Notes on Data Analysis and Machine Learning Tools

Nicky D. van Foreest

September 21, 2026

This work is licensed by the University of Groningen under a Creative Commons Attribution-ShareAlike 4.0 International License.

Contents

Chapter 1

Introduction

These notes teach some parts of *data analysis* and *machine learning* by letting the chapters follow the steps of a data science project, whatever its subject. In the earlier chapters we start with small problems such as matching two translations of the same text, the Wordle game,[1] and making a tool that can plan pleasant walking routes. It is not our aim to cover all of data science and machine learning; these fields are too large to cover in a single course. We develop the machinery these problems need, and no more. We hope that the reader remembers the games, and from the games the techniques we used to solve them.[2] No two projects meet quite the same difficulties, and certainly not all of the ones described here, so we hope too that the reader can abstract from our examples.

[1] Wikipedia: Wordle.

[2] Toy problems make good mnemonics for a general technique.

Along the way we invest in making formal representations of the problems we deal with. Well-designed representations help us to formalize what we want to achieve and to write good pseudocode to compute solutions.

For ease, we will reference nearly only to Wikipedia pages. These pages contain further references if you want to pursue a topic in more detail.

DATA SCIENCE (DS) is the practice of using data to answer empirical questions.[3] Such questions can be:

[3] Wikipedia: Data science.

1. *Descriptive*. What is in the data? For example: How many employees reported illness periods that lasted more than two weeks?
2. *Predictive*. What will happen for a new case? For example: Given a patient's disease and age, how long will this patient remain ill?
3. *Causal*. What would change under an intervention? For example: What would be the effect of a salary cut on the number of sick-leave days?

A data science project should start by *asking the right question*; this includes investigating the problem and settling what a *good answer would be*. After formulating the question, the next task is to analyze how to answer it: what data must be assembled, what must be measured, how records should be stored. Sometimes the available database is enough; sometimes we need additional measurements,[4] or a different way to store future records.

[4] Such as a questionnaire or an experiment.

Raw data rarely contain exactly the quantities needed for the question. A first step is to construct useful *variables* from the raw fields. For instance, from dates one may construct the duration of an illness period; from repeated records one may construct the number of earlier sick-leave periods; from a text code one may construct an indicator for pregnancy-related absence. In practice, data science therefore starts with analyzing the *combination* of the question and the data. This involves checking where the data come from, what the variables mean, whether records are comparable, and whether there are missing values or obvious errors.

Simple *tables* and *graphs* are often the most useful first tools to represent and analyze data: they reveal scale, variation, outliers, and structure, and they expose errors before any formal prediction method is fitted. Real data is messy. For example, a database may report a person as ill even though the person has already returned to work, or it may code pregnancy-related absence as illness even though the underlying interpretation is different.[5] The difficult task of checking, cleaning, filtering, and interpreting the data therefore precedes model fitting.[6] Model fitting and coding matter, but they should *always* come after the harder question of whether the data represent the phenomenon we want to study and are suitable for analyzing it. We skip this step in these notes as it is too problem-specific; general advice is hard to give. Hence, we assume henceforth that data is clean.

[5] For instance, women between 20 and 40 may be reported ill for 16 weeks, while this is pregnancy leave, not illness.

[6] Often much time in data analysis is spent on these tasks and profound domain knowledge is needed.

Often, the primary data is text, and text stays hard to analyze even after cleaning. To show this problem in detail, we consider in Chapter 2 the challenge of matching two translations of the same story. We develop a *dynamic programming (DP)* framework to compute the best matching between chunks of sentences of each of the texts. DP also underlies reinforcement learning, a major branch of machine learning. Besides this, several relevant general machine learning ideas are addressed in passing.

Interestingly, machine translation itself has been one of the driving problems in machine learning. The Transformer paper *Attention Is All You Need*[7] , which led to the architecture behind GPT-style models, was focused on machine translation. Thus, these notes start with a program to match human-made translations; in Section 8.3.4 they end with a far more powerful machine learning tool for working with translated text.

[7] Wikipedia: Attention Is All You Need.

Once the question and the data have been made clear, we can ask what kind of model is useful.

MACHINE LEARNING (ML) develops and analyzes algorithms that learn patterns from data and use these patterns for tasks such as prediction and decision making.[8] In *supervised learning*, the training data contain input-output pairs.[9] In Chapter 3 we study the common supervised learning tasks: *regression*, where the output is numerical, *nominal classification*, where the output is categorical without a natural order, and *ordinal prediction*, where the output is categorical with a natural order.[10] For each category we say what counts as a good prediction by choosing a *loss function*. Fitting is then the problem of finding a predictor that makes the loss small on the data. The chapter also takes up the classical themes of ML. First, *overfitting* and *regularization*. Second, the division of the data into a *training set* to fit on, a *validation set* to choose between

[8] Wikipedia: Machine learning.

[9] Other branches exist: in *unsupervised learning* the data contain no labels, and the goal is to find structure such as clusters or low-dimensional summaries; in *reinforcement learning* an agent learns from rewards obtained by interacting with an environment. These notes focus on supervised learning.

[10] Wikipedia: Supervised learning. Wikipedia: Nominal category Wikipedia: Ordinal data.

candidate rules that have been trained with different hyperparameter settings, and a *test set* that is used only at the very end to evaluate whether the found predictor works also on new data. The last two are *hold-out sets*, which means that they are disjoint from the training data. Third, *imbalanced data*, where one class is so much rarer than the other that a rule can look accurate while never predicting it. If one in a thousand screened patients carries a certain disease, the rule that declares everybody healthy is 99.9% accurate, and probably useless.

SUPERVISED LEARNING REQUIRES LABELED DATA. But where do the labels come from? Sometimes we are lucky and the observations of interest carry labels already. More often they do not, and the labels must be acquired from humans, sensors, or historical records: a radiologist labels X-rays, employees read and classify the feedback on customer evaluation forms, a farmer walks around in a field to check whether the plants flagged as weeds in drone photographs really are weeds. The list is long, and none of this is free. On the contrary, labeling is often the most expensive part of a project.

BUILDING A MACHINE capable of making predictions is the next step. In Chapter 4 we build *decision trees* and their generalization *random forests*, which remove the shortcomings of single trees. This class of predictors is versatile: trees cope with large amounts of data and with mixed types of features, they are robust, and they remain easy to understand.

As in the earlier chapters, a concrete application drives our interest in random forests. We wanted to build a walking-route planner that, given a start point A and an end point B on the map of the Netherlands, produces a nice route between A and B.[11] As it happens, the most expensive steps of a typical data science project have already been done for us. *OpenStreetMap (OSM)*[12] stores edges with features for the entire Netherlands. As labels we can use published tracks, which are also in OSM. With this information we can train a random forest to rank edges from good to bad.

However, a ranking is not yet enough. Routing algorithms need edge costs to find the cheapest route. A difference in rank does not automatically correspond to a real cost difference in how pleasant an edge is to walk. The subject of Chapter 6 is to turn the output of a random forest, i.e., a ranking score, into a cost. We will discuss in particular *isotonic regression*, which is a common technique in statistics.

The last step of a data science project is to put the predictor to work, and to show what it produces, so that the user can judge whether the answer is any good. In the walking app the edge costs go to OSRM,[13] which computes the cheapest route, and a Leaflet map draws that route in the browser. The result can be enjoyed as the webapp *De Wandelvogel*.[14] Chapter B contains the technical details.[15]

END OF 2022 THERE WAS A BREAKTHROUGH. Suddenly machines, i.e., *large language models (LLMs)*, became available that help us do all of the above steps. We can discuss with them to clarify the question and what a good answer would look like. They write all the code necessary to make simple tables for statistical analysis, and simple graphs to obtain insight into outliers and data errors. They do not mind when data is not clean,

[11] This is not the same thing as the shortest path.

[12] Wikipedia: OpenStreetMap.

[13] Open Source Routing Machine.

[14] De Wandelvogel.

[15] That we delegate this step to an appendix does not make it unimportant. It is the counterpart of the first step: having settled with the user what the right question is and what an answer should look like, this is where the answer is finally delivered. However, it is fairly technical; in these notes we concentrate on the ML techniques.

they can deal with ambiguity in language, and they seem to be able to make good translations.[16] What they are capable of is impressive.[17]

In the last two chapters of these notes we explain how neural networks work, because LLMs are neural networks with an enormous number of tunable parameters. Then we discuss the particular architectures of LLMs. We will not touch on the question of why LLMs, which essentially are just machines that are trained to predict the next word[18] of a sequence, are able to do all these tasks.

[16] At least much better than Google translate before 2024.

[17] But see my remarks below.

[18] More precisely: a token.

PREDICTION IS NOT CAUSATION. An econometrician may ask whether x predicts y, but also whether changing x would cause y to change. The distinction between ML and causal models is not a distinction between formulas. Ordinary least squares (OLS) can be used as a prediction method, as an econometric model for estimating an interpretable parameter, or as one component of a causal analysis. The difference lies in the question and in the assumptions. A predictive analysis asks whether a rule predicts well for new observations. An econometric estimation problem asks what parameter is being estimated and how uncertain that estimate is. Causal models are concerned with interventions, not only with prediction.[19] A *causal model* asks what would happen to Y if a treatment D were set to 1 rather than 0.

[19] Wikipedia: Causal inference.

The basic difficulty is that we cannot observe both outcomes for the same unit. If a worker receives a training course, we can observe the worker's wage after the course, but not the wage that same worker would have had without the course. A clean solution is *random assignment*: if some workers receive the course by lottery and others do not, then the two groups are comparable on average before treatment. The difference in their average outcomes can then be interpreted as a causal effect.

Prediction and causality can diverge. A variable can be useful for predicting Y without having the direct causal effect suggested by the prediction: often a third variable, a *confounder*, drives both. For example, houses with more bedrooms tend to sell for higher prices, but this does not mean that changing the living room of a given house into an extra bedroom would raise its price. So, the number of bedrooms in a house correlates with the price, but the confounder is the size of the house, as larger houses tend to have more bedrooms. Conversely, a causal effect can be important even if it adds little predictive power. Prediction accuracy alone therefore does not establish causality.

TWO THEMES UNDERLIE THESE NOTES. The first is *optimization*. Nearly every problem in these notes is formulated as an *objective*, that is, a number saying how good a candidate solution is, together with the set of candidate solutions.

The second theme is *recursion*. A hard problem is reduced to a smaller problem of exactly the same form, and the reduction is then applied to what remains. Dynamic programming aligns two texts by aligning two shorter texts. A guess in Wordle leaves a smaller set of candidate words. A tree splits its data in two and grows a subtree on each part. The algorithm for isotonic regression merges two neighboring blocks and starts afresh on the coarser partition. Backpropagation to train neural networks computes the error at one layer from the error at the layer that follows it.

The intended reader of these notes is a student with some background in linear algebra, probability and statistics.

WHILE DEVELOPING THIS DOCUMENT I used two LLMs[20] . I took the role of architect, and the LLMs as discussion partners and builders. I started by telling them that I wanted to use random forests to make a routing app and took it from there. With these tools I learned many interesting topics, such as probability calibration.

[20] Claude and ChatGPT.

In the writing process I noticed some interesting things. These LLMs are very capable but need a lot of coaching at the same time. If left unguided, they can go south. I needed to work very hard on keeping a good overall structure, and stay alert, so as not to be misled by the confident way in which the LLMs write and argue. However, once I managed to get the structure clear, they helped fill in many details, making figures, searching the Internet for literature, answering small questions, repairing typos, finding accurate vocabulary, and checking the mathematics.

I also noticed that at times the language of LLMs can be annoying. LLMs seem to have learned from a lot of marketing text, which makes their language slick and blabby, sometimes to the extent that I had no clue what it would mean. I had to reorganize and rewrite *every sentence* of this set of notes.

WITH RESPECT TO CODING after more than 30 years of programming, I now discuss coding concepts with LLMs in the form of pseudocode because I still like the underlying elegant algorithmic ideas.[21] Once both of us were satisfied, I followed a *test-driven development* paradigm. That means that before writing the actual code, the coder develops lots of tests. Once the test suite is somewhat complete, the coder builds code until the first test passes, then the second, and so on, until all tests pass.[22] Thus, I discussed in words with one LLM what kind of tests were necessary; then I let the LLM write the tests and another LLM comment on them, and back and forth until both LLMs were satisfied. Finally I let an LLM port the pseudocode to python and work until all tests passed.

[21] I am not so keen on reading production code as this includes many checks and offers many options. Such details are important but often hinder the understanding of the basic ideas.

[22] The code counts then as correct, or at least as correct as the test suite is complete.

1.1 Exercises

Exercise (C) 1.1.1. A company keeps records of the sick-leave periods of its employees. Consider three questions: how many employees reported an illness period longer than two weeks; how long a patient of a given age and disease will remain ill; what a salary cut would do to the number of sick-leave days. Classify each as descriptive, predictive, or causal, and say why the third needs an assumption that the other two do not.

Exercise (C) 1.1.2. The sick-leave database stores one row per reported absence, with the columns employee id, start date, end date, absence code, and date of birth of the employee. The company wants to predict how long an employee who reports ill will stay away from work, and no column of the database holds that number.

Define three quantities that can be computed from these columns: the one to be predicted, and two that can serve as features for that prediction. For each, say which columns it is computed from, and how many rows are needed.

Exercise (C) 1.1.3. A company wants to predict how long an employee who reports ill will stay away from work, and fits a model on its sick-leave records. The database codes pregnancy leave as illness, so women between 20 and 40 appear in those records with absences of 16 weeks. Nobody repairs this before the model is fitted.

What does the model predict for a woman of 30 who reports ill with the flu, and why is that prediction wrong although the model reproduces the records faithfully? Why does a high accuracy on held-out records fail to bring the error to light?

Exercise (T) 1.1.4. One in a thousand screened patients carries a certain disease. Compute the accuracy of the rule that declares every patient healthy. What is this rule worth, and which quantity exposes it?

Exercise (C) 1.1.5. What is the validation set for, what is the test set for, and what is lost when the test set is used to choose between candidate rules?

Exercise (C) 1.1.6. Name the task type of each of the following: predicting how long a patient remains ill; sorting customer feedback forms into topics; grading a review as poor, average, or good. For each, say what a loss function has to charge for.

Exercise (C) 1.1.7. Why is labeling often the most expensive part of a data science project? Give two examples from this chapter, and say in each case who or what produces the label.

Exercise (C) 1.1.8. Houses with more bedrooms sell for higher prices, yet turning the living room of a given house into an extra bedroom will not raise its price. Name the confounder.

Consider next a company that offers its workers a training course, and ask whether the course raises a worker's wage. Of a worker who took the course we observe the wage afterwards, but not the wage that same worker would have earned without it, so the two outcomes are never both available. Explain why, when the course is handed out by lottery, the difference between the average wage of the workers who took it and the average wage of those who did not can nevertheless be read as the effect of the course. Explain also what could go wrong when the workers themselves decide whether to take it.

Chapter 2

Parallel Translations with Dynamic Programming

The problem that underlies this chapter is entirely practical. In 2018 I started learning French for fun. As, at that time, Google Translate was not very good and LLMs were not yet publicly available, I had to use an (electronic) dictionary to look up words, which is a chore, and in harder texts I sometimes lost the meaning of the whole sentence. To make my life a bit easier, I looked up a French text on the internet with an English translation.[1] Then, my idea was that by splitting these texts into sentences and then putting the sentences side by side, I could rely on my understanding of English to re-engineer the meaning of the French text.

So, I picked a text[2] and its translation, and I started to make a matching, but that quickly became boring. My next idea was to delegate this task to a computer.[3] However, this turned out to be quite a bit more challenging than I initially thought. Simply placing two human translations next to each other, sentence by sentence, does not work: translators split, merge, and omit passages,[4] so naive layouts were not useful for my learning purposes.

However, there is structure in texts. Even though human translators insert variation, they do not reorder the whole story: the first paragraph of a French text will not correspond to the last paragraph of the English translation. This means that we can look for an alignment that mostly moves forward through both texts. Then, for each text element in one language, find (chunks of) elements of the other text that are similar. Once the entire matching is done, print these chunks side by side. In more abstract terms, if we can find a *cost* that expresses how much the chunks in each text resemble each other, we can use *dynamic programming (DP)* to make a sequential matching that achieves the lowest overall cost. It seems natural to use *text features*, such as person names, as cost structure. After I had this insight, I reformulated my goal: extract relevant features, define a cost structure, and use dynamic programming to make a good pairing[5] between the elements of a source text and its translation.

THIS IS ALL VERY INTERESTING, but what has this to do with data analysis and ML? Let's spell that out. First, Section 2.1 explains why text analysis[6] in itself is difficult. Second, it is often necessary to summarize text in terms of features.[7] When matching translations, we are confronted with both aspects of text, so our problem is an example of this common part

[1] Of course, these translations were made by humans, not by an LLM.

[2] Grimm Stories, *Hansel and Grethel*.

[3] Computers excel at boring and repetitive tasks.

[4] Sometimes even entire paragraphs.

[5] We should not insist on 'perfect'; that is impossible with human-made translations.

[6] Text analysis is one of the principal forms of data analysis, think for instance about online retailers analyzing customer feedback.

[7] In the case of customer feedback, was the customer happy or not about the product.

of data analysis.

Dynamic programming is a fundamental algorithmic technique. To apply it to our project, we need to introduce many concepts that reappear throughout machine learning. Next, DP is important because reinforcement learning is one of the most active areas of current ML research, and reinforcement learning grew out of DP; we return to the differences below.

As a matter of fact, while we focus here on matching translations, matching related sequences is a very common problem. In DNA sequence alignment, one aligns two strings while allowing insertions, deletions, and substitutions. In speech recognition and part-of-speech tagging, one searches for the most likely hidden sequence behind observed data.[8] In time-series analysis, dynamic time warping aligns two signals that progress at different speeds. Yet another example is plagiarism detection. We have an original text, and the plagiarized text. The perpetrator will not copy all of the original text, and probably insert some new text. We can use the tools we develop in this chapter to find plausible matchings, hence proof (or disproof) of plagiarism. A final example is tree rings dating in wood. In good years the tree rings are thicker than in bad years. By comparing the sequence of thick and thin rings of some piece of wood to a reference, it is possible to find the age of some piece of wood, which was perhaps used in an old boat.

[8] Wikipedia: Hidden Markov model and Wikipedia: Viterbi algorithm.

The example of text matching also illustrates a point that recurs throughout ML. Methods should offer solutions that are good *on average*. In the case of matching texts, rather than attempting to clean each sentence and pair it with its counterpart, we let the DP find an alignment that is good overall. For our purpose that is enough, so that we can avoid the arduous cleaning task altogether.[9]

[9] In this sense, this illustrates the first step of a data science project: our main question is how to get an alignment that works reasonably well for humans.

There are also choices that the dynamic program does not determine by itself. We have to choose which features to use for scoring a text element, how many features to include, how to weight them, and how large a chunk the algorithm is allowed to match at once. These are *hyperparameters*: design choices outside the optimization problem that have to be tuned by inspecting the result. Tuning hyperparameters is a central part of ML practice.

2.1 Text Analysis Is Difficult

We discuss some examples to show that human text in general, and translations in particular, are difficult.

HUMAN TEXT IS UNSTRUCTURED, and replete with ambiguity; often the sense of a (group of) words depends essentially on the context. Here is a list of funny examples.

1. For typos, consider the word *occurrence*; here are some variations: *occurence*, *occurance*, *ocurrence*.

2. The abbreviation *St.* can mean *Saint* or *Street*: *Saint Anthony Hospital* may also be written as *St. Anthony Hospital*. *Data normalization* is a method to standardize naming in databases.

