Mathematical Foundations of Data Science

Author: Jan Schweikert

👋 Hey there! I'm thrilled that you found these materials! Hope they help you on your data science journey.

This collection contains course materials and notes for "Mathematical Foundations of Data Science" focusing on supervised and unsupervised learning techniques, based on the university course

> Online Version Available

📚 Course Contents

The course is structured into three main sections:

1. **Introduction**
   - What is Data Science?
   - Linear Regression
2. **Supervised Learning**
   - Regression Techniques
   - Classification Methods
3. **Unsupervised Learning**
   - Principal Component Analysis
   - Clustering Methods

!! Important Information

Please note that the "Significance of Parameters" (2.1.3) chapter is incomplete and won't be provided in the future. Please refer to the official script given by the course instructor.

⚠️ Disclaimer

These course materials are provided for educational purposes only. While every effort has been made to ensure accuracy, please refer to official course materials and consult with instructors for authoitative information.

1.0 - What is Data Science?

Extract information from Data

Example 1

Years of study / salary relation

1. Understand the relation between an "input" and an "output"
2. Find a function that roughly estimates the data points (called *regression*)

Other examples

- Using user rating to generate movie recommendations
- Label an entire dataset from only a few labels
- Classify data
    - Whole dataset is labeled (e.g. Cats and Dogs)
    - Draw a new sample (take a new image) and classify it (as a Dog or a Cat)

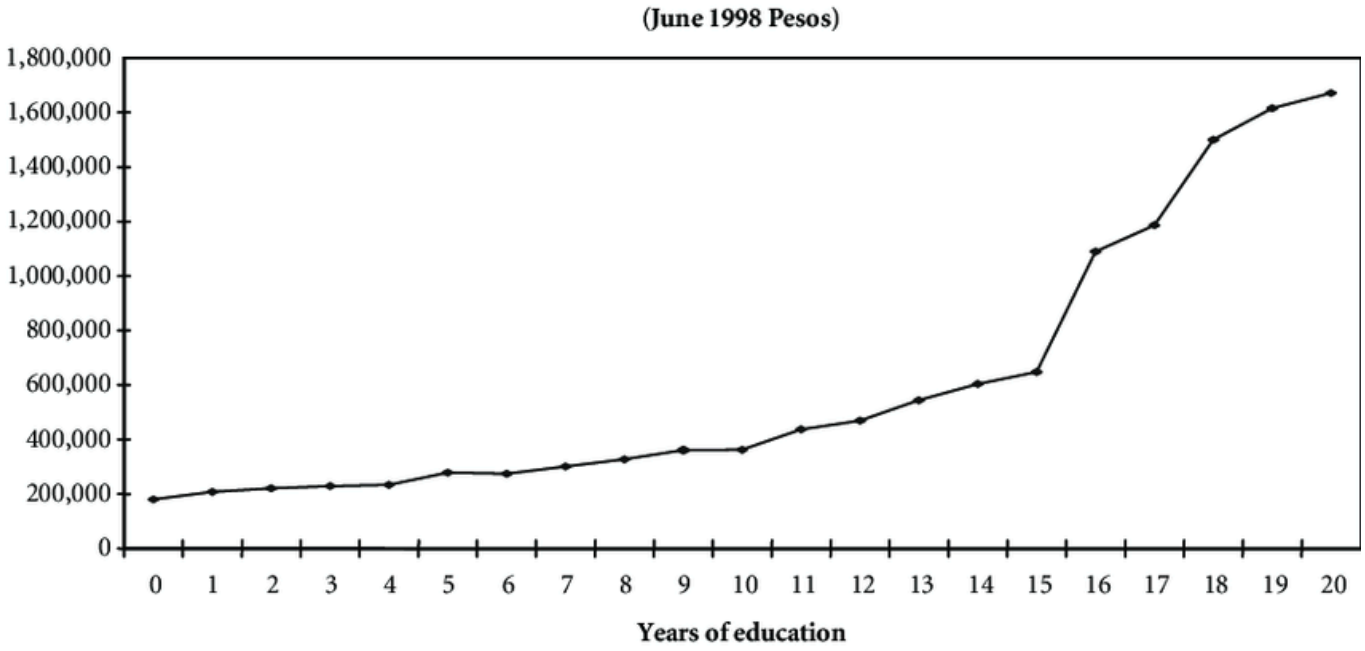
3 Fields

1. Supervised learning
    - classification / regression
2. Unsupervised learning
    - clustering
    - you only have data and no information on it
3. Semi-supervised learning
    - if you have a few labels and you want to infer all the labels

Terminology

step size = learning rate

Example: The income dataset



The data points $(x_i, y_i) \in \mathbb{R}^2$ are supposed to be of the form

$$y_i = f(x_i) + \epsilon \quad (\epsilon \to \text{noise})$$

Remark: In general, the function f is *unknown*
We want to approximate this function f using the data!
Find an approximation $\hat{f}$ of f using $\{(x_i, y_i)\}_{1 \leq i \leq N}$

Nearest neighbours interpolation

The nearest neighbours interpolation of $\{(x_i, y_i)\}_{1 \leq i \leq N}$ is the function

$$x \Rightarrow \hat{f}(x) = y_{i*}$$

This means for an input x you find the nearest data point x_i (i.e. the one with the smallest absolute distance to x) and assign its corresponding value y_i to $\hat{f}(x)$

when $i_x \in \text{argmin}_{1 \leq i \leq N} |x - x_i|$

- Perfectly fits the data, i.e. $\hat{f}(x_i) = y_i$
- However, its sensible to outlayers
    - **Outlayers**: Some point in the graph is somehow not along the expected function (imagine point (18, 40,000)), which is very far from the function we are seeking

1.1 - Linear Regression

Linear regression

We try to approximate f by a function $\hat{f} : \mathbb{R} \to \mathbb{R}$, $x \to wx + b$, where $w, b \in \mathbb{R}$
How to select w and b that approximate the data quite well?

Question: How to find $w, b \in \mathbb{R}$ such that $\hat{f}(x) \sim f(x)$?
--> Find w, b such that $\forall i, \hat{f}(x_i) \sim y_i$

Sum of squared deviations

$$L(w, b) = \frac{1}{2} \sum_{i=1}^{N} |wx_i + b - y_i|^2$$

$(wx_i + b - y_i)$ is the distance from data points to estimated function
This will give an exact fit for the data
We want to find w, b which minimizes $L(w, b)$
You may be wondering why there's a $\frac{1}{2}$ is in the loss function. Well scaling a loss function by a constant does not change the location of its minimum (optimal w and b remain the same whether $\frac{1}{2}$ is included or not). Also it simplifies derivatives because when taking the derivate the square cancels out the $\frac{1}{2}$ which leaves a cleaner expression.

1st observation: L is **differentiable**
2nd observation: L is convex
(w, b) is a minimum of $L \Leftrightarrow \partial_w L(w, b) = \partial_b L(w, b) = 0$
((is the same as)$\Leftrightarrow \nabla L(w, b) = 0$)

1.3.3

$$\partial_w L(w, b) = \frac{1}{2} \sum_{i=1}^{N} (wx_i + b - y_i)^2$$

Apply chain rule $f'(g(x))g'(x)$ where $f(x) = x^2$ and $g(x) = (wx_i + b - y_i)$

$$\text{chain rule} = 2(wx_i + b - y_i) \cdot \frac{\partial}{\partial w}(wx_i + b - y_i)$$

$$= \frac{1}{2} \sum_{i=1}^{N} \cdot 2(wx_i + b - y_i) \cdot x_i$$

$$= \sum_{i=1}^{N} x_i(wx_i + b - y_i)$$

Now expanding $(wx_i + b - y_i)x_i = w_x i^2 + bx_i - y_i x_i$

$$= w\sum_{i=1}^{N} x_i^2 + b\sum_{i=1}^{N} x_i - \sum_{i=1}^{N} x_i y_i$$

$$\partial_b L(w,b) = \sum_{i=1}^{N}(wx_i + b - y_i)$$

Also applying chain rule here:

$$\frac{1}{2}\sum_{i=1}^{N} 2(wx_i + b - y_i) \cdot \frac{\partial}{\partial b}(wx_i + b - y_i)$$

$$= \sum_{i=1}^{N}(wx_i + b - y_i)$$

$$= w\sum_{i=1}^{N} x_i + b\sum_{i=1}^{N} 1 - \sum_{i=1}^{N} y_i$$

$b\sum_{i=1}^{N} = b \cdot N$, thus:

$$= w\sum_{i=1}^{M} x_i + Nb - \sum_{i=1}^{N} y_i$$

The necessary and sufficient optimality conditions for

$$\min_{w,b\in\mathbb{R}} \mathcal{L}(w,b)$$

are given by $\frac{\partial}{\partial w}\mathcal{L}(w,b) = 0$ and $\frac{\partial}{\partial b}\mathcal{L}(w,b)$ and are equivalent to the linear system

1.3.4

$$\begin{pmatrix} \sum_{i=1}^{N} x_i^2 & \sum_{i=1}^{N} x_i \\ \sum_{i=1}^{N} x_i & N \end{pmatrix} \begin{pmatrix} w \\ b \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{N} x_i y_i \\ \sum_{i=1}^{N} y_i \end{pmatrix}$$

Explanation

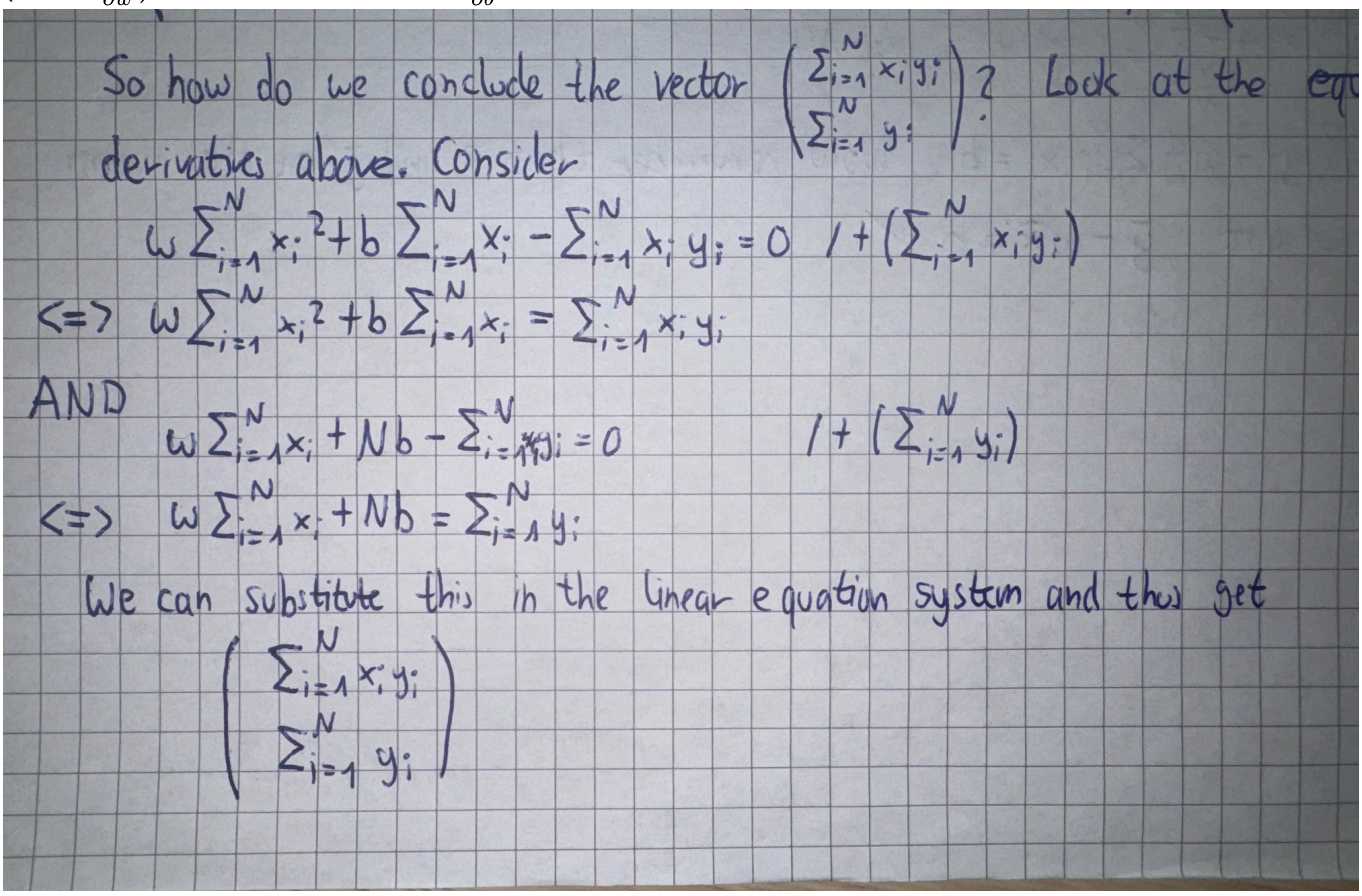
Let's reconsider matrix multiplications to understand the linear system given above:
The first element (11) is multiplicated with (1) of the vector and added up with the product of $\sum i = 1^N x_i * b$. Those products are added up so:

$$\begin{pmatrix} \sum_{i=1}^{N} x_i^2 & \sum_{i=1}^{N} x_i \\ \sum_{i=1}^{N} x_i & N \end{pmatrix} \begin{pmatrix} w \\ b \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{N} wx_i^2 + \sum_{i=1}^{N} x_i b \\ \sum_{i=1}^{N} x_i w + Nb \end{pmatrix} = \begin{pmatrix} w\sum_{i=1}^{N} x_i^2 + b\sum_{i=1}^{N} x_i \\ w\sum_{i=1}^{N} x_i + Nb \end{pmatrix}$$

But how do we conclude to

$$\begin{pmatrix} \sum_{i=1}^{N} x_i y_i \\ \sum_{i=1}^{N} y_i \end{pmatrix}$$

(here: $\frac{\partial}{\partial w}$) and after the AND its $\frac{\partial}{\partial b}$



The explanation above should be enough to conclude that we get to the final equation in 1.3.4

This can be further simplified by defining the averages $\overline{x} := \frac{1}{N} \sum_{i=1}^{N} x_i$ and $\overline{y} = \frac{1}{N} \sum_{i=1}^{N} y_i$ Now by dividing every entry of the matrices by N we get to:

$$\begin{pmatrix} \frac{1}{N} \sum_{i=1}^{N} x_i^2 & \overline{x} \\ \overline{x} & 1 \end{pmatrix} \begin{pmatrix} w \\ b \end{pmatrix} = \begin{pmatrix} \frac{1}{N} \sum_{i=1}^{N} x_i y_i \\ \overline{y} \end{pmatrix}$$

The determinant of the system matrix is:

$$\frac{1}{N} \sum_{i=1}^{N} x_i^2 - \overline{x}^2 = \frac{1}{N} \sum_{i=1}^{N} (x_i - \overline{x})^2$$

and is non-zero **exactly when** there are at least **two distinct** data points x_i. In this case we

can explicitly calculate w and b, and the solution is given by (see the exercise sheet)

Problem 2

We can rewrite it as a system of linear equations:

$$\begin{cases} w \frac{1}{N} \sum_{i=1}^{N} x_i^2 + b \bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i y_i \\ w \bar{x} + b = \bar{y} \end{cases}$$

Where we can immediately derive b:

$$b = \bar{y} - w \bar{x}$$

Let's substitute b into the first equation:

$$w \cdot \frac{1}{N} \sum_{i=1}^{N} x_i^2 + (\bar{y} - w \bar{x}) \bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i y_i \equiv$$

$$\equiv w \cdot \frac{1}{N} \sum_{i=1}^{N} x_i^2 + \bar{x} \bar{y} - w \bar{x}^2 = \frac{1}{N} \sum_{i=1}^{N} x_i y_i \equiv$$

$$\equiv w \left(\frac{1}{N} \sum_{i=1}^{N} x_i^2 - \bar{x}^2 \right) = \frac{1}{N} \sum_{i=1}^{N} x_i y_i - \bar{x} \bar{y}$$

Where $\frac{1}{N} \sum_{i=1}^{N} x_i^2 = E[x_i^2]$, $\bar{x}^2 = E[x_i]^2$,

$\frac{1}{N} \sum_{i=1}^{N} x_i y = E[x \cdot y]$.

We also know, that $Var(X) = E[(x_i - E[x])^2] =$

$= E[x_i^2] - E[x_i]^2$

and $Cov(X, Y) = E[x \cdot y] - E[x] \cdot E[y] =$

$= E[(x_i - E[x])(y_i - E[y])]$.

We can rewrite it:

$$w = \frac{cov(X, Y)}{Var(X)} = \frac{\sum_{i=1}^{N} (x - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{N} (x_i - \bar{x})^2}$$

1.3.5

$$w = \frac{\sum_{i=1}^{N}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

$$b = \overline{y} - w\overline{x}$$

Unlike nearest-neighbor interpolation, the linear regression function $\hat{f}(x) := wx + b$ is continuous and easy to evaluate. Additionally, it is more robust to errors in the data (x_i, y_i)
Remark: Let

$$X = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_N & 1 \end{pmatrix} \in \mathbb{R}^{Nx2}$$

$$\beta = \begin{pmatrix} w \\ b \end{pmatrix} \in \mathbb{R}^2$$

$$y = \begin{pmatrix} y_i \\ . \\ . \\ . \\ y_N \end{pmatrix}$$

we can rewrite the sum of squared deviations 1.3.3

$$L(\beta) = \frac{1}{2}||X\beta - y||^2$$

where $||\cdot||$ denotes the euclidian norm on $\mathbb{R}^N$ (normal Euclidian norm: $\sqrt{x_1^2 + x_2^2 + \cdots + x_N^2}$)

Explanation

Why does this work?
When multiplying $X\beta$ we get

$$X\beta = \begin{pmatrix} x_1 w + b \\ x_2 w + b \\ x_n w + b \end{pmatrix}$$

Remember the euclidian norm above. This effectively replaces the sum $\sum_{i=1}^{N} wx_i + b$. Thus

$$L(\beta) = \frac{1}{2}||(X\beta - y)_i^2 = \frac{1}{2}||X\beta - y||^2$$

Exercise 1.3.1

Show that for a model of the form $\hat{f}(x) = wx$ with parameter $w \in \mathbb{R}$, the least squares solution is given by

$$w = \frac{\sum_{i=1}^{N} x_i y_i}{\sum_{i=1}^{N} x_i^2}$$

Solution: We have $\mathcal{L}(w) := \frac{1}{2} \sum_{i=1}^{N} (wx_i - y_i)^2$
$\frac{\partial}{\partial w} \mathcal{L} = \sum_{i=1}^{N} (wx_i - y_i) x_i$. Since we want to minimize the function we set $\frac{\partial}{\partial w} \mathcal{L} = 0$:

$$\sum_{i=1}^{N} (wx_i - y_i) x_i = 0$$

$$\sum_{i=1}^{N} wx_i^2 - y_i x_i = 0 \quad | + \sum_{i=1}^{N} x_i y_i$$

$$w \sum_{i=1}^{N} x_i^2 = \sum_{i=1}^{N} x_i y_i \quad | \div \sum_{i=1}^{N} x_i^2$$

$$w = \frac{\sum_{i=1}^{N} x_i y_i}{\sum_{i=1}^{N} x_i^2}$$

□

Exercise 1.3.2

Show that the gradient of the function L in 1.3.3 is given by:

$$\nabla L(\beta) = X^T (X\beta - y)$$

Remember that for any vector,

$$|| \cdot ||^2 = v^T v$$

Since $|| \cdot ||^2$ is just adding up the squared matrix elements its equivalent (multiplying each element with itself and then add all elements up)
So:

$$||X\beta - y||^2 = (X\beta - y)^T (X\beta - y)$$

And the loss function would be

$$\mathcal{L}(\beta) = \frac{1}{2} (X\beta - y)^T (X\beta - y)$$

$$(X\beta - y)^T (X\beta - y) = (X\beta)^T X\beta - 2(X\beta)^T y + y^T y$$

Since the scalar transpose ($(X\beta)^T$ would be a $1 \times m$ row vector and we have a property for scalar values: $a^T = a$) does not change the value:
Remember that $(X\beta)^T y = y^T (X\beta)$. Since $(X\beta)^T = (X\beta)$ if its a scalar we can rewrite

$$= \beta^T X^T X\beta - 2y^T X\beta + y^T y$$

We add $\frac{1}{2}$ again:

$$\mathcal{L}(\beta) = \frac{1}{2}[\beta^T X^T X\beta - 2y^T X\beta + y^T y]$$

Thus:

$$\mathcal{L}(\beta) = \frac{1}{2}[\beta^T X^T X\beta - 2y^T X\beta + y^T y]$$

Now for any quadratic from like $\beta^T A\beta$, where A is a symmetric matrix, the gradient with respect to β is:

$$\nabla_\beta(\beta^T A\beta) = 2A\beta$$

Since in $(\beta^T X^T X\beta)$ $X^T X$ is symmetric, applying this formula gives:

$$\nabla_\beta(\beta^T X^T X\beta) = 2X^T X\beta$$

However, we still have $\frac{1}{2}$ in front of it so we get:

$$\nabla_\beta(\frac{1}{2}\beta^T X^T X\beta) = X^T X\beta$$

Differentiating second term $-2y^T X\beta$

For a linear term like $c^T\beta$ the gradient is simply the coefficient:

$$\nabla_\beta(c^T\beta) = c$$

Here, treating $X^T y$ as a constant vector we get

$$\nabla_\beta(\frac{1}{2} - 2y^T X\beta) = -X^T y$$

Now differentiating $y^T y$ which does not contain β this is 0
Combining the results we get

$$\nabla_\beta\mathcal{L}(\beta) = X^T X\beta - X^T y$$
$$\nabla\beta\mathcal{L}(\beta) = X^T(X\beta - y)$$

2.0.1 - Introduction

In this chapter we deal with supervised learning.
This refers to the following situation:

- We are looking for a mapping $f : \mathcal{X} \to \mathcal{Y}$, where we denote $\mathcal{X}$ as the input space and $\mathcal{Y}$ as the output space
- We consider $N \in \mathbb{N}$ data points $(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}$ for $i = 1, \ldots, N$.
- We choose a class of hypotheses $\mathcal{H}$, typically a (subset of a) function space
- We determine $\hat{f} \in \mathcal{H}$ such that $\hat{f}(x_i) \approx y_i$ for all $i = 1, \ldots, N$

Remark: it is often assumed that the data points (x_i, y_i) are samples from an (unknown) probability distribution on $\mathcal{X} \times \mathcal{Y}$
Within supervised learning, a distinction is often made between regression problems and classification problems. The distinguishing feature here is that in the case of quantitative output data, we speak of regression, and in the case of qualitative data, we speak of classification. However, the transition between these categories are fluid, and the terminology is not used consistently.

2.1 Regression

We begin with regression problems and consider some different models of increasing complexity.

Next up: 2.1.1 Linear regression

2.1.1 Linear regression

We have already seen linear regression in the What is Data Science? Introduction but we will consider a more general setting here.
We define

- the input space $\mathcal{X} := \mathbb{R}^n$ and the output space $\mathcal{Y} := \mathbb{R}^m$
- and the hypothesis class

$$\mathcal{H} := \{\hat{f} : \mathbb{R}^n \to \mathbb{R}^m | \exists W \in \mathbb{R}^{m \times n}, \hat{f}(x) = Wx\}$$

To determine $\hat{f}$ from $N \in \mathbb{N}$ data points (x_i, y_i) for $i = 1, \ldots, N$ we consider the problem

$$\min_{W \in \mathbb{R}^{m \times n}} \frac{1}{2} \sum_{i=1}^{N} ||Wx_i - y_i||^2$$

Let us define the data matrix $X \in \mathbb{R}^{N \times n}$ with $X_{ij} := (x_i)_j$, the matrix $Y \in \mathbb{R}^{N \times m}$ with $Y_{ij} := (y_i)_j$, and $\beta := W^T$, then we can rewrite this problem as

2.1.1

$$\min_{\beta \in \mathbb{R}^{n \times m}} \frac{1}{2} ||X\beta - Y||^2_{\text{Fro}}$$

where the Frobenius norm of a matrix $A \in \mathbb{R}^{N \times m}$ is defined as

$$||A||^2_{\text{Fro}} := \sum_{i=1}^{N} \sum_{j=1}^{m} |A_{ij}|^2 = \text{Tr}(AA^T)$$

(The sum of squares of all elements in the matrix)
We will first examine the solvability of the optimization problem

Proposition 2.1.1

If the matrix $X^T X \in \mathbb{R}^{n \times n}$ is invertible, then the unique solution of 2.1.1 is given by

2.1.2

$$\hat{\beta} := (X^T X)^{-1} X^T Y$$

Otherwise, there are infinitely many solutions.

