *Statistics and computing* 14.3 (2004), pp. 199–222.

Index