3. The same meaning can be expressed with different words: *The train left at noon.*, *The train departed at twelve o'clock.*, and *At midday the train pulled out of the station.* A program that matches on common words sees three unrelated sentences.

4. The same words can carry different meanings. *May may arrive in May.*, *I sat on a bank near the river bank.*, *The bass player caught a bass.*, *They lead the expedition that mines lead.*

5. One expression can have opposite meanings: *It's a hell of a film.*, *It's a hell of a mess.*

6. One place name can refer to different places: *Paris, France* and *Paris, Texas.*

7. One place can have several names: *New York*, *New York City*, *NYC*, or even *the Big Apple.*

8. A word can change role: *Mr. Baker is a baker and lives on Baker Street.*

9. Two strings can have exactly the same words, but a different meaning: *This is good, not bad at all.* or *This is bad, not good at all.*[1]

10. Finally, it is surprisingly hard to say where a sentence ends: *Dr. P.A.M. Smith Sr. arrived at 2 p.m. When he met a friend, Prof. H. Jones, M.D., Ph.D., at the Mass. Inst. of Tech., he said: "Hi, Dr. Jones! How are you?".*

[1] This example is taken from Bishop and Bishop, *Deep Learning: Foundations and Concepts*.

So, even before we start matching translations, we see that ordinary text defeats many simple rules for automatic matching and parsing. For decades, people have used tools based on regular expressions such as `grep`.[2] These tools are extremely useful when the pattern is clear, but they can fail badly when the meaning depends on context.[3] In general, manually designed text-processing rules, such as rules for search and replace, are fragile, even when they look reasonable at first.

All in all, whenever you later have to work on human text, we want you to remember from these examples: text analysis is full of pitfalls, and 'home-brewed' rules are usually naive, and often wrong altogether.

[2] Related tools include `awk`, `sed`, `perl`, and `python`.

[3] Interestingly, ChatGPT and Claude are very good at using such tools.

TRANSLATION ADDS ANOTHER LAYER OF DIFFICULTY. At the word level, names change spelling across languages: *Gretel* versus *Grethel* in the fairy tale. Next, translators may replace an explicit name by a description. In *Hansel and Grethel*, the source may say that Hansel stooped down, while the translation says that *the boy* stooped down; the reader has to infer that the boy is Hansel. A source may first describe the children as a *boy* and a *girl*, while a translation may describe them as the *elder* and the *younger*. Consequently, exact word matching is too brittle to use directly for matching translations.

At the sentence level, translators split, merge, and rewrite. This means that sentence i in the source need not correspond exactly to sentence i in the target. To illustrate, here is a French version of the first sentence of *Les Mille et Une Nuits*[4] compared with three English translations and one by ChatGPT. Notice how each human translator restructures

[4] Wikipedia: Les Mille et Une Nuits.

the sentence differently, and adds or removes material. Interestingly, the translation offered by ChatGPT is nearly literal.

> Les chroniques des Sassaniens, anciens rois de Perse, qui avaient étendu leur empire dans les Indes, dans les grandes et petites îles qui en dépendent, et bien loin au delà du Gange jusqu'à la Chine, rapportent qu'il y avait autrefois un roi de cette puissante maison, qui était le plus excellent prince de son temps. —A. Galland

> The chronicles of the Sassanids, the ancient kings of Persia, who had extended their empire into India, across the large and small islands belonging to it, and far beyond the Ganges as far as China, relate that there once lived a king of that powerful dynasty who was the most excellent prince of his time. —ChatGPT 5.6[5]

> In the chronicles of the ancient dynasty of the Sassanidae, who reigned for about four hundred years, from Persia to the borders of China, beyond the great river Ganges itself, we read the praises of one of the kings of this race, who was said to be the best monarch of his time. —A. Lang

> It is related (but God alone is all-knowing, as well as all-wise, and almighty, and all-bountiful,) that there was, in ancient times, a King of the countries of India and China, possessing numerous troops, and guards, and servants, and domestic dependents: and he had two sons; one of whom was a man of mature age; and the other, a youth. —E.W. Lane

> Therein it is related (but Allah is All knowing of His hidden things and All ruling and All honored and All giving and All gracious and All merciful) that, in tide of yore and in time long gone before, there was a King of the Kings of the Banu Sásán in the Islands of India and China, a Lord of armies and guards and servants and dependents. —R.F. Burton

[5] This translation was produced without special instructions.

At a higher level yet, translators sometimes leave out sentences, or even entire paragraphs. They might split paragraphs of the source into several target paragraphs, or merge paragraphs of the source into one paragraph in the target. And finally, the splits and merges they introduce might be in the middle of a paragraph, that is, they do not respect the boundaries of the source paragraphs at all.

This is why we should not try to find a perfect match by exact rules. Instead, we look for a matching with a low total loss, so that the alignment works well in most places. We compare chunks of text by features, such as names, punctuation, length, and other markers, and allow the matching algorithm to skip, split, and merge chunks when that gives a better overall alignment. The next section formalizes this idea.

2.2 Dynamic programming

Now that we have some understanding of the complications that arise in matchmaking, we discuss the concept of dynamic programming and then model text matching in this framework.

A DYNAMIC PROGRAMMING PROBLEM in discrete time with time horizon N contains the following elements.[1] The variable x_k represents the *state of the system* at time k.[2] The *state space* is the set of all possible states. A *control variable*[3] u_k modifies the state through the function $T_k(x_k, u_k)$ to produce the state at the next time moment

$$x_{k+1} = T_k(x_k, u_k).$$

The control u lies in a *control set* $U(x)$ that depends on the state x. Finally, the function $g(x,u)$ specifies the *running cost*[4] of exercising control u in state x, and g_N is the *terminal cost*.

Let us illustrate this with a single-item inventory model. The state x_k is the number of items in stock at the end of period k. We assume we can order any amount we like so $U(x) = \{0, 1, \ldots\}$. If we order Q units as soon as the inventory drops to or below the *reorder level* r, then $u_k = Q\,\mathbb{1}\{x_k \le r\}$. Then, with d the constant demand per period,

$$x_{k+1} = T_k(x_k, u_k) = x_k - d + u_k.$$

If placing an order costs ordering cost K and the function $h(x) \ge 0$ represents the inventory cost, then $g(x,u) = h(x) + K\,\mathbb{1}\{u \ge 1\}$. If we finish after N periods, we set the cost g_N to zero.

As the evolution is deterministic, the entire state trajectory $x_1, x_2, \ldots, x_N$ is determined by the starting state x_1, the sequence of controls $u_1, u_2, \ldots, u_{N-1}$, and the rule $x_{k+1} = T_k(x_k, u_k)$. The *total cost* of this trajectory is

$$J(x_1; u_1, u_2, \ldots, u_{N-1}) = g_N(x_N) + \sum_{k=1}^{N-1} g_k(x_k, u_k),$$

where g_N is the terminal cost.

The *value function* $J_k^*(x)$ is the minimum remaining cost from state x at time k, optimized over all future controls:

$$J^*(x) = \min_{u \in U} J(x; u),$$

where $U = (U_1, U_2, \ldots, U_N$ and $u = (u_1, u_2, \ldots, u_N)$. It can be proven[5] that the value functions satisfy the *dynamic programming equations* (*DPE*):[6]

$$\begin{aligned} J_N^*(x_N) &= g_N(x_N), \\ u_k^*(x_k) &\in \operatorname*{argmin}_{u_k \in U_k(x_k)} \left\{ g_k(x_k, u_k) + J_{k+1}^*(T_k(x_k, u_k)) \right\}, \\ J_k^*(x_k) &= \min_{u_k \in U_k(x_k)} \left\{ g_k(x_k, u_k) + J_{k+1}^*(T_k(x_k, u_k)) \right\} \\ &= g_k(x_k, u_k^*) + J_{k+1}^*(T_k(x_k, u_k^*)). \end{aligned}$$

The first equation is the *terminal condition*; the second provides the *optimal control*; the third is the *backward recursion*; the fourth follows from the second. In essence, dynamic programming combines recurrence and optimization: if we know J_{k+1}^* we can optimize to find J_k^*.

[1] Bertsekas, *A Course in Reinforcement Learning*, 2026.

[2] In this section we follow the notation that is standard in the dynamic-programming literature: x is a state. In later sections, x denotes a feature vector.

[3] Also called *action*.

[4] Or *one-step cost* or *reward*.

[5] See the many books of Bertsekas on dynamic programming.

[6] Also called *Bellman equations*.

2.3 Examples for recursion and DP

The DP code is recursive, and understanding recursion takes some getting used to. To help you practice with this way of thinking, we first discuss a few algorithms that are very elegant, but also very useful to know and understand.[1]

[1] The trees we discuss later are also defined recursively.

TWO FAMOUS RECURSIVE ALGORITHMS are Quicksort and the K-median. Quicksort takes a list and returns its elements in increasing order. K-Median takes a list and a number k, and returns the element that would stand at position k if the list were sorted; k is called the *rank* of that element. Since the element halfway the sorted list is a median, we obtain a median of an unsorted list of n elements by asking for the element of rank $\lceil n/2 \rceil$.

Both algorithms reduce a question about a list $\mathcal{S}$ to the same question about shorter lists. For this the *pivot* p of Alg. 2.3.1 is used to split $\mathcal{S}$ into a set $\mathcal{L}$ with the elements smaller than p, $\mathcal{M}$ with elements equal to p, and $\mathcal{R}$ with elements larger than p. Taking p from $\mathcal{S}$ itself ensures that the part equal to p is not empty, hence that the other two parts are both shorter than $\mathcal{S}$, so the recursion reaches a list of at most one element. We take p uniformly at random, so that no particular input, a sorted list for instance, makes the split lopsided time and again.

Algorithm 2.3.1 Choose a pivot uniformly from a finite set.

```
Input: a non-empty finite set S.
Output: an element of S.
procedure ChoosePivot(S)
    return RandomElement(S)
```

QUICKSORT SORTS THE PART below p and the part above p by the same procedure; the part equal to p needs no sorting, Alg. 2.3.2. Every element of the first part is smaller than every element of the second, and likewise for the second and the third, so the concatenation of the three sorted parts is sorted.[2]

[2] For lists u and v, the list $u+v$ consists of the elements of u followed by the elements of v.

Algorithm 2.3.2 Sort a list $\mathcal{S}$ in increasing order.

```
1:  Input: a list S.
2:  Output: the elements of S in increasing order.
3:  procedure QuickSort(S)
4:      if |S| ≤ 1 then
5:          return S
6:      p ← ChoosePivot(S)
7:      L ← [x ∈ S : x < p]
8:      M ← [x ∈ S : x = p]
9:      R ← [x ∈ S : x > p]
10:     return QuickSort(L) + M + QuickSort(R)
```

THE K-MEDIAN FINDS the element of rank k without sorting $\mathcal{S}$. After splitting $\mathcal{S}$ with a pivot p as in Quicksort, the elements of $\mathcal{L}$ take the ranks $1,\dots,|\mathcal{L}|$, the copies of p take the ranks $|\mathcal{L}|+1,\dots,|\mathcal{L}|+|\mathcal{M}|$, and the elements of $\mathcal{R}$ take the ranks above that. Comparing k with $|\mathcal{L}|$ and $|\mathcal{L}|+$

$|\mathcal{M}|$ therefore tells us in which part the element of rank k lies, without looking at the parts themselves. If $k \le \mathcal{L}$, its rank there is again k. If instead $k \le |\mathcal{L}| + |\mathcal{M}|$, it is p, and we are done. If it lies in $\mathcal{R}$, the $|\mathcal{L}| + |\mathcal{M}|$ elements that precede $\mathcal{R}$ can be discarded because all these elements are smaller than the kth, so the rank we search for drops to $k - |\mathcal{L}| - |\mathcal{M}|$.

Algorithm 2.3.3 Find the element of rank k of a list $\mathcal{S}$.

```
1:  Input: a non-empty list S and a rank k.
2:  Output: the element of S with rank k when S is sorted increasingly.
3:  procedure KMedian(S, k)
4:      p ← ChoosePivot(S)
5:      L ← [x ∈ S : x < p]
6:      M ← [x ∈ S : x = p]
7:      R ← [x ∈ S : x > p]
8:      if k ≤ |L| then
9:          return KMedian(L, k)
10:     if k ≤ |L| + |M| then
11:         return p
12:     return KMedian(R, k − |L| − |M|)
```

The median of a set can now be efficiently computed too. Alg. 2.3.4 computes the median of a list: the rank $\lceil |\mathcal{S}|/2 \rceil$ is, for a list of even length, that of the lower of the two middle elements. This is why the output is *a* median rather than *the* median.

Algorithm 2.3.4 Compute a median of a list $\mathcal{S}$.

```
Input: a non-empty list S.
Output: a median of S.
procedure Median(S)
    return KMedian(S, ⌈|S|/2⌉)
```

The shortest path problem is to find the shortest path from a set of nodes $\mathcal{N}$ to a target $t \in \mathcal{N}$ where the cost is $c(i,j) \ge 0$ for every arc $(i,j) \in \mathcal{C}$. We assume that the graph is directed and has cycles.[3]

For the DPE, the state is the node we stand on, the controls in state i are the arcs leaving i, the running cost of an arc is $c(i,j)$, and the terminal cost at t is zero. With $J^*(i)$ the cost of a cheapest path from i to t, the DPE reads

$$J^*(t) = 0, \quad J^*(i) = \min_{j:(i,j)\in\mathcal{C}} \{c(i,j) + J^*(j)\}.$$

The recursion of Alg. 2.3.5 follows this equation directly: to know $J^*(i)$ we need $J^*(j)$ for every node j that i points to. One point needs care. A node j can be pointed to by many nodes i, so a plain recursion computes $J^*(j)$ again for each of them. To prevent recomputing $J^*(j)$ time and again, we store it as soon as it is known:[4] the procedure keeps the set *seen_nodes* and the map J^*, and returns the stored value at once when the node lies in the set.

[3] If there can be cycles, then we need some more advanced algorithms.

[4] Storing the value of a procedure that has already been called on the same input is called *caching* or *memoization*. It is crucially important in recursive procedures.

Algorithm 2.3.5 Cost of a cheapest path from node i to the target t.

```
1:  Input: a node i.
2:  Output: the cost J*(i) of a cheapest path from i to t.
3:  procedure SHORTESTPATH(i)
4:      if i ∈ seen_nodes then
5:          return J*(i)
6:      if i = t then
7:          J*(i) ← 0
8:          seen_nodes.ADD(i)
9:          return J*(i)
10:     best_cost ← ∞
11:     best_next ← NONE
12:     for j such that (i, j) ∈ 𝒞 do
13:         cost ← c(i, j) + SHORTESTPATH(j)
14:         if cost < best_cost then
15:             best_cost ← cost
16:             best_next ← j
17:     J*(i) ← best_cost
18:     u*(i) ← best_next
19:     seen_nodes.ADD(i)
20:     return J*(i)
```

IN THE TRAVELLING SALESMAN PROBLEM (TSP) we are given n cities and a cost $c(i,j)$ of travelling from city i to city j, and we look for a tour that starts at city 1, visits every city once, and returns to city 1 at minimal total cost. The city we stand in is not enough to decide how to continue: where we should go next also depends on which cities we still have to visit. The state is therefore the pair $(i,\mathcal{V})$, with i the current city and $\mathcal{V}$ the set of cities not yet visited. The controls in this state are the cities $j \in \mathcal{V}$, the running cost of going to j is $c(i,j)$, and the next state is $(j, \mathcal{V}\setminus\{j\})$. Writing $J^*(i,\mathcal{V})$ for the cost of a cheapest route from i back to city 1 that visits all of $\mathcal{V}$ on the way, the DPE is

$$J^*(i,\varnothing) = c(i,1), \quad J^*(i,\mathcal{V}) = \min_{j\in\mathcal{V}} \{c(i,j) + J^*(j,\mathcal{V}\setminus\{j\})\},$$

and the cost of a cheapest tour is $J^*(1,\{2,\dots,n\})$, see Alg. 2.3.6.

Algorithm 2.3.6 Cost of a cheapest tour through n cities.

```
1:  Input: the current city i and the set V of cities still to visit.
2:  Output: the cost J*(i, V) of a cheapest route from i to city 1 visiting
    all of V.
3:  procedure TSP(i, V)
4:      if (i, V) ∈ seen_states then
5:          return J*(i, V)
6:      if V = ∅ then
7:          J*(i, V) ← c(i, 1)
8:          seen_states.ADD((i, V))
9:          return J*(i, V)
10:     best_cost ← ∞
11:     best_next ← NONE
12:     for j ∈ V do
13:         cost ← c(i, j) + TSP(j, V \ {j})
14:         if cost < best_cost then
15:             best_cost ← cost
16:             best_next ← j
17:     J*(i, V) ← best_cost
18:     u*(i, V) ← best_next
19:     seen_states.ADD((i, V))
20:     return J*(i, V)
```

TSP shows that solving the problem by brute force, that is, by computing the cost of every tour and keeping the cheapest, is not the way to go. There are $(n-1)!$ tours, against $(n-1)2^{n-1}$ states,[5] and 59! is about 10^{80} while $59 \cdot 2^{59}$ is about $3 \cdot 10^{19}$. The DP is faster because the cost to go from $(i, \mathcal{V})$ depends on the *set* of cities already visited, not on the order in which they were visited, so all $(n-1-|\mathcal{V}|)!$ orders that arrive at $(i, \mathcal{V})$ share one stored value. That cache, however, holds one number per state, so the time we save is paid for in memory: at $n = 30$ the map J^* already has more than 10^{10} entries. Brute force fails on time, the DP fails on memory, and neither settles for $n = 60$. In other words: we need heuristics to solve problems that are serious,[6] and, as we will see, not only for the TSP, but much more generally.

[5] The first element of the states runs over all states i except t, and the second element over all possible sets that do not include the city i when we are at city i.

[6] That is, not toy problems.

Now that you practiced with recursion, we can continue with the DP for text sequencing.

2.4 Model for text matching

Before we can formulate the text matching problem in the above DP terms[1] , we need some definitions. A *text* is a *list* of *elements* $(e_1, e_2, \ldots, e_n)$. In our example, the elements are the sentences that make up the text. A *chunk* is a contiguous subsequence of elements $(e_a, e_{a+1}, \ldots, e_b)$. When $a = b$ the chunk is a single element, and when $b < a$ the chunk is *empty*.

[1] States, control, cost.

Features are numerical text markers that help describe an element. They can count words or punctuation marks, record the occurrence of names, or indicate the presence of distinctive objects or animals. For each element e, we compute a feature vector $f(e) = (f_1(e), \ldots, f_p(e))$, where each component corresponds to one selected marker.

In practice, it works best to tune the features iteratively.[2]

1. Start with a few simple features, such as some names and the number of question marks.
2. Let the computer use the DP below to make a match.
3. Where the match goes wrong, look for more descriptive keywords and add these as features.
4. Run the matcher again with the augmented list, and repeat.

[2] As said, engineering hyperparameters is standard in such problems.

CASTING THE TEXT MATCHING problem in the form of a DP requires us to formulate a state x, the update function T_k, the control sets U_k, and the cost functions g_k.

We have a *left* text L, in French say, and a *right* text R, in English.[3] We write $|L|$ and $|R|$ for the *lengths* of the texts, that is, for the number of elements each contains. The state $x = (\ell, r)$ points to the first element of each text that has not yet been matched, so $1 \le \ell \le |L|+1$ and $1 \le r \le |R|+1$. The value $\ell = |L|+1$ means that the left text is used up, and $(|L|+1, |R|+1)$ is the *terminal state*.