Proof. We define the objective function $f(\beta) := \frac{1}{2}\|X\beta - Y\|_{\text{Fro}}^2$. Since f is a convex function of β, β solves the minimization problem (2.1.1) if and only if $\nabla f(\beta) = 0$. From Exercise 1.3.2, we know that the gradient of f is given by

$$\nabla f(\beta) = X^T(X\beta - Y).$$

Thus, $\nabla f(\beta) = 0$ is equivalent to $X^T X\beta = X^T Y$. This linear equation system has the unique solution $\hat{\beta} := (X^T X)^{-1} X^T Y$ if $X^T X$ is invertible, and infinitely many solutions otherwise. □

Remark 2.1.1 (Invertibility of $X^T X$)

Let us denote the columns of the matrix $X \in \mathbb{R}^{N\times n}$ by $a_i \in \mathbb{R}^N$ for $i = 1, \dots, n$. A fact from linear algebra states that $X^T X$ is **invertible if and only if the vectors** $(a_i)_{i=1}^n$ are linearly **independent** in $\mathbb{R}^N$. If $n >> N$, this is a "very likely" event and is referred to in data science as "independent features".
It is also important to note that the size of the matrix $X^T X \in \mathbb{R}^{n\times n}$ is independent of the number of data points. Thus, the difficulty of inversion does not increase with more data.

Next we will try to understand the case where $X^T X$ is not invertible and to construct a "meaningful" solution for 2.1.1. To do this we recall

Definition 2.1.1 (Singular Value Decomposition)

A singular value decomposition of a matrix $X \in \mathbb{R}^{N\times n}$ is a decomposition of the form $X = U\Sigma V^T$ with orthogonal matrices $U \in \mathbb{R}^{N\times N}$ and $V \in \mathbb{R}^{n\times n}$ and a matrix $\Sigma \in \mathbb{R}^{N\times n}$ of the form

$$\Sigma = \begin{pmatrix} \sigma_1 & \dots & \dots & \dots & \dots & \dots \\ \dots & \ddots & \dots & \dots & \dots & \vdots \\ \dots & \dots & \sigma_k & \dots & \dots & 0 \\ \dots & \dots & \dots & 0 & \dots & \vdots \\ \dots & \dots & \dots & & \ddots & \vdots \\ \dots & \dots & 0 & \dots & \dots & 0 \end{pmatrix}$$

with $k \le n$ singular values $\sigma_1 \ge \sigma_2 \ge \dots \ge \sigma_k > 0$ (Singular values are ordered by largest to smallest along the diagonal).
We define the pseudo inverse of X as

$$X^\dagger = V\Sigma^\dagger U^T$$

where $\Sigma^\dagger \in \mathbb{R}^{n\times N}$ is given by

$$\Sigma^\dagger = \begin{pmatrix} \frac{1}{\sigma_1} & \dots & \dots & \dots & \dots & \dots \\ \dots & \ddots & \dots & \dots & \dots & \vdots \\ \dots & \dots & \frac{1}{\sigma_k} & \dots & \dots & 0 \\ \dots & \dots & \dots & 0 & \dots & \vdots \\ \dots & \dots & \dots & & \ddots & \vdots \\ \dots & \dots & 0 & \dots & \dots & 0 \end{pmatrix}$$

Remark 2.1.2 The existence of the SVD can be proven by applying the Spectral Theorem to the matrix $X^T X$

Proposition 2.1.2. The following holds:

$$XX^\dagger X = X, \quad X^\dagger X X^\dagger = X^\dagger$$

$$X^\dagger X \text{ and } XX^\dagger \text{ are symmetric}$$

We observe that for the SVD $X = U\Sigma V^T$, it holds that

$$X^T X = (U\Sigma V^T)^T U\Sigma V^T = V\Sigma^T U^T U\Sigma V^T = V\Sigma^T \Sigma V^T$$

Definition of an Orthogonal matrix

An orthogonal matrix U is a **square matrix** whose columns (and rows) form an orthonormal set. This means:

- Orthonormal columns: Each column vector has a unit length ($||u_i|| = 1$) and any two distinct columns are perpendicular ($u_i \cdot u_j = 0$ for $i \neq j$)
- Inverse property: The inverse of U is equal to its transpose: $U^T = U^{-1}$
  - if $i = j : u_i \cdot u_j = 1$ (unit vectors)
- Orthogonal matrices represent rotations or reflections in space

Back to the previous definition

Thus, we have found a diagonalization (and in particular, a singular value decomposition) of $X^T X$. The positive eigenvalues of $X^T X$ are exactly given by σ_i^2 for $i = 1, \dots, k$.
In the case that $k = n$, i.e. there are n singular values, $X^T X$ is invertible with $(X^T X)^{-1} = VDV^T$ where $D = \text{diag}(\sigma_1^{-2}, \dots, \sigma_n^{-2})$. Otherwise, we can use the pseudoinverse of $X^T X$ to calculate the solution of 2.1.1 with minimal norm.

Proposition 2.1.3

If the matrix $X^T X \in \mathbb{R}^{n \times n}$ is not invertible, then the solution of 2.1.1 with minimal norm is given by

2.1.3

$$\hat{\beta} := (X^T X)^\dagger X^T Y$$

(similarly see 2.1.2)
where $(X^T X)^\dagger = VDV^T$ with (D := squares of the singular values of X)

$$D := \begin{pmatrix} \frac{1}{\sigma_1^2} & \cdots & \cdots & \cdots \\ \cdots & \ddots & \cdots & \cdots \\ \cdots & \cdots & \frac{1}{\sigma_k^2} & \cdots \\ \cdots & \cdots & \cdots & 0_{n-k,n-k} \end{pmatrix}$$

How do we know if $X^T X$ is invertible?

$$X = \begin{pmatrix} \cdots & x_1^T & \cdots \\ & \vdots & \\ \cdots & x_N^T & \cdots \end{pmatrix}$$

$C = X^T X$ is the covariance matrix. If you have data for which the variance of one direction is zero, you know that the eigenvalue for this direction will be 0 and C will not be invertible.
--> A (covariance?) Matrix with an eigenvalue of 0 is not invertible

2.1.2 Regularized Linear Models

Now we will briefly discuss so-called regularized linear models. In the case of a very large number of input variables, one may only be interested in a few variables that have the strongest influence. Thus, the model becomes

$$\min_{\beta \in \mathbb{R}^{n \times m}} \frac{1}{2} ||X\beta - Y||^2 \text{ such that most entries of } \beta \text{ are small or zero}$$

To enforce that the entries of β are small, we add a penalty term and obtain

2.1.4

$$\min_{\beta \in \mathbb{R}^{n \times m}} \frac{1}{2} ||X\beta - Y||^2 + \frac{\lambda}{2} ||\beta||^2$$

where $\lambda \geq 0$ is a freely selectable parameter. This model is also called Ridge Regression. Here, $|| \cdot ||$ denotes the Frobenius norms, but we omit the corresponding designation.
What is remarkable about this model is that for all $\lambda > 0$, it has a unique solution, regardless of the invertibility of $X^T X$

Proposition 2.1.4

For all $\lambda > 0$, the problem 2.1.4 has a unique solution given by

2.1.5

$$\hat{\beta}_\lambda := (X^T X + \lambda \mathbb{I})^{-1} X^T Y$$

Also, $\hat{\beta}_\lambda$ represents the **Lasso estimate** of the regression coefficient β for a given regularization parameter λ. Note that β is the true coefficient in a regression model, $\hat{\beta}_\lambda$ is the estimated value of β obtained by solving the Lasso optimization problem.
(This is derived by expanding the squared norms from 2.1.4, then compute the gradient w.r.t beta and rearranging terms)

Expanding

The squared Frobenius norm $\|X\beta - Y\|^2$ expands as follows:

$$\|X\beta - Y\|^2 = \text{Trace}\left((X\beta - Y)^T(X\beta - Y)\right).$$

Expanding the product:

$$(X\beta - Y)^T(X\beta - Y) = \beta^T X^T X\beta - \beta^T X^T Y - Y^T X\beta + Y^T Y.$$

Taking the trace (linear operator):

$$\|X\beta - Y\|^2 = \underbrace{\text{Trace}(\beta^T X^T X\beta)}_{\text{Quadratic Term}} - \underbrace{2\text{Trace}(Y^T X\beta)}_{\text{Linear Term}} + \underbrace{\text{Trace}(Y^T Y)}_{\text{Constant}}.$$

Key Simplifications:

1. **Quadratic Term**:
   $\text{Trace}(\beta^T X^T X\beta) = \beta^T X^T X\beta$ (if β is a vector).
2. **Linear Term**:
   $\text{Trace}(Y^T X\beta) = Y^T X\beta$ (if β is a vector).
3. **Constant Term**:
   $\text{Trace}(Y^T Y) = \|Y\|^2$.

For simplicity, assume β is a vector ($m = 1$). Then:

$$\|X\beta - Y\|^2 = \beta^T X^T X\beta - 2Y^T X\beta + Y^T Y.$$

2. Expanding the Regularization Term

The regularization term $\|\beta\|^2$ is the Frobenius norm squared of β:

$$\|\beta\|^2 = \beta^T \beta \quad (\text{for } \beta \text{ as a vector}).$$

---

3. Full Loss Function

Substitute the expanded terms into the loss function:

$$\mathcal{L}(\beta) = \frac{1}{2}\left(\beta^T X^T X\beta - 2Y^T X\beta + Y^T Y\right) + \frac{\lambda}{2}\beta^T \beta.$$

---

4. Computing the Gradient

To find the optimal β, take the derivative of $\mathcal{L}(\beta)$ with respect to β and set it to zero:

1. **Quadratic Term**:
$$\frac{\partial}{\partial \beta} \left(\frac{1}{2} \beta^T X^T X \beta \right) = X^T X \beta.$$
2. **Linear Term**:
$$\frac{\partial}{\partial \beta} \left(-Y^T X \beta \right) = -X^T Y.$$
3. **Regularization Term**:
$$\frac{\partial}{\partial \beta} \left(\frac{\lambda}{2} \beta^T \beta \right) = \lambda \beta.$$
4. **Constant Term**:
$\frac{\partial}{\partial \beta} \left(\frac{1}{2} Y^T Y \right) = 0.$

Combine these results:

$$\nabla \mathcal{L}(\beta) = X^T X \beta - X^T Y + \lambda \beta = 0.$$

---

5. Solving for β

Rearrange the equation:

$$(X^T X + \lambda \mathbb{I}) \beta = X^T Y.$$

Since $X^T X + \lambda \mathbb{I}$ is always invertible for $\lambda > 0$, the solution is:

$$\boxed{\hat{\beta}_\lambda = (X^T X + \lambda \mathbb{I})^{-1} X^T Y}.$$

Back to the script

Proof

The objective function above 2.1.4 is convex, thus the optimality condition is

2.1.6

$$0 = X^T (X\hat{\beta} - Y) + \lambda \hat{\beta} = (X^T X + \lambda \mathbb{I}) \hat{\beta} - X^T Y$$

(Note that the first part $0 = X^T(X\hat{\beta} - Y) + \lambda\hat{\beta}$ is the gradient which must be 0)which is necessary and sufficient for minimality. We assert that the square matrix $X^TX + \lambda\mathbb{I}$ is invertible. To show this, it suffices to check injectivity. Thus, assume that $(X^TX + \lambda\mathbb{I})\beta = 0$ or equivalently $X^TX\beta = -\lambda\beta$. Then we obtain

$$-\lambda||\beta||^2 = \langle X^TX\beta, \beta\rangle = \langle X\beta, X\beta\rangle = ||X\beta||^2 \geq 0$$

Since $\lambda > 0$, this is only possible if $\beta = 0$, which means injectivity. Thus, the optimality condition 2.1.6 has the unique solution $\hat{\beta}_\lambda = (X^TX + \lambda\mathbb{I})^{-1}X^TY$We can also express the solution 2.1.5 using the pseudoinverse. Let $X = U\Sigma V^T$ be a singular value decomposition of X. Then we have

$$\begin{aligned} X^TX + \lambda\mathbb{I} &= (U\Sigma V^T)^T(U\Sigma V^T) + \lambda VV^T \\ &= V\Sigma^TU^TU\Sigma V^T + \lambda VV^T \\ &= V\Sigma^T\Sigma V^T + \lambda VV^T \\ &= V(\Sigma^T\Sigma + \lambda)V^T \end{aligned}$$

The matrix in parantheses has the following form:

$$\Sigma^T\Sigma + \lambda = \text{diag}(\sigma_1^2 + \lambda, \ldots, \sigma_k^2 + \lambda, \lambda \ldots, \lambda)$$

In particular, we obtain

$$\begin{aligned} (X^TX + \lambda\mathbb{I})^{-1}X^T &= V\text{diag}(\frac{1}{\sigma_1^2 + \lambda}, \ldots, \frac{1}{\sigma_k^2 + \lambda}, \frac{1}{\lambda}, \ldots \frac{1}{\lambda})V^T \cdot (U\Sigma V^T)^T \\ &= V\text{diag}(\frac{1}{\sigma_1^2 + \lambda}, \ldots, \frac{1}{\sigma_k^2 + \lambda}, \frac{1}{\lambda}, \ldots \frac{1}{\lambda})V^TV\text{diag}(\sigma_1, \ldots, \sigma_k, 0, \ldots, 0)U^T \\ &= \text{now } \mathrm{V}^TV \text{ cancels out and we then multiply diag with the other diag:} \\ &= V\text{diag}(\frac{\sigma_1}{\sigma_1 + \lambda}, \cdots, \frac{\sigma_k}{\sigma_k + \lambda}, 0, \ldots, 0)U^T \end{aligned}$$

Corollary 2.1.1

As $\lambda \to 0$, the solution $\hat{\beta}_\lambda$ of 2.1.4 converges to the minimum-norm solution $\hat{\beta} := (X^TX)^\dagger X^TY$

Proof

According to the above calculation, we have

$$= V\text{diag}(\frac{\sigma_1}{\sigma_1 + \lambda}, \cdots, \frac{\sigma_k}{\sigma_k + \lambda}, 0, \ldots, 0)U^T$$

As $\lambda \to 0$ we have

$$\lim_{\lambda \to 0} \frac{\sigma_i}{\sigma_i^2 + \lambda} = \frac{\sigma_i}{\sigma_i^2} = \frac{1}{\sigma_i}$$

Thus:

$$\begin{aligned}\lim_{\lambda \to 0}(X^TX + \lambda \mathbb{I})^{-1}X^T &= V\text{diag}(\sigma_1^{-1}, \dots, \sigma_k^{-1}, 0, \dots, 0)U^T \\ &= (X^TX)^{\dagger}X^T\end{aligned}$$

Thus, the claim follows from Proposition 2.1.3
It turns out that the Frobenius/Euclidian norms in 2.1.4 are not suitable for reducing the entries of $\hat{\beta}_\lambda$ to zero.

Example 2.1.1

The solution of

$$\min_{\beta \in \mathbb{R}} \frac{1}{2}(x\beta - y)^2 + \frac{\lambda}{2}\beta^2$$

is given by the derivative (chain rule)

$$x(x\beta - y) + \lambda\beta = 0$$

Now solving for β

$$\begin{aligned}x^2\beta - xy + \lambda\beta &= 0 \\ \Rightarrow \beta(x^2 + \lambda) &= xy \quad | \div (x^2 + \lambda) \\ \hat{\beta}_\lambda &= \frac{xy}{x^2 + \lambda}\end{aligned}$$

This can easily be seen by differentiation. Thus, for all $\lambda \in [0, \infty)$, it holds that $\hat{\beta}_\lambda \neq 0$.In contrast, let us consider the problem

$$\min_{\beta \in \mathbb{R}} \frac{1}{2}(x\beta - y)^2 + \lambda|\beta|$$

This problem can be equivalently reformulated as

$$\min_{\beta \in \mathbb{R}} f(\beta)$$

where

$$\begin{aligned} f(\beta) &= \frac{1}{2}(x^2\beta^2 - 2yx\beta + y^2) + \lambda|\beta| \\ &= \frac{1}{2}x^2\beta^2 - xy\beta + \frac{1}{2}y^2 + \lambda|\beta| \qquad\qquad |\div x^2 \\ &= \frac{1}{2}\beta^2 - \frac{y}{x}\beta + \frac{y^2}{2x^2} + \frac{\lambda}{x^2}|\beta| \\ &\text{since constants do not affect the minimizer, we drop} \frac{y^2}{2x^2} \\ f(\beta) &:= \frac{1}{2}\beta^2 - \frac{y}{x}\beta + \frac{\lambda}{x^2}|\beta| \end{aligned}$$

Suppose that $\frac{y}{x} > 0$. Then the solution $\hat{\beta}_\lambda > 0$ because otherwise, putting $\beta = -\beta_\lambda$ would yield $f(\beta) < f(\hat{\beta}_\lambda)$. Then for $\beta \geq 0$, we can rewrite

$$f(\beta) := \frac{1}{2}\beta^2 - \frac{y}{x}\beta + \frac{\lambda}{x^2}\beta$$

Differentiation leads to

$$f'(\beta) = \beta - \left(\frac{y}{x} - \frac{\lambda}{x^2}\right)$$

If $(\frac{y}{x} - \frac{\lambda}{x^2}) < 0$ then $f'(\beta) > 0$ for all $\beta > 0$ hence f is increasing on $[0, +\infty]$ and the minimum is attained for $\hat{\beta}_\lambda = 0$. On the other hand, if $(\frac{y}{x} - \frac{\lambda}{x^2}) > 0$, then $f'(\hat{\beta}_\lambda) = 0 \implies \hat{\beta}_\lambda = \frac{y}{x} - \frac{\lambda}{x^2}$. We can then pout

$$\hat{\beta}_\lambda = \max\left\{\frac{y}{x} - \frac{\lambda}{x^2}, 0\right\}$$

Now supposing that $\frac{y}{x} \leq 0$ and using the same reasoning, we can show that

$$\hat{\beta}_\lambda = -\max\left\{-\frac{y}{x} - \frac{\lambda}{x^2}, 0\right\}$$

Using this formula, we see that for $\lambda \geq |xy|$, the solution $\hat{\beta}_\lambda$ is equal to zero.
Motivated by this example, instead of 2.1.4, we now consider the following model

2.1.7 (LASSO regression)

$$\min_{\beta \in \mathbb{R}^{n \times m}} \frac{1}{2}||X\beta - Y||^2 + \lambda||\beta||_1$$

where $||\beta||_1 := \sum_{i,j} |\beta_{i,j}|$. This model is also known as LASSO regression, where LASSO stands for *Least Absolute Shrinkage and Selection Operator*Since the objective function above is not differentiable, we cannot derive optimality conditions using our methods. Moreover, (except for the uninteresting case where X is an orthogonal matrix) it it is not possible to derive an explicit solution. Solutions must be approximated using an iterative procedure. Here, we

consider the **ISTA** (**I**terative **S**oft **T**hresholding **A**lgorithm), which successively reduces the first and second terms in 2.1.7. For a step size $\tau > 0$, one computes:

2.1.8 (ISTA Algorithm Steps)

$$\begin{cases} \text{Initialize } \beta^0 \\ \beta^{k-1/2} := \beta^{k-1} - \tau X^T(X\beta^{k-1} - Y) \\ \beta^k := \mathcal{S}_{\tau\lambda}\left(\beta^{k-1/2}\right) \end{cases}$$

As an initialization one might choose for example, $\beta^0 = 0$ or $\beta^0 = (X^T X)^\dagger X^T Y$.The operator $\mathcal{S}_\lambda$ is the so-called *Soft Thresholding* operator, which is applied entry-wise and is defined as follows:

2.1.9

$$\mathcal{S}_\mu(x) = \operatorname{sgn}(x)\max\{|x| - \mu, 0\}$$

The optimal step size τ, which guarantees the convergence of 2.1.8 to the solution of 2.1.7 can be precomputed and is given by

2.1.10 (step size computation)

$$\tau := \frac{1}{\sigma_1(X)^2}$$

where $\sigma_1(X)^2$ is the largest singular value of X. An interesting special case is $\lambda = 0$ and thus $\mathcal{S}_{\tau\lambda} = \mathrm{id}$. In this case, 2.1.8 reduces to

$$\begin{cases} \text{Initialize } \beta^0 \\ \beta^k := \beta^{k-1} - \tau X^T(X\beta^{k-1} - Y) \quad k \in \mathbb{N} \end{cases}$$

which is a very slow algorithm for iteratively solving the linear system $X\beta = Y$. The use of the pseudoinverse or better iterative methods is preferred for $\lambda = 0$

2.1.3 Significance of Parameters

Let say I want to buy a new home. However, before buying it, I want to have a general idea of how much the number of square meters affects the price of a house. To this end, I model the relationship between the price of a house and its ground surface as linear, i.e.:

2.1.11

$$y = wx + b$$

where x is the number of square meters and y is the price.
However, I do not have access to the actual coefficients w and b (the ground truth). However, I can visit several houses, and for the i-th house, measure the number of square meters x_i, and ask the owners for the price y_i. The relationship between x_i and y_i can be modeled as

$$y_i = wx_i + b + \epsilon_i, \quad i = 1, \dots, N$$

where ϵ_i denotes measurement or model error, which in this case might be influenced by several factors such as:

- how old the house is
- how nice the neighborhood is
- how greedy the owner is
- with which amount of precision the ground surface was measured
- etc

I can then estimate the parameters w and b using linear regression. The corresponding least squares problem is

$$\min_{w,b \in \mathbb{R}} \frac{1}{2} \sum_{i=1}^{N} |wx_i + b - y_i|^2$$

and we know that the solution ($\hat{w}$, $\hat{b}$) to this problem is

2.1.13

$$\hat{w} = \frac{\sum_{i=1}^{N}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{N}(x_i - \overline{x}^2)}$$

$$\hat{b} = \overline{y} - \hat{w}\overline{x}$$

To save time, I visit 10 houses and ask a good friend of mine to visit 10 others. Then we each

compute the linear regression corresponding to our own data, and draw the curve we each obtain

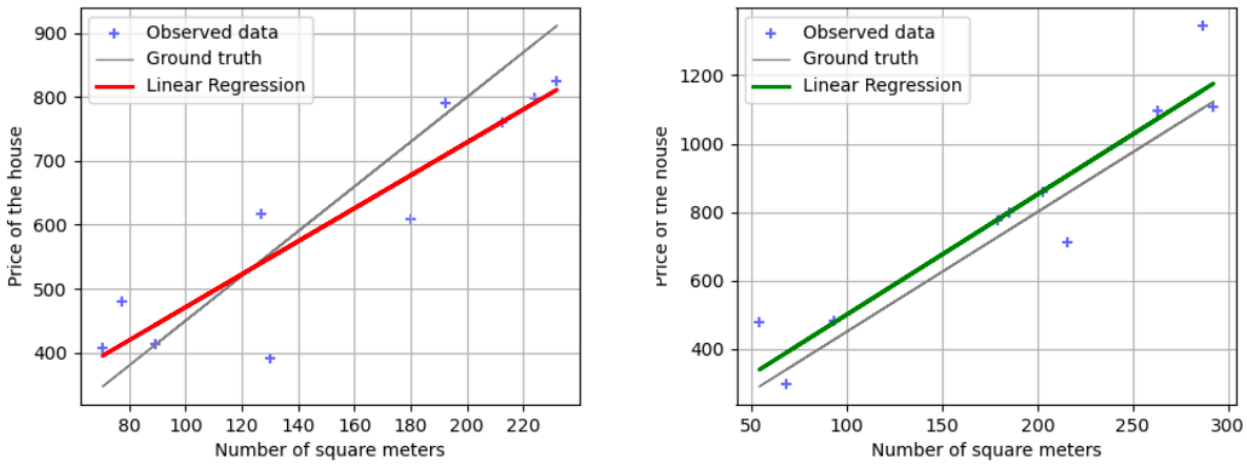


Figure 2.1: Linear regression curve I computed on the data I observed(left, in red) vs. the curve my friend obtains (right, in green). The ground truth (in grey) is unknown.

Surprisingly, the curves we obtain are pretty different. I think that this can be linked to different factors, like the few number of data points (x_i, y_i) my friend and I considered (called the sample size) or the amount of noise ϵ_i. In order to get another opinion, I ask a real estate agency for their data. They know the surface and price of over 100 different houses, and used this data to compute the linear regression showed below.

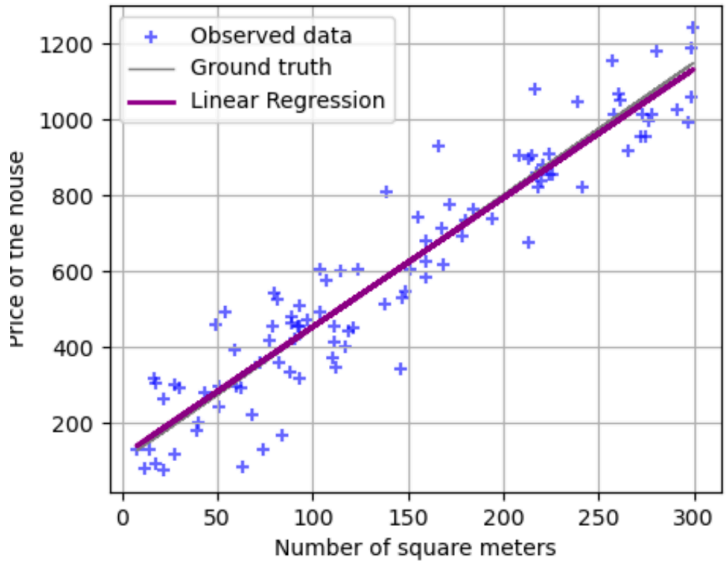


Figure 2.2: Linear regression curve the real estate agency computed on 100 samples

As we can see, this curve is different from the previous ones

Question

Since I do not know the actual values of w and b, which values of $\hat{w}$ and $\hat{b}$ should I trust more? Mine, my friend's or the one of the real estate agency?
This type of question is at the heart of *statistical tests* and forms the basis of modern scientific method. It is used in areas like pharmacology, epidemiology, sociology, psychology, finance, particle physics, etc.

Reminder of probability theory

Definition 2.1.2 (Random Variable)

Let $(\Omega, \mathbb{P})$ be a probability space. A random variable X is a function from Ω to $\mathbb{R}$. It assigns a real number to each outcome in probability space
Purpose: It translates abstract outcomes (e.g. "heads", "tails") into numerical values (e.g. 1 for heads, 0 for tails) so we can analyze probabilities mathematically

Example:

- Let $\Omega = \{\text{rain}, \text{snow}, \text{sunny}\}$
- Define $X(\text{rain}) = 0, X(\text{snow}) = 1, X(\text{sunny}) = 2$
- X is a random variable representing weather numerically

Definition 2.1.3 (Cumulative distributive function)

Let X be a random variable. Its cumulative distribution function $F_X : \mathbb{R} \to [0, 1]$ is defined as

$$F_X(x) := \mathbb{P}(X \leq x)$$

It gives the probability that X takes a value **less than or equal to** x

- $F_X(x)$ is non-decreasing
- $\lim_{x \to -\infty} F_X(x) = 0, \lim_{x \to +\infty} F_X(x) = 1$

Example
For a fair 6-sided die:

- $F_X(3) = \mathbb{P}(X \leq 3) = \frac{3}{6} = 0.5$
- $F_X(6) = \mathbb{P}(X \leq 6) = 1, F_X(0) = 0$

Definition 2.1.4 (Probability density function)

Let X be a random variable and suppose that F_X is differentiable. The probability density function is defined as $f_X = F_X'$
What it is: The derivative of the CDF:

$$f_X(x) = \frac{d}{dx} F_X(x)$$

Purpose: Describes the "density" of probability at a point x for **continuous** random variables

- Probabilities are calculated by integrating the PDF

$$\mathbb{P}(a \leq X \leq b) = \int_a^b f_X(x)dx$$

Example:
For a normal distribution, the PDF is the bell-shaped curve

$$f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Remark 2.1.3 Suppose that X can only take values in a discrete set $\{x_i\}_{i\in\mathbb{N}} \subset \mathbb{R}$.
Then X is called a discrete random variable. A random variable that can take all values in $\mathbb{R}$ is called a *continuous random variable*
Discrete Random Variables
X takes a value in a countable set $\{x_i\}$
Instead of a PDF, discrete variables use a PMF: $\mathbb{P}(X = x_i) = p_i$
e.g. $\mathbb{P}(X = 1) = \frac{1}{6}$ for a fair die

Continuous Random Variables
X can take uncountably many values in $\mathbb{R}$
CDF has a smooth and differentiable function

Definition 2.1.5 (Expectation)

Let X be a continuous random variable with probability density function f_X. The expectation of X is defined as

$$\mathbb{E}(X) := \int_{\mathbb{R}} x f_X(x)dx$$

If X is a discrete random variable, its expectation is defined as

$$\mathbb{E}(X) := \sum_{i\in\mathbb{N}} x_i \mathbb{P}(X = x_i)$$

Exercise 2.1.1

Let X, Y be random variables and $a \in \mathbb{R}$. Show that

$$\mathbb{E}(aX + Y) = a\mathbb{E}(X) + \mathbb{E}(Y)$$

$$\mathbb{E}(aX + Y) = \mathbb{E}(aX) + \mathbb{E}(Y)$$

$$= a\mathbb{E}(X) + \mathbb{E}(Y)$$

Definition 2.1.6 (Variance and covariance)

The variance of a random variable X is defined as

$$\mathrm{Var}(X) := \mathbb{E}[(X - \mathbb{E}(X))^2]$$

or

$$\mathrm{Var}(X) := \mathbb{E}(X^2) - \mathbb{E}(X)^2$$

The covariance of two random variables X, Y is defined as

$$cov(X, Y) := \mathbb{E}[(X - \mathbb{E}(X))(Y - \mathbb{E}(Y))]$$

Exercise 2.1.2

Show that

$$\mathbb{V}(X + Y) = V(X) + V(Y) + 2cov(X, Y)$$

Step 1:

$$V(X + Y) = \mathbb{E}((X + Y)^2) - (\mathbb{E}(X + Y))^2$$

Step 2: Look at $\mathbb{E}((X + Y)^2)$ and expand $(X + Y)^2$

$$(X + Y)^2 = X^2 + 2XY + Y^2$$

-->

$$\mathbb{E}((X + Y)^2) = \mathbb{E}(X^2) + 2\mathbb{E}(XY) + \mathbb{E}(Y^2)$$

Step 3: Look at second part $(\mathbb{E}(X + Y))^2$ and expand $\mathbb{E}(X + Y)$
The expectation of a sum is the sum of expectations

$$\mathbb{E}(X + Y) = \mathbb{E}(X) + \mathbb{E}(Y)$$

So:

$$(\mathbb{E}(X + Y)^2) = (\mathbb{E}(X) + \mathbb{E}(Y))^2 = \mathbb{E}(X)^2 + 2\mathbb{E}(X)\mathbb{E}(Y) + \mathbb{E}(Y)^2$$

Step 4: Substitute into variance formula

$$V(X + Y) = \mathbb{E}(X^2) + 2\mathbb{E}(XY) + \mathbb{E}(Y^2) - (\mathbb{E}(X)^2 + 2\mathbb{E}(X)\mathbb{E}(Y) + \mathbb{E}(Y)^2)$$

Rearranging gets us to

$$= (\mathbb{E}(X^2) - \mathbb{E}(X)^2) + (\mathbb{E}(Y^2) - \mathbb{E}(Y)^2) + 2(\mathbb{E}(XY) - \mathbb{E}(X)\mathbb{E}(Y))$$

Recognize Variance and Covariance

$V(X) = \mathbb{E}(X^2) - \mathbb{E}(X)^2$,
$V(Y) = \mathbb{E}(Y^2) - \mathbb{E}(Y)^2$
$cov(x, y) = \mathbb{E}(XY) - \mathbb{E}(X)\mathbb{E}(Y)$

Substituting this gets us to

$$V(X + Y) = V(X) + V(Y) + 2cov(X, Y)$$

Definition 2.1.7 (Independence)

We say that two random variables X, Y are independent if for all $U, V \subset \mathbb{R}$

$$\mathbb{P}(X \in U, Y \in V) = \mathbb{P}(X \in U)\mathbb{P}(Y \in V)$$

Proposition 2.1.5 (Independance and covariance)

If X and Y are independent then

$$cov(X, Y) = 0$$

Remark 2.1.4 The converse is not true in general!!

Example 2.1.2 (Common distributions)

1. Let $p \in [0, 1]$. We say that X is a Bernoulli random variable (denoted $X \sim \mathcal{B}(p)$) if and only if

   $$\mathbb{P}(X = 1) = p$$

   and

   $$\mathbb{P}(X = 0) = 1 - p$$

   We have in this case:

   $$\mathbb{E}(X) = p$$

   and

   $$V(X) = p(1 - p)$$

   For $p = 1/2$, this can for instance model the behavior of a coin flip.
2. We say that a continuous random variable X is uniformly distributed on $[0, 1]$, and denote $X \sim Unif([0, 1])$, when the probability distribution function is $f_X(x) = 1$. In this case, we have

   $$\mathbb{E}(X) = \frac{1}{2}$$

   and

$$V(X) = \frac{1}{12}$$

3. Let $\mu, \sigma \in \mathbb{R}$. We say that X follows a Gaussian probability distribution (denoted by $X \sim \mathcal{N}(\mu, \sigma^2)$) when the probability density function of X is

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

We have in this case:

$$\mathbb{E}(X) = \mu$$

and

$$V(X) = \sigma^2$$

Statistical significance

In this section we consider the even simpler model of only estimating the mean of the observed data, i.e., we use the model

2.1.14

$$y_i = b + \epsilon_i, \quad i = 1, \ldots, N$$

A real-world application might be the following: suppose that you run an online store, and you want to change the design of your website from its original design A to a design B. You want that this change in design to have some impact on the amount your customers usually spend on your website. You already know that when presented with the original design A, a user usually spends in average b_A euros. You want to estimate the amount b_B an average user would spend when presented the design B. In order to do so, you present the new design B to a sample of N random users, and model the amount y_i a user spend by

2.1.15

$$y_i = b_B + \epsilon_i, \quad i = 1, \ldots, N$$

We want to estimate the value of b_B and compare it to b_A. If $b_B \neq b_A$, we will definitely use the design B. If $b_B = b_A$, we keep the design A.
The hypothesis we are testing is $b_B = b_A$. This is what we call the **null hypothesis**, and is usually denoted by H_0. The alternative hypothesis (usually, the one we want to prove: here, $b_A \neq b_B$) is denoted by H_1. The idea is to reject the null hypothesis in such a way that the probability of wrongly rejecting it is very small.
However, we do not have access to the true value of b_B. The best we can do is to estimate it by its empirical mean obtained by least square approximation

$$\hat{b}_B = \frac{1}{N}\sum_{i=1}^{N} y_i$$

The way we will test if H_0 is likely to be true or false is to quantify how probable it is to observe the empirical value $\hat{b}_B$ under the assumption that H_0 is true. If this probability is small, then we will reject H_0. In order to do so, we need to use a test statistic, i.e. a random variable for which the distribution is known, provided that H_0 is true. To derive the test statistic, we make the strong assumptions that ϵ_i are independent and normally distributed random variables with $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ and that we know their variance σ^2. In this case, the y_i are i.i.d. (independent and identically distributed) according to $\mathcal{N}(b_B, \sigma^2)$ and therefore $\hat{b}_B \sim \mathcal{N}(b_B, \frac{\sigma^2}{N})$
The test statistic in this case

$$T := \frac{\hat{b}_B - b_a}{\sqrt{V(\hat{b}_B)}} = \frac{\sqrt{N}}{\sigma}(\hat{b}_B - b_a)$$

Under the null hypothesis $H_0 := (b_B = b_A)$, we see that

2.1.16

$$T = \frac{\sqrt{N}}{\sigma}(\hat{b}_B - b_a)$$

and we can show that $T \sim \mathcal{N}(0, 1)$

Exercise 2.1.3

Show that we indeed have

$$\mathbb{E}(T) = 0$$

and

$$V(T) = 1$$

Step 1: Expected Value of T

We want to show $\mathbb{E}(T) = 0$.

1. **Expectation of $\hat{b}_B$:**
   Since $y_i = b_B + \epsilon_i$, the expected value of y_i is:

   $$\mathbb{E}(y_i) = b_B$$

   Therefore, the expectation of $\hat{b}_B$ (the sample mean) is:

   $$\mathbb{E}(\hat{b}_B) = \mathbb{E}\left(\frac{1}{N}\sum_{i=1}^{N} y_i\right) = \frac{1}{N}\sum_{i=1}^{N}\mathbb{E}(y_i) = b_B$$

2. **Under H_0:**
   If H_0 is true ($b_B = b_A$), then:

   $$\mathbb{E}(\hat{b}_B) = b_A$$

3. **Expectation of T:**
   Substitute $\mathbb{E}(\hat{b}_B) = b_A$ into T:

   $$\mathbb{E}(T) = \mathbb{E}\left(\frac{\sqrt{N}}{\sigma}(\hat{b}_B - b_A)\right) = \frac{\sqrt{N}}{\sigma}\left(\mathbb{E}(\hat{b}_B) - b_A\right) = \frac{\sqrt{N}}{\sigma}(b_A - b_A) = 0$$

   Result: $\mathbb{E}(T) = 0$.

Step 2: Variance of T

We want to show $V(T) = 1$.

1. **Variance of $\hat{b}_B$:**
   Since y_i are i.i.d. with variance σ^2:

$$V(\hat{b}_B) = V\left(\frac{1}{N}\sum_{i=1}^{N} y_i\right) = \frac{1}{N^2}\sum_{i=1}^{N} V(y_i) = \frac{1}{N^2} \cdot N\sigma^2 = \frac{\sigma^2}{N}$$

2. **Variance of $\hat{b}_B - b_A$:**
   b_A is a constant (not random), so:

$$V(\hat{b}_B - b_A) = V(\hat{b}_B) = \frac{\sigma^2}{N}$$

3. **Variance of T:**
   Use the property $V(aX) = a^2V(X)$ for a constant a:

$$V(T) = V\left(\frac{\sqrt{N}}{\sigma}(\hat{b}_B - b_A)\right) = \left(\frac{\sqrt{N}}{\sigma}\right)^2 V(\hat{b}_B - b_A)$$

   Substitute $V(\hat{b}_B - b_A) = \frac{\sigma^2}{N}$:

$$V(T) = \left(\frac{N}{\sigma^2}\right) \cdot \frac{\sigma^2}{N} = 1$$

   Result: $V(T) = 1$.

Remark 2.1.5

If the null hypothesis is false, i.e. $b_B \neq b_A$, then we can show that

$$T \sim \mathcal{N}\left(\frac{\sqrt{N}}{\sigma}(b_B - b_a), 1\right)$$

Hence, with large N and small σ, we see that T "deviates" more and more from 0.
We then choose a number $\alpha \in (0, 1)$, called the statistical significance, which has the property that

$$\mathbb{P}(H_0 \text{ is rejected } | H_0 \text{ is true}) \leq \alpha$$

i.e. the probability of wrongly rejecting H_0 is smaller than α. Usually, α is taken equal to 0.05, 0.01 or much lower depending on the field of study
The last thing we need is to actually define what it means to reject H_0. To this purpose, we need to define a *rejection rule*.
Using Equation 2.1.16, we see that T is very likely to be close to 0 when H_0 is true. Hence, we decide to reject H_0 when T is far enough from 0. However, T being a random variable, we decide to reject H_0 by choosing $M > 0$ such that

$$\mathbb{P}(|T| > M | H_0 \text{ is true}) \leq \alpha$$

When H_0 is true, we know that $T \sim \mathcal{N}(0,1)$. Hence,

$$\begin{aligned}\mathbb{P}[\text{H0 is rejected} \mid \text{H0 is true}] &:= \mathbb{P}[|T| \geq M \mid \text{H0 is true}] \\ &= \mathbb{P}_{t\sim\mathcal{N}(0,1)}[|t| \geq M]\end{aligned}$$

In the specific case for $\alpha = 0.02$, we can show that taking $M \approx 2.33$ guarantees that $\mathbb{P}_{t\sim\mathcal{N}(0,1)}[|t| \geq M]$
Consequently, we reject the null hypothesis if $|T| \geq 2.33$ (or equivalently $|\hat{\beta} - b_A| \geq 2.33\frac{\sigma}{\sqrt{N}}$) and declare the value of $\hat{b}_B$ significant at the level α
Important It is incorrect to conclude that in this case the probability of the null hypothesis being true is 2%!
A quantity similar to the significance level is the so-called p-value, which in this specific case is defined as $p(T) = \mathbb{P}_{t\sim\mathcal{N}(0,1)}[|t| \geq |T|]$. In contrast to $\alpha, p(T)$ is a random variable and is not predetermined
The p-value $p(T)$ is the probability of observing a test statistic as extreme as T under H_0. For $\alpha = 0.02$, we reject H_0 if $|T| \geq 2.33$ corresponding to $p(T) \leq 0.02$

Skipped a few things here :)

Definition 2.1.8 (Hypothesis Test)

A hypothesis test for a null hypothesis H_0 consists of a test statistic T along with a decision rule against or in favor of H_0 based on T.
In the case of a real test statistic T and a decision rule of the form $T \geq M$ (or $T \leq M$) against H_0, we refer to it as a right tailed (or left-tailed) test. A decision rule of the form $T \geq M$ and $T \leq N$ for $M, N \in \mathbb{R}$ with $M \leq N$ refers to a two-tailed test.

Next, we define the p-value. For this purpose, we restrict ourselves to simple null hypotheses H_0 (e.g., $b = 0$, but not $b \leq 0$), which uniquely determine the distribution of $T|H_0$

Definition 2.1.19 (p-value)

In the case of a real test statistic T with an (unknown) distribution μ and a simple null hypothesis H_0, the p-value of T is defined as

- $p(T) := \mathbb{P}_{t\sim\mu}[t \geq T|H_0]$ for right-tailed tests
- $p(T) := \mathbb{P}_{t\sim\mu}[t \leq T|H_0]$ for left-tailed tests
- $p(T) :=. 2\min\{\mathbb{P}_{t\sim\mu}[t \leq T|H_0], \mathbb{P}_{t\sim\mu}[t \geq T|H_0]\}$ for a two-tailed test. In the case of a distribution of T that is symmetric around zero, it holds that $p(T) = \mathbb{P}_{t\sim\mu}[|t| \geq |T|\ |H_0]$

The p-value is the probability of observing data even more "extreme" than T under the assumption that H_0 is true.

Remark 2.1.6 It is important to note that by definition, the p-value $p(T)$ of a test statistic T is a random variable. Therefore, its values can vary significantly and are not suitable for a posteriori determination of a significance level!

2.1.4 Nonlinear Regression

The approximation of data points (x_i, y_i) by a linear function of the form $\hat{f}(x) := wx + b$ is parametric, meaxning that the sought linear function depends only on certain parameters w, b that need to be determined. Once w and b are known, $wx + b$ can be easily computed for any input x.
However, if data pairs do not follow the model $y_i = wx_i + b + \epsilon_i$, but more generally $y_i = f(x_i) + \epsilon_i$ for $i = 1, \ldots, N$ with a nonlinear function f, linear regression is not a good model

Polynomial Regression

One possible solution is polynomial regression, meaning one chooses the basis function

$$\hat{f}(x) := \sum_{j=0}^{p} \beta_j x^j$$

(where p is the polynomial degree)
and tries to determine β_j. x_j are the polynomial terms(e.g. $x^0 = 1, x^1 = x, x^2$ etc) The corresponding least squares problem still has the form

$$\min_{\beta \in \mathbb{R}^{p+1}} \frac{1}{2} ||X\beta - y||^2$$

(p+1 because "+1" comes from the **constant term** (β_0) which corresponds to x^0. Even if the polynomial degree is p, we still need to include the constant term, making the total number of coefficients $p + 1$)
with a matrix X of the form

$$X = \begin{pmatrix} 1 & x_1 & \ldots & x_1^p \\ \vdots & & \vdots & \\ 1 & x_N & & x_N^p \end{pmatrix}$$

Each row corresponds to a data points (x_i, y_i) and each column corresponds to a polynomial term x^j. Since there are $p + 1$ polynomial terms $(x^0, x^1, \ldots, x^p)$, the matrix X has $p + 1$ columns
The matrix X is known as a Vandermonde matrix and has the following properties:

1. If $N = p + 1$ (number of data points equals number of coefficients), then X is invertible if and only if all x_i are different
2. If $N > p + 1$ (more data points than coefficients), then $X^T X$ is invertible if and only if there are p different x_i
3. If $N < p + 1$ (fewer data points than coefficients), then $X^T X$ is not invertible

Mathematically, polynomial regression is thus very similar to linear regression. Disadvantages of polynomial regression are that it tends to suffer from overfitting, meaning that if the polynomial degree p is too high, noise in the data is exactly represented by the polynomial. Additionally, generalization to inputs $x_i \in \mathbb{R}^d$ is non-trivial

Next: 2.1.4.1 Neural Networks

2.1.4.1 Neural Networks

The idea of neural networks is to generate nonlinearity through a so-called *activation function* $\sigma : \mathbb{R} \to \mathbb{R}$. Popular activation functions include $\mathrm{ReLU} \sigma(t) = \max(t, 0), \mathrm{Sigmoid}\ \sigma(t) = \frac{1}{1+e^{-t}}$ (squashes values to the range $[0, 1]$ which is useful for probabilities), the Heaviside function $\sigma(t) = 1_{t>0}$, or the hyperbolic tangent $\sigma(t) = \tanh(t)$ (squashes values to $[-1, 1]$, similar to sigmoid but centered at 0).
Now consider the basis function $x \mapsto \sigma(\langle w, x \rangle + b)$, which is also called a neuron. (takes an input x applies a weight w, adds bias b and passes the result through an activation function σ). This function is nonlinear but corresponds only to a shift and stretch of the activation function. (The dot product $\langle w, x \rangle + b$ shifts and stretches the input linearly). The activation function σ then bends this linear combination into a **nonlinear** output. To obtain a better basis function, we form a linear combination of n of these basis functions or neurons:

2.1.21

$$\hat{f}(x) := \sum_{j=1}^{n} a_j \sigma(\langle w_j, x \rangle + b_j)$$

Here:

- n: Number of neurons (the **width** of the network)
- a_j: Coefficient for the j-th neurons output
- w_j, b_j: Weight and bias for the j-th neuron

Key terminology

1. **Neuron**: The function $x \mapsto \sigma(\langle w_j, x \rangle + b_j)$
2. **Width** (n): The number of neurons in the hidden layer
3. **Weights** (w_j): Parameters controlling how inputs are combined
4. **Biases** (b_j): Parameters shifting the activation threshold
5. **Layers**:
   1. **Input layer**: Raw data x
   2. **Hidden layer**: Neurons ($j = 1, \ldots, n$) processing inputs
   3. **Output layer**: The final prediction $\hat{f}(x)$
   Each neuron contributes d weights ($w_j \in \mathbb{R}^d$ for d-dimensional input), 1 bias (b_j) and 1 coefficient (a_j).
   The function $\hat{f}$ now depends on $(2 + d) \cdot n$ many parameters that need to be determined.

Example

For $n = 10$ neurons and $d = 3$ inputs, parameters = $(2 + 3) \cdot 10 = 50$

Back to the script

Equation 2.1.21 is referred to as a neural network with two layers (or also with one hidden layer). The following terminology is very common:

1. The function $x \mapsto \sigma(\langle w_j, x \rangle + b_j)$ is referred to as the j-th neuron
2. $n \in \mathbb{N}$ is the number of neurons or the width of the neural network
3. $\{w_j\}_{j=1,\dots,n}$ are referred to as weights, and $\{b_j\}_{j=1,\dots,n}$ are referred to as bias terms or biases

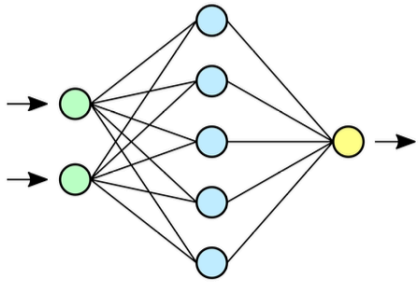


Figure 2.5: Illustration of a neural network with one hidden layer.

Universal Approximation

A special feature of neural networks is that they have to so-called universal approximation property. It means that it can approximate any continuous function on compact subset of $\mathbb{R}^d$.