[3] Each is assumed to be a list of sentences obtained by splitting the source text.

A control u is the matching of elements of the left and right text into chunks. We can match one element of either side to 0 or more elements of the other side. Matching one element to more than *MAX_MATCH* elements of the other side seems so implausible that we prevent it.[4] We take $MAX\_MATCH = 4$. To illustrate, if $u = (2,1)$ and $x = (\ell, r)$, then we match the chunk of elements $(\ell, \ell+1)$ of the left text with the single element (r, r) of the right text. The next state becomes

[4] This is a hyperparameter, whose value we set from our understanding of the problem.

$$T(x,u) = T((\ell, r), (i, j)) = (\ell + i, r + j).$$

We restrict the controls so that exactly one left element is matched with several right elements, or exactly one right element with several left elements; a matching of several with several is not allowed.[5] We write $U_L(x)$ for the controls that advance the left text by exactly one element, that is, those that match the left element ℓ with the elements of the right text that follow position r. Then, with $x = (\ell, r)$,

[5] This is a modeling choice rather than a necessity. It keeps the number of controls in a state proportional to *MAX_MATCH*, and a genuine many-to-many correspondence can still be approximated by two consecutive matches. The price is that a merge-and-resplit of the kind described in Section 2.1 cannot be represented in a single step.

$$U_L(x) = \begin{cases} \{(1, j) : 0 \le j \le \min\{MAX\_MATCH, |R| + 1 - r\}\} & \text{if } \ell \le |L|, \\ \varnothing & \text{if } \ell = |L| + 1. \end{cases}$$

The minimum with $|R| + 1 - r$ prevents the match from running past the end of the right text, and the condition $\ell \le |L|$ says that there has to be a left element to match at all. The lower limit $j = 0$ matters. The control $(1, 0)$ matches the left element ℓ with an *empty* chunk on the right, and this is how the model represents an element that the translator dropped. Such a move is not free: its cost compares the features of the left chunk with the zero vector, so the matcher pays the full feature mass of the skipped element. Skipping is therefore always charged, and the matcher will only do it when every alternative is worse. Likewise,

$$U_R(x) = \begin{cases} \{(i, 1) : 0 \le i \le \min\{MAX\_MATCH, |L| + 1 - \ell\}\} & \text{if } r \le |R|, \\ \varnothing & \text{if } r = |R| + 1. \end{cases}$$

Therefore, the control set is

$$U(x) = U_L(x) \cup U_R(x).$$

The running cost is based on the features of the chunks that make up the match controlled by u. The cost of a match should express how much the *chunks* of each text resemble each other.[6] For each chunk we add up, for each feature separately, the values over all its elements. For example, consider the English chunk *The dog chased the cat. The cat ran away.* and the features *cat*, *dog*, and *mouse*. The first sentence contributes $[1,1,0]$, and the second contributes $[1,0,0]$, so the chunk feature vector is $[2,1,0]$. If the corresponding French chunk is *Le chien a poursuivi le chat.*, and the French features are *chat*, *chien*, and *souris*, then its feature vector is $[1,1,0]$. More generally, let $F_L(a,b)$ be the feature vector of the chunk starting at position a of the left text up to and including element b; similarly $F_R(c,d)$ is the feature vector of the right chunk for the right elements $[c,\ldots,d]$.

[6] Recall, we match chunks, because there may not be a one-to-one correspondence, due to splitting, merging, or discarding, between the elements of the left and the right text.

For the cost we take the absolute value of the difference of the features. Clearly, some features may be more important than others, which we achieve by multiplying the contribution of the jth feature by a weight $w_j > 0$. If $T((\ell,r),u) = (\ell',r')$, then

$$g((\ell,r),u) = \sum_{j=1}^{p} w_j \left| F_{L,j}(\ell,\ell'-1) - F_{R,j}(r,r'-1) \right|.$$

Observe that g is still a function of the state and the control. The functions F_L and F_R lift these to a *feature space*. The optimizer never sees the text itself. If we add or remove features, we only have to modify F_L and F_R; the DP needs no change.

Observe that this problem needs no index k to track time. Each control advances at least one of the two positions, so the state (ℓ,r) can never return to an earlier value, in other words, the position in the two texts *is* the time index.

REINFORCEMENT LEARNING starts from the exact DP formulation[7] above, but the differences are profound. Exact DP assumes that the state is known, that $U(x)$ can be enumerated, and that the value function $J^*(x)$ can be computed for all relevant states in the state space. These assumptions do not hold in most problems of practical relevance. The state space may be too large, the control space may be too large to search exhaustively, or the state may be only partly observed. In *Markov decision processes* there are additional difficulties: the next state and the one-step cost, or reward, are random. The dynamic programming equation then contains an expectation over these random quantities. We do not pursue these concepts further here.

[7] Bertsekas, *A Course in Reinforcement Learning*, 2026, and the references therein for further discussion.

2.5 From Model to Pseudocode

The above gives a mathematical description of the matching problem. For actual computations, however, we need code. Here we specify the ideas in pseudocode. We implement the above model with the TEXT class and the TEXTDP class.

THE TEXT CLASS in Alg. 2.5.1 receives the elements of a text in one language and the names of the features that matter in that language, such as [*cat, dog, mouse*] in English and [*chat, chien, souris*] in French. Upon initialization of the class, it computes the feature matrix F, where the ith row F_i is the feature vector of element i.

Algorithm 2.5.1 A Text object.

```
Input: a list elements and a list of feature names.
Output: a text object.
class TEXT(elements, features)
  instance variables:
    F ← COMPUTEFEATUREMATRIX()        ▷ Initialize the feature matrix.
  methods:
    ELEMENTFEATURES(element)          ▷ Compute features for an element.
    COMPUTEFEATUREMATRIX()
    CHUNKFEATURES(a, b)               ▷ Add up the features in a chunk.
```

The procedure TEXT.ELEMENTFEATURES, Alg. 2.5.2, computes one row of the feature matrix F. It first converts the element to lowercase, for uniformity. Then it builds a feature vector whose first component is the length of the text element, in our case the number of characters of the sentence. The length feature is necessary: it prevents feature-empty sentences from being absorbed into a neighboring match for free. COUNTFEATURE counts the number of occurrences of a feature in an element.[1] Each of these counts is added to the list v.[2]

[1] A standard function; we will not build it.

[2] If v is a list, then $v + [z]$ denotes the list obtained by appending the single element z to v.

Algorithm 2.5.2 Feature vector of one element.

```
1: Input: an element element.
2: Output: the feature vector of element.
3: procedure TEXT.ELEMENTFEATURES(element)
4:     element ← LOWERCASE(element)
5:     v ← [LENGTH(element)]
6:     for 1 ≤ j ≤ p do
7:         v ← v + [COUNTFEATURE(element, features_j)]
8:     return v
```

The procedure TEXT.COMPUTEFEATUREMATRIX applies TEXT.ELEMENTFEATURES to every element in the list of elements given to the class at initialization. Thus row i of F is the feature vector of $elements_i$, see Alg. 2.5.3.

Algorithm 2.5.3 Compute the feature matrix.

```
1: Input: none.
2: Output: the feature matrix F.
3: procedure TEXT.COMPUTEFEATUREMATRIX
4:     F ← matrix with |elements| rows and |features| + 1 columns
5:     for 1 ≤ i ≤ |elements| do
6:         F_i ← ELEMENTFEATURES(elements_i)        ▷ Set row i of F.
7:     return F
```

The procedure TEXT.CHUNKFEATURES[3] represents a chunk by

[3] We follow the usual vector-style convention: expressions such as αv are understood componentwise, just as functions such as $\exp(x)$.

adding the feature vectors of its elements. If the chunk is empty, it returns the zero vector of length $|\mathit{features}| + 1$, see Alg. 2.5.4.

Algorithm 2.5.4 Feature vector of a chunk.

```
1: Input: indices a and b; the chunk is empty when b < a.
2: Output: the feature vector of the chunk elements_i with a ≤ i ≤ b.
3: procedure TEXT.CHUNKFEATURES(a, b)
4:     if b < a then
5:         return zero vector of length |features| + 1
6:     return Σ_{i=a}^{b} F_i                          ▷ This is a vector sum.
```

THE DYNAMIC-PROGRAMMING CLASS stores the value functions J^* and the optimal controls u^*. Its methods are the DP objects from the model: the transition T, the feasible control set U, the running cost g, the terminal cost, and the procedure for solving the DPE. The class interface is in Alg. 2.5.5.

Algorithm 2.5.5 Dynamic-programming object for text matching.

```
Input: left text L, right text R.
Output: a dynamic-programming object.
class TEXTDP(L, R)
    instance variables:
        J* ← empty map                          ▷ Optimal value function
        u* ← empty map                          ▷ Optimal controls
        seen_states ← ∅                         ▷ States for which J* is known
    methods:
        T(x, u)          ▷ Next state after applying control u in state x.
        U(x)                                    ▷ Feasible controls in state x.
        G(x, u)                                 ▷ Cost of applying control u in state x.
        G_TERMINAL(x)                           ▷ Terminal cost.
        SOLVE(x)                                ▷ Solve the DPE from state x.
        OPTIMALPATH()                           ▷ Recover the matching by following u*.
```

The transition TEXTDP.T adds the control $u = (i, j)$ to the current state $x = (\ell, r)$. The next state is therefore $(\ell + i, r + j)$, see Alg. 2.5.6.

Algorithm 2.5.6 State transition for text matching.

```
1: Input: state x = (ℓ, r) and control u = (i, j).
2: Output: the next state.
3: procedure TEXTDP.T(x, u)
4:     (ℓ, r) ← x
5:     (i, j) ← u
6:     return (ℓ + i, r + j)
```

The feasible control set $U(x)$ contains matches in which at least one side contributes exactly one element. The bounds prevent overruns beyond the two texts, see Alg. 2.5.7.

Algorithm 2.5.7 Feasible controls for text matching.

Input: state $x = (\ell, r)$.
Output: the feasible control set $U(x)$.
procedure TEXTDP.U(x)
    $(\ell, r) \leftarrow x$
    $U_L \leftarrow \varnothing$
    $U_R \leftarrow \varnothing$
    if $\ell \le |L|$ **then**
        $U_L \leftarrow \{(1, j) : 0 \le j \le \min\{MAX\_MATCH, |R| + 1 - r\}\}$
    if $r \le |R|$ **then**
        $U_R \leftarrow \{(i, 1) : 0 \le i \le \min\{MAX\_MATCH, |L| + 1 - \ell\}\}$
    return $U_L \cup U_R$

The running cost $g(x, u)$, Alg. 2.5.8, quantifies the cost of feature vector differences between the left chunk and the right chunk. This is only meaningful when the left and right chunk feature vectors have the same length, and coordinate j counts the same kind of thing on either side.

Algorithm 2.5.8 Running cost for text matching.

Input: state $x = (\ell, r)$ and control u.
Output: the running cost $g(x, u)$.
procedure TEXTDP.G(x, u)
    $(\ell, r) \leftarrow x$
    $(\ell', r') \leftarrow \mathrm{T}(x, u)$
    $v_L \leftarrow L$.CHUNKFEATURES$(\ell, \ell' - 1)$
    $v_R \leftarrow R$.CHUNKFEATURES$(r, r' - 1)$
    $c \leftarrow 0$
    for $1 \le j \le |v_L|$ **do**
        $c \leftarrow c + weights_j |v_L(j) - v_R(j)|$
    return c

The terminal cost is zero once both texts have been consumed. Any unmatched remainder has already been charged by the preceding running costs, see Alg. 2.5.9.

Algorithm 2.5.9 Terminal cost for text matching.

Input: terminal state $x = (|L| + 1, |R| + 1)$.
Output: the terminal cost.
procedure TEXTDP.G_TERMINAL(x)
    return 0

The function TEXTDP.SOLVE, see Alg. 2.5.10, solves the DPE by recursion, and caches the result as SHORTESTPATH does, Alg. 2.3.5: the same state (ℓ, r) is reached by several sequences of controls. For this purpose the procedure keeps the set *seen_states* and the map J^*. If a state has already been solved, the stored value $J^*(x)$ is returned instead of recomputed. Otherwise the procedure minimizes $g(x, u) + J^*(T(x, u))$ over feasible controls, stores both $J^*(x)$ and $u^*(x)$, and marks x as seen. Here the cycle question does not arise: every control advances ℓ or r, so a state is never revisited.

Algorithm 2.5.10 Solve the dynamic programming equation.

Input: state $x = (\ell, r)$.
Output: the optimal value $J^*(x)$.
procedure TEXTDP.SOLVE(x)
    if $x \in$ *seen_states* **then**
        return $J^*(x)$
    $(\ell, r) \leftarrow x$
    if $\ell = |L| + 1$ and $r = |R| + 1$ **then**
        $J^*(x) \leftarrow$ G_TERMINAL(x)
        seen_states.ADD(x)
        return $J^*(x)$
    best_cost $\leftarrow \infty$
    best_control $\leftarrow$ NONE
    for $u \in \mathrm{U}(x)$ **do**
        $c \leftarrow \mathrm{G}(x, u) +$ SOLVE($\mathrm{T}(x, u)$)
        if $c <$ *best_cost* **then**
            best_cost $\leftarrow c$
            best_control $\leftarrow u$
    $J^*(x) \leftarrow$ *best_cost*
    $u^*(x) \leftarrow$ *best_control*
    seen_states.ADD(x)
    return $J^*(x)$

Once the DP is solved, we use the stored controls $u^*(x)$ to construct the optimal matching of chunks. TEXTDP.OPTIMALPATH(), see Alg. 2.5.11, recovers this list.

Algorithm 2.5.11 Recover the optimal path.

Input: none.
Output: the optimal list of matched chunks.
procedure TEXTDP.OPTIMALPATH
    path $\leftarrow$ []
    $(\ell, r) \leftarrow (1, 1)$
    while $(\ell, r) \neq (|L| + 1, |R| + 1)$ **do**
        $(\ell', r') \leftarrow \mathrm{T}((\ell, r), u^*(\ell, r))$
        path $\leftarrow$ *path* $+ \left[((\ell, \ell' - 1), (r, r' - 1))\right]$
        $(\ell, r) \leftarrow (\ell', r')$
    return *path*

2.6 Example

In this section we apply the above DP to make a matching for the fairy tale of Hansel and Grethel[1] . We take French as source, and English as target. If you make the effort to compare the stories in the two languages on that site, you'll quickly see that making matches is a real challenge, as the English translation does not follow the French one sentence by sentence.[2] Getting a perfect match is simply impossible, but with the DP we can try to get the least distracting result.

With the model above, we only have to specify the feature lists and the weights to make an optimal matching, see Alg. 2.6.1. The first of the 7

[1] Grimm Stories, *Hansel and Grethel*.

[2] Both are translations of the German original, made independently of each other.

features is the length of the element e, so $f_1(e) = |e|$. The other six count how often a term occurs in e.[3] For example, with the English feature list in Alg. 2.6.1, the sentence *When they had gone a little way Hansel stood still and looked back towards the house, and this he did again and again, till his father said to him, "Hansel, what are you looking at? take care not to forget your legs."* has feature vector

$$(217, 4, 0, 0, 2, 1, 0),$$

where the first entry is the number of characters of the lowercased sentence.

The content-word features receive a larger weight than the punctuation features, because we guess that names and salient objects are more important story markers than punctuation.[4] As the above feature vector shows, character counts are much larger than term counts. By setting a weight of 0.05 we ensure that the length feature gets a small total weight. Like this, length differences still discourage empty or wildly unbalanced matches, but they do not dominate the other features.

[3] The terms are lowercase, because each sentence is lowercased before the counts are computed.

[4] We picked the hyperparameter 10 somewhat arbitrarily.

Algorithm 2.6.1 Term lists for the Hansel and Grethel example.

```
MAX_MATCH ← 4
weights ← [0.05, 1, 1, 10, 10, 10, 10]
french_terms ← [" ", "!", gretel, hansel, père, canard]
english_terms ← [" ", "!", grethel, hansel, father, duck]
```

To run the matcher on the French source and the English target, the full procedure is now short. Load the texts, split them into elements,[5] build the two TEXT objects, solve the DPE, and recover the optimal path, see Alg. 2.6.2. The result is a list of matched chunks.[6]

[5] With the `python` package `spacy`.

[6] With the LaTeX `paracol` package we make the two-column document.

Algorithm 2.6.2 Run the matcher on the French and English texts.

```
1: Input: French source text and English target text.
2: Output: a list of matched chunks.
3: french_elements ← SPLITINTOELEMENTS(french_source)
4: english_elements ← SPLITINTOELEMENTS(english_target)
5: L ← TEXT(french_elements, french_terms)
6: R ← TEXT(english_elements, english_terms)
7: dp ← TEXTDP(L, R)
8: dp.SOLVE((1, 1))
9: path ← dp.OPTIMALPATH()
10: return path
```

The final French-English parallel translation, and the Dutch-English version, are in Chapter C. Whether the result is acceptable is up to the user.[7] If it is not, the sentence splitter can be improved: the French splitter is not very good, as the result shows. We can also add more features, or enlarge the control set so that several left elements can be matched to several right elements. This would capture the merge-and-resplit that the current one-to-many controls cannot represent.

[7] For me it sufficed when I needed it.

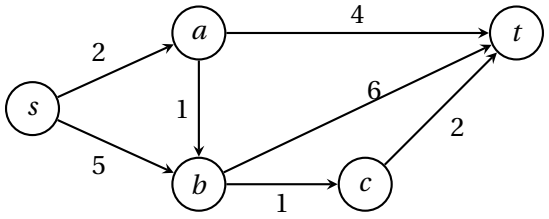
2.7 Exercises

Exercise (C) 2.7.1. What are the components of a dynamic progrogramming problem? Explain all functions involved.

Exercise (A) 2.7.2. Complete the ??? part.

```
1: Input: a non-empty list S and a rank k.
2: Output: the element of S with rank k when S is sorted increasingly.
3: procedure KMEDIAN(S, k)
4:     p ← CHOOSEPIVOT(S)
5:     L ← [x ∈ S : x < p]
6:     M ← [x ∈ S : x = p]
7:     R ← [x ∈ S : x > p]
8:     if k ≤ |L| then
9:         return KMEDIAN(L, k)
10:    if k ≤ |L| + |M| then
11:        return p
12:    return ???
```

Exercise (T) 2.7.3. Consider the following directed graph, with target node (t).



Using the dynamic-programming equation

$$J^*(t) = 0, \qquad J^*(i) = \min_{j:(i,j)\in\mathcal{E}} \{c(i,j) + J^*(j)\},$$

compute (J^*(s),J^*(a),J^*(b),J^*(c)), and give a shortest path from (s) to (t).

Chapter 3

Supervised Learning, Prediction, and Evaluation

Supervised learning uses examples (x_i, y_i), where both the features x_i and the outcome y_i are observed, to learn a prediction rule. Given the features of a new case, this rule predicts its unknown outcome. The observed outcomes provide the supervision: they let us measure prediction errors and use these errors to fit the rule.

This chapter studies prediction rules, their losses, and the classes from which we choose them. We consider regression, nominal classification, and ordinal prediction using constant rules and rules based on affine functions. After these three prediction problems, we return to regression to study how nearest neighbors and polynomial degree control flexibility. Regularization provides another way to control it. Finally, validation and test data help us choose and assess a fitted rule, including when classes are imbalanced.