Theorem 2.1.3

Let the activation function $\sigma : \mathbb{R} \to \mathbb{R}$ be piecewise continuous and not a polynomial, and let $f : [0, 1]^d \to \mathbb{R}$ be any continuous function. Then for every $\epsilon > 0$, there exists $n \in \mathbb{N}$ along with weights $\{w_j\}_{j=1,\dots,n}$ and biases $\{b_j\}_{j=1,\dots,n}$ such that for the corresponding neural network of the form 2.1.21, it holds that

$$\max_{x \in [0,1]^d} |f(x) - \hat{f}(x)| < \epsilon$$

for any $\epsilon > 0$, there exists a neural network $\hat{f}$ such that the network $\hat{f}$ can approximate f **uniformly** over $[0, 1]^d$ with arbitrary small error

The proof is relatively simple but requires some profound theorems from functional analysis. It is clear that for polynomial activation functions σ, the theorem cannot hold. If σ is a polynomial of degree p, then the same holds for $\hat{f}$ regardless of n, w_i or b.
An intuition for why the approximation property holds is the following argument, which we

formulate for simplicity in the case $d = 1$: Every continuous function f can partition $[0, 1]$ into $n \in \mathbb{N}$ intervals of the form $I_j := [\frac{j-1}{n}, \frac{j}{n}]$ for $j = 1, \dots, n$. We define the step function $f_n : [0, 1] \to \mathbb{R}$ by

$$f_n(x) := f\left(\frac{j}{n}\right)$$

where $j \in \{1, \dots, n\}$ is chosen such that $x \in I_j$. Now let $\epsilon > 0$ be given. Since f is continuous on $[0, 1]$ and therefore uniformly continuous, there exists $n \in \mathbb{N}$ such that for all $x, y \in [0, 1]$ with $|x - y| \leq \frac{1}{n}$, it holds that $|f(x) - f(y)| < \epsilon$. Let $x \in [0, 1]$ be arbitrary and let $j \in \{1, \dots, n\}$ be such that $x \in I_j$. Then it follows that

$$|f(x) - f_n(x)| = |f(x) - f\left(\frac{j}{n}\right)| < \epsilon$$

Since x in $[0, 1]$ was arbitrary, we obtain

$$\sup_{x \in [0,1]} |f(x) - f_n(x)| < \epsilon$$

It remains to show that each step function can be approximated by neural networks of the form 2.1.21. Here, we also restrict ourselves to a special case and consider the Heaviside activation $\sigma(t) = 1_{t>0}$. It suffices to show that such neural networks can approximate a step function. Let $f(x) = 1_{s<x\leq t}$ for $s < t$ be a step function. Then choose
$n = 2, a_1 = 1, w_1 = 1, b_1 = -s, a_2 = -1, w_2 = 1, b_2 = -t$ and obtain according to 2.1.21

$$\sum_{j=1}^{2} a_j \sigma(\langle w_j, x \rangle + b_j)$$

$$= 1\sigma(\langle 1, x \rangle - s) + (-1\sigma(\langle 1, x \rangle - t))$$

$$= \sigma(x - s) - \sigma(x - t)$$

Its the same as written below (copied from script)

$$\hat{f}(x) = \sigma(x - s) - \sigma(x - t) = 1_{s<x\leq t} = f(x)$$

Training Neural Networks

As with linear regression, we can use the least squares method to determine the parameters of $\hat{f}$. For this, we consider the problem

2.1.22

$$\min_{\substack{a \in \mathbb{R}^n, \\ W \in \mathbb{R}^{d \times n} \\ b \in \mathbb{R}^n}} \left\{ \mathcal{L}(a, W, b) := \frac{1}{2N} \sum_{i=1}^{N} |a^T \sigma(W^T x_i + b) - y_i|^2 \right\}$$

where we abbreviate $a = (a_1, \dots, a_n), W = (w_1, \dots, w_n)$ and $b = (b_1, \dots, b_n)$ and we used the fact that

$$\sum_{j=1}^{n} a_j \sigma(\langle w_j, x \rangle + b_j) = a^T \sigma(W^T x + b)$$

To determine the parameters a, W, and b, we use a gradient descent method.
Let $L : \mathbb{R}^P \to \mathbb{R}$ be a continuously differentiable function. Then we denote

$$\begin{cases} \text{Initialize } x^0 \in \mathbb{R}^P \\ x^k := x^{k-1} - \tau \nabla L(x^{k-1}), \quad k \in \mathbb{N} \end{cases}$$

This is the gradient descent method for L with step size $\tau > 0$. We want to apply this function $\mathcal{L}(a, W, b)$ and compute the gradients with respect to the three groups of variables.
To do this, we define the residual

$$\text{res}(x_i) := a^T \sigma(W^T x_i + b) - y_i$$

and we obtain

$$\nabla_a \mathcal{L}(a, W, b) = \frac{1}{N} \sum_{i=1}^{N} [\text{res}(x_i) \sigma(W^T x_i + b)] \in \mathbb{R}^n$$

$$\nabla_W \mathcal{L}(a, W, b) = \frac{1}{N} \sum_{i=1}^{N} [\text{res}(x_i) a \odot \sigma'(W^T x + b) x_i^T]^T \in \mathbb{R}^{d \times n}$$

$$\nabla_b \mathcal{L}(a, W, b) = \frac{1}{N} \sum_{i=1}^{N} [\text{res}(x_i) a \odot \sigma'(W^T x + b)] \in \mathbb{R}^n$$

where $\odot$ defines the Hadamard-Product (element-wise product)

Definition (Hadamard product)

For two matrices A and B of the same dimension $m \times n$, the Hadamard product $A \odot B$ is a matrix of the same dimension as the operands, with elements given by

$$(A \odot B)_{ij} = (A)_{ij} (B)_{ij}$$

For matrices of different dimensions the Hadamard product is undefined.
For example, the Hadamard product for two arbitrary 2×3 matrices is:

$$\begin{bmatrix} 2 & 3 & 1 \\ 0 & 8 & -2 \end{bmatrix} \odot \begin{bmatrix} 3 & 1 & 4 \\ 7 & 9 & 5 \end{bmatrix} = \begin{bmatrix} 2 \cdot 3 & 3 \cdot 1 & 1 \cdot 4 \\ 0 \cdot 7 & 8 \cdot 9 & -2 \cdot 5 \end{bmatrix} = \begin{bmatrix} 6 & 3 & 4 \\ 0 & 72 & -10 \end{bmatrix}$$

Properties:

- $A \odot B = B \odot A$

- $A \odot (B \odot C) = (A \odot B) \odot C$
- $A \odot (B + C) = A \odot B + A \odot C$
- $(kA) \odot B = A \odot (kB) = k(A \odot B)$
- $A \odot 0 = 0$

Explanation how the gradients are derived

Remember the Loss function is the MSE:

$$\mathcal{L}(a, W, b) = \frac{1}{2N} \sum_{i=1}^{N} |a^T \sigma(W^T x_i + b) - y_i|^2$$

Explanation for gradient w.r.t. a

Remember the chain rule and set $f(x) = x^2$ and $g(x) = a^T \sigma(W^T x_i + b) - y_i$
By applying $f'(g(x)) \cdot g'(x)$
Focus on single weight a_j (the j-th component of a):

$$\frac{\partial \mathcal{L}}{\partial a_j} = \frac{1}{N} \sum_{i=1}^{N} (a^T \sigma(W^T x_i + b) - y_i) \cdot \sigma(\langle w_j, x_i \rangle + b_j)$$

where $(a^T \sigma(W^T x_i + b) - y_i)$ is the residual (prediction error for x_i)
Alternatively we could do it like this:

$$\mathcal{L}(a, W, b) = \frac{1}{2N} \sum_{i=1}^{N} \text{res}(x_i)^2$$

Now compute $\nabla_a \mathcal{L}$ by differentiating $\mathcal{L}$ with respect to each component a_j of a
Applying the **chain rule**
For each term $\text{res}(x_i)^2$:

$$\frac{\partial}{\partial a_j} \text{res}(x_i)^2 = 2 \cdot \text{res}(x_i) \cdot \frac{\partial}{\partial a_j} \text{res}(x_i)$$

The derivative of $\text{res}(x_i)$ with respect to a_j is:

$$\frac{\partial}{\partial a_j} [a^T \sigma(W^T x_i + b) - y_i] = \sigma(w_j^T x_i + b_j)$$

Here, w_j is the j-th column of W, and $\sigma(W^T x_i + b)$ evaluated element-wise
Now combining those components we get:

$$\nabla_a \mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \text{res}(x_i) \cdot \sigma(W^T x_i + b)$$

Explanation for gradient w.r.t. W

Remember,

$$\mathcal{L}(a, W, b) = \frac{1}{2N} \sum_{i=1}^{N} \text{res}(x_i)^2, \quad \text{res}(x_i) = a^T \sigma(W^T x_i + b) - y_i$$

Expanding $\text{res}(x_i)$ using the columns w_j of W we get:

$$\text{res}(x_i) = \sum_{j=1}^{n} a_j \sigma(w_j^T x_i + b_j) - y_i$$

Now computing the gradient for the derivative of $\mathcal{L}$ w.r.t one column w_j

$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{1}{N} \sum_{i=1}^{N} \text{res}(x_i) \cdot \frac{\partial}{\partial w_j} (a^T \sigma(W^T x_i + b) - y_i)$$

Now focus on the last term where we have to compute the gradient for the residual:

$$\frac{\partial}{\partial w_j} (a^T \sigma(W^T x_i + b) - y_i) = \frac{\partial}{\partial w_j} (a^T \sigma(W^T x_i + b)) - \frac{\partial}{\partial w_j} y_i$$

Notice that y_i is a constant and does not depend on w_j thus it disappears
So we have to focus on

$$\frac{\partial}{\partial w_j} (a^T \sigma(W^T x_i + b)) = \frac{\partial}{\partial w_j} (a_j \sigma(w_j^T x_i + b_j)) = a_j \sigma'(w_j^T x_i + b_j) \cdot \frac{\partial}{\partial w_j} (w_j^T x_i + b_j)$$

The derivative of

$$\frac{\partial}{\partial w_j} (w_j^T x_i + b_j) = x_i$$

since w_j is a vector and $w_j^T x_i$ is linear in w_j
Combining the results we get:

$$\frac{\partial}{\partial w_j} (a^T \sigma(W^T x_i + b)) = a_j \sigma'(w_j^T x_i + b_j) x_i$$

The **almost finished gradient** $\nabla_W \mathcal{L}$ aggregates derivatives for all columns $w_1, \ldots, w_n$. For each data point x_i:

$$\nabla_W \mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \text{res}(x_) \cdot [a \odot \sigma'(W^T x_i + b)] x_i^T$$

where $[a \odot \sigma'(W^T x_i + b)]$ is a vector with components $a_j \sigma'(w_j^T x_i + b_j)$
The outer product $x_i^T \in \mathbb{R}^{1 \times d}$ ensures the gradient $\nabla_W \mathcal{L}$ matches W's dimensions.
$[a \odot \sigma'(W^T x_i + b)] \in \mathbb{R}^{n \times 1}$. BUT this doesn't match the dimensions of W ($d \times n$) for now, the

gradient gives us a $n \times d$ matrix because of the dimensions described above. Because of this we have to transpose everything, so the **FULL** gradient is:

$$\nabla_W \mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} [\text{res}(x_i) a \odot \sigma'(W^T x + b)] x_i^T]^T, \quad \mathbb{R}^{d \times n}$$

Explanation for gradient w.r.t. b

Remember

$$\mathcal{L}(a, W, b) = \frac{1}{2N} \sum_{i=1}^{N} \text{res}(x_i)^2, \quad \text{res}(x_i) = a^T \sigma(W^T x_i + b) - y_i$$

$$\frac{\partial}{\partial b} = \frac{1}{N} \sum_{i=1}^{N} \text{res}(x_i) \cdot \frac{\partial}{\partial b} (a^T \sigma(W^T x_i + b) - y_i)$$

Compute the derivative of the activation function using the chain rule

$$\frac{\partial}{\partial b} \sigma(W^T x_i + b) = \sigma'(W^T x_i + b)$$

The gradient of $\frac{\partial}{\partial x}(a^T \sigma(x)) = a \odot \sigma'(x)$, since this is out outer function. Remember the chain rule $f'(g(x)) \cdot g'(x)$
The $\odot$ symbol represents the Hadamard product (element-wise multiplication) between two vectors. The reason the Hadamard is taken is that when you differentiate the scalar output $a^T \sigma(W^T x_i + b)$ with respect to b is essentially a vector where each element corresponds to the gradient of the respective output dimension. The Hadamard product allows you to combine this vector with the residual, where element-wise multiplication is necessary to correctly propagate the gradient.

The final gradient is thus:

$$= \frac{1}{N} \sum_{i=1}^{N} \text{res}(x_i) \cdot a \odot \sigma'(W^T x_i + b)] \in \mathbb{R}^n$$

Now back to the script

Thus, the gradient descent procedure in this case is given by

$$\begin{cases} \text{Initialize } a \in \mathbb{R}^n, W \in \mathbb{R}^{d \times n}, b \in \mathbb{R}^n \\ \text{For } k \in \mathbb{N} \text{ iterate :} \\ \\ \quad a^k := a^{k-1} - \tau \nabla_a \mathcal{L}(a^{k-1}, W^{k-1}, b^{k-1}) \\ W^k := W^{k-1} - \tau \nabla_W \mathcal{L}(a^{k-1}, W^{k-1}, b^{k-1}) \\ \quad b^k := b^{k-1} - \tau \nabla_b \mathcal{L}(a^{k-1}, W^{k-1}, b^{k-1}) \end{cases}$$

a, W and b start with random values (e.g. sampled from a normal distribution)

Since the number of data points $N \in \mathbb{N}$ is typically very large, a stochastic gradient method with batch size $B << N$ is usually employed:

$$\begin{cases} \text{Initialize } a \in \mathbb{R}^n, W \in \mathbb{R}^{d \times n}, b \in \mathbb{R}^n \\ \text{For } k \in \mathbb{N} \text{ iterate :} \\ \text{Choose a random } B\text{-element subset } \mathcal{I} \text{ from } \{1, \dots, N\} \\ \\ a^k := a^{k-1} - \tau \hat{\nabla}_a \mathcal{L}(a^{k-1}, W^{k-1}, b^{k-1}) \\ W^k := W^{k-1} - \tau \hat{\nabla}_W \mathcal{L}(a^{k-1}, W^{k-1}, b^{k-1}) \\ b^k := b^{k-1} - \tau \hat{\nabla}_b \mathcal{L}(a^{k-1}, W^{k-1}, b^{k-1}) \end{cases}$$

where

$$\hat{\nabla}_a \mathcal{L}(a, W, b) := \frac{1}{B} \sum_{i \in \mathcal{I}} [\text{res}(x_i) \sigma(W^T x_i + b)]$$

and analogously for the W and b variables. (just change N with the batch size B)

$$\hat{\nabla}_W \mathcal{L}(a, W, b) = \frac{1}{B} \sum_{i=1}^{N} [\text{res}(x_i) a \odot \sigma'(W^T x + b) x_i^T]^T$$

$$\hat{\nabla}_b \mathcal{L}(a, W, b) = \frac{1}{B} \sum_{i=1}^{N} [\text{res}(x_i) a \odot \sigma'(W^T x + b)]$$

The random subset $\mathcal{I}$ changes in each iteration
Although neural networks of the form 2.1.21 have the universal approximation property, deep neural networks are often used in practice. These are obtained by nesting individual layers as shown in 2.1.21
To this end, we choose a depth $L \in \mathbb{N}$, activation functions $\sigma_l : \mathbb{R} \to \mathbb{R}$ (activation function for layer l, bias terms $b_l \in \mathbb{R}^{n_l}$ and weight matrices $W^{d_l \times n_l}$ (weight matrix or layer l). Here $d_l \in \mathbb{N}$ is the input dimension and $n_l \in \mathbb{N}$ is the number of neurons in the l-th layer. We define the l-th layer of the neural network as

$$\Phi_l(x) := \sigma_l(W_l^T x + b_l), \quad x \in \mathbb{R}^{d_l}$$

and define a neural network as

$$\hat{f}(x) := \Phi_L \circ \Phi_{L-1} \circ \cdots \circ \Phi_1(x), \quad x \in \mathbb{R}^d$$

where we set $d_1 := d$ (the first layer takes the raw input $x \in \mathbb{R}^d$ and also assume that $d_{l+1} = n_l$ (the output of layer l becomes the input to layer $l+1$) for $l \in \{1, \dots, L-1\}$. The output dimension n_L is equal to 1 for the scalar-valued neural networks but can also be greater than 1 in the case of multidimensional regression.
Example for $L = 3$:

$$\hat{f}(x) = \sigma_3(W_3^T\sigma_2(W_2^T\sigma(W_1^Tx + b_1) + b_2) + b_3)$$

Typically, the activation functions $\sigma_1, \dots, \sigma_{L-1}$ are chosen to be the same (e.g. ReLU or $\mathrm{Sigmoid}$). However, for σ_L, it is usually chosen as $\sigma_L(x) = x$ to preserve the universal approximation property of the inner layers
We now consider a generalization of the least squares problem 2.1.22 by using more general loss functions

2.1.24

$$\min_{\substack{W_l \in \mathbb{R}^{d_l \times n_l} \\ b_l \in \mathbb{R}^{n_l} \\ l=1,\dots,L}} \left\{ \mathcal{L}(a, W, b) := \frac{1}{N} \sum_{i=1}^{N} \ell(\hat{f}(x_i), y_i) \right\}$$

where $\ell : \mathbb{R}^{n_L \times n_L}$ denotes a loss function (e.g. $\ell(\tilde{y}, y) = \frac{1}{2}|\tilde{y} - y|^2$), and we suppress in our notation that $\hat{f}$ depends on the sought parameters W_l, b_l (all weight matrices and biases across layers)

Backpropagation

$$\frac{d\ell(\hat{f}(x), y)}{d(W_l)_{ij}} = (z_{l-1})_i(\delta_l)_j$$

$$\frac{d\ell(\hat{f}(x), y)}{d(b_l)_{ij}} = (\delta_l)_j$$

Where z_{l-1} is the output from the previous layer!
Backpropagation computes gradients of the loss $\ell(\hat{f}(x), y)$ with respect to weights W_l and biases b_l. It works by:

- Forward Pass: Compute predictions $\hat{f}(x)$ and intermediate layer outputs z_l
- Backward Pass: Propagate the "error" δ_l backward through the network to compute the gradients

$\delta_l \in \mathbb{R}^n_l$: The "error" at layer l, defined as:

$$\delta_l = \frac{\delta_l}{\delta a_l}$$

This quantifies how sensitive the loss is to changes in the pre-activation a_l
Now on to explaining $\frac{d\ell(\hat{f}(x), y)}{d(W_l)_{ij}} = (z_{l-1})_i(\delta_l)_j$

- $(z_{l-1})_i$: Activation from neuron i in layer $l-1$
- $(\delta_l)_j$: Error at neuron j in layer l
- **Why?**

- The weight $(W_l)_{ij}$ connects neuron i in layer $l-1$ to neuron j in layer l. The gradient depends on:
  - How much neuron i in layer $l-1$ was activated (z_{l-1})
  - How much error is attributed to neuron j in layer l (δ_l)

Now on to explaining $\frac{d\ell(\hat{f}(x),y)}{d(b_l)_{ij}} = (\delta_l)_j$

- This is the error at neuron j in layer l
- **Why?**
  - The bias $(b_l)_j$ directly offsets the pre-activation a_l. Its gradient is purely the error at neuron j

2.2 Classification

In this chapter, we address classification problems. These differ from regression problems mainly in that the output data is **qualitative** rather than quantitative. Typically, each output is referred to as a label. In the case of two labels, which are usually modeled by $\mathcal{Y} = \{0, 1\}$ or $\mathcal{Y} = \{-1, 1\}$, we speak of **binary classification**.
We consider training data $(x_i, y_i)_{i=1}^N \subset \mathcal{X} \times \mathcal{Y}$, where $\mathcal{Y} = \{l_1, \dots, l_C\}$ denotes the set of all possible labels. The abstract goal is to find disjoint sets $\mathcal{M}_1, \dots, \mathcal{M}_C \subset \mathcal{X}$ with $\mathcal{X} = \bigcup_{c=1}^C \mathcal{M}_c$ ($\mathcal{X}$ is the input set, $\mathcal{M}_c$ should all be disjoint), such that most training points labeled l_c lie in the set $\mathcal{M}_c$. If we then have a new data point $x \in \mathcal{X}$, we determine the unique set $\mathcal{M}_c$ with $x \in \mathcal{M}_c$ and assign the label l_c to x.
We typically consider the case of binary classification with $C = 2$. The general case with $C > 2$ classes can be solved by one of the following approaches:

1. (one-vs one) We compute $\binom{C}{2} = \frac{C!}{2!(C-2)!} = \frac{C(C-1)}{2}$ binary classifiers that compare all pairs of classes. To classify a new data point, we choose the class that is most frequently assigned by these classifiers
2. (one-vs-all) We compute C classifiers that classify the points of a fixed class $c \in \{1, \dots, C\}$ against all other points (the rest). We also assume that this gives us a probability for belonging to the c-th class. To classify a new data point, we choose the class with the highest probability value

2.2.1 Linear Classification

In linear classification, we start with the binary case and make the assumption that the sought (to search) sets are half-spaces in $\mathbb{R}^d$ separated by a hyperplane of the form $\{x \in \mathbb{R}^d | \langle w, x \rangle + b\}$.

We consider the following function $\hat{f} : \mathbb{R}^d \to \{-1, 1\}$, which assigns a label $\hat{f}(x) \in \{-1, 1\}$ to a data point $x \in \mathbb{R}^d$:

2.2.1

$$\hat{f}(x) := \begin{cases} 1 & \text{if } \langle w, x \rangle + b \geq 0 \\ -1 & \text{if } \langle w, x \rangle + b < 0 \end{cases}$$

As with regression problems, we now need to determine the parameters $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$ so that $\hat{f}(x_i) \approx y_i$ for all data points $i = 1, \ldots, N$.

At this point, we emphasize that $\hat{f}(x) = \operatorname{sign}(\langle w, x \rangle + b)$ and thus the activation function $\hat{f}$ is a neural network with one neuron and an activation function $\sigma = \operatorname{sign}$, where

2.2.2

$$\operatorname{sign}(t) := \begin{cases} 1 & \text{if } t \geq 0, \\ -1 & \text{if } t < 0 \end{cases}$$

is the sign function with the convention $\operatorname{sign}(0) = 1$

2.2.1.1 Rosenblatt's perceptron

Rosenblatt's perceptron

The function $\hat{f}$ in 2.2.1 is also called the Perceptron and dates back to Warren McCulloch and Walter Pitts in 1943. Due to the discontinuity of $\hat{f}$, gradient methods for determining w and b based on a least squares method are not possible. An alternative method was proposed by Frank Rosenblatt in 1958. For this purpose, the perceptron loss is defined as

2.2.3

$$\mathcal{L}(w,b) := \frac{1}{N}\sum_{i=1} \max(0, -y_i(\langle w, x_i\rangle + b))$$

Intuition

- For correctly classified points ($y_i(\langle w, x_i\rangle) \geq 0$), the loss is 0
- For misclassified points ($y_i(\langle w, x_i\rangle) + b) < 0$) the loss is $-y_i(\langle w, x_i\rangle + b)$, which is proportional to the (unnormalized) distance from x_i to the hyperplane. Points farther from the hyperplane incur large penalties

Back to the script

Note that in the case where $\langle w, x_i\rangle + b$ and y_i have the same sign, the i-th term in this sum is equal to zero. If the signs are different, the corresponding summand is equal to $\langle w, x_i\rangle + b$ and thus proportional to the distance from x_i to the hyperplane.

Explanation:

If y_i is negative (e.g. -1) and $\langle w, x_i\rangle + b$ is also negative (e.g. -2) then:

$$-y_i(\langle w, x_i\rangle + b) = -(-1)\cdot(-2) = -2$$

and thus the max is 0!
Now if the signs are different e.g. y_i is 1 and $\langle w, x_i\rangle + b$ is still -2 then:

$$-y_i(\langle w, x_i\rangle + b) = -(1)\cdot(-2) = 2$$

(which is equal to $\langle w, x_i\rangle + b$ and thus proportial to the distance from x_i to the hyperplane)

Back to the script

Therefore, misclassified points x_i that are further away from the hyperplane are penalized more heavily than those that are close.
We note that 2.2.3 is a convex function in the sought variables w and b. Unfortunately is is not differentiable in w and b at $y_i(\langle w, x_i\rangle + b) = 0$.
However, we still calculate the derivative of the function

$$f_i(w, b) := \max(0, -y_i(\langle w, x_i\rangle) + b)$$

with respect to the two variables $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$ and ignore the fact that the maximum function is not differentiable. At such points, we set the derivative to the value 0. The resulting derivative is called a *subgradient* of f_i, and is given by

$$\partial_w f_i(w, b) := -y_i x_i 1_{y_i(\langle w, x_i\rangle + b) < 0}$$

$$\partial_b f_i(w, b) := -y_i 1_{y_i(\langle w, x_i\rangle + b) < 0}$$

How are those computed?

Explanation

Case Analysis for Gradients

The loss has two cases depending on whether the point is **correctly classified** or **misclassified**
Case 1: correctly classified:
If $y_i(\langle w, x_i\rangle + b) \geq 0$

- The term inside $\max(0, -y_i(\langle w, x_i\rangle + b))$ is ≤ 0 (because of the negative sign in front of the y_i), so $f_i(w, b) = 0$
- **Gradients**: Since the loss is a constant (0), the gradients are

$$\partial_w f_i(w, b) = 0, \quad \partial_b f_i(w, b) = 0$$

Case 2: Misclassified
If $y_i(\langle w, x_i\rangle + b) < 0$:

- The term inside $\max(0, -y_i(\langle w, x_i\rangle + b))$ is > 0 (because of the negative sign in front of the y_i), so $f_i(w, b) = -y_i(\langle w, x_i\rangle + b)$
- **Gradients**: The loss is linear in w and b, so we differentiate
  - With respect to w

$$\partial_w f_i(w, b) = \partial_w(-y_i(\langle w, x_i\rangle + b)) = -y_i x_i$$

  - With respect to b

$$\partial_b f_i(w, b) = \partial_b(-y_i(\langle w, x_i\rangle) + b) = -y_i$$

  Now combining the cases and using the indicator function 1:

$$\partial_w f_i(w,b) = -y_i x_i \cdot 1_{y_i(\langle w, x_i\rangle + b) < 0}$$

$$\partial_w f_i(w,b) = -y_i \cdot 1_{y_i(\langle w, x_i\rangle + b) < 0}$$

Back to the script

Rosenblatt's algorithm is now a stochastic subgradient method with step size $\tau > 0$ and batch size $B \in \{1, \dots, N\}$ applied to $L(w,b)$:

$$\begin{cases} \text{Initialize } w_0 \in \mathbb{R}^d, b_0 \in \mathbb{R} \\ \text{For } k \in \mathbb{N} \text{ iterate:} \\ \text{Choose a random B-element subset } \mathcal{I} \subset \{1, \dots, N\}, \\ \text{Define } \mathcal{M} := \{i \in \mathcal{I} : y_i(\langle w, x_i\rangle + b) < 0\} (\text{misclassified points}), \\ w_k := w_{k-1} + \frac{\tau}{B} \sum_{i \in \mathcal{M}} y_i x_i, \\ b_k := b_{k-1} + \frac{\tau}{B} \sum_{i \in \mathcal{M}} y_i \end{cases}$$

The main problem with Rosenblatts algorithm is that we have no control over which hyperplane it converges to. In the case that the data are linearly separable, i.e. there exist $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$ such that $y_i(\langle w, x_i\rangle + b) \geq 0$ for all $i = 1, \dots, N$, every hyperplane of this form is a fixed point of the algorithm (the algorithm stops updating once all points are correctly classified!)

2.2.1.2 Support Vector Machines

The question now is whether and how one can select an optimal hyperplane. Our goal will be to find a hyperplane that has the maximum possible distance from the nearest points of the two classes. For this, we first derive an expression for the distance of a point to a hyperplane.

Proposition 2.2.1

Let $x_0 \in \mathbb{R}^d$ and $\mathcal{H} := \{x \in \mathbb{R}^d : \langle w, x\rangle + b = 0\}$ ($w \in \mathbb{R}^d$ is the normal vector (perpendicular to the hyperplane), $b \in \mathbb{R}$ is the bias term (offset from the margin)): Then it holds that

$$\operatorname{dist}(x_0, \mathcal{H}) := \inf\{||x_0 - x|| : x \in \mathcal{H}\} = \frac{|\langle w, x_0\rangle + b|}{||w||}$$

(The shortest distance from a point x_0 to a hyperplane is the length of the perpendicular line from x_0 to $\mathcal{H}$ $\frac{|\langle w, x_0\rangle + b|}{||w||}$. This arises from projecting the vector $x_0 - x$ (where $x \in \mathcal{H}$) onto the normal vector w
Proof. The proof will be provided in the exercise

We assume that the data are linearly separable and now consider the following optimization problem

2.2.5

$$\max\left\{M : w \in \mathbb{R}^d, b \in \mathbb{R}, y_i(\langle w, x_i\rangle + b \geq M||w||, \forall i = 1, \ldots, N\right\}$$

(The constraint ensures all points are classified correctly and are at least M away from the hyperplane).
The value M is the distance of the data points to the hyperplane that we want to maximize. The constraint $y_i(\langle w, x_i\rangle + b) \geq M||w||$ is, according to Proposition 2.2.1 equivalent to all points being correctly classified and having a distance of at least M from the hyperplane. Since we assume that the data are linearly separable, the constraints are at least satisfiable with $M = 0$. Since the constraints are scale-invariant, we can, without loss of generality, set $||w|| = \frac{1}{M}$ and obtain the problem

$$\max\left\{\frac{1}{||w||} : w \in \mathbb{R}^d, y_i(\langle w, x_i\rangle + b) \geq 1, \forall i = 1, \ldots, N\right\}$$

which is usually given in the following equivalent standard form (maximizing $\frac{1}{||w||}$, is equivalent to minimizing $||w||^2$)

2.2.6

$$\min\left\{\frac{1}{2}||w||^2 : w \in \mathbb{R}^d, b \in \mathbb{R}, y_i(\langle w, x_i\rangle + b) \geq 1, \forall i = 1, \dots, N\right\}$$

(objective function $\frac{1}{2}||w||^2$ minimizes the norm of w, which maximizes the margin, the constraints ensure that every data point is correctly classified and at least distance 1 from the hyperplane)
This is a quadratic optimization problem with linear constraints and can be solved using classical numerical optimization methods. A classifier of the form $\hat{f}(x) = \text{sign}(\langle w, x\rangle + b)$, where w and b are solutions to 2.2.6, is called a **support vector machine** (SVM). The reason for this is that the classifier essentially depends only on the points that are closest to the hyperplane $\langle w, x\rangle + b$ and thus satisfy $y_i(\langle w, x_i\rangle + b) = 1$ (points that lie **exactly on the margin**), the so-called support vectors (points nearest to the hyperplane). If the points **that are further away**, i.e. $y_i(\langle w, x_i\rangle + b) > 1$, are slightly altered, the SVM remains unchanged.
So far, we have assumed that the data are linearly separable, which ensures that the constraints in 2.2.5 or 2.2.6 are satisfiable.
In most realistic applications, this assumption does not hold (there is usually overlap between classes, making constraints like above impossible to satisfy for all points)!
Nevertheless, we would like to find a hyperplane in these situations that violates the constraints for as few points as possible. To do this, we introduce non-negative slack variables $\xi_i \geq 0$ for $i = 1, \dots, N$ which measure the deviation from $y_i(\langle w, x_i\rangle + b) \geq 1$ by replacing this constraint with $y_i(\langle w, x_i\rangle + b) \geq 1 - \xi_i$.

- If $\xi_i = 0$: The point is correctly classified outside the margin
- If $0 < \xi_i < 1$: The point is correctly classified inside the margin
- If $\xi_i \geq 1$: The point is misclassified
  Those slack variables are introduced one for each data point and relax the constraint to

At the same time, we want the slack variables to be as small as possible and therefore consider the following optimization problem

2.2.7

$$\min\left\{\frac{1}{N}\sum_{i=1}^{N}\xi_i + \frac{\lambda}{2}||w||^2 : w \in \mathbb{R}^d, b \in \mathbb{R}, \xi_i \in \mathbb{R},\right.$$

$$\left. y_i(\langle w, x_i\rangle + b) \geq 1 - \xi_i, \xi_i \geq 0, \forall i = 1, \dots, N\right\}$$

- the term $\frac{1}{N}\sum_{i=1}^{N}\xi_i$ penalizes constraint violations - minimizing this keeps most points correctly classified

- the term $\frac{\lambda}{2}||w||^2$ maintains a large margin - we still want to keep the margin as wide as possible
- The parameter λ controls trade-off between margin-size and misclassification allowance.
  - Large λ prioritizes a large margin, allowing fewer misclassifications
  - Small λ allows more misclassifications to better fit noisy data

where $\lambda > 0$ is a selectable parameter. Formally, we see that in the linearly separable case 2.2.7 reduces to 2.2.6 in the limit as $\lambda \to \infty$
In practice 2.2.7 has an equivalent formulation as an optimization problem without constraints.

To do this, we note that we can replace the constraint $y_i(\langle w, x_i \rangle + b) \geq 1 - \xi_i$ with an equality constraint. This follows from the following simple argument:
If the constraint is satisfied with a strict inequality at the optimum, we could reduce ξ_i to create an equality while also reducing the objective function.
Thus, we can perform the variable substitution $\xi_i := 1 - y_i(\langle w, x_i \rangle + b)$.
From the non-negativity condition, we finally obtain $\xi_i = \max(0, 1 - y_i(\langle w, x_i \rangle + b))$. This term is also called **Hinge Loss**. Substituting this into 2.2.7 yields the equivalent problem

2.2.8

$$\min_{\substack{w \in \mathbb{R}^d \\ b \in \mathbb{R}}} \left\{ \frac{1}{N} \sum_{i=1}^{N} \max(0, 1 - y_i(\langle w, x_i \rangle + b)) + \frac{\lambda}{2}||w||^2 \right\}$$

Note that this is a convex problem in the variables w and b we are looking for. Unfortunately, the objective function, similar to the Perceptron loss in 2.2.3, is not differentiable in w and b when $y_i(\langle w, x_i \rangle + b) = 1$. We again compute subgradients of the function

$$f_i(w, b) := \max(0, 1 - y_i(\langle w, x_i \rangle + b)) + \frac{\lambda}{2}||w||^2$$

with respect to the two variables $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$:

$$\partial_w f_i(w, b) := -y_i x_i 1_{y_i(\langle w, x_i \rangle + b) < 1} + \lambda w$$

$$\partial_b f_i(w, b) := -y_i 1_{y_i(\langle w, x_i \rangle + b) < 1}$$

The objective function in 2.2.8 is given by $\frac{1}{N} \sum_{i=1}^{N} f_i(w, b)$, and thus we can now perform a deterministic or stochastic subgradient method:

$$\begin{cases} w_{k+1} := w_k - \frac{\tau}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} \partial_w f_i(w_k, b_k) \\ b_{k+1} := b_k - \frac{\tau}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} \partial_b f_i(w_k, b_k) \end{cases}$$

Here, $\mathcal{I} = \{1, \ldots, N\}$ corresponds to the deterministic case, and in the stochastic case, $\mathcal{I}$ is a batch of size $B \leq N$
The natural question arises: how can SVMs be generalized for classification of data that are not

or poorly linearly separable? One conceptual approach for this is so-called kernel SVMs, which replace all inner products $\langle x, y \rangle$ for $x, y \in \mathbb{R}^d$ with values of a symmetric function $k(x, y)$. Addressing this is beyond the scope of this lecture.

In fact, this approach is equivalent to embedding the data points $x_i \in \mathbb{R}^d$ into a higher-dimensional space and thus classifying points $\phi(x_i) \in \mathbb{R}^D$. A possible embedding would be $\phi : \mathbb{R}^d \to \mathbb{R}^{2d}$ with

$$\phi(x) := (x_1, \dots, x_d, x_1^2, \dots, x_d^2)^T$$

Here, x_i denotes the i-th coordinate of $x \in \mathbb{R}^d$ and not the i-th data point $x_i \in \mathbb{R}^d$!
A hyperplane in $\mathbb{R}^{2d}$ then, in the original coordinates, is expressed as:

$$0 = b + \sum_{i=1}^{d} w_i x_i + \sum_{i=1}^{d} w_{d+1} x_i^2 = b + \langle w, x \rangle + x^T D x$$

with $w = (w_1, \dots, w_d)^T \in \mathbb{R}^d$ and $D := \operatorname{diag}(w_{d+1}, \dots, w_{2d}) \in \mathbb{R}^{d \times d}$.
This is a quadratic equation in $\mathbb{R}^d$ and the solutions are parabolas. In a similar way, arbitrarily complex classifiers can be generated in $\mathbb{R}^d$.
As another example, consider the embedding $\phi : \mathbb{R}^2 \to \mathbb{R}^3$ with $\phi(x) := (x_1, x_2, x_1^2 + x_2^2)$. The following hyperplane

$$0 = b + (x_1^2 + x_2^2)$$

with $b \leq 0$ is a circle equation in $\mathbb{R}^2$

2.2.2 Neural Networks (for Classification)

Of course, neural networks can also be used for classification. In fact, the Perceptron $\hat{f}(x) = \text{sign}(\langle w, x \rangle + b)$ was already a simple neural network
We again consider neural networks of the form

$$\hat{f}(x) = \Phi_L \circ \dots \Phi_1(x)$$

($\circ$ means function composition: $(f \circ g)(x) = f(g(x))$, which means applying g first, then applying f to the result)

- In the context of neural networks
  - Each $\Phi_l(x)$ represents a layer transformation
  - The function $\hat{f}(x)$ is a composition of these transformation, applied sequentially!
  - So, the output of layer 1 becomes the input to layer 2 and so on!
  with $\Phi_l(x) := \sigma_l(W_l^T x + b_l)$. For classification of $C \geq 2$ classes (**2 or more classes!**, data of the form $(x_i, y_i) \in \mathbb{R}^d \times \mathbb{R}^C$ is usually used, where we assume that the y_i are in so-called one-hot encoding, i.e., they are vectors of the form $(0, \dots, 0, 1, 0, \dots, 0) \in \mathbb{R}^C$ with a 1 in the c-th position, where $c \in \{1, \dots, C\}$ denotes the class. Choosing an appropriate loss function is very important for classification.
  One possible choice in $\ell(z, y) := \frac{1}{2}||z - y||^2$ for $y, z \in \mathbb{R}^C$. Much more common is the so-called cross-entropy. We assume the last activation function σ_L is the identity and that $n_L = C$. We then define the so-called Softmax operation

$$\text{softmax} : \mathbb{R}^C \to \mathbb{R}^C, \quad \text{softmax}(x)_c := \frac{\exp(x_c)}{\sum_{i=1}^C \exp(x_i)}$$

  Note that for all $x \in \mathbb{R}^C$, the vector $\text{softmax}(x) \in \mathbb{R}^C$ is a discrete probability distribution, since it holds that

$$0 \leq \text{softmax}(x)_c \leq 1 \forall c = 1, \dots, C, \quad \sum_{c=1}^C \text{softmax}(x)_c = 1$$

  Furthermore, we define the cross-entropy $H : \mathbb{R}^C \times \mathbb{R}^C \to \mathbb{R}$ as

$$H(x, y) := \begin{cases} -\sum_{c=1}^C y_c \log(x_c) & \text{if } x_c > 0 \ \forall c = 1, \dots, C, \\ \infty & \text{otherwise} \end{cases}$$

  x_c is the predicted probability for class c (from softmax!), y_c is 1 for the correct class and 0 otherwise.
  If y is a one-hot encoding of the c-th class, then $H(x, y)$ is small if x_c is close to or equal to 1. (remember: $\log(1) = 0$ and if y_c has 1 in the correct class position, CE is computed solely

by $-\log(x_c)$. Low loss --> good prediction)
E.g. if the correct class has high probability x_c then $\log(x_c)$ is close to 0, meaning low loss. If the correct class has low probability, $\log(x_c)$ is large and negative, meaning high loss!
We obtain a suitable loss function by computing the Softmax operation and the cross-entropy

$$\ell(z, y) := H(\text{softmax}(z), y) = -\sum_{c=1}^{C} y_c \log\left(\frac{\exp(z_c)}{\sum_{i=1}^{C} \exp(z_i)}\right)$$

To train a neural network with this loss function, we need the derivative with respect to z. This has a particularly simple form.

Proposition 2.2.2

Let $x, y \in \mathbb{R}^C$ and also $y_c \geq 0$ for all $c \in \{1, \ldots, C\}$ and $\sum_{c=1}^{C} y_c = 1$. Then for all $c \in \{1, \ldots, C\}$, it holds that

$$\frac{\partial \ell(z, y)}{\partial z_c} = \text{softmax}(z)_c - y_c$$

The gradient is the difference between predicted probability and actual label! This makes gradient descent simple: If the prediction is too high, the weight decreases, and if it's too low, the weight increases!
Proof. The proof is carried out in the exercise

2.2.3 Bayes and Naive Bayes Classification

Now we consider non-parametric models for classification, i.e. in contrast to the previous approaches, we do not try to determine the parameters of a separating hyperplane or a neural network. Instead, we aim to use the data to approximate their distribution.

To motivate this, consider the training dataset

Class:	Weight:	Color:
Apple	162g	red
Apple	186g	yellow
Banana	112g	yellow
Banana	142g	green
Banana	128g	yellow

Table 2.1: Fruit Dataset

We want to use this data to classify a new fruit, e.g. $x = (195g, \text{yellow})$. What do you think, is it more likely to be an apple or a banana?

Subtopics: 2.2.3.1 Bayes Classification

2.2.3.1 Bayes Classification

In an ideal world, we have for each class $c = 1, \dots, C$ the probability for x to belong to the c-th class.
For example if we are classifying fruits, K_1 could be "Banana", K_2 could be "Apple" etc
When we write $\mathbb{P}(K_c|x)$ we are asking: "Given that we observed x, what is the probability that it belongs to class c?"
If we denote the event of belonging to class c by K_c, we know the conditional probabilities $\mathbb{P}(x|K_c)$ (in English: *conditional distributions*), e.g. $\mathbb{P}(1003g, \text{red}|\text{Banana}) = 0\%$. Thus, we can use Bayes' theorem to obtain

$$\mathbb{P}(K_c|x) = \frac{\mathbb{P}(K_c)\mathbb{P}(x|K_c)}{\mathbb{P}(x)} = \frac{\mathbb{P}(K_c)\mathbb{P}(x|K_c)}{\sum_{i=1}^{C}\mathbb{P}(K_i)\mathbb{P}(x|K_i)}$$

Explanation

- $\mathbb{P}(K_c)$ is the **prior probability** of class c, i.e. how common this class is in general. It is denoted as π_c
- $\mathbb{P}(x|K_c)$ is the **likelihood**, meaning the probability of observing x given that it belongs to class c
- $\mathbb{P}(x)$ is the **total probability** of x over all possible classes, which is computed as $\sum_{i=1}^{C}\mathbb{P}(K_i)\mathbb{P}(x|K_i)$

The probability $\mathbb{P}(K_c)$ of the c-th class is usually abbreviated as π_c (in English: *class prior*), leading to

2.2.9

$$\mathbb{P}(K_c|x) = \frac{\pi_c\mathbb{P}(x|K_c)}{\sum_{i=1}^{C}\pi_c\mathbb{P}(x|K_i)}$$

Classification is then done by choosing the most probable class, i.e.

$$\arg\max_{c=1}^{C}\frac{\pi_c\mathbb{P}(x|K_c)}{\sum_{i=1}^{C}\pi_c\mathbb{P}(x|K_i)} = \arg\max_{c=1}^{C}\pi_c\mathbb{P}(x|K_c)$$

(Since the denominator is the same for all classes, we can just compute the numerator!)
This means:

- Compute $\pi_c\mathbb{P}(x|K_c)$ for each class
- Pick the class with the highest value!

Example: Classifying fruits

Imagine we want to classify a fruit x based on **weight** and **color**. Suppose we have

- Bananas (K_1): 90% of bananas are yellow and 10% are green
- Apples (K_2): 50% are red, 40% are green and 10% are yellow
- Cherries (K_3): 95% are red, 5% are yellow

If we observe $x = (1003g, red)$, the probability of observing that for each fruit is

- $\mathbb{P}(x|K_1)$ (prob of a 1003g red banana) is 0% --> bananas are never red
- $\mathbb{P}(x|K_2)$ is 50%
- $\mathbb{P}(x|K_3)$ is 95%
  If apples and cherries are equally common, we classify x as a **cherry** since it has the highest probability

Back to the script

The mapping $x \mapsto \arg\max_{c=1}^{C} \pi_c \mathbb{P}(x|K_c)$ is also called the Bayes classifier.
This means that given some feature x, we assign it to the class K_c that maximizes $\pi_c \mathbb{P}(x|K_c)$
Note that this is independent of the denominator in 2.2.9
Often, x is a continuous random variable, making it meaningless to work with $\mathbb{P}(x|K_c)$: If $x|K_c$ is, for example, normally distributed, then $\mathbb{P}(x|K_c) = 0$ for all $x \in \mathbb{R}^d$
In this case, however we can use Bayes' theorem for densities and obtain

2.2.10

$$\mathbb{P}(K_c|x) = \frac{\pi_c p(x|K_c)}{\sum_{i=1}^{C} \pi_c p(x|K_i)}$$

where $p(x|K_c)$ are the conditional **densities** ($p(x|K_c)$ now is a PDF of x given class K_c). Note that the left side still represents a probability and not a density since the possible classes form a discrete and finite probability space!
In practe, we unfortunately have neither π_i nor $\mathbb{P}(x|K_c)$ or $p(x|K_c)$ available.
Both quantities must be approximated using the data. To this end, one can use the class proportions:

$$\hat{\pi}_c := \frac{\#\{i = 1, \dots, N \mid y_i = l_c\}}{N}$$

This just means:

- l_c represents the label for the class
- $y_i = l_c$ means "the i-th sample is in class c"

- Count how many times class c appears in the training data
- Divide by the total number of samples N
  ... and is used regardless of discrete or continuous case (see below!)

Additionally, the conditional probabilities $\mathbb{P}(x|K_c)$ or densities $p(x|K_c)$ must be suitably approximated by $\hat{P}(x|K_c)$ or $\hat{p}(x|K_c)$.

- $\hat{P}(x|K_c)$ is an approximation of the probability $\mathbb{P}(x|K_c)$ when x is discrete (e.g. categorical features)
- $\hat{p}(x|K_c)$ is an approximation of the probability density function $p(x|K_c)$ when x is continuous (e.g. real-valued features)

We then obtain the approximate Bayes classifier:

$$x \mapsto \arg\max_{c=1}^{C} \hat{\pi}\hat{P}(x|K_c) \quad \text{or} \quad x \mapsto \arg\max_{c=1}^{C} \hat{\pi}_c\hat{p}(x|K_c)$$

This means:

1. Compute the estimated prior π_c+
2. Compute the estimated probability (depending on the case discrete or continuous)
3. Assign x to the class c that maximizes

Discrete Random Variables

If $x|K_c$ is a discrete random variable (e.g. colors red, green, yellow), one can approximate $\mathbb{P}(x|K_c)$ by proportions in the training data, e.g.

$$\hat{P}(\text{red}|K_c) := \frac{\#\{i|x_i = \text{red}, y_i = l_c\}}{\#\{i|y_i = l_c\}}$$

This means:

- **Numerator**: number of samples where x_i (the feature) is red **and** the label y_i is class c
- **Denominator**: total number of samples in class c

Example

Sample	Color (x_i)	Fruit Class (y_i)
1	Red	Apple
2	Green	Apple
3	Yellow	Banana
4	Red	Apple
5	Yellow	Banana
6	Red	Cherry
7	Green	Apple
8	Yellow	Banana
9	Red	Apple

There are 3 red apples and 4 apples in total so

$$\hat{P}(\text{red}|K_{\text{apple}}) = \frac{3}{4} = 0.75$$

This means that 75% of apples in our training data are red

Gaussian Mixture Model

For continuous variables, a common assumption is that $x|K_c \sim \mathcal{N}(\mu_c, \sigma_c^2)$ is normally distributed with means μ_c and variances σ_c^2 for $c = 1, \ldots, C$. For simplicity, we will only consider the one-dimensional case here.
We can approximate the densities $p(x|K_c)$ for $x \in \mathbb{R}$ by

$$\hat{p}(x|K_c) := \frac{1}{\sqrt{2\pi\hat{\sigma_c}^2}} \exp\left(-\frac{(x-\hat{\mu}_c)^2}{2\hat{\sigma_c}^2}\right)$$

for $c = 1, \ldots, C$ where

$$\mu_c = \frac{1}{\#\{i|y_i = l_c\}} \sum_{i|y_i=l_c} x_i \qquad \hat{\sigma}_c^2 := \frac{1}{\#\{i|y_i = l_c\}} \sum_{i|y_i=l_c} (x_i - \mu_c)^2$$

are the mean and discrete variance of the points with label l_c

- μ_c is just the average of all feature values x_i that belong to class c
  - The denominator is the number of training samples in class c
  - The numerator sums all x_i values for class c
- $\hat{\sigma_c}^2$ is the variance estimate, which calculates the **spread** of values around the meaen

- Each sample's difference from the mean is squared and summed, then divided by the number of samples of training points in class c

Kernel Density Classification

If the distribution of $x|K_c$ is more complicated, the Gaussian Mixture Model may not be a good choice. In this case, one can approximate it using a so-called kernel density estimator.
We also restrict ourselves here to the case $d = 1$, i.e. the inputs are **one-dimensional**.
We define

2.2.12

$$\hat{p}(x|K_c) := \frac{1}{\#\{i|y_i = l_c\}} \sum_{i|y_i=l_c} K_\lambda(x - x_i)$$

where $\lambda > 0$ is the bandwidth, $K_\lambda(x) := \frac{1}{\lambda} K(\frac{x}{\lambda})$ and $K : \mathbb{R} \to \mathbb{R}$ is a non-negative function with $\int_{\mathbb{R}} K(x)dx = 1$. Possible choices are the
Gaussian kernel

$$K(x) := (2\pi)^{-1/2} \exp(-x^2/2)$$

or $K(x) := \frac{1}{2} 1_{|x| \leq 1}$. However note that for the second choice, it may happen that $\hat{p}(x|K_c) = 0$ for all $c = 1, \dots, C$ which does not allow for class assignment.
This approach depends on the choice of bandwidth λ.
In particular, for $0 < \lambda \ll 1$, $\hat{p}(x|K_c) \approx 0$ if x is not a training data point x_i.
On the other hand, if λ is very large, $\hat{p}(\cdot|K_c)$ is nearly constant.
It can be shown that an optimal choice of λ for large values of $n \in \mathbb{N}$ takes the form $\lambda = Cn^{-\frac{1}{5}}$.
For normally distributed data $x_i \sim \mathcal{N}(\mu, \sigma^2)$, the optimal constant is given by $C = (\frac{4}{3})^{\frac{1}{5}} \sigma$