3.1 Prediction, loss and risk

We formulate supervised prediction as a statistical decision problem with four components: a *random feature vector* $X \in \mathcal{X}$, an *outcome* $Y \in \mathcal{Y}$, a *prediction function* $f : \mathcal{X} \to \mathcal{A}$, and a *loss function* ℓ. The set $\mathcal{A}$ is the *prediction set*: it contains the objects our rule is allowed to output. In regression this is usually $\mathbb{R}$; in classification it can be a set of labels, a vector of class scores, or a vector of class probabilities.

Sometimes the object returned by f is not itself the final label or value we report to the user. We then use a *decision map*

$$d : \mathcal{A} \to \mathcal{Y}, \qquad \hat{y} = d(f(x)). \tag{3.1.1}$$

For example, in classification $f(x)$ may be a vector of class probabilities, while $d(f(x))$ is the class with largest probability. To compare a prediction with an observed outcome, we use a *label-lifting map*

$$r : \mathcal{Y} \to \mathcal{A} \tag{3.1.2}$$

that embeds y into the prediction set. We choose r so that $d(r(y)) = y$.[1] In regression, usually $\mathcal{A} = \mathcal{Y} = \mathbb{R}$, and both d and r are the identity.

The loss function

$$\ell : \mathcal{A} \times \mathcal{A} \to \mathbb{R}_+$$

[1] In general $r(d(a)) \neq a$: a hard label does not contain enough information to reconstruct a probability vector or score vector.

assigns a cost $\ell(r(y), f(x))$ to predicting $f(x)$ when the observed outcome is represented by $r(y)$. The order matters: losses need not be symmetric. The loss specifies the costs of different errors in the application. We will always take it to be nonnegative, and usually zero for a perfect prediction. In regression, where differences such as $y - \hat{y}$ are meaningful, common losses increase as the prediction moves farther away from the observed value. In classification, the prediction may be a label or a probability vector, so the loss need not be based on numerical distance.

Let $F_{X,Y}$ be the joint distribution of X and Y. This distribution is typically unknown, but it is useful as the population from which the training samples

$$\mathcal{T} = [(x_1, y_1), \ldots, (x_n, y_n)]$$

are drawn independently. Here x_i is a *feature vector*[2] and y_i is the *observed outcome*[3]. We write $\mathcal{T}$ as a list rather than a set because the same sample may occur more than once.[4] The order itself has no meaning; it only lets us refer to sample i. More generally, $\mathcal{S}$ denotes any finite list of samples, such as validation or test data. The jth feature coordinate of x is written $x(j)$.

The assumption that the training samples and future samples come from the same distribution matters. If the population changes after fitting, then a predictor can perform poorly on new[5] observations even when it did not overfit the training set. This problem is called *distribution shift*.

[2] Also called a vector of *predictors*, *covariates*, *regressors*, *explanatory variables*, or *inputs*. The precise name depends on the field and on whether the task is predictive or causal.

[3] Also called the *target*, *response*, *label*, *dependent variable*, or *supervisory signal*.

[4] We will later see that bootstrap samples can contain the same sample multiple times.

[5] By new we mean an observation that does not occur in the training set $\mathcal{T}$.

FOR A FIXED PREDICTION FUNCTION f, the *population risk* under the loss ℓ is

$$R(f) = \mathsf{E}\left[\ell(r(Y), f(X))\right] = \int \ell(r(y), f(x)) \, \mathrm{d}F_{X,Y}(x, y). \tag{3.1.3}$$

This is the expected loss on a new sample from the population. The *Bayes optimal predictor* for the chosen loss is

$$f^* \in \arg\min_f R(f) = \arg\min_f \mathsf{E}\left[\ell(r(Y), f(X))\right] \tag{3.1.4}$$

The value $R(f^*)$ is called the *Bayes risk*. It is the smallest risk attainable for the prediction problem under the chosen loss, hence the *irreducible loss* that remains even for an optimal predictor.

If $F_{X,Y}$ were known and the optimization over all prediction functions were feasible, the Bayes predictor would be the natural target. In practice we only have the training set $\mathcal{T}$, so we replace the population risk by the *training risk*

$$\hat{R}_{\mathcal{T}}(f) = \frac{1}{n} \sum_{i=1}^{n} \ell(r(y_i), f(x_i)).$$

This leads to the optimization problem

$$\arg\min_f \hat{R}_{\mathcal{T}}(f).$$

If we minimize training risk over all possible functions, the result can fit the training samples perfectly while performing arbitrarily poorly on new samples. We want the predictor to *generalize*: it should also have low risk on new samples drawn from the same distribution.

WE THEREFORE RESTRICT the search to a *model class* $\mathcal{F}$, a specified set of prediction functions.[6] The model class determines which rules can be fitted. A restrictive class allows only a limited range of functions; a larger class may also contain functions that fit random variation in the training sample.

[6] A model is just a function, but in machine learning the word *model* is often used for the fitted prediction rule together with its parametrization or construction method.

We begin with constant rules and rules based on affine functions. The *constant* model class is

$$\mathcal{F}_{\text{const}} = \{f_c : c \in \mathcal{A}\}, \qquad f_c(x) = c.$$

Each rule returns the same prediction for every feature vector. The prediction c can be a numerical value or a probability vector, depending on the prediction set. Decision trees also use constant rules within their leaves.

For numerical features $x \in \mathcal{X} \subseteq \mathbb{R}^p$, the class of real-valued *affine* rules is[7]

$$\mathcal{F}_{\text{aff}} = \{f_{\beta_0,\beta} : \beta_0 \in \mathbb{R}, \beta \in \mathbb{R}^p\}, \qquad f_{\beta_0,\beta}(x) = \beta_0 + \langle \beta, x \rangle .$$

[7] $\langle beta, x\rangle = \sum_i \beta_i x_i$, i.e., the inner product.

In regression, this value is the prediction itself. In classification, we use affine functions as scores and transform them into probabilities, as described below. The resulting probability predictions are generally not affine functions of x. Later chapters consider more flexible model classes, such as trees and neural networks.

The restriction of the optimization to $\mathcal{F}$ gives two more prediction rules besides the Bayes predictor:

$$f^*_{\mathcal{F}} \in \operatorname*{arg\,min}_{f \in \mathcal{F}} R(f), \qquad f^*_{\mathcal{T}} \in \operatorname*{arg\,min}_{f \in \mathcal{F}} \hat{R}_{\mathcal{T}}(f). \tag{3.1.5}$$

The function $f^*_{\mathcal{F}}$ is the *best-in-class predictor*: it is the best rule in $\mathcal{F}$ for the true population risk. The function $f^*_{\mathcal{T}}$ is the *empirical risk minimizer*: it is the rule in $\mathcal{F}$ with the smallest training risk. Since the Bayes predictor optimizes over all prediction functions, while $f^*_{\mathcal{F}}$ optimizes only over $\mathcal{F}$, we have

$$R(f^*_{\mathcal{F}}) \geq R(f^*).$$

These definitions give the standard decomposition of the *excess risk*:

$$R(f^*_{\mathcal{T}}) - R(f^*) = \underbrace{R(f^*_{\mathcal{F}}) - R(f^*)}_{\text{approximation error}} + \underbrace{R(f^*_{\mathcal{T}}) - R(f^*_{\mathcal{F}})}_{\text{estimation error}}. \tag{3.1.6}$$

The approximation error is the increase in risk caused by restricting the optimization to the model class $\mathcal{F}$. If $\mathcal{F}$ is too small, even the best rule in the class cannot achieve a risk close to the Bayes risk. The estimation error results from choosing a rule that optimizes $\hat{R}$ rather than R.

This decomposition explains a tradeoff in model choice. Enlarging $\mathcal{F}$ can reduce the approximation error, because we minimize risk over more functions. But a larger class can also increase the estimation error, because the empirical risk minimizer can adapt more strongly to random noise in $\mathcal{T}$. The training risk $\hat{R}_{\mathcal{T}}(f^*_{\mathcal{T}})$ cannot increase when $\mathcal{F}$ is enlarged, but the population risk $R(f^*_{\mathcal{T}})$ can.[8] This is the overfitting problem.

[8] To be explicit, note how $\hat{R}$ and R are used here.

Many texts discuss this tradeoff in terms of *bias* and *variance*. The usual bias-variance decomposition is a squared-loss identity that averages over possible training samples. It separates risk in a different way

from the approximation/estimation decomposition above. We derive it when we specialize to squared loss below.

The next three sections apply these definitions to regression, nominal classification, and ordinal prediction.

3.2 Regression

3.2.1 Setup

Regression problems aim to predict numerical values such as a price, a travel time, or a demand level.[1] Therefore, in a *regression* problem, $\mathcal{Y} = \mathcal{A} = \mathbb{R}$, and the decision map d and label-lifting map r are identity maps.[2]

We next consider two common regression losses: *squared loss* and *absolute loss*. For each loss, we first derive the Bayes predictor, then find the best predictors within the constant and affine model classes. Replacing population risk by training risk gives practical fitting procedures: squared loss leads to the sample mean and ordinary least squares, while absolute loss leads to sample medians and least absolute deviations regression.

[1] Wikipedia: Regression analysis.

[2] In classification problems, this is no longer the case.

3.2.2 Squared loss

The most common loss is *squared loss*,

$$\ell(y, \hat{y}) = (y - \hat{y})^2.$$

The empirical average of squared loss over a list S is called the *mean squared error* (MSE).[3]

[3] Wikipedia: Mean squared error. In practice one often reports the *root mean squared error* (RMSE), the square root of the MSE, which has the same units as y.

TO FIND THE OPTIMAL RULE for this loss, suppose we know $F_{X,Y}$ and may choose any prediction function. As a first step, we decompose the squared risk in a way that will be useful throughout this section. It separates the part of the error that can be reduced by choosing a better prediction rule from the variation in Y that remains once X is known.

Theorem 3.2.1. Let $m(x) = \mathsf{E}[Y \mid X = x]$ be the conditional expectation of Y given $X = x$, and suppose $\mathsf{E}\left[Y^2\right] < \infty$. Then, for every $f : \mathcal{X} \to \mathbb{R}$ with $\mathsf{E}\left[f(X)^2\right] < \infty$,

$$R(f) = \mathsf{E}\left[(Y - f(X))^2\right] = \underbrace{\mathsf{E}\left[\left(m(X) - f(X)\right)^2\right]}_{\text{squared approximation error}} + \underbrace{\mathsf{E}\left[\mathsf{V}[Y \mid X]\right]}_{\text{irreducible noise}}. \tag{3.2.1}$$

—

Proof. Split the prediction error at m:

$$\begin{aligned}\mathsf{E}\left[(Y - f(X))^2\right] &= \mathsf{E}\left[\left((Y - m(X)) + \left(m(X) - f(X)\right)\right)^2\right] \\ &= \mathsf{E}\left[\left(m(X) - f(X)\right)^2\right] + 2\,\mathsf{E}\left[\left(m(X) - f(X)\right)(Y - m(X))\right] \\ &\quad + \mathsf{E}\left[(Y - m(X))^2\right].\end{aligned}$$

By conditioning on X in the middle term,

$$\mathsf{E}\left[\left(m(X) - f(X)\right)(Y - m(X)) \,\middle|\, X\right] = \left(m(X) - f(X)\right)\mathsf{E}\left[Y - m(X) \mid X\right] = 0,$$

because $m(X) = \mathsf{E}[Y \mid X]$. With respect to the third term, conditioning on X yields

$$\begin{aligned}\mathsf{E}\left[(Y - m(X))^2 \,\middle|\, X\right] &= \mathsf{E}\left[Y^2 \,\middle|\, X\right] - 2m(X)\,\mathsf{E}[Y \mid X] + m(X)^2 \\ &= \mathsf{E}\left[Y^2 \,\middle|\, X\right] - m(X)^2.\end{aligned}$$

From the definition of conditional variance

$$\mathsf{V}[Y \mid X] = \mathsf{E}[(Y - \mathsf{E}[Y \mid X] \mid X] = \mathsf{E}\left[Y^2 \mid X\right] - m(X)^2.$$

Taking expectations completes the proof. □

The *Bayes regressor* under squared loss now follows easily.

Corollary 3.2.2. Under the same conditions as Th. 3.2.1, $R(f) \geq R(m)$. Consequently, the conditional expectation $f^*(x) = m(x) = \mathsf{E}[Y \mid X = x]$ attains the minimal risk $R(f^*) = R(m) = \mathsf{E}[\mathsf{V}[Y \mid X]] \leq \mathsf{V}(\mathrm{Y})$.[4]

—

FITTING THE BEST CONSTANT FUNCTION requires minimizing $\mathsf{E}\left[(m(X) - c)^2\right]$ in (3.2.1). Expanding the square and using that $\mathsf{E}[m(X)] = \mathsf{E}[Y]$, gives

$$\mathsf{E}\left[m(X)^2\right] + c(c - 2\,\mathsf{E}[Y]).$$

This is minimal at $c^* = \mathsf{E}[Y]$, and then the risk $R(c^*)$ becomes

$$c^* = \mathsf{E}[Y], \quad R(c^*) = \mathsf{E}[\mathsf{V}[Y \mid X]] + \mathsf{E}\left[m(X)^2\right] - (\mathsf{E}[Y])^2 = \mathsf{V}[Y], \quad (3.2.2)$$

where the last step follows from Eve's law.

To obtain the empirical risk minimizer and minimal empirical risk, we replace $F_{X,Y}$ by the measure induced by $\mathcal{T}$ to get

$$\hat{c} = \hat{y} = \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} y, \qquad \hat{R}_{\mathcal{T}}(\hat{c}) = \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} (y - \hat{c})^2. \qquad (3.2.3)$$

FOR AN AFFINE PREDICTION RULE the population problem is to minimize the approximation term in (3.2.1), since the irreducible-noise term does not depend on the affine parameters:

$$\mathsf{E}\left[\left(m(X) - f_{\beta_0,\beta}(X)\right)^2\right] = \mathsf{E}\left[\left(m(X) - \beta_0 - \langle \beta, X\rangle\right)^2\right].$$

Setting the partial derivatives wrt β and β_0 to zero gives the *population normal equations*:

$$\begin{aligned}\beta_0 + \langle \beta, \mathsf{E}[X]\rangle &= \mathsf{E}[Y], \\ \mathsf{E}\left[\left(m(X) - \beta_0 - \langle \beta, X\rangle\right) X(j)\right] &= 0, \qquad j = 1, \ldots, p.\end{aligned}$$

Replacing $F_{X,Y}$ by the measure induced by $\mathcal{T}$ gives

$$\begin{aligned}\hat{\beta}_0 + \langle \hat{\beta}, \hat{x}\rangle &= \hat{y}, \\ \sum_{(x,y)\in\mathcal{T}} \left(y - \hat{\beta}_0 - \langle \hat{\beta}, x\rangle\right) x(j) &= 0, \qquad j = 1, \ldots, p.\end{aligned}$$

These solve the *ordinary least squares (OLS)* problem:

$$(\hat{\beta}_0, \hat{\beta}) \in \operatorname*{arg\,min}_{\beta_0,\beta} \hat{R}_{\mathcal{T}}(f_{\beta_0,\beta}), \qquad (3.2.4)$$

$$\hat{R}_{\mathcal{T}}(f_{\beta_0,\beta}) = \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \left(y - \beta_0 - \langle \beta, x\rangle\right)^2. \qquad (3.2.5)$$

Restricting the affine class to $\beta = 0$ recovers the constant class: the population equation gives $\beta_0^* = \mathsf{E}[Y]$, and the sample equation gives $\hat{\beta}_0 = \hat{c}$ of (3.2.3).

[4] Eve's law, also known as the law of total variance, states that $\mathsf{V}[Y] = \mathsf{E}[\mathsf{V}[Y \mid X]] + \mathsf{V}[\mathsf{E}[Y \mid X]]$. I prefer the name Eve's law because it is a law.

BIAS AND VARIANCE are two forms of error that appear through the approximation of $m(x)$ by the empirical risk minimizer $\hat{f}_{\mathcal{T}}$, (3.1.5). Write

$$\bar{f}(x) = \mathsf{E}_{\mathcal{T}}\left[\hat{f}_{\mathcal{T}}(x)\right]$$

for the average prediction at x over the iid training samples that can obtained from F_{XY}. The *bias* is $\bar{f}(x) - m(x)$: the systematic discrepancy between the average fitted prediction and the conditional mean. The *variance* measures how much the fitted prediction changes when we draw a new training sample.

There are two sources of randomness to distinguish. The training sample determines the fitted rule; a *new* observation (X^0, Y^0), independent of $\mathcal{T}$ and with distribution $F_{X,Y}$, determines where and against which outcome we evaluate that rule.

Theorem 3.2.3. The risk of the empirical risk minimizer can be decomposed as: [5]

$$\mathsf{E}_{\mathcal{T}}\left[R(\hat{f}_{\mathcal{T}})\right] = \underbrace{\mathsf{E}\left[\mathsf{V}\left[Y \mid X\right]\right]}_{\text{irreducible noise}} + \underbrace{\mathsf{E}\left[\left(\bar{f}(X) - m(X)\right)^2\right]}_{\text{squared bias}} + \underbrace{\mathsf{E}\left[\mathsf{V}_{\mathcal{T}}\left[\hat{f}_{\mathcal{T}}(X)\right]\right]}_{\text{variance}}. \quad (3.2.6)$$

[5] An expectation with subscript $\mathcal{T}$ averages over training samples.

—

Proof. Fix x. In the first term in (3.2.1) add and subtract the average fitted prediction to get

$$m(x) - \hat{f}_{\mathcal{T}}(x) = \left(m(x) - \bar{f}(x)\right) + \left(\bar{f}(x) - \hat{f}_{\mathcal{T}}(x)\right).$$

Squaring, expanding and taking the expectation over $\mathcal{T}$ gives

$$(m(x) - \bar{f}(x))^2 + 2(m(x) - \bar{f}(x))\,\mathsf{E}_{\mathcal{T}}\left[\bar{f}(x) - \hat{f}_{\mathcal{T}}(x)\right] + \mathsf{E}_{\mathcal{T}}\left[(\bar{f}(x) - \hat{f}_{\mathcal{T}}(x))^2\right].$$

The cross term vanishes: by the definition of $\bar{f}$,

$$\mathsf{E}_{\mathcal{T}}\left[\bar{f}(x) - \hat{f}_{\mathcal{T}}(x)\right] = \bar{f}(x) - \mathsf{E}_{\mathcal{T}}\left[\hat{f}_{\mathcal{T}}(x)\right] = 0.$$

The third term is the variance of the fitted prediction across training samples.

Taking the expectation over a new observation X, combining this with (3.2.1), and using the independence of X and $\mathcal{T}$ proves the result. □

FEATURE SELECTION affects both bias and variance. Suppose for this paragraph that the feature coordinates are uncorrelated, so that leaving one out does not disturb the estimates of the others. A feature with $\beta_j = 0$ has no contribution to the conditional mean, so estimating its coefficient adds variance without reducing bias. Omitting it therefore lowers the variance of the fitted rule without increasing bias. Omitting a feature with $\beta_j \neq 0$ leaves its contribution $\beta_j x(j)$ inside the residual, which raises the residual variance, and every remaining coefficient is estimated against that larger noise; omitting it therefore raises the variance of the remaining estimates and introduces bias. Feature selection aims to retain useful features and omit those that add only estimation noise. The lasso in Section 3.5 is one method for selecting features.

3.3 Nominal classification

3.3.1 Setup

Nominal classification problems have categorical labels without a natural order. This means that for K-class nominal classification, the label lies in the set[1]

$$\mathcal{Y} = \mathcal{K} = \{1, \dots, K\}.$$

[1] It is common to assume that labels like 'red' and 'green' are already converted to numbers.

Besides the prediction $\hat{y}$ for the label y, we will be concerned with *probability prediction*

$$\hat{p} = (\hat{p}_1, \dots, \hat{p}_K) \in \Delta_K,$$

where Δ_K *is the /probability simplex*

$$\Delta_K = \left\{ p \in \mathbb{R}^K : p_k \geq 0,\ \sum_{k=1}^{K} p_k = 1 \right\}.$$

Here $\hat{p}_k$ is interpreted as the predicted probability of class k.

Throughout this section the prediction set $\mathcal{A} = \Delta_K$, and the fitted rule returns $\hat{a} = \hat{p}$. When a single class has to be returned based on $\hat{p}$, the decision map is the *argmax rule*[2]

$$\hat{y} = d(\hat{p}) \in \arg\max_k \hat{p}_k. \tag{3.3.1}$$

[2] This rule is the same as *majority vote* when $\hat{p}$ is the vector of class frequencies of a sample list $\mathcal{S}$: the label with the highest probability is then the most frequent one.

In the other direction, the observed class y is lifted into the simplex by the *one-hot encoding* of y,

$$a = q = r(y), \qquad q_k = \mathbb{1}\{y = k\},$$

which puts probability one on the observed class and zero on all others.[3]

[3] Observe that, as required, $d(r(y)) = y$.

For any list $\mathcal{S}$ whose labels lie in $\mathcal{K}$, $\mathcal{S}_k = [(x, y) \in \mathcal{S} : y = k]$ is the sublist of samples of class k. Write p for the vector with components[4] $p_k = \mathsf{P}\{Y = k\}$, and $p(x)$ be the vector of conditional class probabilities with components $p_k(x) = \mathsf{P}\{Y = k \mid X = x\}$. For the non-empty training list $\mathcal{T}$, write $\mathcal{T}_k = [(x, y) \in \mathcal{T} : y = k]$ and $\hat{p}_k = |\mathcal{T}_k|/|\mathcal{T}|$.

[4] That is, the marginal probabilities.

3.3.2 Zero-one loss

Zero-one loss compares samples based on class:

$$\ell(a, \hat{a}) = \ell(q, \hat{p}) = \mathbb{1}\{d(q) \neq d(\hat{p})\} = \mathbb{1}\{y \neq d(\hat{p})\}. \tag{3.3.2}$$

The next numerical example shows an important problem with zero-one loss. Take two classes, $K = 2$, and suppose the observed class is $y = 1$, which converts into the one-hot vector $q = (1, 0)$. Compare the two probability predictions

$$\hat{p}^A = (0.51, 0.49), \qquad \hat{p}^B = (0.99, 0.01).$$

The two predictions receive the same zero-one loss, but this loss does not show confident the probability vector is. As a consequence, optimizing under zero-one loss is difficult because small changes in the probabilities often leave the predicted label unchanged.

FOR ANY RULE $f : \mathcal{X} \to \Delta_K$ write $g = d \circ f : \mathcal{X} \to \mathcal{K}$. Then for the conditional risk

$$\begin{aligned}\mathsf{E}\left[\ell(r(Y), f(X)) \mid X = x\right] &= \mathsf{E}\left[Y \neq g(X) \mid X = x\right] \\ &= \mathsf{P}\left\{Y \neq g(X) \mid X = x\right\} \\ &= 1 - \mathsf{P}\left\{Y = g(X) \mid X = x\right\}.\end{aligned}$$

Therefore,

$$\arg\min_g \mathsf{E}\left[\mathbb{1}\left\{Y \neq g(X) | X = x\right\}\right] = \arg\max_{k \in K} \mathsf{P}\left\{Y = k | X = x\right\}.$$

Using (3.3.1), we see that the *Bayes classifier* f^* has components

$$f_k^*(x) = \mathsf{P}\{Y = k \mid X = x\} = p_k(x). \tag{3.3.3}$$

FOR A CONSTANT PREDICTOR $d(f(x)) = k$, the loss becomes

$$R(f) = \mathsf{E}[Y \neq k] = 1 - \mathsf{P}\{Y = k\}.$$

Thus the best constant predictor is the *majority class* and the risk is the *misclassification rate*,

$$\hat{y} \in \arg\max_k |\mathcal{T}_k|, \qquad \hat{R}_{\mathcal{T}}(r(\hat{y})) = 1 - \max_k \hat{p}_k. \tag{3.3.4}$$

3.3.3 Brier loss

The Brier loss[5]

$$\ell(q, \hat{p}) = \sum_{k=1}^{K} (q_k - \hat{p}_k)^2 \tag{3.3.5}$$

[5] Wikipedia: Brier score.

compares the one-hot representation $q = r(y)$ to the prediction $\hat{p}$.

A NUMERICAL EXAMPLE shows the advantage of Brier loss over zero-one loss. Take two classes, $K = 2$, and suppose the observed class is $y = 1$, which converts into the one-hot vector $q = (1, 0)$. Compare the two probability predictions

$$\hat{p}^A = (0.51, 0.49), \qquad \hat{p}^B = (0.99, 0.01).$$

For the Brier loss, both probability components enter the loss. For $\hat{p}^A$,

$$\begin{aligned}\ell(q, \hat{p}^A) &= (q_1 - \hat{p}_1^A)^2 + (q_2 - \hat{p}_2^A)^2 \\ &= (1 - \hat{p}_1^A)^2 + (0 - \hat{p}_2^A)^2 \\ &= (1 - 0.51)^2 + (0 - 0.49)^2 \\ &= 0.49^2 + 0.49^2 \\ &\approx 0.48.\end{aligned}$$

For $\hat{p}^B$,

$$\begin{aligned}\ell(q, \hat{p}^B) &= (q_1 - \hat{p}_1^B)^2 + (q_2 - \hat{p}_2^B)^2 \\ &= (1 - \hat{p}_1^B)^2 + (0 - \hat{p}_2^B)^2 \\ &= (1 - 0.99)^2 + (0 - 0.01)^2 \\ &= 0.01^2 + 0.01^2 \\ &= 0.0002.\end{aligned}$$

Thus the second prediction receives a much smaller Brier loss, because its probability vector is closer to the one-hot vector for the observed class.

The maximum value 2 is attained when the prediction puts all probability on the wrong class.

FOR THE BAYES PREDICTOR we split the risk in a similar way as Th. 3.2.1. As the kth component of $r(Y)$ is $\mathbb{1}\{Y = k\}$, the Brier loss becomes

$$\mathsf{E}\big[\ell(r(Y), f(X))\big] = \mathsf{E}\left[\sum_{k=1}^{K} \big(\mathbb{1}\{Y = k\} - f_k(X)\big)^2\right] = \sum_{k=1}^{K} \mathsf{E}\Big[\big(\mathbb{1}\{Y = k\} - f_k(X)\big)^2\Big].$$

To minimize this, we can minimize each of the terms in the summation, which allows us to apply Th. 3.2.1 to each term separately with $\mathbb{1}\{Y = k\}$ in the role of Y for the kth term. By straight comparision, its conditional expectation is $\mathsf{E}[\mathbb{1}\{Y = k\} \mid X] = p_k(X)$. The conditional variance[6] is $\mathsf{V}[\mathbb{1}\{Y = k\} \mid X] = p_k(X)\big(1 - p_k(X)\big)$. Summing the variances over k and using that the $p_k(x)$ add up to one,

[6] An indicator with success probability p has variance $p(1-p)$.

$$\sum_{k=1}^{K} p_k(x)\big(1 - p_k(x)\big) = 1 - \sum_{k=1}^{K} p_k(x)^2.$$

Summarizing:

Theorem 3.3.1. For every $f : \mathcal{X} \to \Delta_K$,

$$R(f) = \mathsf{E}\big[\ell(r(Y), f(X))\big] = \underbrace{\mathsf{E}\left[\sum_{k=1}^{K} \big(f_k(X) - p_k(X)\big)^2\right]}_{\text{squared error}} + \underbrace{\mathsf{E}\left[1 - \sum_{k=1}^{K} p_k(X)^2\right]}_{\text{Gini impurity}}.$$

—

From the above and Cor. 3.2.2 we obtain the following corollary.

Corollary 3.3.2. The minimizer of the Brier loss is the *Bayes predictor* $f^* = p$, that is

$$f_k^*(x) = p_k(x) = \mathsf{P}\{Y = k \mid X = x\}.$$

—

A CONSTANT VECTOR AS A PREDICTOR has the form $f_c(x) = c \in \Delta_{\mathcal{K}}$. As each term f_k is constant, we can use (3.2.2) term by term. Again, with $\mathbb{1}\{Y = k\}$ in the role of Y for the kth term, the optimal constant is $\mathsf{E}[\mathbb{1}\{Y = k\}] = p_k$ and the variance is $\mathsf{V}[\mathbb{1}\{Y = k\}] = p_k(1 - p_k)$. Thus the best-in-class predictor and its risk are

$$c^* = p, \qquad R(c^*) = 1 - \sum_{k=1}^{K} p_k^2.$$

Replacing the marginal probabilities by sample frequencies gives

$$\hat{c}^* = \hat{p}, \qquad \hat{R}_{\mathcal{T}}(c^*) = 1 - \sum_{k=1}^{K} \hat{p}_k^2. \tag{3.3.6}$$

3.4 Model flexibility

We have now studied losses and constant or affine rules for regression, nominal classification, and ordinal classification. We next return to *regression under squared loss* to examine two other ways of constructing a fitted rule. Both aim to approximate the conditional mean $m(x) = \mathsf{E}[Y \mid X = x]$, but they control flexibility differently. Nearest neighbors fit a constant locally; polynomial regression fits a function from a class whose degree we choose.

3.4.1 Nearest neighbors

If several training samples have exactly the feature vector x, we can estimate $m(x)$ by averaging their outcomes. For continuously distributed features, exact matches to a specified x generally do not occur. Instead, we can average outcomes at feature vectors *near* x, using the idea that their conditional means should also be close.

Choose a number of neighbors k, with $1 \le k \le n$, and measure closeness by Euclidean distance:

$$\|x_i - x\|_2 = \left(\sum_{j=1}^{p} (x_i(j) - x(j))^2 \right)^{1/2}.$$

Let $i_1(x), \dots, i_n(x)$ be the training indices ordered by increasing distance to x. Break equal-distance ties by the original sample index, so that the first k indices always identify exactly k samples, including repeated samples when present. The *k-nearest-neighbor* (KNN)[1] prediction is

$$\hat{f}_{k,\mathcal{T}}(x) = \frac{1}{k} \sum_{a=1}^{k} y_{i_a(x)}. \tag{3.4.1}$$

[1] We use lowercase k for the number of neighbors; uppercase K continues to denote the number of classes.

This is precisely the fitted constant of (3.2.3), but fitted to the local list $\mathcal{T}_k(x) = [(x_{i_1(x)}, y_{i_1(x)}), \dots, (x_{i_k(x)}, y_{i_k(x)})]$. The list changes with the prediction point. The resulting rule can therefore follow a nonlinear conditional mean even though each individual prediction is obtained by fitting a constant.

Algorithm 3.4.1 Predict by averaging the outcomes of exactly k nearest training samples.

Input: a non-empty training list $\mathcal{T}$, a feature vector x, and an integer $1 \le k \le |\mathcal{T}|$.
Output: a numerical prediction at x.
procedure KNNPREDICT($\mathcal{T}, x, k$)
    Order the indices $i_1, \dots, i_n$ by increasing $\|x_i - x\|_2$, breaking ties by index.
    return AVERAGE($[y_{i_1}, \dots, y_{i_k}]$)

Distances depend on the units of the features. A coordinate measured in euros can dominate one measured in kilometers even if both are useful. One common choice is to center each feature and divide it by its training standard deviation before calculating distances. The same

[2] We apply the last transformation only when $\sigma(j) > 0$; constant features can be omitted.

transformation, with the same fitted means and standard deviations, is then applied to new inputs. For $1 \le j \le p$, let[2]

$$\mu(j) = \frac{1}{n}\sum_{i=1}^{n} x_i(j), \quad \sigma(j) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i(j)-\mu(j))^2}, \quad \tilde{x}_i(j) = \frac{x_i(j)-\mu(j)}{\sigma(j)}. \tag{3.4.2}$$

A NUMERICAL EXAMPLE demonstrates clearly the effect of k. Consider the conditional mean

$$m(x) = 2 - \frac{1}{2}x^2 + 3x + \sin(4x), \qquad -1 \le x \le 2. \tag{3.4.3}$$

Generate $n = 200$ independent samples according to

$$X_i \sim \mathrm{Unif}(-1,2), \qquad Y_i = m(X_i) + \varepsilon_i, \qquad \varepsilon_i \sim \mathcal{N}(0, 1.5^2),$$

with the noises independent of the inputs and of one another. Then $\mathsf{E}[Y_i \mid X_i = x] = m(x)$.[3]

Fig. 3.1 compares $k = 200, 100, 50, 10, 5, 1$ on the same sample. At $k = 200$ the rule is constant and cannot follow the trend or the oscillations. At $k = 1$, the fitted rule reproduces each training outcome, including its noise, which is clearly overfitting.

WHEN THE NUMBER OF FEATURES p is large, e.g. $p \ge 10$, the nearest neighbors of x are no longer close to x. Two computations show this for samples that are uniformly distributed on $[0,1]^p$.

First, consider a sub-cube with sidelength[4] a contains a fraction s of the samples. Then, $a = s^{1/p}$. For $s = 0.01$ and $p = 10$ this side length is $0.01^{1/10} \approx 0.63$: a neighborhood that holds 1% of the samples spans 63% of the range of each feature.

Second, let X and X' be independent uniform points in $[0,1]^p$. Then

$$\mathsf{E}\left[\|X - X'\|_2^2\right] = \sum_{j=1}^{p} \mathsf{E}\left[(X(j) - X'(j))^2\right] = \frac{p}{6}, \tag{3.4.4}$$

$$\mathsf{V}\left[\|X - X'\|_2^2\right] = \sum_{j=1}^{p} \mathsf{V}\left[(X(j) - X'(j))^2\right] = \frac{7p}{180}, \tag{3.4.5}$$

where the second line uses that the coordinate differences are independent. The squared coefficient of variation[5] is

$$\frac{\mathsf{V}\left[\|X - X'\|_2^2\right]}{\mathsf{E}\left[\|X - X'\|_2^2\right]^2} = \frac{7}{5p}. \tag{3.4.6}$$

Thus, as p grows, the variation of squared distances relative to their mean becomes small: all points lie roughly equally far away. The k nearest samples are then barely closer to x than the other samples, so their average says little more about $m(x)$ than an average over arbitrary samples. Yet using all n observations is also undesirable, since it gives the global constant predictor.

This is the *curse of dimensionality*: local averaging becomes difficult in high dimension. An alternative restricts the rule to a class $\mathcal{F}$ with a fixed number of parameters. All n samples then contribute to fitting each parameter, not only the k samples nearest to x.

[3] Because this is a constructed example, we know m; in an application we only observe the pairs (x_i, y_i).

[4] Hence with volume a^p.

[5] The variance divided by the squared mean.

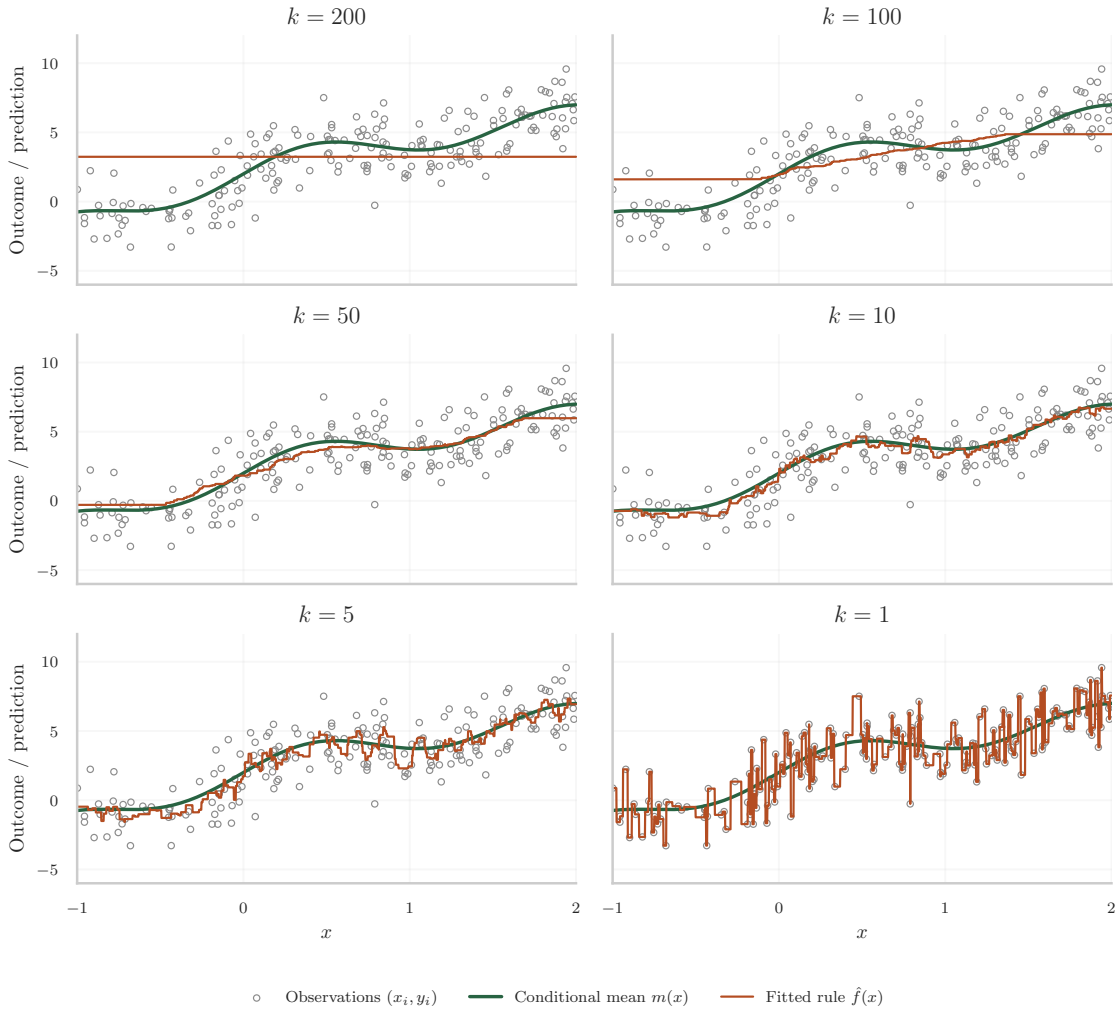


FIG. 3.1: Nearest-neighbor regression for six values of k, using the same 200 observations in every panel. The green curve is $m(x)$; the brown curve is the fitted rule. At $k = 200$ the fit is constant. At $k = 1$ it follows individual noisy outcomes.

POLYNOMIAL REGRESSION is one such class. For a nonnegative integer degree q, define

$$\mathcal{F}_q = \left\{ f_\beta : f_\beta(x) = \beta_0 + \sum_{j=1}^{q} \beta_j x^j,\ \beta \in \mathbb{R}^{q+1} \right\}. \tag{3.4.7}$$

The cases $q = 0$ and $q = 1$ recover the constant and affine classes for one numerical input. For $q > 1$, the rule is generally nonlinear in x, but remains affine in the transformed features $(x, x^2, \ldots, x^q)$ and linear in the coefficients. Fitting therefore uses the same least-squares method as before:

$$\hat{\beta} \in \operatorname*{argmin}_{\beta \in \mathbb{R}^{q+1}} \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{q} \beta_j x_i^j \right)^2.$$

The coefficients β are *parameters*: least squares fits them for a given degree. The degree q is a *hyperparameter*: it fixes the class $\mathcal{F}_q$ in which we fit.