2.2.3.2 Naive Bayes Classification

The problem with Kernel density estimation is that it performs very poorly in high dimensions. While one can calculate 2.2.3.1 Bayes Classification > 2.2.12 for $x \in \mathbb{R}^d$ by using a non-negative function $K_\lambda : \mathbb{R}^d \to \mathbb{R}$ with $\int_{\mathbb{R}^d} K_\lambda(x)dy = 1$, an accurate estimation requires an extremely large number of data points, meaning N must be very large.
Another problem that we cannot address with the previous methods is the approximation of the conditional probabilities $\mathbb{P}(x|K_x)$ **for mixed variables** $x \in \mathbb{R}^d$, e.g. with qualitative (color) and quantitative (weight) component.
An alternative approach that circumvents both difficulties is naive Bayes classification, which operates under the **strong assumption** that the individual features x_{ij} for $j = 1, \dots, d$ for all data points x_i are independent within a class. In this case, we can use Bayes' theorem to get:

$$\mathbb{P}(K_c|x) = \frac{\mathbb{P}(K_c)\mathbb{P}(x|K_c)}{\sum_{i=1}^C \mathbb{P}(K_i)\mathbb{P}(x|K_i)} = \frac{\pi_c \prod_{j=1}^d \mathbb{P}(x_j|K_c)}{\sum_{i=1}^C \pi_i \prod_{j=1}^d \mathbb{P}(x_j|K_i)}$$

remember that π_c was considered as the probability $\mathbb{P}(K_c)$ and the estimator was $\hat{\pi}_c := \frac{\#\{i=1,\dots,N|y_i=l_c\}}{N}$

or for densities

$$\mathbb{P}(K_c|x) = \frac{\pi_c \prod_{j=1}^d p(x_j|K_c)}{\sum_{i=1}^C \pi_i \prod_{j=1}^d p(x_j|K_i)}, \quad x \in \mathbb{R}^d$$

Thus, in this case although $x \in \mathbb{R}^d$, we only need to approximate one-dimensional probabilities or densities, namely $\mathbb{P}(x_j|K_c)$ or $p(x_j|K_c)$ for $j = 1, \dots, d$. This can be achieved using one of the three approaches discussed above. The approximated naive Bayes classifier then becomes

$$x \mapsto \arg\max_{c=1}^C \hat{\pi}_c \prod_{j=1}^d \hat{P}(x_j|K_c), \quad \text{or} \quad x \mapsto \arg\max_{c=1}^C \hat{\pi}_c \prod_{j=1}^d \hat{p}(x_j|K_c)$$

Exercise 2.2.1

Calculate the naive Bayes classifier for the dataset below for x = (195,g, yellow). Use a Gaussian Mixture Model for the weight variable and a discrete model for the color

Class:	Weight:	Color:
Apple	162g	red
Apple	186g	yellow
Banana	112g	yellow
Banana	142g	green
Banana	128g	yellow

Table 2.1: Fruit Dataset

3.1 Principal Component Analysis

Principal component analysis (PCA) is the simplest and most widely used method for dimensionality reduction. It transforms the original variables or features (initial data dimension) into uncorrelated ones.
In the original dataset, some variables might be correlated (to be in a connection with each other), for example Height and Weight might be positively correlated (taller people tend to weigh more); correlated features provide redundant information, which PCA aims to reduce.
PCA creates new variables that are linearly uncorrelated:

- The **correlation** between **any** two principal **components is zero**
- This makes each principal component represent unique information from the data

We consider a data matrix $X \in \mathbb{R}^{N \times D}$ which contains $N \in \mathbb{N}$ feature vectors, each with $D \in \mathbb{N}$ features (N: number of data points or feature vectors (row vectors), D number of features).
Please see example below
That is, the i-th row for $i = 1, \dots, N$ contains the feature vector $x_i \in \mathbb{R}^D$. We assume that X has centered features, i.e. $\sum_{i=1}^{N} X_{ij} = 0$ for all $j = 1, \dots, D$.
This can always be achieved by replacing the data $(x_i)_{i=1}^{N}$ with $x_i - \overline{x}$, where $\overline{x} := \frac{1}{N} \sum_{i=1}^{N} x_i$

Example

Consider a data matrix $X \in \mathbb{R}^{3 \times 2}$ ($N = 3$ data points / samples (3 row vectors) and $D = 2$ features for each data point (2 columns))

$$X = \begin{bmatrix} 170 & 60 \\ 180 & 75 \\ 160 & 50 \end{bmatrix}$$

Here the first row $[170, 60]$ is the feature vector (D not really but only symbolically) for the first data point, the second row $[180, 75]$ is the feature vector for the second data point and $[160, 50]$ represents the third data point.^

We have two features:

- Feature 1: $[170, 180, 160]$ being for example Heights
- Feature 2: $[60, 75, 50]$ being for example Weights

So each data point is described by 2 features (one from each column)

Centering the features in X

Reconsider: Replacing the data $(x_i)_{i=1}^N$ with $x_i - \overline{x}$, where $\overline{x} := \frac{1}{N}\sum_{i=1}^N x_i$
Lets compute the mean for column 1:

$$\frac{170 + 180 + 160}{3} = 170$$

and the mean for column 2:

$$\frac{60 + 75 + 50}{3} = 61,66$$

So we need to do:

$$X - \overline{X} = \begin{bmatrix} 170 & 60 \\ 180 & 75 \\ 160 & 50 \end{bmatrix} - \begin{bmatrix} 170 & 61.66 \\ 170 & 61.66 \\ 170 & 61.66 \end{bmatrix} = \begin{bmatrix} 0 & -1.67 \\ 10 & 13.33 \\ -10 & -11.67 \end{bmatrix}$$

Now the features are centered because if you add each column they are 0:
$(0 + 10 + (-10)) = 0$ and $(-1.67 + 13.33 + (-11.67)) = 0$

3.1.1

The first goal is to transform the original features into uncorrelated ones. To do this, we consider the empirical covariance matrix

$$C := X^T X = \sum_{i=1}^N x_i x_i^T \in \mathbb{R}^{D \times D}$$

Since the features are assumed to be centered (i.e., $\sum_{i=1}^N x_i = 0$) the entry C_{ij} measures the empirical correlation between the i-th and j-th feature. We are now looking for a system of uncorrelated features. Note that C is a symmetric real-valued matrix ($C = C^T$ and all entries in the matrix are real numbers).
This means that the entry at row i, column j is equal to the entry at row j at column i.
$C_{ij} = C_{ji}$
An example for such a matrix would be

$$C = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix}, \quad C^T = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix}$$

Because of the matrix being symmetric real-valued, it is orthogonally diagonalizable, i.e., there exists an orthogonal matrix $P \in \mathbb{R}^{D \times D}$ such that $P^T X P$ is diagonal (all non-diagonal entries are 0).
Why is that matrix orthogonally diagonalizable?

1. All eigenvalues $\lambda_1, \ldots, \lambda_D$ are **real**
2. Eigenvectors corresponding to different eigenvalues are **orthogonal**

3. Using these eigenvectors, we can construct an orthogonal matrix P, whose columns are the eigenvectors of C
   An orthogonal matrix P satisfies $P^TP = I$, where I is the identity matrix. This means the columns of P are **orthonormal vectors**, i.e.:

- Each column has a length of 1 (normalization)
- Columns are perpendicular (senkrecht) to each other (orthogonality)
  Example

$$P = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad P^TP = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The columns of $P := (v_1, \dots, v_D)$ are also eigenvectors of C, i.e., there are numbers $(\lambda_j)_{j=1}^D \subset \mathbb{R}_+$ such that $Cv_j = \lambda_j v_j$ (remember: $Cv = \lambda v$.
For example:

$$C = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \lambda = 2$$

then:

$$Cv = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix} = \lambda v = 2 \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

We can assume without loss of generality that the columns of P are ordered such that $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_D$.

Example: Compute the covariance matrix

We have the centered X matrix already calculated above with

$$X - \overline{X} = \begin{bmatrix} 170 & 60 \\ 180 & 75 \\ 160 & 50 \end{bmatrix} - \begin{bmatrix} 170 & 61.66 \\ 170 & 61.66 \\ 170 & 61.66 \end{bmatrix} = \begin{bmatrix} 0 & -1.67 \\ 10 & 13.33 \\ -10 & -11.67 \end{bmatrix} \$\$Nowwecalculatetheempiricalcovariancem$$

3.1.2

We now want to use the eigenvectors v_j as new features. To do this, we consider the transformed data

$$Z := XP \in \mathbb{R}^{N \times D}$$

(XP gives the transformed data Z which now has the features in terms of the Eigenvectors of the covariance matrix C; P contains the eigenvectors of C)
Expanding the above matrix product, we obtain for $i = 1, \dots, N$ and $j = 1, \dots, D$

3.1.3

$$Z_{ij} = \sum_{k=1}^{D} X_{ik} P_{kj} = \sum_{k=1}^{D} (x_i)_k (v_j)_k = \langle x_i, v_j \rangle$$

(X_{ik} represents the k-th feature of the i-th data point, P_{kj} represents the k-th entry of the eigenvector v_j). The dot product means we're projecting the data point x_i onto the direction defined by the eigenvector v_j
$(x_i)_k$ refers to the k-th element of the row vector x_i, which is the i-th row of X:

$$x_1 = (X_{i1}, X_{i2}, \ldots, X_{iD})$$

Thus, the j-th new feature of the i-th new data point is given by the dot product of the old data point x_i with the j-th eigenvector of C. Another interpretation is that the new features are weighted sums of the old features, where the weights are given by the entries of the respective eigenvector.
With these insights, we can now achieve dimensionality reduction by considering only $j = 1, \ldots, d$ with $d < D$ in 3.1.3 (considering only the first d eigenvectors that correspond to the largest eigenvalues).
These eigenvectors capture the most variance in the data, and by using only the top d eigenvectors, we can reduce the number of features while preserving most of the information.
Equivalently, we can define the matrix $P^{(d)}$ as $P^{(d)} := (v_1, \ldots, v_d) \in \mathbb{R}^{D \times d}$ and instead of 3.1.2, consider the data transformation

3.1.4

$$Z^{(d)} := XP^{(d)} \in \mathbb{R}^{N \times d}$$

(this is the reduced dimensionality data)
Note that for $d = D, P^{(d)} = P$ and $Z^{(d)} = Z$. However, for $d < D, Z^d$ is a dimension-reduced representation of the data. Note that the components of $Z^{(d)}$ are exactly given by 3.1.3, i.e. for $i = 1, \ldots, N$ and $j = 1, \ldots, d, Z_{ij}^{(d)} = Z_{ij}$.

We first prove that the transformed data are indeed uncorrelated

Proposition 3.1.1

For all $j = 1, \ldots, d$ we have $\sum_{i=1}^{N} Z_{ij}^{(d)} = 0$. Moreover, the empirical covariance matrix $(Z^{(d)})^T Z^{(d)}$ is diagonal.

Proof: According to 3.1.3, we have

$$\sum_{i=1}^{N} Z_{ij}^{(d)} = \sum_{i=1}^{N} \langle x_i, v_j \rangle = \left\langle \sum_{i=1}^{N} x_i, v_j \right\rangle = 0$$

for all $j = 1, \ldots, d$. (the data is assumed to be centered!).
For the covariance matrix, we have

$$(Z^{(d)})^T Z^{(d)} = (XP^{(d)})^T XP^{(d)} = (P^{(d)})^T X^T XP^{(d)} = (P^{(d)})^T CP^{(d)}$$

Recall that $X^T X = C$, the empirical covariance matrix of the original data.
In the case where $d = D$, $P^T CP$ is diagonal, as P is chosen as the diagonalizing matrix for C.
In the general case, we note that we can write $P^{(d)} = PS$, where the matrix $S \in \mathbb{R}^{(D \times d)}$ is given by

$$S = \begin{pmatrix} \mathbb{I}_{d \times d} \\ 0_{D-d \times d} \end{pmatrix}$$

(S is a block matrix where the top block is the identity matrix and the bottom block is a zero matrix which is a $D - d \times d$ filled matrix with zeros)
Let's assume $D = 5$ and $d = 3$. Then S would look like

$$S = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Thus, we obtain (substitute $P^{(d)}$ with PS)

$$(Z^{(d)})^T Z^{(d)} = S^T P^T CPS = S^T \Lambda S$$

where $\Lambda \in \mathbb{R}^{D \times D}$ is a diagonal matrix. It is now easy to see that $S^T \Lambda S \in \mathbb{R}^{d \times d}$ is also a diagonal matrix, specifically the upper-left corner of Λ

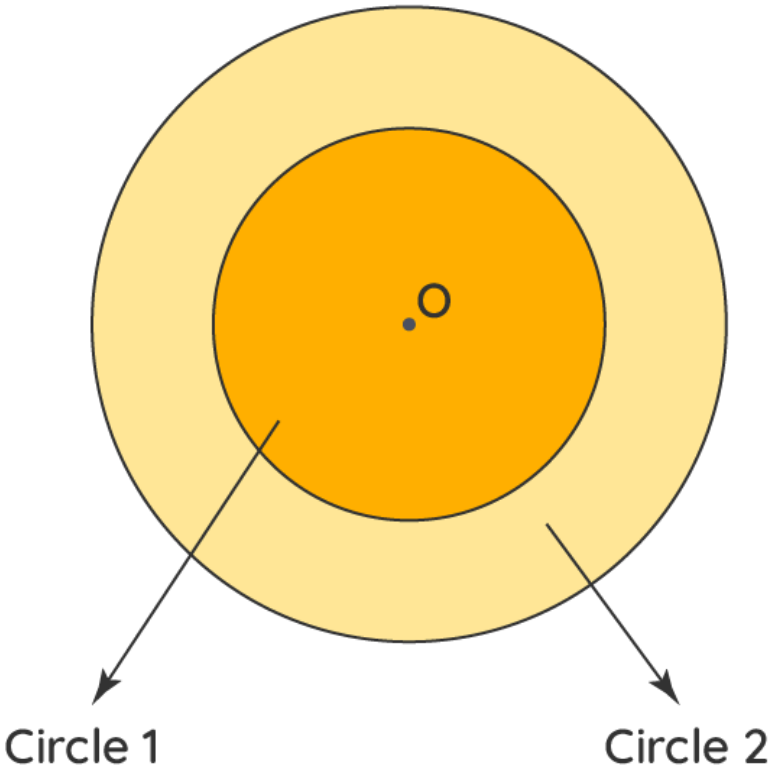
Summary

We summarize the principal component algorithm for dimensionality reduction as follows:

1. Create a data matrix $X \in \mathbb{R}^{N \times D}$ and normalize it so that all columns sum to zero
2. Compute the eigenvectors and eigenvalues of the covariance matrix $C = X^T X$
3. Retain the eigenvectors corresponding to the d largest eigenvalues and combine them into a matrix $P^{(d)} \in \mathbb{R}^{D \times d}$
4. Compute the transformed data $Z^{(d)} := XP^{(d)}$

Note that Principal Component Analysis assumes that the data can be well described by the empirical covariance matrix C. This is not always the case, for example, in the case of data points distributed on concentric circles

Concentric Circles



In this case, with appropriate scaling, C is proportional to the identity matrix, and it's eigenvectors are simply the coordinate directions. However, a more meaningful feature would be the distance of a data point from the origin. To extract this, it is useful as in Section 2.2.1, to first embed the data into a higher dimensional space and then perform PCA in this space. For instance, this can be done using the mapping $\phi : \mathbb{R}^2 \to \mathbb{R}^3$ with $\phi(x) := (x_1, x_2, x_1^2 + x_2^2)$. For the data described above, the dominant component would be the third coordinate, which corresponds precisely to the distance from the origin. Such situations can be handled much more systematically using the so-called kernel-based PCA, which goes beyond the scope of this lecture.

3.2 Clustering

We now proceed with clustering methods. As previously discussed, clustering refers to the division of data into groups. As before, we are given $N \in \mathbb{N}$ data points $\mathcal{D} := \{x_i\}_{i=1}^N \subset \mathbb{R}^d$. We also assume a fixed number $K \in \mathbb{N}$ of groups (clusters?)

We begin with a wish list. We are looking for:

1'. **Assignment of data points to cluster centers**: $x_i \mapsto k_i \in \{1, \dots, K\}$ for all $i = 1, \dots, N$

1. General assignment rule: $x \mapsto k(x) \in \{1, \dots, K\}$ for all $x \in \mathbb{R}^d$
2. Reconstruction rule: $k \mapsto m_k \in \mathbb{R}^d$ for all $k \in \{1, \dots, K\}$

We note that 1 is a special case of 1. Also notable is 2, where we seek a kind of inverse mapping (more precisely, a right inverse), which assigns a representative element m_j to each group k. We also refer to m_j as the **mean** of the k-th group. It is also reasonable to choose this mapping so that m_k belongs to the k-th group.

Next up: 3.2.1 K-means

3.2.1 K-means

The goal here is to place each data point in a group whose center it is closest to, and then adjust the group centers to better represent the points in them

The first clustering method we will consider is the so called K-means or Lloyd's algorithm. To derive the algorithm, we fix a maximum number of groups $K \in \mathbb{N}$ and consider two variables: a clustering variable $C := (C_1, \ldots, C_K)$ (represents the clusters or groups of data points) and a mean variable $m := (m_1, \ldots, m_k)$. Here $C_k \subset \mathcal{D}$ (Each C_k is a subset of $\mathcal{D}$ (the whole dataset)) for $k = 1, \ldots, K$ are subsets of the data points with $\bigcup_{k=1}^{K} C_k = \mathcal{D}$ (the union of all clusters should include all the data points: every data point must belong to one of the clusters and no point is left out) and $C_i \cap C_j = \emptyset$ for $i \neq j$ (clusters are disjoint, no data point can belong to more than one cluster). These subsets are interpreted as the k-th group. Additionally, $m_k \in \mathbb{R}^d$ for $k = 1, .., K$ and m_k is the mean of the k-th group.

The K-means algorithm is an iterative method that alternates between updating the variables C and m, holding the other constant
Given a clustering $C = (C_1, \ldots, C_K)$ the k-th mean m_k is updated as the mean of the data point in C_k :

3.2.1

$$m_k \leftarrow \frac{1}{|C_k|} \sum_{\substack{i=1,\ldots,N \\ x_i \in C_k}} \forall k = 1, \ldots, K$$

where $|C_k| := \#\{i = 1, \ldots, N | x_i \in C_k\}$ is the number of data points in the k-th group. This corresponds to the reconstruction rule 2
(The mean of each cluster is updated using the average of all points in that cluster, where $|C_k|$ represents the number of data points in cluster C_k)

On the other hand, fixing the means $m = (m_1, \ldots, m_K)$, the k-th group C_k is defined as the set of all points that are closer to m_k than to any other m_j for $j \neq k$.
To achieve this, we assign each data point to the nearest mean:

3.2.2

$$k_i \in \arg \min_{k \in \{1,\ldots,K\}} ||x_i - m_k||, \quad i = 1, \ldots, N,$$

resolving any ambiguity by choosing a group assignment (e.g., randomly). This is the assignment rule 1'.
(Here each data point is reassigned to the nearest cluster center. For each point x_i find the

cluster k whose mean m_k is closest to x_i. If multiple clusters are equidistant, a random assignment can be made)
Using this assignment to the nearest mean, we can now update the clustering as follows

3.2.3

$$C_k \leftarrow \{x_i | i = 1, \dots, N, k_i = k\}$$

(Update cluster C_k to contain all points x_i, such that the assigned cluster index $k_i = k$)
By definition $C_i \cap C_j = \emptyset$ for $i \neq j$ and $\bigcup_{k=1}^{K} C_k = \mathcal{D}$.
The K-means algorithm iterates through the steps 3.2.1 to 3.2.3 for a certain number of iterations or until the means and clustering change little or not at all, see:

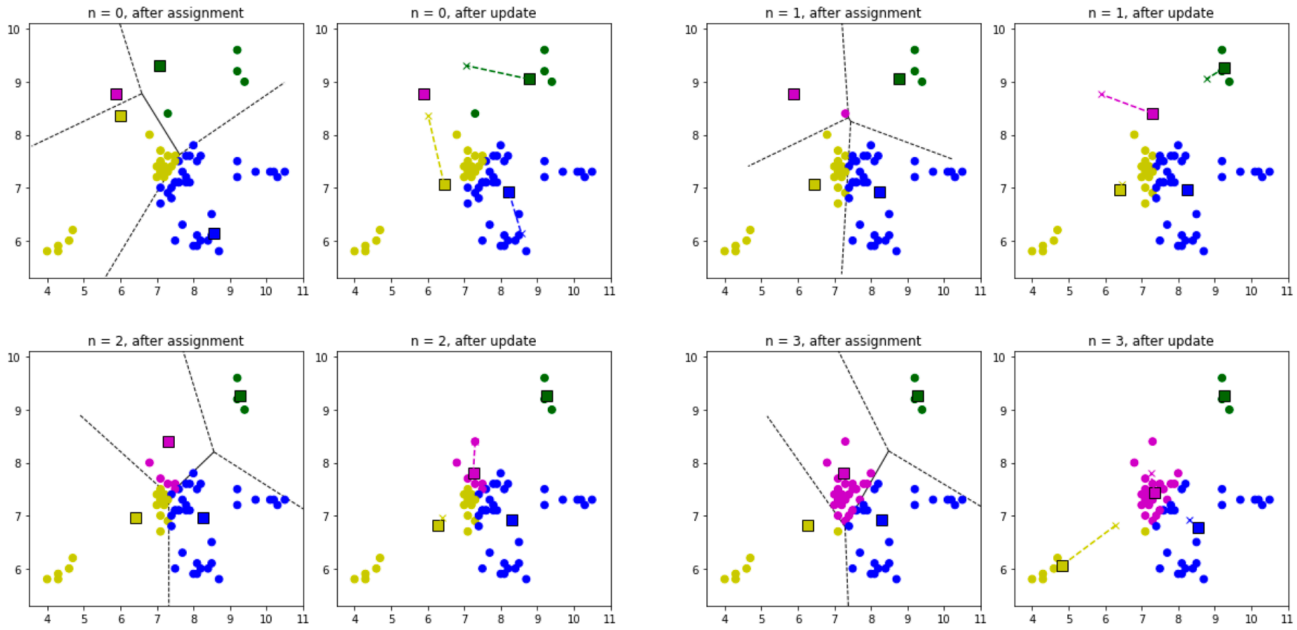


Figure 3.1: Four iterations of the K-means algorithm for $K = 4$.

The general assignment rule 1 for arbitrary points $x \in \mathbb{R}^d$ is obtained exactly as in 3.2.2 via

$$x \mapsto k(x) \in \arg \min_{k \in \{1,\dots,K\}} ||x - m_k||$$

where in case of ambiguity, a k in the argmin is chosen (e.g. randomly)

Remark 3.2.1 (Voronoi Cells)

The discrete clustering $C = (C_1, \dots, C_K)$ can also be replaced by so-called **Voronoi cells** of the means $m = (m_1, \dots, m_K)$. These are defined by

3.2.5

$$V_k := \{x \in \mathbb{R}^d | \ \ ||x - m_k|| \leq ||x - m_j|| \, \forall j = 1, \dots, K\}, k = 1, \dots, K$$

The cell V_k consists of all points that are closer to m_k than to any other means, see Figure blow.

- A voronoi cell V_k is the set of all points in the space ($\mathbb{R}^D$) that are closer to the centroid m_k than to any other centroid m_j for $j \neq k$
- If you imagine placing K cluster centers (means), the space around each center is divided into regions, where each region contains all the points that are closest to that specific center. These regions are called **Voronoi cells**.
  Note that the interiors of these Voronoi cells are not necessarily disjoint! For example, if two means are the same, the cells coincide.
  In fact, the K-means algorithm 3.2.1 to 3.2.3 has the property that it decreases the so-called clustering energy $E(C, m)$ (how good the clustering is) at each step. This energy is defined as

$$E(C, m) := \sum_{k=1}^{K} \sum_{\substack{i=1,\ldots,N \\ x_i \in C_k}} ||x_i - m_k||^2$$

  But what does this formula mean?
  The clustering Energy $E(C, m)$ measures how well the centroids represent the data points in each cluster

1. For each cluster C_k compute the squared distance between each data point x_i in that cluster and the cluster's mean m_k
2. Sum these squared distances across all clusters
   Why squared distance?
   It penalizes points that are far from the centroid more than closer ones, encouraging tighter clusters
   A **lower energy** means that the clusters are well formed (points are close to centroids)
   A **higher energy** means that the clustering isn't very good (points far from their assigned centroids)

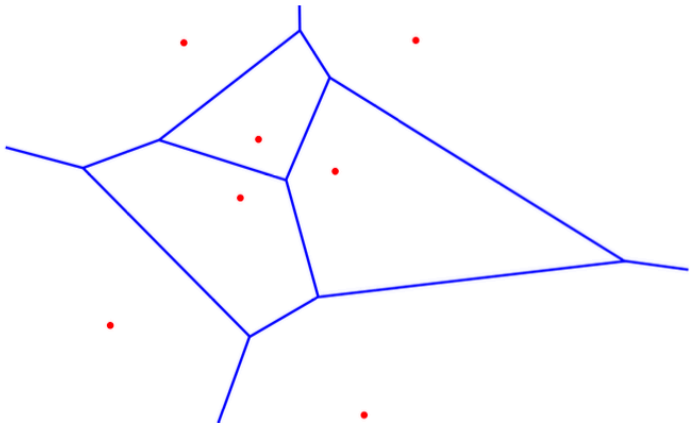


Figure 3.2: Voronoi cells regarding the red data points.

Note that the dependence on the clustering variable C is quite complicated. Finally, C is neither a vector nor a number but a collection of K disjoint subsets of $\mathcal{D}$. Thus, one cannot use a gradient-based algorithm to minimize E with respect to C.
Instead, the following holds:

Proposition 3.2.1

For all clusterings C and for m defined as in 3.2.1, it holds that

$$E(m, C) \leq E(n, C)$$

for all other n.
This means: If we fix the clustering C and update the centroids m, the energy $E(m, C)$ will always decrease or stay the same. This means updating the centroids to the mean of the assigned points is always an improvement!