Fig. 3.2 compares degrees $q = 1, 4, 10, 30$ on the same observations used for KNN. An affine rule, $q = 1$, cannot follow the oscillations in m. A

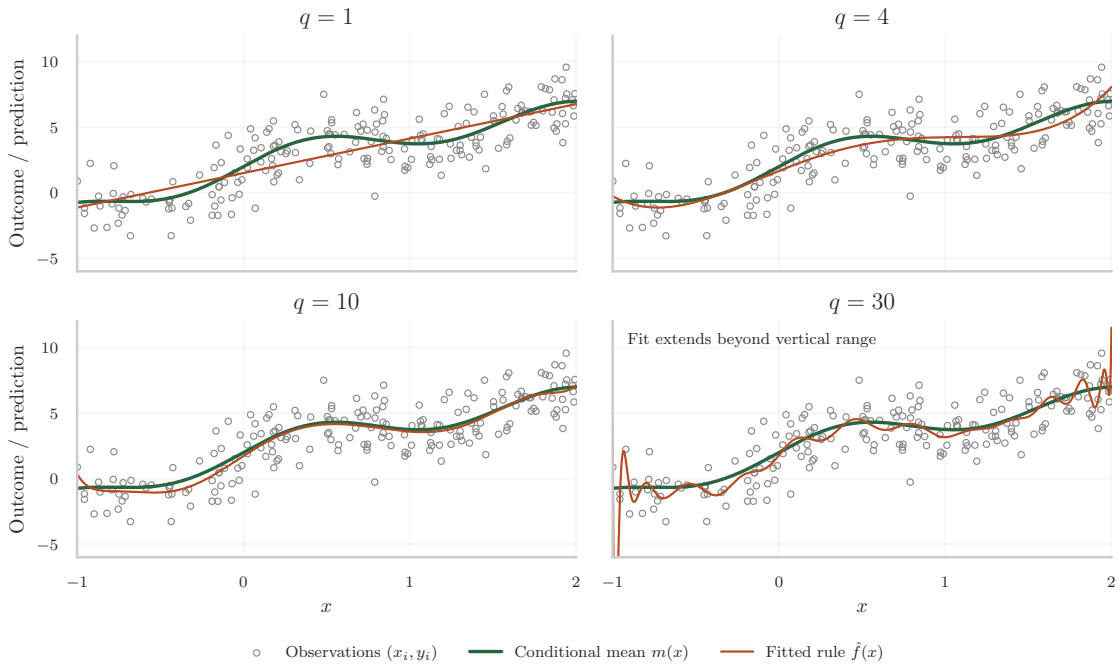


FIG. 3.2: Polynomial least-squares fits of degrees $q = 1, 4, 10, 30$. The observations are the same as in Fig. 3.1. The degree-one fit misses the nonlinear pattern. The degree-thirty fit shows additional oscillations from fitting noise. Portions of that curve extend beyond the displayed vertical range.

polynomial of degree q has at most $q-1$ turning points, so a higher degree can improve the approximation of m, but it also lets the fit respond more strongly to the noise in the sample. The $q = 10$ fit follows m closely; the $q = 30$ fit adds oscillations, especially near the endpoints.

3.5 Regularization

Regularization addresses the problem that a method can be too flexible. When the training outcomes contain noise, a flexible method can adapt not only to stable patterns, but also to accidental variation in the training sample. One way is to restrict the fitted rule deliberately by making $\mathcal{F}$ small; another way is to make complicated rules more expensive in the fitting problem.[1]

[1] Wikipedia: Regularization (mathematics).

Starting from the OLS problem (3.2.4), *ridge regression*[2] replaces the least-squares criterion by

[2] Wikipedia: Ridge regression.

$$\min_{\beta_0,\beta} \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} (\beta_0 + \langle \beta, x\rangle - y)^2 + \frac{\lambda}{2} \|\beta\|_2^2, \qquad \|\beta\|_2^2 = \sum_j \beta_j^2, \tag{3.5.1}$$

where $\lambda \geq 0$ is a *penalty parameter*, or *regularization parameter*. A larger λ makes large coefficients more expensive. *Lasso*[3] uses the same idea, again starting from (3.2.4), but replaces the squared 2-norm penalty by a 1-norm penalty:

[3] Wikipedia: Lasso (statistics).

$$\min_{\beta_0,\beta} \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} (\beta_0 + \langle \beta, x\rangle - y)^2 + \lambda \|\beta\|_1, \qquad \|\beta\|_1 = \sum_j |\beta_j|.$$

These penalties shrink the fitted coefficients and can reduce variance, at the cost of some bias—an instance of the bias-variance tradeoff. The

two penalties differ in one important respect: the lasso can set coefficients exactly to zero, so it also performs *variable selection* in the sense of Section 3.2, while ridge regression only shrinks the coefficients.

Before adding a penalty term, we should standardize each feature using its training mean and standard deviation, as in (3.4.2).

Note that standardization is unnecessary for unregularized OLS: the fitted predictions are unchanged by rescaling a feature. Here, however, it is essential: rescaling changes the coefficient and therefore the penalty applied to it.

3.6 Validation and testing

Above we discussed several predictors and risks. We next discuss methods to assses the quality of the fitting process.

3.6.1 Training, validation, and test data

Choosing and assessing a prediction rule takes three steps. First, we select a function class $\mathcal{F}$ over which we fit. A function in $\mathcal{F}$ is specified by hyperparameters and parameters. For instance, for the union of the polynomial classes (3.4.7), the degree q is a hyperparameter and the coefficients $\beta_0, \ldots, \beta_q$ are the parameters. Here we discuss how to tune the hyperparameters and the coeficients and assess how well the final result would do on a new observation X. To tune a hyperparameter, we fix in advance a finite set of values to compare, for instance the degrees $\Gamma = \{0, 1, \ldots, 30\}$. Each value in this set is a *candidate*, and the rule fitted at a candidate is a *candidate rule*.

For independent observations from a common population, we can randomly divide the available sample into three disjoint lists:

- The *training list* $\mathcal{T}$: used to fit candidate rules.
- The *validation list* $\mathcal{V}$: used to select a candidate.
- The *test list* $\mathcal{S}_{\text{test}}$: kept aside to assess the final rule after selection and fitting are complete.

Splitting the samples into these subsets affects both fitting accuracy and evaluation precision: more training data can improve a fitted rule, while more evaluation data make its measured risk more precise. There is no single split proportion that is appropriate for every sample size and prediction problem.

Let Γ be a finite collection of candidate settings and let $\hat{f}_{\gamma,\mathcal{T}}$ be the rule fitted with setting γ. Now we use the validation set $\mathcal{V}$ to find the best γ. Writing the risk *on the validation set* as $\hat{R}_{\mathcal{V}}(\hat{f}_{\gamma,\mathcal{T}})$,[1] the best candidate is

$$\hat{\gamma} \in \arg\min_{\gamma \in \Gamma} \hat{R}_{\mathcal{V}}(\hat{f}_{\gamma,\mathcal{T}}). \tag{3.6.1}$$

[1] Note: each candidate is fitted without using the validation observations.

After choosing $\hat{\gamma}$, we can refit on the concatenated list $\mathcal{D} = \mathcal{T} + \mathcal{V}$. The final rule is then $\hat{f}_{\hat{\gamma},\mathcal{D}}$.

Finally, we use the test list to assess the final rule. Its test risk

$$\hat{R}_{\mathcal{S}_{\text{test}}}(\hat{f}_{\hat{\gamma},\mathcal{D}})$$

is the average loss of $\hat{f}_{\hat{\gamma},\mathcal{D}}$ over the test samples. The test samples are used neither to fit the parameters nor to select $\hat{\gamma}$. For the final rule they are therefore new observations, and the test risk estimates its risk $R(\hat{f}_{\hat{\gamma},\mathcal{D}})$ on a new observation (X, Y). Since the test risk is an average of losses on independent samples, a larger test list gives a more precise estimate.

PREPROCESSING IS PART OF FITTING . Means and standard deviations used to scale inputs must be estimated on the training part, then applied unchanged to the validation part. Feature selection, missing-value imputation, and other transformations estimated from data must follow the same procedure. Allowing the validation or test data to influence fitting is called *data leakage*.

3.6.2 Cross-validation

In the procedure above the candidate rules are fitted on $\mathcal{T}$ only, and $\mathcal{V}$ is never used for this purpose. *Cross-validation* divides $\mathcal{D} = \mathcal{T} + \mathcal{V}$ instead into several parts. Each part serves once as validation list, while the rules are fitted on the other parts. Every sample in $\mathcal{D}$ is then used for fitting and for validation, but never both in the same fit.

Let $N = |\mathcal{D}|$, and divide its sample indices into F disjoint non-empty folds $\mathcal{I}_1, \ldots, \mathcal{I}_F$, usually of nearly equal size. For fold b, write

$$\mathcal{D}_b = [(x_i, y_i) : i \in \mathcal{I}_b], \qquad \mathcal{D}_{-b} = \mathcal{D} \setminus \mathcal{D}_b = [(x_i, y_i) : i \notin \mathcal{I}_b].$$

In *F-fold cross-validation*, we fit a candiate γ on $\mathcal{D}_{-b}$ and we evaluate its risk on $\mathcal{D}_b$.[2] Writing $\hat{f}_{\gamma,\mathcal{D}_{-b}}$ for such a fit, the cross-validation criterion for candidate γ is

[2] Use the same folds for every candidate, so their errors are compared on the same validation observations.

$$\begin{aligned} \mathrm{CV}(\gamma) &= \frac{1}{N} \sum_{b=1}^{F} \sum_{i \in \mathcal{I}_b} \ell(r(y_i), \hat{f}_{\gamma,\mathcal{D}_{-b}}(x_i)) \\ &= \sum_{b=1}^{F} \frac{|\mathcal{D}_b|}{N} \hat{R}_{\mathcal{D}_b}(\hat{f}_{\gamma,\mathcal{D}_{-b}}). \end{aligned} \tag{3.6.2}$$

Every sample contributes one loss, from a rule fitted without that sample's fold. The weights account for unequal fold sizes. Now select

$$\hat{\gamma} \in \operatorname*{arg\,min}_{\gamma \in \Gamma} \mathrm{CV}(\gamma).$$

Again, once $\hat{\gamma}$ is computed, fit the model on $\mathcal{D}$ for this candidate.

A special case is when $F = N$; this is called *leave-one-out cross-validation*: each fit omits just one sample.

Algorithm 3.6.1 Select a setting by cross-validation and refit on all development data.

Input: a development list $\mathcal{D}$, candidate settings Γ, and $2 \le F \le |\mathcal{D}|$.
Output: a selected setting and the final fitted rule.
procedure CROSSVALIDATE($\mathcal{D}, \Gamma, F$)
    Divide the indices of $\mathcal{D}$ into folds $\mathcal{I}_1, \ldots, \mathcal{I}_F$.
    for $\gamma \in \Gamma$ **do**
        $\mathrm{CV}(\gamma) \leftarrow 0$
        for $b = 1, \ldots, F$ **do**
            Fit all preprocessing and the rule at γ on $\mathcal{D}_{-b}$.
            $\mathrm{CV}(\gamma) \leftarrow \mathrm{CV}(\gamma) + \dfrac{|\mathcal{D}_b|}{|\mathcal{D}|} \hat{R}_{\mathcal{D}_b}(\hat{f}_{\gamma,\mathcal{D}_{-b}})$
    $\hat{\gamma} \leftarrow$ a minimizer of $\mathrm{CV}(\gamma)$ over Γ
    Refit preprocessing and the rule at $\hat{\gamma}$ on all of $\mathcal{D}$.
    return $(\hat{\gamma}, \hat{f}_{\hat{\gamma},\mathcal{D}})$

THE NUMBER OF FOLDS F affects cross-validation in three ways.

First, with equal folds, each fold rule is fitted on $N(1-1/F)$ samples, while the final rule $\hat{f}_{\hat{\gamma},\mathcal{D}}$ is fitted on all N. A rule fitted on fewer samples has a larger estimation error, so with cross validation we obtain usually less good predictors. Hence $\mathrm{CV}(\gamma)$ tends to be somewhat larger than the risk of the rule fitted at γ on all of $\mathcal{D}$.

Second, cross-validation fits every candidate F times, so $F|\Gamma|$ fits in total, plus the final refit. Hence, CV increases the compuational load.

Third, the random variation of $\mathrm{CV}(\gamma)$. Whatever F, each sample of $\mathcal{D}$ is validated exactly once, so $\mathrm{CV}(\gamma)$ is always an average of N losses. What changes with F is how much the fold rules have in common. For large F, any two fitting lists $\mathcal{D}_{-b}$ share all but $2N/F$ samples, so the fold rules are nearly the same rule. If that rule happens to be poor, all its losses are large together, and averaging them removes less of this chance variation than averaging over rules fitted on different samples. Taking $F = 5$ or $F = 10$ keeps the fold training size at 80 or 90 percent of N, with 5 or 10 fits per candidate.

3.7 Assessing a classifier

CLASSIFICATION ASSESSMENT addresses the problem that classification quality cannot always be summarized by the *accuracy* on a list $\mathcal{S}$, the fraction of its label predictions that are correct,

$$\text{accuracy} = \frac{1}{|\mathcal{S}|} \sum_{(x,y)\in\mathcal{S}} \mathbb{1}\{\hat{y}(x) = y\}. \tag{3.7.1}$$

In a two-class problem it is customary to call the class of special interest[1] *positive* and the other class *negative*. If only 1% of cases are positive, always predicting the negative class already gives 99% accuracy, but it misses every positive case. This is the problem of *imbalanced data*.

[1] This is often the rare class, such as a disease in a screening problem.

For two classes, the four possible outcomes of a prediction are collected in the *confusion matrix*: a *true positive* (TP) is a positive case predicted positive, a *false positive* (FP) is a negative case predicted positive, and *true negatives* (TN) and *false negatives* (FN) are defined analogously.

A screening example makes the terminology concrete. In breast-cancer screening, call the screening result *positive* when the X-ray is suspicious, and *negative* otherwise. After follow-up, we can compare this screening result with whether cancer was actually present. A true positive is a suspicious X-ray for a woman who has breast cancer; a false positive is a suspicious X-ray where no cancer is found; a false negative is a non-suspicious X-ray for a woman who does have breast cancer; and a true negative is a non-suspicious X-ray where no cancer is found.

In this example, *recall* is the fraction of women with breast cancer whose X-rays are flagged as suspicious. A test with high recall misses few cancer cases. *Precision* is the fraction of suspicious X-rays for which breast cancer is actually present. A test with high precision gives few false alarms among the women sent for further tests. The *false-positive rate* is the fraction of women without breast cancer whose X-rays are nevertheless flagged as suspicious.

A good screening test should have high recall, because missed cancer cases are serious. High precision is also desirable, but it can be hard to achieve when the disease is rare. In practice there is often a tradeoff: flagging more X-rays as suspicious can increase recall, but it may also create more false positives and lower precision.

Counting these outcomes over the evaluation set gives three useful rates. The *true-positive rate*, or recall,[2] $\mathrm{TP}/(\mathrm{TP}+\mathrm{FN})$, is the fraction of positive cases that are found. The false-positive rate, $\mathrm{FP}/(\mathrm{FP}+\mathrm{TN})$, is the fraction of negative cases that are wrongly flagged. Precision, $\mathrm{TP}/(\mathrm{TP}+\mathrm{FP})$, is the fraction of predicted positives that are correct. In the example above, always predicting the negative class has recall 0: none of the positive cases are found. The accuracy is high only because positive cases are rare.

[2] The name comes from information retrieval: recall measures what fraction of all relevant cases the method retrieves, or "recalls". Here the relevant cases are the positive cases.

The always-negative rule is an example of a *baseline*: a simple reference method, such as always predicting the most frequent class in classification, or the training mean in regression. Reporting the performance of a fitted method alongside a baseline shows whether it improves on this simple reference under the chosen evaluation measure.

3.8 Exercises

Exercise (T) 3.8.1. Assume $\mathsf{E}\left[Y^2\right] < \infty$, and let $m(x) = \mathsf{E}[Y \mid X = x]$. Using the squared-loss decomposition (3.2.1), derive the Bayes predictor and the Bayes risk. Also express the excess risk of an arbitrary predictor f.

Exercise (T) 3.8.2. Under squared loss, derive the optimal constant predictor c^* and its population risk.

Exercise (C) 3.8.3. Explain the difference between the Bayes predictor, the best-in-class predictor, and the empirical-risk minimizer.

Exercise (T) 3.8.4. Derive the approximation–estimation decomposition of the excess risk $R(f_{\mathcal{F}}^*) - R(f^*)$.

Exercise (T) 3.8.5. Derive the bias–variance decomposition for squared loss using the average prediction $\bar{f}(x) = \mathsf{E}_{\mathcal{T}}\left[\hat{f}_{\mathcal{T}}(x)\right]$.

Exercise (T) 3.8.6. For binary classification with Brier loss, derive the Bayes predictor and explain what quantity it predicts.

Exercise (C) 3.8.7. Suppose $\mathcal{F}_1 \subseteq \mathcal{F}_2$. What can be said about their training risk and their population risk after empirical-risk minimization?

Chapter 4

Trees, Bags, and Random Forests

Instead of paying for a *route planner* such as Komoot,[1] why not try to build our own? OpenStreetMap (OSM) offers map data. From this we can get edges and features of edges such as highway classification (`path`, `track`, `primary road`, ...), surface (`sand`, `asphalt`, `gravel`, ...), and access restrictions. Binary features indicate whether an edge lies in a protected area or crosses a bridge or tunnel. We also compute *spatial features* ourselves, for instance the distance to the nearest forest, water, or motorway. Then we need labeled edges that tell us whether an edge is good (0) or bad (1). We label edges in published walks such as the Pieterpad as good. We call an edge bad if it lies in an industrial zone or next to a motorway. With these labeled edges, we can train a classifier that can score new, unlabeled edges. We convert these scores into edge costs, find a minimum-cost path from a point A where we want to start our walk to the finish B, and plot it on a map. In this chapter we show how to set up a classifier for our map, which contains about 10^7 edges.

[1] A mobile app for navigation and route planning, Wikipedia: Komoot.

A *binary decision tree* can score an edge by asking yes/no questions about its features: *is the highway type a major road? is the distance to the nearest forest below 100 m? is the surface paved?* Each answer selects a branch. At a leaf, the tree returns the fraction of training edges in that leaf labeled bad. For our walking app, this fraction lies in $[0, 1]$ by construction.

In slightly more abstract terms,[2] a tree maps a feature vector x to a prediction $\hat{a} = T(x)$. Each internal node stores a split rule and two child nodes; each leaf stores a prediction.

[2] Wikipedia: Decision tree learning.

A single tree is easy to interpret, but deep trees can be unstable: small changes in the training data can change their splits and predictions. This is high variance. Bagging, random forests, and boosting are useful ensemble methods that combine trees to reduce this instability.

4.1 Decision trees

Consider two features: distance to a forest, $x(1)$, and distance to a primary road, $x(2)$, both in meters.[1] The left panel in Fig. 4.1 shows a few edges that have been labeled: $y = 0$ means that the edge is a good edge, and $y = 1$ means that it is a bad edge. We place the labels in the

[1] For the real app, we use many more features.