For all means m and for $C := (C_1, \ldots, C_K)$ defined in 3.2.2 and 3.2.3, it holds that

$$E(m, C) \leq E(m, D)$$

for all other clusterings $D = (D_1, \ldots, D_K)$ with $\bigcup_{k=1}^{K} D_k = \mathcal{D}$ (all points in dataset must be included in one of the clusters of D_k and $D_i \cap D_j = \emptyset$ for $i \neq j$ (no clusters overlap; each data point belongs to exactly one cluster)
This means: If we fix the centroids m and update the clusters C by assigning points to the closest centroid, the energy will always decrease or stay the same. This means the reassignment step (placing each point in the closest cluster) improves or maintains clustering quality

The K-means algorithm also has some disadvantages. On the one hand, it struggles with more complex data, as illustrated in the Figure in the next chapter. Another problem is that K-means

does not provide information about the certainty of the group assignment. It would be desirable to obtain a probability vector with K probabilities for each data point.

3.2.2 EM-Clustering

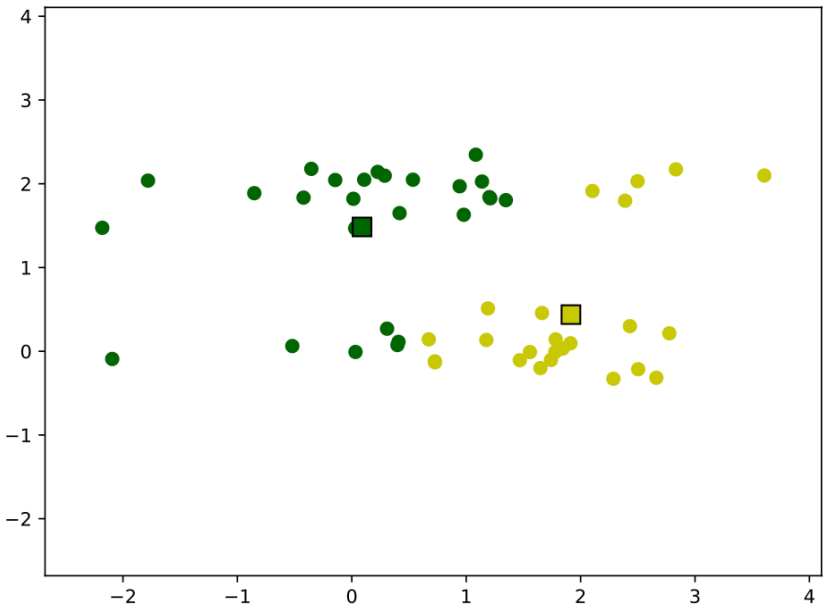


Figure 3.3: K-means fails to recognize the elongated structures in the data.

To address the difficulties mentioned in the last chapter with the K-means algorithm, we consider a stochastic approach. We assume that the data is distributed according to a Gaussian Mixture Model, i.e. that the points of a cluster are normally distributed around a mean m_k. In mathematical terms, this means that the data of the k-th group C_k has a density function of the form

3.2.9

$$p_k(x) := \frac{1}{\sqrt{(2\pi)^d \det \Sigma_k}} \exp\left(-\frac{1}{2}(x-m_k)^T \Sigma_k^{-1}(x-m_k)\right), \quad x \in \mathbb{R}^d$$

where $m_k \in \mathbb{R}^d$ is a mean of cluster k and $\sum_k \in \mathbb{R}^{d \times d}$ is a covariance matrix for $k = 1, \ldots, K$ (describes shape and spread of the cluster). $(x-m_k)^T \sum_k{}^{-1}(x-m_k)$ measures the **Mahalanobis** distance, which generalizes Euclidian distance by considering correlations between variables.

$\frac{1}{\sqrt{(2\pi)^d \det \sum_k}}$ normalizes the Gaussian function so that the probability integrates to 1

Note that in constrast to Section 2.2.1, we must work with multivariate normal distributions here. In the case of $d = 1$ and for $\sum_k := \sigma_k^2$ (3.2.9) simplifies to the familiar density of a scalar normal distribution with mean $m_k \in \mathbb{R}$ and variance σ_k^2:

$$p_k(x) = \frac{1}{\sqrt{2\pi\sigma_k^2}} \exp\left(-\frac{(x-m_k)^2}{2\sigma_k^2}\right)$$

The distribution of all data can then be modeled by a density function of the form
(Instead of a single Gaussian function, the whole dataset is modeled as a **mixture** of multiple Gaussian distributions)

3.2.10

$$p(x) := \sum_{k=1}^{K} \pi_k p_k(x), \quad x \in \mathbb{R}^d$$

where $0 \leq \pi_k \leq 1$ with $\sum_{k=1}^{K} \pi_k = 1$.

Explanation

- π_k
  - These are the **mixing coefficients**, representing the probability that a randomly chosen data point belongs to cluster k
  - They must satisfy $0 \leq \pi_k \leq 1$ and sum to 1!
- $p_k(x)$
  - The probability density function for cluster k
  Thus the full data distribution $p(x)$ is a weighted sum of **individual** Gaussian distributions, where each cluster contributes proportionally to its prior probability π_k

As in Bayesian classification, the numbers π_k represent the probabilities of the k-th group C_k. Our goal is to calculate the means m_k, the covariance matrices $\sum_k$, and the group probabilities π_k by utilizing the data

For this purpose, the so-called ***responsibilities*** (determine probability that point x belongs to cluster k) of the k-th group for the point x, denoted as $\gamma(x,k)$ are helpful. These are defined as

3.2.11

$$\gamma(x,k) := \mathbb{P}(C_k|x) = \frac{\pi_k p_k(x)}{p(x)} = \frac{\pi_k p_k(x)}{\sum_{k=1}^{K} \pi_k p_k(x)}$$

where we have used Bayes' theorem for densities.

Exkurs

$$\mathbb{P}(C_k|x) = \frac{\mathbb{P}(x|C_k)\mathbb{P}(C_k)}{\mathbb{P}(x)}$$

- $\mathbb{P}(x|C_k) \to p_k(x)$:
  - The likelihood of observing the point x given that it belongs to cluster C_k. Modeled as Gaussian distribution for cluster k
- $\mathbb{P}(C_k) \to \pi_k$
  - The prior probability of cluster k, representing the proportion of data expected in cluster k
- $\mathbb{P}(x) \to p(x)$
  - The total probability of observing x across **all clusters**, which serves as the denominator

Explanation

The numerator $\pi_k p_k(x)$ represents the probability that x belongs to cluster k, considering the prior π_k and the likelihood $p_k(x)$
The denominator is

$$p(x) = \sum_{k=1}^{K} \pi_k p_k(x)$$

This means that $p(x)$ is the weighted sum of the likelihoods of x under each cluster, weighted by their respective priors π_k

The so-called EM (Expectation Maximization) algorithm now starts from an initial initialization of the parameters $\pi_k, m_k, \sum_k$ and updates these using the following rules:

Initialization

Basically figure out the square or space you are in and either place points randomly or if you have two clusters, you compute a mean point, and pertubate this point into two points then and thus initialize the parameters. Elsewise often K-Means is just used for initialization

First, we define a kind of weight for the k-th group as the sum of $\gamma(x, k)$ over all data points via

3.2.12

$$N(k) \leftarrow \sum_{i=1}^{N} \gamma(x_i, k)$$

Explanation: weight for k-th group is calculated by summing the responsibilities for each point x_i with respect to cluster k. $N(k)$ is the total weight assigned to cluster k, based on how much each point belongs to that cluster. Remember $\gamma(x_i, k)$ is the **responsibility**, which indicates the probability that the point x_i, belongs to cluster k.

We can now update the group probabilities π_k by

3.2.13

$$\pi_k \leftarrow \frac{N(k)}{N}$$

Explanation: update prior probability for each cluster. N is the total number of data points, $N(k)$ is the sum of responsibilities for the k-th cluster (see above!)

Next, we update the means through a weighted average of the points, where the weights are the responsibilities

3.2.14

$$m_k \leftarrow \frac{1}{N(k)} \sum_{i=1}^{N} \gamma(x_i, k) x_i \in \mathbb{R}^d$$

Thus, the points contribute significantly to the k-th mean for which the k-th group has a large responsibility. (Points that have a higher responsibility for cluster k will have more influence on the new mean of that cluster).

Finally, we update the covariance matrices using a weighted empirical covariance matrix:

3.2.15

$$\Sigma_k \leftarrow \frac{1}{N(k)} \sum_{i=1}^{N} \gamma(x_i, k)(x_i - m_k)(x_i - m_k)^T \in \mathbb{R}^{d \times d}$$

Explanation

The covariance matrix is updated to reflect the spread of the data points around the new mean for each cluster, weighted by their responsibilities, where $(x_i - m_k)$ is the vector of deviations from the mean point for x_i. Multiplying this with its transposed form gives a covariance matrix! This is then weighed by the responsibility. We divide by the total effective weight of cluster k because this normalizes the sum, resulting in the **weighted empirical covariance matrix** for cluster k

An iteration of the EM algorithm thus executes the steps 3.2.12 to 3.2.15 for all $k = 1, .., K$. In the next iteration, the responsibilities $\gamma(x, k)$ defined in 3.2.11 are calculated using the updated parameters of the Gaussian Mixture Model. This is iterated for a fixed number of iterations or until the parameters change only slightly. See figure below:

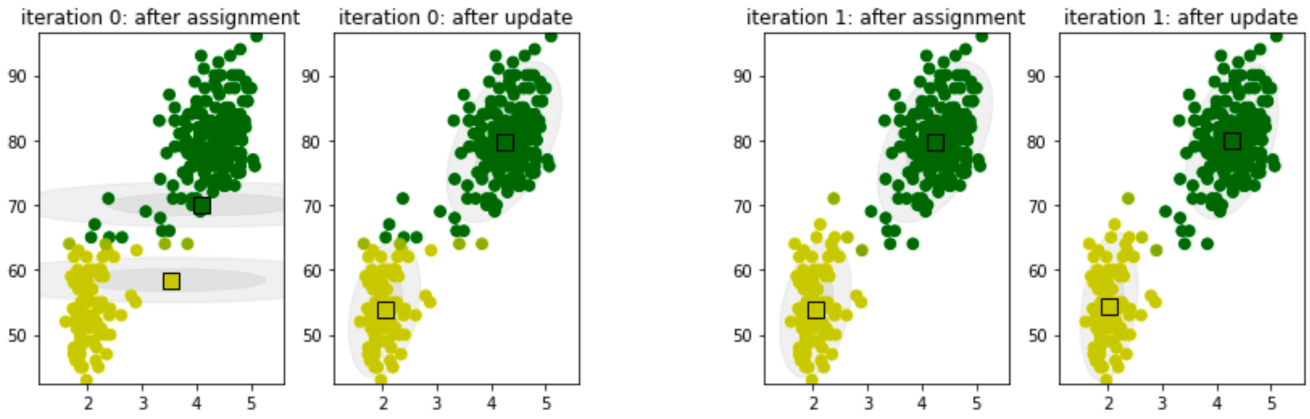


Figure 3.4: Two iterations of the EM algorithm for $K = 2$.

The main difference between K-means and EM lies in the covariance matrices Σ_k, which allow the algorithm to adapt locally to the data, similar to what we observed in principal component decomposition. It should also be noted that the vector $(\gamma(x, k))_{k=1}^{K}$ is the sought probability vector that contains the group probabilities for a fixed data point $x \in \mathbb{R}^d$

3.2.3 Spectral Clustering

https://www.youtube.com/watch?v=zkgm0i77jQ8

The last clustering method we will discuss is called spectral clustering. In contrast to the previous two methods, we require data in the form of a weighted graph for this method. However, it is capable of clustering significantly more complex datasets (e.g. two concentric circles)
A weighted graph is a tuple $G = (V, E, W)$ with a set of vertices V, a set of edges $E \subset V \times V$, and a weight function $W : E \to [0, \infty]$. We use the data points as the vertex set, i.e. $V = \{x_i\}_{i=1}^N \subset \mathbb{R}^d$ (**Explanation** the vertices are the data points in $\mathbb{R}^d$, meaning each data point is a node in the graph). Edges are drawn between "similar" data points. For this, we choose a weight function of the form $W(x, y) := \eta(||x - y||)$ which takes large values when $x, y \in V$ are similar (i.e., $||x - y||$ is small) and small values when they are dissimilar.

Explanation: If $||x - y||$ (Euclidian distance) is small, the similarity should be **large**, meaning the points are closely related.
If $||x - y||$ is **large**, the similarity should be **small**, meaning the points are far apart and less related
Popular choices are $\eta(s) = 1_{0 \leq s \leq \epsilon}$ for $\epsilon > 0$

Explanation: This means an edge is drawn between two points if their distance is within ϵ or $\eta(s) = \exp(-s^2/\sigma^2)$ for $\sigma > 0$

Example
If $\epsilon = 3$ and $||x - y|| = 2$ then $W(x, y) = \eta(2)$ and $\eta(2) = 1$ because $0 \leq 1 \leq 3$

Explanation: This function smoothly decreases similarity as distance increases. σ controls the decay; larger σ means more points are considered similar. We can define the edge set as $E = \{(x, y) \in V \times V | W(x, y) > 0\}$.

Explanation: This means that an edge exists between two points if the weight is greater than zero.
Since $W(x, y) = W(y, x)$, the resulting graph is undirected or symmetric, i.e., $(x, y) \in E \iff (y, x) \in E$. We can identify the graph G with its adjacency or weight matrix $W \in \mathbb{R}^{N \times N}$, where $W_{ij} := W(x_i, x_j)$ for $i, j = 1, \ldots, N$. This matrix is also symmetric. The interesting case for clustering is when $(x_{i_1}, x_{i_2}, \ldots, x_{i_m})$ (this represents a sequence of data points (or path in the graph starting from x_{i_1})) with $m \in \mathbb{N}$ and $i_j \in \{1, \ldots, N\}$ (this represents an index in the dataset) for $j = 1, \ldots, m$ such that $x_{i_1} = x, x_{i_m} = y$ (this means that the first point

in the sequence is x and the last point is y indicating a connection between them through intermediate points), and also $(x_{i_j}, x_{i_{j+1}}) \in E$ for all $j = 1, \ldots, m-1$

Explanation
$(x_{i_j}, x_{i_{j+1}}) \in E$ ensures that every consecutive (one after another) pair of points in the sequence is directly connected by an edge in the graph

Example
If we are given $x_{i_1} = x_2$ and $x_{i_m} = x_5$ it means that we are checking if there's a path from x_2 to x_5 through intermediate connections
The subscripts $i_1, i_2, \ldots, i_m$ indicate indices of data points forming a path in the graph

We first consider the case where we want to find $K = 2$ clusters. To do this, we want to assign a **value** to each graph **node**, with **similar nodes receiving similar values**.

Explanation: If two nodes are connected by a high-weight edge (indicating strong similarity) their assigned values should be close to each other.
We denote the collection of these values by $u \in \mathbb{R}^N$. A sensible assumption is that the mean of u is zero, i.e. $\sum_{i=1}^N u_i = 0$. If we define the vector $\mathbf{1} := (1, \ldots, 1) \in \mathbb{R}^N$, this is equivalent to $\langle u, \mathbf{1} \rangle = 0$ $(= u_1 \cdot 1 + u_2 \cdot 1 + \ldots = \sum_{i=1}^N u_i$, now we assumed above that this is $0)$
To enforce that similar nodes receive similar values, we use a minimization approach: We try to determine u such that

3.2.16

$$E(u) := \frac{1}{2} \sum_{i,j=1}^{N} W_{ij} |u_i - u_j|^2$$

is small, while also satisfying $\langle u, \mathbf{1} \rangle = 0$ (in geometric sense, if u is orthogonal to $\mathbf{1}$ then the dot product of both is 0). However, this problem has an uninteresting solution, namely $u = 0$.
Explanation
$E(u)$ measures the total squared difference between nodes x_i and x_j. The weights W_{ij} capture the similarity between nodes x_i and x_j. The goal is to minimize this function which means minimizing differences in u_i and u_j for strongly connected nodes (with large W_{ij})
To ensure that u takes positive and negative values (and not trivially 0), we require the Euclidian norm of the vector u equals 1, i.e., $||u||^2 := \sum_{i=1}^N u_i = 1$
(This is done to prevent trivial solution and force optimization to produce meaningful partition of nodes)
Thus we obtain the optimization problem (which we want to **solve**)

3.2.17

$$\min\{E(u) \mid u \in \mathbb{R}^N, \ ||u|| = 1, \ \langle u, 1 \rangle = 0\}$$

$$(= \min_{||u||_2=1} \langle u, Lu \rangle = \lambda_2)$$

Explanation
Minimize function $E(u)$ ensuring that similar nodes have similar values. Ensure that the sum of values is zero and ensure that the vector u has unit length (normalization constraint)

It turns out that the optimization problem 3.2.17 is equivalent to determining a certain eigenvector of the so-called graph Laplacian matrix. This is defined as

3.2.18

$$L := D - W \in \mathbb{R}^{N \times N}$$

where W is the **weight** matrix and D is the so-called degree matrix of the graph
This is a diagonal matrix with

3.2.19

$$D_{ij} := \begin{cases} \sum_{k=1}^{N} W_{ik} & \text{if } i = j, \\ 0 & \text{if } i \neq j \end{cases}$$

(diagonal elements D_{ii} of D are computed as the sum of the entries in the corresponding row i of the weight matrix W. This sum represents the **degree** (total connection strength) of node i)
Interpretation: You're summing across all nodes k that are connected to node i, which gives the **total weight** of all edges incident to node i. This forms the diagonal element D_{ii}, capturing the **degree (sum of edge weights)** of node i

Example
Consider following weight matrix W

$$W = \begin{bmatrix} 0 & 2 & 1 \\ 2 & 0 & 3 \\ 1 & 3 & 0 \end{bmatrix}$$

Row 1 sum: $0 + 2 + 1 = 3$
Row 2 sum: $2 + 0 + 3 = 5$
Row 3 sum $1 + 3 + 0 = 4$
So the degree matrix D is

$$D = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 4 \end{bmatrix}$$

We can now compute the Laplacian matrix L:

$$L = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 4 \end{bmatrix} - \begin{bmatrix} 0 & 2 & 1 \\ 2 & 0 & 3 \\ 1 & 3 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 3 & -2 & -1 \\ -2 & 5 & -3 \\ -1 & -3 & 4 \end{bmatrix}$$

Using the graph Laplacian matrix, we can express the function E as an inner product:

Proposition 3.2.2

It holds that $E(u) = \langle Lu, u \rangle$ for all $u \in \mathbb{R}^N$
Proof: It holds that $\langle Lu, u \rangle = \sum_{i=1}^{N} (Lu)_i u_i$ and also

$$(Lu)_i = \sum_{j=1}^{N} L_{ij} u_j$$

L is Laplacian matrix, which represents the **graph structure**, the vector $u \in \mathbb{R}^N$ contains values assigned to each node in the graph. The subscript i on $(Lu)_i$ means the i-th entry of the **vector** obtained by multiplying the Laplacian matrix L with the vector u.
Expanding this we get:

$$(Lu)_i = \sum_{j=1}^{N} L_{ij} u_j$$

This is the matrix vector multiplication of L and u, calculated as the dot product of the i-th row of L with the entire vector u. Each entry in the resulting vector Lu is a weighted sum of the components of u, where the weights come from the matrix L

Thus, we obtain

$$\langle Lu, u \rangle = \sum_{i=1}^{N} (Lu)_i u_i$$

Substituting $(Lu)_i$ in from above we get:

$$\langle Lu, u \rangle = \sum_{i=1}^{N} \sum_{j=1}^{N} L_{ij} u_j u_i$$

This equation represents a double summation over all pairs (i, j) where each term multiplies the weight L_{ij} with the corresponding values u_j and u_i.

Now we know that $L = D - W$ and $D_{ii} = \sum_{k=1}^{N} W_{ik}$ so we can replace L with $D - W$:

$$= \sum_{i=1}^{N} \sum_{j=1}^{N} D_{ij} u_j u_i - \sum_{i=1}^{N} \sum_{j=1}^{N} W_{ij} u_j u_i$$

We also know that D is diagonal, so the off-diagonal terms vanish, leaving:

$$= \sum_{i=1}^{N} D_{ii} u_i^2 - \sum_{i=1}^{N} \sum_{j=1}^{N} W_{ij} u_j u_i$$

Exkurs
Now using the definition of $D_{ii} = \sum_{j=1}^{N} W_{ij}$ we can rewrite:

$$\sum_{i=1}^{N} D_{ii} u_i^2 = \sum_{i=1}^{N} \left(\sum_{j=1}^{N} W_{ij} \right) u_i^2$$

Rearranging gets us to

$$\sum_{i=1}^{N} \sum_{j=1}^{N} W_{ij} u_i^2$$

Now **back to the actual proof**:

$$= \sum_{i=1}^{N} \sum_{j=1}^{N} W_{ij} u_i^2 - \sum_{i=1}^{N} \sum_{j=1}^{N} W_{ij} u_j u_i$$

We can now factor out the common term W_{ij}. This is done by factoring out the common term in both terms of the sum (W_{ij}). The terms u_i^2 and $u_j u_i$ inside the parentheses are distinct components!

$$\sum_{i,j=1}^{N} = W_{ij} (u_i^2 - u_j u_i)$$

We factor our u_i from u_i^2

$$= \sum_{i,j=1}^{N} W_{ij} u_i (u_i - u_j)$$

Exkurs
Recall the squared difference identity: $|u_i - u_j|^2 = (u_i - u_j)^2 = u_i^2 - 2u_i u_j + u_j^2$
Now we observe that we don't have this form and only have $(u_i^2 - u_j) = u_i(u_i - u_j)$.
So we observe that the squared difference has an extra u_j^2 term, which is missing from our current expression. We add and subtract u_j^2 to balance the equation:

$$u_i^2 - u_i u_j = u_i^2 - 2u_i u_j + u_j^2 - u_j^2 + u_i u_j$$

(If you are still not sure: notice that $+u_j^2 - u_j^2$ would cancel out and $+u_i u_j$ would add to $-2u_i u_j$.

But we don't want to do this here. We want to rearrange)
We can now group the terms:

$$(u_i^2 - 2u_iu_j + u_j^2) + (-u_j^2 + u_iu_j)$$

In the first parantheses we recognize: $(u_i^2 - 2u_iu_j + u_j^2) = (u_i - u_j)^2$
Thus, the expression simplifies to $(u_i - u_j)^2 + (-u_j^2 + u_iu_j)$
Now we can factor out the remaining term $-u_j^2 + u_iu_j$

$$(-u_j^2 + u_iu_j) = u_j(-u_j + u_i) = u_j(u_i - u_j)$$

So we **finally** have

$$(u_i - u_j)^2 + u_j(u_i - u_j)$$

Now **back to the actual proof**:
Using the expansion in the sum

$$\sum_{i,j=1}^{N} W_{ij}u_i(u_i - u_j)$$

becomes

$$\sum_{i,j=1}^{N} W_{ij}((u_i - u_j)^2 + u_j(u_i - u_j))$$

Now we can split the summation into two separate sums:

$$\sum_{i,j=1}^{N} W_{ij}(u_i - u_j)^2 + \sum_{i,j=1} W_{ij}u_j(u_i - u_j)$$

This effectively is the same as

$$= \sum_{i,j=1}^{N} W_{ij}|u_i - u_j|^2 + \sum_{i,j=1}^{N} W_{ij}u_j(u_i - u_j)$$

The change to absolute value is simply a notational way to emphasize that we are always dealing with positive values here!
The first term measures the squared differences between values of u, weighted by the corresponding graph weight W_{ij}. The second term balances the contribution of the neighboring node's value u_j.
Now recall that $E(u)$ is defined as

$$E(u) := \frac{1}{2}\sum_{i,j=1}^{N} W_{ij}|u_i - u_j|^2$$

So we can just do $2E(u)$ to eliminate the $\frac{1}{2}$ and then get the equation

$$= 2E(u) - \sum_{i,j=1}^{N} W_{ij} u_i (u_i - u_j)$$

and since

$$\langle Lu, u \rangle = \sum_{i,j=1}^{N} W_{ij} u_i (u_i - u_j)$$

(if you're wondering why, just look at what we started with! It was $\langle Lu, u \rangle$ the whole time!!!) So we get

$$= 2E(u) - \langle Lu, u \rangle$$

where in the penultimate step, we used the symmetry $W_{ij} = W_{ji}$ of the weight matrix. By rearranging, we obtain $\langle Lu, u \rangle = E(u)$

END OF PROOF

Now we collect some important properties of the graph Laplacian matrix, which mostly follow from Proposition 3.2.2

Proposition 3.2.3

The graph Laplacian matrix L has the following properties

1. It is symmetric, i.e., $L^T = L$
   - Proof from exercise:
     - Let $u \neq 0$ and let the Graph be connected. $\Leftrightarrow \forall i, j = \{1, \ldots, N\}$, there exists $(i_1, \ldots, i_l)$ s.t. $i = i_1$ and $j = i_l$ and $\forall p \in \{1, \ldots, l-1\}, W_{seeexercise} > 0$
     - Let $(i_1, \ldots, i_l)$ be a path from x_1 to x_j for $j \in \{1, \ldots, N\}$
     - Then $0 \leq \frac{1}{2} \sum_{k=1}^{l}$
2. It is positive semidefinite, meaning all eigenvalues are non-negative
3. Its smallest eigenvalue is $\lambda = 0$, and if the graph is connected, all corresponding eigenvectors are of the form $u = c\mathbf{1}$ with $c \neq 0$
   - Consider the vector $\mathbf{1} = (1, 1, \ldots, 1) \in \mathbb{R}^N$
   - Applying L to $\mathbf{1}$
     - $L\mathbf{1} = (D - W)\mathbf{1} = D\mathbf{1} - W\mathbf{1}$
     - Since D is the degree matrix (and its diagonal! meaning if you sum a row, theres only one value $\neq 0$!), $D\mathbf{1}$ gives a vector which is the degree of the corresponding node

    - $W\mathbf{1}$ will give the same vector, because each row of W represents the weights of edges connected to a node, and summing those weights will give the same result (matrix-vector multiplication!)
- Therefore $L\mathbf{1} = 0$. This shows that $\mathbf{1}$ is an eigenvector corresponding to the eigenvalue $\lambda = 0$
- Now to the connected graph thing (meaning that there is a path between any pair of nodes in the garph)
    - For a connected graph, the corresponding eigenvector for the eigenvalue $\lambda = 0$ is always a **constant** vector $u = c\mathbf{1}$, where c is some constant (typically non-zero). This is because the Laplacian matrix measures the difference between a node and its neighbors. A constant vector, where all elements are the same, has no difference between any node and its neighbors, which means it corresponds to an eigenvalue of zero
        - The constant eigenvector $u = c\mathbf{1}$ corresponds to the fact that if all nodes are assigned the same value, the difference between connected nodes is zero, which minimizes the energy function and satisfies the eigenvalue equation $Lu = 0$

4. Let $u, v \in \mathbb{R}^N$ be eigenvectors corresponding to two different eigenvalues λ and μ. Then $\langle u, v \rangle = 0$
   - The Laplacian matrix L satisfies the eigenvalue problem $Lu = \lambda u$
   - Similarly for a different eigenvector v we have $Lv = \mu v$
   - Since the Laplacian is symmetric it's eigenvectors corresponding to distinct eigenvalues are orthogonal. (due to symmetry)
   - For example, consider this matrix where each column vector is an eigenvector
   -

$$\begin{bmatrix} -0.44 & -0.71 & 0.56 & 0.0 \\ -0.56 & 0.0 & -0.44 & -0.71 \\ -0.56 & 0.0 & -0.44 & 0.71 \\ -0.44 & 0.71 & 0.56 & 0 \end{bmatrix}$$

   - and consider this matrix where each value along the diagonal is an eigenvalue
   -

$$\begin{bmatrix} 2.6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1.6 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

   - Now if you you take any different eigenvalue λ and μ, $\langle u, v \rangle$ will be 0
   - Example:

-

$$\begin{pmatrix} -0.44 & -0.56 & -0.56 & -0.44 \end{pmatrix} \begin{pmatrix} -0.71 \\ 0.0 \\ 0.0 \\ 0.71 \end{pmatrix}$$

-

$$\langle u, v \rangle = (-0.44 \cdot -0.71) + (-0.56 \cdot 0.0) + (-0.56 \cdot 0.0) + (-0.44 \cdot 0.71)$$

-

$$= 0.3124 + 0 + 0 - 0.3124 = 0$$

From Proposition 3.2.3 we conclude that the first eigenvalue of L is equal to zero. For connected graphs, the dimension of the corresponding eigenspace is one, and thus the second eigenvalue of L is positive. It can be shown relatively easy that the minimum value in 3.2.17 is exactly the second eigenvalue. Furthermore, the corresponding eigenvectors are minimizers.

Thus, spectral clustering consists of determining a minimizer of 3.2.17, or equivalently, an eigenvector corresponding to the second eigenvalue of the graph Laplacian matrix L.

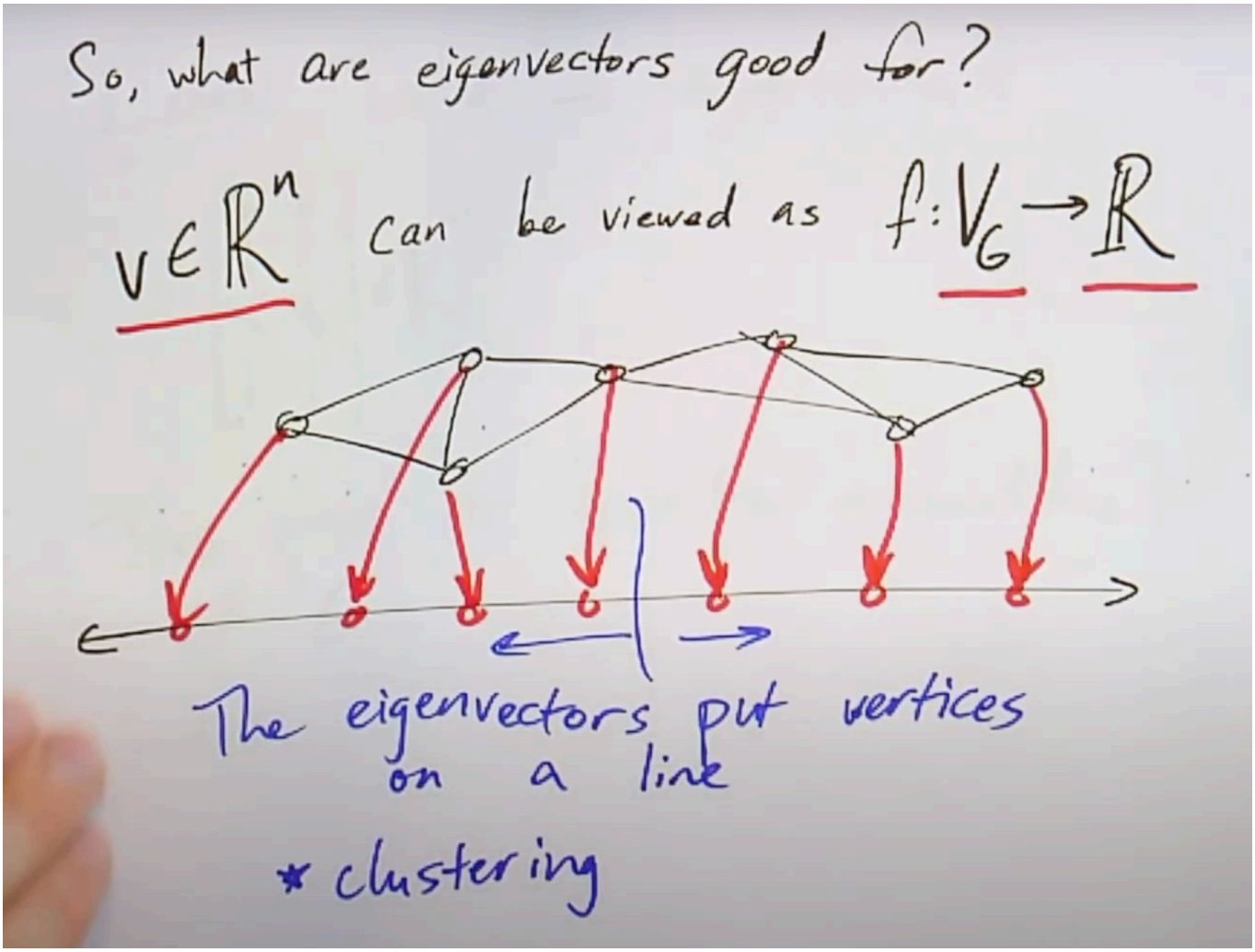
We obtain a trivial clustering for $K = 1$ by using the first eigenvector, e.g., $u^{(1)} = \mathbf{1}$. A clustering for $K = 2$ is obtained by calculating a second eigenvector $u^{(2)}$ (which is orthogonal to $u^{(1)}$, i.e. $\langle u^{(2)}, u^{(1)} \rangle = 0$). The cluster assignment for the i-th data point is then determined by the sign of $u_i^{(2)}$ (for example if node i has $u_i^{(2)} > 0$ it goes to Cluster A, else it goes to Cluster B)

To deal with larger numbers of clusters, we compute the so-called spectral embedding. This replaces the data matrix $X \in \mathbb{R}^{N \times d}$, where the i-th row of X is given by x_i, with the spectral embedding $Z \in \mathbb{R}^{N \times m}$, where the j-th column of Z contains the j-th eigenvector of L for $j = 1, \ldots, m$ with $m \leq N$. On these transformed data Z, we can then apply a clustering method such as K-means or EM again

!Missing Contents!

Need to know what y_i is computed for

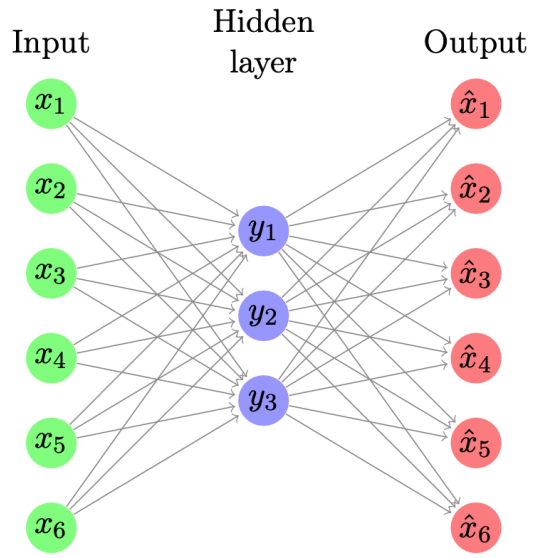
Spectral clustering



Now from here you can measure the distance of those points?

3.3 Autoencoders

An autoencoder is a particular network architecture allowing to do dimension reduction. The idea is to learn to *encode* some vector $x \in \mathbb{R}^D$ in the feature space by another latent vector $y \in \mathbb{R}^d$, and then decode y into a vector $\hat{x}$ with the hope that $\hat{x} \approx x$.
By taking $d << D$, it would in principle allow to do dimension reduction.
The architecture of a 1-hidden layer neural network is given in this figure



In this case, the autoencoder can be written as

$$\hat{x} := \hat{f}(x) := W_2\sigma(W_1x + b_1) + b_2$$

Remember: x is input data, W_1 is a weight matrix that maps input from $\mathbb{R}^d$ to $\mathbb{R}^D$ (the number of columns in W_1 must match the dimension of the input vector) where D (number of columns) is the original dimension and d is the reduced dimension. Analogue, W_2 maps the compressed representation ($\mathbb{R}^d$) back to the original space $\mathbb{R}^d$
with $x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{d\times D}, W_2 \in \mathbb{R}^{D\times d}, b_1 \in \mathbb{R}^d, b_2 \in \mathbb{R}^D$

Example

Suppose $D = 4, d = 2$
$W_1 \in \mathbb{R}^{2\times 4}$: Maps from $\mathbb{R}^4$ to $\mathbb{R}^2$ (Matrix with 2 rows and 4 columns), $W_2 \in \mathbb{R}^{4\times 2}$ maps back to $\mathbb{R}^4$
Now multiplying for example

$$W_1 = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

So multiplying

$$z = W_1 x = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + w_{14}x_4 \\ w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + w_{24}x_4 \end{bmatrix}$$

results in a 2 dimensional vector! (reduced dimension). So $W \in \mathbb{R}^{d \times D} x \in \mathbb{R}^D$ results in a d-dimensional vector. **Remember**: The number of columns in any matrices A must equal to the number of rows in B, else the matrix multiplication won't work. That's why the dimension of W are defined like that.

The latent vector is $y = \sigma(W_1 x + b_1)$ (compressed representation). In a classical autoencoder, since the purpose is to reconstruct x from itself, the classical loss function is chosen to be, for a dataset $\{x_i\}_{1 \leq i \leq N}$:

$$\mathcal{L}_{AE}(\theta) := \frac{1}{N} \sum_{i=1}^{N} ||\hat{x}_i - x_i||^2$$

Explanation

You may be wondering why θ is passed since no parameters are used in the equation but remember that $\hat{x}_i = W_2 \sigma(W_1 x_i + b_1) + b_2$. The reconstruction error $||\hat{x}_i - x_i||^2$ depends on $\hat{x}_i$ which in turn depends on θ.

Remember

$$||v||^2 = v_1^2 + v_2^2 + \ldots v_D^2$$

So here, it's

$$(\hat{x}_{i1} - x_{i1})^2 + (\hat{x}_{i2} - x_{i2}) + \ldots$$

where i refers to the i-th data point in the dataset. The subscript 1 or 2 etc refer to the first component of the vector x_i

Example

Suppose you have a dataset of 3 data points

$$x_1 = [1.0 \quad 2.0 \quad 3.0 \quad 4.0], \quad x_2 = [5.0 \quad 6.0 \quad 7.0 \quad 8.0], \quad x_3 = [9.0 \quad 10.0 \quad 11.0 \quad 12.0]$$

So for example $x_{11} = 1.0, x_{23} = 5.0, x_{32} = 10.0$

Back to the script (formula)

where θ denotes the parameters of the network (such as W_1, b_1, W_2, b_2 So $\theta = \{W_1, b_1, W_2, b_2\}$).

Minimizing $\mathcal{L}_{AE}$ will force $\hat{x}$ to be as similar as possible to x, while being represented by a vector y of lower dimension. (the last part here means that the autoencoder doesnt just reconstruct x directly. Instead, it first compresses x into latent space y)
Apart from dimension reduction, autoencoders can be used for other purposes, for instance image denoising. **In this setup** (when using autoencoders for image denoising), we perturb ($\sim$ disturb) each data point x_i by some random noise ϵ_i (usually Gaussian) and learn to reconstruct x_i from the corrupted image $x_i + \epsilon_i$ by minimizing

$$\mathcal{L}_{AE} := \frac{1}{N} \sum_{i=1}^{N} ||\hat{f}(x_i + \epsilon_i) - x_i||^2$$

($\hat{f}(x_i + \epsilon_i)$ just returns a reconstructed *denoised* image). By minimizing L_{AE} the autoencoder learns to remove noise and reconstruct the original image!
While usually, autoencoders verifies the bottleneck property (latent representation) $d << D$, it may be interesting to allow for $d \geq D$ (latent space can be larger than input space) while forcing some sparsity on y.

What does sparsity mean?

Sparsity (=spärlich) means most of the elements in the latent vector y are **zero** or close to zero
This forces the autoencoder to **focus** on a small number of **important** features in the data, rather than using all available dimensions.

Back to the script

This can be achieved by taking the loss $\mathcal{L} = \mathcal{L}_{AE} + \mathcal{L}_{sp}$ ($\mathcal{L}_{sp}$ is the sparsity penalty) penalizes large values $y_i = \sigma(W_1 x_i + b_i)$. For instance,

$$\mathcal{L}_{sp}(\theta) = \frac{1}{N} \sum_{i=1}^{N} ||y_i||_1$$

leads to the L^1-sparse autoencoder.

Explanation

$||y_i||_1$ Is the sum of absolute values of the elements in y_i. y_i is the **latent representation** of the i-th data point. $||y_i||_1$ is the L^1-norm of y_i, which is the sum of absolute values of its elements.
For example (values are just imaginary): $||y_i||_1 = |0.1| + |-0.3| + |0.0| + |0.7| = 1.1$
Minimizing $\mathcal{L}_{sp}$ encourages y_i to have many zeros.

L_{sp} is the average L^1-norm over all data points in the dataset (bc we average $||y_i||_1$ over all data points!)

3.3.1 Linear autoencoder

In this section, we treat the case of linear autoencoder, i.e. the case where σ is the identity function ($\sigma(z) = z$). This case may seem uninteresting at first glance, but we will see that it nicely links to a previously seen algorithm.
In this particular case, the network reduces to

$$\hat{f}(x) = W_2\sigma(W_1x + b_1) + b_2$$

Since $\sigma(z) = z$, just multiply everything out:

$$\hat{f}(x) = W_2W_1x + W_2b_1 + b_2$$

while the loss is

$$\mathcal{L}_{AE}(b_1, b_2, W_1, W_2) = \frac{1}{N}\sum_{i=1}^{N}||W_2W_1x_i + W_2b_1 + b_2 - x_i||^2$$

(Notice that above we had θ but here since σ is the identity function we can just substitute $\hat{x}_i$ with our network)
and we aim at solving

$$\min_{b_1, b_2, W_1, W_2} \mathcal{L}_{AE}(b_1, b_2, W_1, W_2)$$

We want to make the optimization easier, first we eliminate b_2 because this reduces the number of variables we need to optimize.
First, by taking the gradient w.r.t b_2 and using the optimality condition $\nabla_{b_2}\mathcal{L}_{AE} = 0$, we get that $b_2 = \overline{x} - W_2W_1\overline{x} - W_2b_1$.

Gradient calculation

$$\nabla_{b_2}\mathcal{L}_{AE} = \frac{2}{N}\sum_{i=1}^{N}(W_2W_1x_i + W_2b_1 + b_2 - x_i)$$

Setting the gradient to zero:

$$\frac{2}{N}\sum_{i=1}^{N}(W_2W_1x_i + W_2b_1 + b_2 - x_i) = 0$$

Simplifying:

$$\sum_{i=1}^{N}(W_2W_1x_i + W_2b_1 + b_2 - x_i) = 0$$

Splitting the sum:

$$W_2W_1\sum_{i=1}^{N}x_i + W_2b_1N + b_2N - \sum_{i=1}^{N}x_i = 0$$

Let $\overline{x} = \frac{1}{N}\sum_{i=1}^{N}x_i$ (mean of data points, then)

$$W_2W_1(N\overline{x}) + W_2b_1N + b_2N - N\overline{x} = 0$$

Dividing by N:

$$W_2W_1\overline{x} + W_2b_1 + b_2 - \overline{x} = 0$$

Solving for b_2:

$$b_2 = \overline{x} - W_2W - \overline{x} - W_2b_1$$

Back to the script

Thus the problem is reduced to

$$\min_{W_1,W_2}\frac{1}{N}\sum_{i=1}^{N}||W_2W_1(x_i - \overline{x}) - (x_i - \overline{x})||^2$$

Expanded version

$$\min_{W_1,W_2}\frac{1}{N}\sum_{i=1}^{N}||W_2W_1x_i + W_2b_1 + (\overline{x} - W_2W_1 - \overline{x} - W_2b_1) - x_i||^2$$

(substituted b_2 here into original loss function)
Notice that

$$W_2W_1x_i + W_2b_1 + \overline{x} - W_2W_1 - \overline{x} - W_2b_1 - x_i$$

needs to be simplified. Notice that W_2b_1 and $-W_2b_1$ cancel out:

$$W_2W_1x_i + \overline{x} - W_2W_1\overline{x} - x_i = W_2W_1x_i - W_2W_1\overline{x} + \overline{x} - x_i$$

Factoring out we get

$$W_2W_1(x_i - \overline{x}) + \overline{x} - x_i$$

We can rewrite the last two terms and thus get

$$W_2W_1(x_i - \overline{x}) - (x_i - \overline{x})$$

Back to the script

$$\min_{W_1,W_2}\frac{1}{N}\sum_{i=1}^{N}||W_2W_1(x_i - \overline{x}) - (x_i - \overline{x})||^2$$

Note that both the dependency in b_2 and b_1 have disappeared. Now let $X_0 \in \mathbb{R}^{D \times N}$ be the matrix having $x_i - \overline{x}$ as the i-th column. The previous optimization problem can be rewritten as (dropping the $1/N$ factor because scaling doesn't affect the minimization)

$$\min_{W_2} \min_{W_1} ||W_2 W_1 X_0 - X_0||^2_{\text{Fro}}$$

We now aim at finding a closed-form solution of the inner problem

$$\min_{W_1} ||W_2 W_1 X_0 - X_0||^2_{\text{Fro}}$$

Let $\phi(W_1) = ||W_2 W_1 X_0 - X_0||^2_{\text{Fro}}$. We recall that the gradient of ϕ at W_1 is defined as the matrix $\nabla \phi(W_1)$ such that for all $H \in \mathbb{R}^{d \times D}$, $D\phi(W_1).H = \langle \phi(W_1), H \rangle_{\text{Fro}} = Tr(\nabla \phi(W_1) H^T)$. Going back to the definition of the differential, we can show that

$$\nabla \phi(W_1) = 2X_0(X_0 - W_2 W_1 X_0)^T W_2$$

Hence, the optimality condition is $X_0 X_0^T W_2 = X_0 X_0^T W_1^T W_2^t W_2$. Remark that the matrix $X_0 X_0^T$ is the covariance matrix of the data. Under the assumption that this matrix is invertible, the previous optimality conditions reduces to

$$W_2^T = W_2^T W_2 W_1$$

Lemma 3.3.1

Let $A \in \mathbb{R}^{n \times m}$. Then

$$A^T = A^T A A^\dagger$$

Proof: Write the SVD $A = U \Sigma V^T$
First calculate $A^T A$

$$A^T = (U \Sigma V^T)^T = (V \Sigma^T U^T), \quad A = (U \Sigma V^T)$$
$$A^T A = (V \Sigma^T U^T)(U \Sigma V^T) = V \Sigma^T \Sigma V^T$$

Now compute $A^\dagger$

$$A^\dagger = (V \Sigma^\dagger U^T)$$

$$\begin{aligned} A^T A A^\dagger &= (V \Sigma^T \Sigma V^T)(V \Sigma^\dagger U^T) \\ &= V \Sigma^T \Sigma \Sigma^\dagger U^T \end{aligned}$$

Now lets look at $\Sigma^T \Sigma \Sigma^\dagger$
We know that $\Sigma^\dagger$ is $\frac{1}{\sigma_k}$ on its diagonal and Σ is σ_k on its diagonal, thus $\Sigma \Sigma^\dagger = I$ and we get:

$$A^T A^\dagger = V \Sigma^T U^T = A^T$$

The previous lemma allows us to take $W_1 = W_2^\dagger$, which further reduces the problem to

3.3.1

$$\min_{W_2} ||W_2 W_2^\dagger X_0 - X_0||_{\text{Fro}}^2$$

Proposition 3.3.1

3.3.1 is equivalent to

3.3.2

$$\max_{W_2} Tr(W_2^\dagger X_0 X_0^T W_2)$$

Proof: Developing:

$$||W_2 W_2^\dagger X_0 - X_0||_{\text{Fro}}^2 = ||X_0||_{\text{Fro}}^2 - 2\langle X_0, W_2, W_2^\dagger X_0\rangle_{\text{Fro}} + ||W_2 W_2^\dagger X_0||_{\text{Fro}}^2$$

we can rewrite $\langle X_0, W_2, W_2^\dagger X_0\rangle_{\text{Fro}} = Tr(W_2 W_2^\dagger X_0 X_0^T) = Tr(W_2^\dagger X_0 X_0^T W_2)$
In a similar way,

$$\begin{aligned} ||W_2 W_2^\dagger X_0||_{\text{Fro}}^2 &= Tr((W_2 W_2^\dagger X_0)(W_2 W_2^\dagger X_0)^T) \\ &= Tr(W_2 W_2^\dagger X_0 X_0^T (W_2^\dagger)^T W_2^T) \\ &= Tr(W_2^\dagger X_0 X_0^T (W_2^\dagger)^T W_2^T W_2) \\ &= Tr(W_2^\dagger X_0 X_0^T W_2) \end{aligned}$$

where we **used the fact that** $(W_2^\dagger)^T W_2^T W_2 = W_2$ using the previous Lemma. Hence, 3.3.1 can be rewritten as

$$\min_{W_2} ||X_0||_{\text{Fro}}^2 - Tr(W_2^\dagger X_0 X_0^T W_2)$$

which is equivalent to 3.3.2
We now restrict to the extreme case where $d = 1$. In this case $W_2 = w_2 \in \mathbb{R}^D$.
By noticing that for all $w_2, Tr(w_2^\dagger X_0 X_0^T w_2) = ||w_2 w_2^\dagger X_0||_{\text{Fro}}^2 \geq 0$, we can assume that $w_2 \neq 0$. In this case, $w_2^\dagger = \frac{1}{||w_2||_2^2} w_2^T$ and thus the problem becomes

$$\max_{w_2 \in \mathbb{R}^D \setminus \{0\}} \frac{w_2^T X_0 X_0^T w_2}{||w_2||_2^2}$$

which is equivalent to

$$\max_{||w_2||_2 = 1} w_2^T X_0 X_0^T w_2$$

As we have seen in the exercises, the optimal w_2 is nothing else that the first eigenvalue of the covariance matrix $X_0 X_0^T$. In other words: a linear autoencoder performs PCA!

With a little more work it is possible to show that this is actually the case for all $d \in \{1, \ldots, D\}$. Hence, a general (nonlinear) autoencoder can be seen as a generaliztion of the PCA