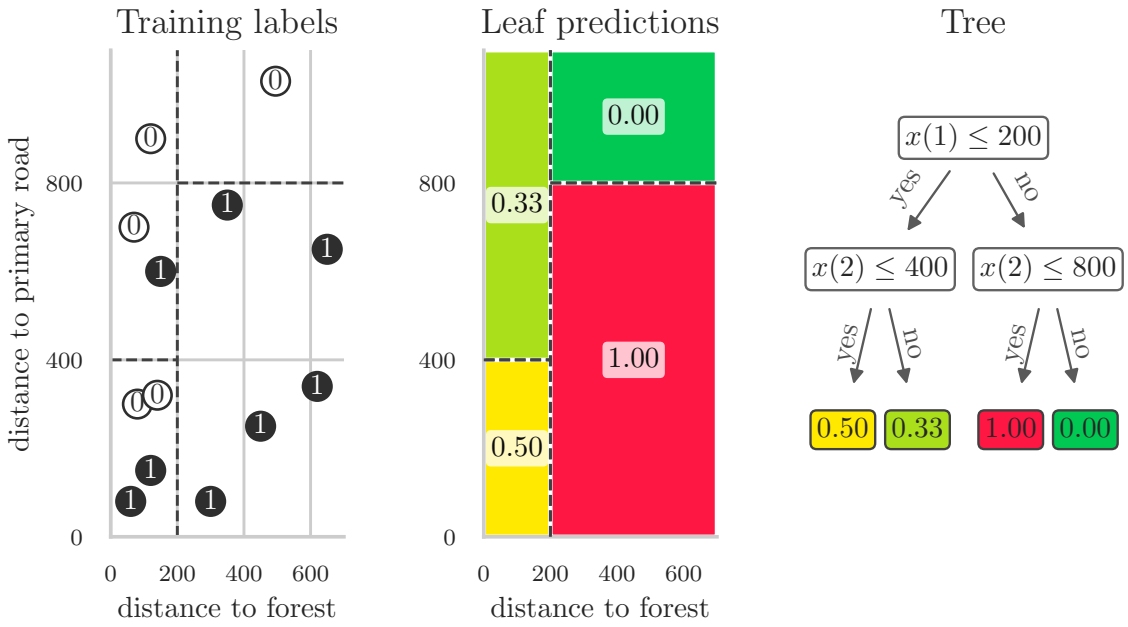


FIG. 4.1: A depth-two tree. Left: labeled training edges in feature space. Middle: the four leaf regions and their mean labels. Right: the corresponding split rules.

figure such that $x(1)$ measures `dist_to_forest` and $x(2)$ measures `dist_to_primary_road`, both in meters. If we use 200 to split on the distance to forest, and then use 400 or 800 for the primary road on the left or right branch respectively, the splits place 4 samples in the lower-left rectangle. Their mean label is 0.5, which becomes the leaf prediction. The tree predicts an estimated probability 0.5 for an edge routed to this rectangle. For the new edge $x_{\text{new}} = (60, 700)$, the tree follows the left branch and then the right branch, reaching the leaf with prediction 0.33. The right panel shows this traversal.

FITTING A TREE means choosing its splits and leaf predictions to minimize training risk. A leaf gives a constant prediction c to every sample that reaches it, so the quantities we need are exactly those of Chapter 3: the empirical risk $\hat{R}_{\mathcal{S}}(c)$ of using c throughout a list $\mathcal{S}$ of samples, the best such constant $\hat{c}(\mathcal{S})$, and the risk $\hat{R}_{\mathcal{S}}(\hat{c}(\mathcal{S}))$ that it attains. For short, write

$$\hat{R}(\mathcal{S}) = \hat{R}_{\mathcal{S}}(\hat{c}(\mathcal{S})).$$

For regression, these are given by (3.2.3), while for classification we use .

The leaves of a fitted tree form a partition $\mathcal{P}_{\mathcal{T}}(T) = \{\mathcal{T}_1, \ldots, \mathcal{T}_m\}$ of the training set $\mathcal{T}$ such that $\mathcal{T}_t$ is the list of samples belonging to a leaf t. The training risk of T is therefore[2]

$$\hat{R}_{\mathcal{T}}(T) = \frac{1}{|\mathcal{T}|} \sum_{t=1}^{m} \sum_{(x,y)\in\mathcal{T}_t} \ell(r(y), \hat{c}(\mathcal{T}_t)) = \frac{1}{|\mathcal{T}|} \sum_{t=1}^{m} |\mathcal{T}_t| \, \hat{R}(\mathcal{T}_t). \tag{4.1.1}$$

[2] Recall, on a leaf $\hat{c}(\mathcal{T}_t)$ is constant.

The training goal can now be stated as an optimization problem[3]

$$\min_{T\in\mathcal{F}} \sum_{\mathcal{S}\in\mathcal{P}_{\mathcal{T}}(T)} \frac{|\mathcal{S}|}{|\mathcal{T}|} \hat{R}(\mathcal{S}), \tag{4.1.2}$$

[3] The summation runs over all leaves $\mathcal{S}$ of the tree T that form the partition of $\mathcal{T}$.

where the model class $\mathcal{F}$ is the set of all trees of a given maximal depth.

IN GENERAL the best tree by exhaustive search is impossible because the number of candidate trees grows rapidly with depth. Let us estimate that growth.

For a binary tree we use a feature coordinate to split the n samples into a left set with l elements and a right set with $n-l$ elements. Assuming distinct feature values, the number of candidate splits at this node is $(n-1)p$.[4] . Assume the root node has depth at most h.[5] Then the number of trees $N_h(n)$ when the depth is h and the number of samples is n satisfies the recursion

$$N_h(n) = p \sum_{l=1}^{n-1} N_{h-1}(l)\, N_{h-1}(n-l), \qquad N_0(n) = 1;$$

when $h = 0$ we stop splitting so there is just one tree left. Suppose $N_h(n) \sim n^{u_h} p^{v_h}$. Matching the exponents of p in the above master equation gives

$$v_h = 1 + 2v_{h-1}, \qquad v_0 = 0, \qquad \Longrightarrow\ v_h = 2^h - 1.$$

For u_h, we should solve

$$n^{u_h} = \sum_{l=1}^{n-1} l^{u_{h-1}} (n-l)^{u_{h-1}}.$$

Approximating the sum by an integral,

$$\sum_{l=1}^{n-1} l^u (n-l)^u = n^{2u} \sum_{l=1}^{n-1} \left(\frac{l}{n}\right)^u \left(1 - \frac{l}{n}\right)^u \simeq n^{2u+1} \int_0^1 x^u (1-x)^u \,\mathrm{d}x.$$

The (beta) integral does not depend on n. So, again matching exponents in the master equation gives

$$u_h = 1 + 2u_{h-1}, \qquad u_0 = 0, \qquad \Longrightarrow\ u_h = 2^h - 1.$$

Therefore, the rough growth estimate for the number of trees with at most h levels of splits is

$$N_h(n) \sim (n\,p)^{2^h - 1}.$$

For our case, $n \approx 10^5$, $p = 30$, and $h = 5$, this is about 10^{200}.

GREEDY SPLITTING is the customary way around the search over all possible trees in (4.1.2). Suppose the current subtree is to be fitted on $\mathcal{T}$. Then we compare two options: make the current node a leaf, or split $\mathcal{T}$ into a left part $\mathcal{L}$ and a right part $\mathcal{R}$ and assume that each part becomes a leaf. The risk for the first option is $\hat{R}(\mathcal{T})$, that of the second is

$$\frac{|\mathcal{L}|}{|\mathcal{T}|}\, \hat{R}(\mathcal{L}) + \frac{|\mathcal{R}|}{|\mathcal{T}|}\, \hat{R}(\mathcal{R}). \tag{4.1.3}$$

If we can split $\mathcal{T}$ such that the risk of the second option is smaller than that of the first, we should split and fit left and right subtrees on their respective parts. By recursion, the risk of the left branch is $\hat{R}(\mathcal{L})$, or smaller if it is better to split $\mathcal{L}$ too. Since the argument also holds for $\mathcal{R}$, we obtain $\hat{R}_{\mathcal{T}}(T) \le \hat{R}(\mathcal{T})$.

The task is therefore to find the split that minimizes the risk for the second option. For each feature j, try thresholds at its observed values. A threshold θ sends z left when $z(j) \le \theta$, and right otherwise. Keep the split with the lowest weighted leaf risk, provided it improves on leaving the node unsplit. If none does, return NIL. The threshold can instead be placed at a midpoint between consecutive distinct feature values.[6]

[4] We can use each of the p features to split, and then split between any of the n samples.

[5] That is, the number of decisions until reaching a leaf node.

[6] This gives the same partition of the training samples.

THE TREE CLASS is implemented in Alg. 4.1.1. The constructor takes a maximum depth and a minimum split size *min_elements*.[7] A node becomes a leaf when its remaining depth is zero or it contains fewer than *min_elements* samples. Each child receives one less unit of remaining depth. This minimum split size does not prevent a split from producing a singleton leaf. In Fig. 4.1, the initial tree has depth 2, and the leaf node for the lower-left rectangle has depth 0. The effect of the depth parameter is illustrated in Fig. 4.2, where regression trees of depth 0, 1, and 2 are fitted to the function $f(x) = x^2$ on $[-2, 2]$.

[7] These are the tree's hyperparameters.

Algorithm 4.1.1 A binary tree class.

Input: the maximum depth and the minimum number of samples required to split a node.
Output: a tree with fit and predict methods.
class TREE(*depth*, *min_elements*)
  instance variables:
    left_tree ← NIL
    right_tree ← NIL
    leaf? ← FALSE ▷ *Is this node a leaf?*
    prediction ← NIL ▷ *The prediction when the node is a leaf.*
  methods:
    PREDICT(x) ▷ *Return the prediction for the feature vector x.*
    FIT($\mathcal{T}$) ▷ *Fit the tree on a training set $\mathcal{T}$.*
    CHOOSEBRANCH(x) ← NIL ▷ *Is x in the left or right subtree?*

Alg. 4.1.2 shows how to set up and use a TREE for the example in Fig. 4.1. As in Chapter 3, the training data form a list $\mathcal{T}$ of observations (x, y). Here x is the feature vector of an edge and y is its observed label. Once fitted, the tree can return a numerical prediction for a feature vector x by asking a short sequence of questions about each feature coordinate $x(j)$.

Algorithm 4.1.2 Fit a tree on the training data $\mathcal{T}$of Fig. 4.1, then compute a prediction.

tree ← TREE(*depth* = 2, *min_elements* = 5)
tree.FIT($\mathcal{T}$) ▷ *Fit the tree on the training set.*
$x \leftarrow [60, 700]$
prediction ← *tree*.PREDICT(x) ▷ *Returns the prediction 0.33.*

THE FUNCTION TREE.PREDICT returns $\hat{a} = T(x)$ for a feature vector x, see Alg. 4.1.3. If the current node is a *leaf*, i.e., when the member variable *leaf?* is TRUE,[fn:: Read *leaf?* like a question: is this a leaf? More generally, any function or variable with a question mark should be read like "Is this ...?".} the method returns the member variable *prediction*.[8] The leaf node in Fig. 4.1 corresponding to the lower-left rectangle has prediction 0.5. If the current node is not a leaf, the function CHOOSEBRANCH delegates the prediction to the left or right subtree.[9]

[8] *leaf?* and *prediction* are set by TREE.FIT.

[9] Note, each tree has its own private CHOOSEBRANCH function.

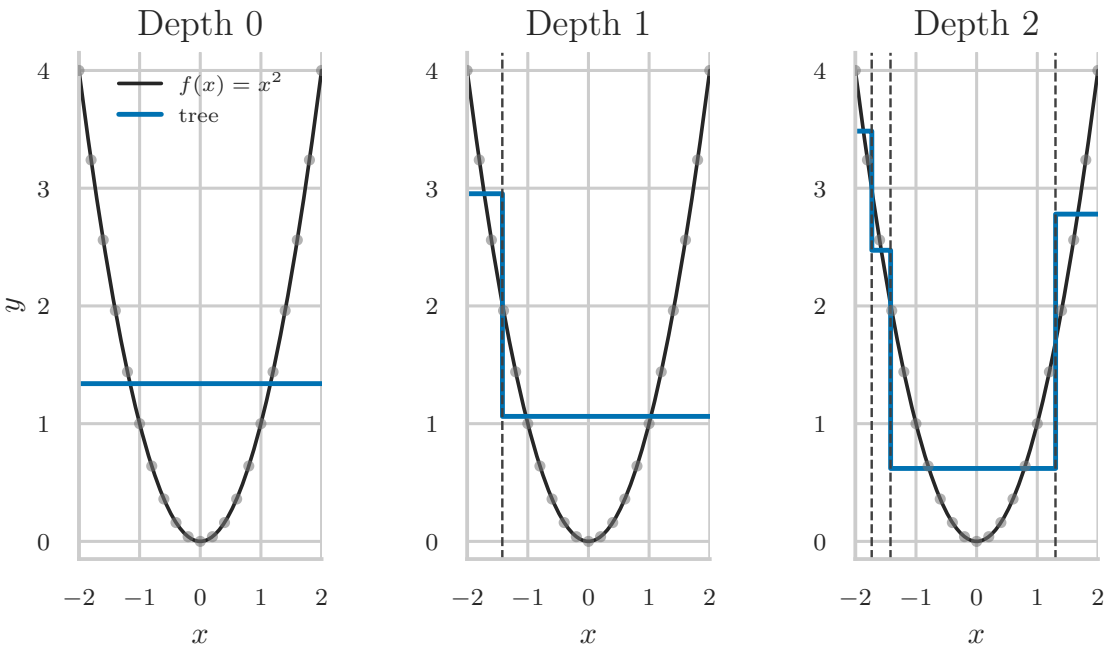


FIG. 4.2: Regression trees of depth 0, 1, and 2 fitted to $f(x) = x^2$ on $[-2, 2]$. Each panel shows the target function, the fitted piecewise-constant tree prediction, and the split points selected by the greedy tree-fitting procedure.

Algorithm 4.1.3 Predict for a feature vector x using the fitted tree.

Input: a feature vector x
Output: a prediction
procedure TREE.PREDICT(x)
    if *leaf?* **then**
        return *prediction*
    if CHOOSEBRANCH(x) = LEFT **then**
        return *left_tree*.PREDICT(x)
    return *right_tree*.PREDICT(x)

Algorithm 4.1.4 Find the best split rule for a training set $\mathcal{T}$.

Input: a non-empty training set $\mathcal{T}$.
Output: a split rule, or NIL if no split into two non-empty parts lowers the training risk
procedure BESTSPLITRULE($\mathcal{T}$)
    $min\_risk \leftarrow \hat{R}(\mathcal{T})$ ▷ *Risk if the node becomes a leaf.*
    $best\_rule \leftarrow$ NIL
    for $1 \le j \le p$ **do** ▷ *Loop over the feature coordinates*
        for $(x, y) \in \mathcal{T}$ **do** ▷ *Loop over the training samples*
            $\theta \leftarrow x(j)$ ▷ *Value of feature coordinate j of this sample*
            CHOOSEBRANCH $\leftarrow \lambda : z \mapsto$ LEFT if $z(j) \le \theta$ else RIGHT
            $\mathcal{L} \leftarrow [(u, v) \in \mathcal{T} : \text{CHOOSEBRANCH}(u) = \text{LEFT}]$
            $\mathcal{R} \leftarrow [(u, v) \in \mathcal{T} : \text{CHOOSEBRANCH}(u) = \text{RIGHT}]$
            if $|\mathcal{L}| > 0$ and $|\mathcal{R}| > 0$ **then**
                $risk \leftarrow \frac{|\mathcal{L}|}{|\mathcal{T}|} \hat{R}(\mathcal{L}) + \frac{|\mathcal{R}|}{|\mathcal{T}|} \hat{R}(\mathcal{R})$
                if $risk < min\_risk$ **then**
                    $min\_risk \leftarrow risk$
                    $best\_rule \leftarrow$ CHOOSEBRANCH
    return *best_rule*

This implementation takes $O(n^2 p)$ time: it tries np thresholds and scans all n samples for each.

TREE.FIT USES RECURSION to fit the tree, see Alg. 4.1.5.[10] If a stopping condition holds, Fit stores a leaf prediction. Otherwise, it finds the best split and fits each child on the samples sent to it. Each child receives one less unit of remaining depth, so a stopping condition is eventually reached. SETPREDICTION($\mathcal{T}$) returns the sample mean $\hat{c} = \hat{y} = \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} y$ for recursion, and the class frequencies $\hat{c}_p = |\,[(x, y) \in \mathcal{T} : y = k]\,|/|\mathcal{T}|$.

[10] In this pseudocode, FIT is called once on a newly constructed object.

Algorithm 4.1.5 Fit a tree recursively to a training set $\mathcal{T}$.

Input: a non-empty training set $\mathcal{T}$.
Output: none
procedure TREE.FIT($\mathcal{T}$)
    if STOPSPLITTING?($\mathcal{T}$, *depth*, *min_elements*) **then**
        leaf? $\leftarrow$ TRUE
        prediction $\leftarrow$ SETPREDICTION($\mathcal{T}$)
        return
    CHOOSEBRANCH $\leftarrow$ BESTSPLITRULE($\mathcal{T}$)
    left_tree $\leftarrow$ TREE(*depth* -1, *min_elements*)
    left_tree.FIT($[(x, y) \in \mathcal{T} : \text{CHOOSEBRANCH}(x) = \text{LEFT}]$)
    right_tree $\leftarrow$ TREE(*depth* -1, *min_elements*)
    right_tree.FIT($[(x, y) \in \mathcal{T} : \text{CHOOSEBRANCH}(x) = \text{RIGHT}]$)

STOPSPLITTING?, see Alg. 4.1.6, tells the current node whether to stop or continue splitting. When *depth* $= 0$ or the training set becomes too small, the node must become a leaf. It can also happen that BESTSPLITRULE returns NIL to indicate that there is no good split. Then the node must also become a leaf.[11]

[11] Compute the best split once and reuse it.

Algorithm 4.1.6 Decide whether the node should become a leaf.

Input: a non-empty training set $\mathcal{T}$, a depth, and a minimal split size.
Output: a Boolean.
procedure StopSplitting?($\mathcal{T}$, *depth*, *min_elements*)
    if *depth* = 0 **then**
        return True
    if $|\mathcal{T}| <$ *min_elements* **then**
        return True
    if BestSplitRule($\mathcal{T}$) = Nil **then**
        return True
    return False

4.2 Bags

Bagging reduces the variability of deep trees by averaging predictions from trees fitted to bootstrap samples. The name is short for *bootstrap aggregating*.[1]

[1] [[https://en.wikipedia.org/wiki/Bootstrap_aggregating][Wikipedia: Bootstrap aggregating].]

From the training set we draw a *bootstrap sample*: a new list formed by sampling n samples from $\mathcal{T}$ *with replacement*. We grow one tree on each such bootstrap sample and aggregate their predictions.

Why resample rather than simply cut $\mathcal{T}$ into as many parts as we want trees? Partitioning the data into B parts leaves each tree only n/B observations. Bootstrap samples overlap and retain much more of the training data for each tree. This can weaken each tree.

Bootstrap draws $|\mathcal{S}|$ elements from $\mathcal{S}$ with replacement.

Algorithm 4.2.1 Make a bootstrap sample of a list $\mathcal{S}$.

Input: a list $\mathcal{S}$.
Output: a list with $|\mathcal{S}|$ elements randomly selected with replacement from $\mathcal{S}$.
procedure Bootstrap($\mathcal{S}$)
    return SampleWithReplacement($\mathcal{S}$, $|\mathcal{S}|$)

Alg. 4.2.2 stores the fitted trees and their bootstrap indices. Its constructor takes the tree parameters and the number of trees.

Algorithm 4.2.2 A bag of binary decision trees.

Input: the number of trees in the bag, and the maximal depth and minimal split size of each tree.
Output: a bag with fit and predict methods.
class Bag(*num_trees*, *depth*, *min_elements*)
    instance variables:
        $\mathcal{B} \leftarrow []$ ▷ *The list of (tree, index list) pairs.*
    methods:
        Fit($\mathcal{T}$) ▷ *Fit the bag on a training set $\mathcal{T}$.*
        Predict(x) ▷ *Return a prediction for the feature vector x.*
        OobPredict(i) ▷ *Predict sample i with the trees that did not see it.*
        OobRisk($\mathcal{T}$) ▷ *Return the out-of-bag risk.*

The bag uses Alg. 4.2.3 to predict by letting every tree in the bag compute a prediction for the feature vector x, and then averaging the results. Each element of $\mathcal{B}$ is a pair $(T, \mathcal{I})$ where $\mathcal{I}$ is the list of indices the tree T was fitted on. The index list is not used in the prediction; we need it only for the out-of-bag risk below.

Algorithm 4.2.3 Predict for a feature vector x using the fitted bag.

Input: a feature vector x
Output: a prediction
procedure BAG.PREDICT(x)
    return AVERAGE($[T.\text{PREDICT}(x) : (T, \mathcal{I}) \in \mathcal{B}]$)

Alg. 4.2.4 grows the bag one tree at a time. For each tree, draw n training indices with replacement, fit the tree to those observations, and store both the tree and its index list. Different bootstrap samples can produce different trees.

Algorithm 4.2.4 Fit a bag to a training set $\mathcal{T}$.

Input: a non-empty training set $\mathcal{T}$
Output: none
procedure BAG.FIT($\mathcal{T}$)
    while $|\mathcal{B}| <$ *num_trees* **do**
        $\mathcal{I} \leftarrow$ BOOTSTRAP($[1, \ldots, n]$)
        $T \leftarrow$ TREE(*depth*, *min_elements*)
        T.FIT($[(x_i, y_i) : i \in \mathcal{I}]$)
        $\mathcal{B} \leftarrow \mathcal{B} + [(T, \mathcal{I})]$

The trees of a bag are usually grown deep: the averaging controls the variance that depth introduces, so each tree may fit its own bootstrap sample closely. In practice this means barely restricting the recursion at all. Rather than choosing a *depth*, one drops the depth test of Alg. 4.1.6 and keeps the other two: the node becomes a leaf when it holds fewer than *min_elements* samples, with *min_elements* set to a small number, or when BESTSPLITRULE finds no split that lowers $\hat{R}(\mathcal{T})$.

OUT-OF-BAG VALIDATION lets us tune the bag without a separate validation set. Call the b-th tree *out-of-bag* for sample i when $i \notin \mathcal{I}_b$, that is, when the tree was not fitted on that sample. This happens with probability $(1 - 1/n)^n$, which tends to $e^{-1} \approx 0.37$ as n grows.[2] About a third of the trees are therefore out-of-bag for sample i, and their average prediction for it can be compared with y_i without setting any data aside. Thus, we can use this risk to select the hyperparameters of the bag. Afterward, reserve untouched test data for final assessment.

Two elements of the bag serve this purpose. First, a tree must know which samples it did *not* train on, which is why Alg. 4.2.4 stores the index list $\mathcal{I}_b$ next to the tree. Second, the bag needs a prediction method of its own: Alg. 4.2.3 averages over *all* trees, and using it here would evaluate each tree on data it was fitted on. Alg. 4.2.5 therefore averages over the out-of-bag trees only, and Alg. 4.2.6 compares those averages with the labels. We state the risk for regression. The bag retains the training data, so OOBPREDICT can retrieve x_i from the input index i.

[2] One draw picks sample i with probability $1/n$, hence misses it with probability $1 - 1/n$. The draws are independent, so m draws all miss it with probability $(1 - 1/n)^m$. A bootstrap has as many draws as $\mathcal{T}$ has samples, so $m = n$.

Algorithm 4.2.5 Out-of-bag prediction for sample i.

Input: an index i.
Output: a prediction for sample i.
procedure BAG.OOBPREDICT(i)
    $\mathcal{V} \leftarrow [T : (T, \mathcal{I}) \in \mathcal{B}, i \notin \mathcal{I}]$ ▷ *The trees that are out-of-bag for sample i.*
    if $\mathcal{V} = []$ **then**
        return NIL
    return AVERAGE($[T.\text{PREDICT}(x_i) : T \in \mathcal{V}]$)

Algorithm 4.2.6 The out-of-bag risk of the bag; the regression case.

1: **Input:** the training set $\mathcal{T}$ the bag was fitted on.
2: **Output:** the out-of-bag risk.
3: **procedure** BAG.OOBRISK($\mathcal{T}$)
4: $\quad\mathcal{W} \leftarrow []$
5: $\quad$**for** $1 \le i \le n$ **do**
6: $\quad\quad\hat{y} \leftarrow$ BAG.OOBPREDICT(i)
7: $\quad\quad$**if** $\hat{y} \neq$ NIL **then**
8: $\quad\quad\quad\mathcal{W} \leftarrow \mathcal{W} + \left[(\hat{y} - y_i)^2\right]$
9: $\quad$**if** $\mathcal{W} = []$ **then**
10: $\quad\quad$**return** NIL
11: $\quad$**return** AVERAGE($\mathcal{W}$)

The NIL case occurs when sample i was drawn into every bootstrap, so that no tree is out-of-bag for it. Such samples are skipped so that the average runs only over the samples that have at least one out-of-bag tree. If all samples are skipped, OOBRISK returns NIL.[3]

The remaining hyperparameter *num_trees* is not chosen this way, since (4.2.1) below shows that the variance decreases in the number of trees; we take as many trees as we can afford to compute.

[3] A given sample has no out-of-bag tree with probability $(1-(1-1/n)^n)^B \approx 0.63^B$, which is about 10^{-4} for $B = 20$ trees.

THE VARIANCE OF A BAG has an interesting form. For a fixed training set $\mathcal{T}$ Alg. 4.2.4 draws for the b-th pass of the loop a bootstrapped index list $\mathcal{I}_b = (I_{b,1}, \ldots, I_{b,n})$, and then Alg. 4.1.5 fits a deterministic tree on the samples $\left[(x_i, y_i) : i \in \mathcal{I}_b\right]$. Hence, there is a function h such that $T_b = h(\mathcal{I}_b, \mathcal{T})$ is a predictor function.

Lemma 4.2.7. Conditional on $\mathcal{T}$, the predictors $T_1, \ldots, T_B$ are iid.

Proof. Condition on $\mathcal{T}$, so that $h(\cdot, \mathcal{T})$ is a fixed function. The index lists $\mathcal{I}_1, \ldots, \mathcal{I}_B$ are independent and identically distributed, each consisting of n uniform draws on $\{1, \ldots, n\}$. Measurable functions of independent random variables are independent, hence so are the $T_b = h(\mathcal{I}_b, \mathcal{T})$, $b = 1, \ldots, B$, and they are identically distributed because the $\mathcal{I}_b$ are. □

For a new observation X the bag predicts the average

$$\bar{T}(X) = \frac{1}{B} \sum_{b=1}^{B} T_b(X).$$

We can use Eve's law to split the variance into

$$\mathsf{V}\left[\bar{T}(X)\right] = \mathsf{E}\left[\mathsf{V}\left[\frac{1}{B}\sum_{b=1}^{B} T_b(X)\,\middle|\,\mathcal{T}\right]\right] + \mathsf{V}\left[\mathsf{E}\left[\frac{1}{B}\sum_{b=1}^{B} T_b(X)\,\middle|\,\mathcal{T}\right]\right].$$

With respect to the right most term:[4]

$$\mathsf{E}\left[\frac{1}{B}\sum_{b=1}^{B} T_b(X)\,\middle|\,\mathcal{T}\right] = \mathsf{E}\left[T_1(X)\,|\,\mathcal{T}\right],$$

because the T_b are identically distributed as T_1. For the other term, we use that the T_b are iid conditional on $\mathcal{T}$,

$$\mathsf{V}\left[\frac{1}{B}\sum_{b=1}^{B} T_b(X)\,\middle|\,\mathcal{T}\right] = \frac{1}{B^2}\sum_{b=1}^{B}\mathsf{V}\left[T_b(X)\,|\,\mathcal{T}\right] = \frac{1}{B}\mathsf{V}\left[T_1(X)\,|\,\mathcal{T}\right].$$

Thus, by increasing B, the variance of the prediction of X becomes smaller. All in all

$$\mathsf{V}\left[\bar{T}(X)\right] = \frac{1}{B}\mathsf{E}\left[\mathsf{V}\left[T_1(X)\,|\,\mathcal{T}\right]\right] + \mathsf{V}\left[\mathsf{E}\left[T_1(X)\,|\,\mathcal{T}\right].\right] \tag{4.2.1}$$

Consequently, the variance of the expected prediction of $T_1(X)$ due to fitting on samples of the training set cannot be removed by increasing the size of the bag.

We can rewrite the above in a slightly different way. By Eve's law

$$\mathsf{V}\left[T_1(X)\right] = \mathsf{E}\left[\mathsf{V}\left[T_1(X)\,|\,\mathcal{T}\right]\right] + \mathsf{V}\left[\mathsf{E}\left[T_1(X)\,|\,\mathcal{T}\right]\right],$$

so that

$$\mathsf{V}\left[\bar{T}(X)\right] = \frac{1}{B}\mathsf{V}\left[T_1(X)\right] + \frac{B-1}{B}\mathsf{V}\left[\mathsf{E}\left[T_1(X)\,|\,\mathcal{T}\right]\right]$$

Next, two trees of one bag are identically distributed, so $\mathsf{V}\left[T_2\right] = \mathsf{V}\left[T_1\right]$, and therefore the correlation can be written as

$$\rho = \frac{\mathsf{Cov}\left[T_1(X), T_2(X)\right]}{\sqrt{\mathsf{V}\left[T_{1(X)}\right]\mathsf{V}\left[T_{2(X)}\right]}} = \frac{\mathsf{Cov}\left[T_1(X), T_2(X)\right]}{\mathsf{V}\left[T_1(X)\right]}.$$

From the law of total covariance,[5]

$$\mathsf{Cov}\left[T_1(X), T_2(X)\right] = \mathsf{E}\left[\mathsf{Cov}\left[T_1(X), T_2(X)\,|\,\mathcal{T}\right]\right] + \mathsf{Cov}\left[\mathsf{E}\left[T_1(X)\,|\,\mathcal{T}\right], \mathsf{E}\left[T_2(X)\,|\,\mathcal{T}\right]\right].$$

Conditional independence makes the first term zero, and in the second both conditional expectations have the same distribution, hence,

$$\mathsf{Cov}\left[T_1(X), T_2(X)\right] = \mathsf{V}\left[\mathsf{E}\left[T_1(X)\,|\,\mathcal{T}\right]\right].$$

And so,

$$\rho = \frac{\mathsf{V}\left[\mathsf{E}\left[T_1(X)\,|\,\mathcal{T}\right]\right]}{\mathsf{V}\left[T_1(X)\right]} \qquad \Longrightarrow \mathsf{V}\left[\mathsf{E}\left[T_1(X)\,|\,\mathcal{T}\right]\right] = \rho\,\mathsf{V}\left[T_1(X)\right].$$

Substitution in the above expression for the variance gives

$$\mathsf{V}\left[\bar{T}(X)\right] = \rho\,\mathsf{V}\left[T_1(X)\right] + \frac{1-\rho}{B}\mathsf{V}\left[T_1(X)\right].$$

[4] Here we pick the first, but any of the trees would do.

[5] For random variables U, V and W, $\mathsf{Cov}\left[U,V\right] = \mathsf{E}\left[\mathsf{Cov}\left[U,V\,|\,W\right]\right] + \mathsf{Cov}\left[\mathsf{E}\left[U\,|\,W\right], \mathsf{E}\left[V\,|\,W\right]\right]$. With $U = V$ this is the law of total variance.

[1] Wikipedia: Random forest.

4.3 Random forests

A *random forest*[1] extends bagging with one further source of randomness. When one feature dominates, many bagged trees choose it for their first split. A random forest samples a subset of candidate features at each node.

The conceptual change in the code is small. Where a normal tree uses the full feature set in the feature loop of Alg. 4.1.4, a random forest only considers *num_candidates* of them. We use Alg. A.0.9

$$\text{SAMPLEWITHOUTREPLACEMENT}(\left[1,\ldots,p\right], \mathit{num\_candidates})$$

to select the features randomly. Common choices are $\mathit{num\_candidates} = \sqrt{p}$ or $\mathit{num\_candidates} = p/3$, rounded to the nearest integer. A forest is otherwise a bag: it grows *num_trees* trees on bootstrap samples and averages them, and its out-of-bag risk is Alg. 4.2.6.

Using fewer candidate features can reduce correlation, but can also weaken the individual trees. Choose the number of candidates by out-of-bag risk. Using all p features recovers bagging.

4.4 Exercises

Exercise (T) 4.4.1. Consider a version of the first tree figure in which the text inside the tree nodes has been removed, while the tree structure, the arrows, and the "yes"/"no" labels remain visible.

Complete the tree by filling in the split rule at each internal node and the prediction at each leaf. Use the training labels and leaf regions shown in the figure. The left branch corresponds to "yes" and the right branch to "no".

Exercise (A) 4.4.2. Complete the ??? part.

```
Input: a feature vector x
Output: a prediction
procedure TREE.PREDICT(x)
    if leaf? then
        return prediction
    if ???? then
        return left_tree.PREDICT(x)
    return right_tree.PREDICT(x)
```

Exercise (A) 4.4.3. Complete the ??? part.

Input: a non-empty training set $\mathcal{T}$.
Output: a split rule, or NIL if no split into two non-empty parts lowers the training risk
procedure BESTSPLITRULE($\mathcal{T}$)
    $min\_risk \leftarrow \hat{R}(\mathcal{T})$ ▷ *Risk if the node becomes a leaf.*
    $best\_rule \leftarrow$ NIL
    for $1 \leq j \leq p$ **do** ▷ *Loop over the feature coordinates*
        for $(x, y) \in \mathcal{T}$ **do** ▷ *Loop over the training samples*
            $\theta \leftarrow x(j)$ ▷ *Value of feature coordinate j of this sample*
            CHOOSEBRANCH $\leftarrow$????
            $\mathcal{L} \leftarrow [(u, v) \in \mathcal{T} : \text{CHOOSEBRANCH}(u) = \text{LEFT}]$
            $\mathcal{R} \leftarrow [(u, v) \in \mathcal{T} : \text{CHOOSEBRANCH}(u) = \text{RIGHT}]$
            if $|\mathcal{L}| > 0$ and $|\mathcal{R}| > 0$ **then**
                $risk \leftarrow \frac{|\mathcal{L}|}{|\mathcal{T}|} \hat{R}(\mathcal{L}) + \frac{|\mathcal{R}|}{|\mathcal{T}|} \hat{R}(\mathcal{R})$
                if $risk < min\_risk$ **then**
                    $min\_risk \leftarrow risk$
                    $best\_rule \leftarrow$ CHOOSEBRANCH
    return *best_rule*

Exercise (A) 4.4.4. Complete lines 5 and 6 of the TREE.FIT algorithm.

Input: a non-empty training set $\mathcal{T}$.
Output: none
procedure TREE.FIT($\mathcal{T}$)
    if STOPSPLITTING?($\mathcal{T}$, *depth*, *min_elements*) **then**
        ???
        ???
        return
    CHOOSEBRANCH $\leftarrow$ BESTSPLITRULE($\mathcal{T}$)
    left_tree $\leftarrow$ TREE(*depth* -1, *min_elements*)
    left_tree.FIT($\big[(x, y) \in \mathcal{T} : \text{CHOOSEBRANCH}(x) = \text{LEFT}\big]$)
    right_tree $\leftarrow$ TREE(*depth* -1, *min_elements*)
    right_tree.FIT($\big[(x, y) \in \mathcal{T} : \text{CHOOSEBRANCH}(x) = \text{RIGHT}\big]$)

Exercise (A) 4.4.5. In Alg. 4.2.6, why does the algorithm test whether $\hat{y} \neq$ NIL before adding a loss to $\mathcal{W}$? Explain what 'Nil' means in this context and why including such a case in the average would give an incorrect out-of-bag risk.

Exercise (A) 4.4.6. Random forests use random feature selection at each node. Which line in Alg. 4.1.4 must be changed to turn the split rule into the one used by a random forest, and how should it be changed?

Exercise (C) 4.4.7. Ordinary least squares fits one affine function $\beta_0 + \langle \beta, x \rangle$ to the whole feature space. A tree is not of this form. Explain why a tree can capture a non-linear relation between the features and the label, and say what shapes it still has trouble with.

Exercise (C) 4.4.8. Does a shallow t Answer in terms of the number of leaves and the number of samples per leaf, and use the trees of depth 0, 1, and 2 in Fig. 4.2 as an illustration. What does the answer say about the depth at which the trees of a bag should be grown?

Exercise (T) 4.4.9. Derive (4.2.1).

Exercise (C) 4.4.10. In what sense is a random forest different from a bag? Why?

Exercise (C) 4.4.11. Explain (4.1.1) and (4.1.2).

Chapter 5

Gradient Descent and Gradient Boosting

The first part of this chapter shows how to use gradient descent to fit the parameters of prediction functions in the model class $\mathcal{F}$ to the training data $\mathcal{T}$. The second part discusses *boosting* which moves the prediction function to better fits rather than a parameter vector.

5.1 Gradient descent

5.1.1 The descent step

Write the model class as a family f_θ indexed by a parameter vector θ.[1] We write

$$\ell(\theta; x, y) = \ell(r(y), f_\theta(x))$$

for the loss of the parameters θ on the single sample (x, y). The training risk $\hat{R}_\mathcal{T}(f_\theta)$ then becomes a function of θ alone,

$$\hat{R}_\mathcal{T}(\theta) = \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \ell(\theta; x, y),$$

and we look for a θ that makes it small. We assume that $\hat{R}_\mathcal{T}$ is differentiable in θ, so that its gradient $\nabla\hat{R}_\mathcal{T}(\theta)$ exists.

THE IDEA is to read the gradient as advice on where to step next, see Fig. 5.1. From Taylor's formula, for a small step $\Delta\theta$,

$$\hat{R}_\mathcal{T}(\theta + \Delta\theta) \simeq \hat{R}_\mathcal{T}(\theta) + \left\langle \nabla\hat{R}_\mathcal{T}(\theta), \Delta\theta \right\rangle.$$

If $\nabla\hat{R}_\mathcal{T}(\theta) \neq 0$, then θ is not a minimizer.[2] The inner product tells us how the risk changes for a small step $\Delta\theta$. Among all steps of a given length, the one opposite to the gradient makes this inner product as negative as possible,[3] so this is the direction in which the risk drops fastest.[4] So take

$$\Delta\theta = -\eta\nabla\hat{R}_\mathcal{T}(\theta)$$

for a *learning rate* $\eta > 0$ that is small enough to keep Taylor's approximation valid. Then

$$\hat{R}_\mathcal{T}(\theta - \eta\nabla\hat{R}_\mathcal{T}(\theta)) \simeq \hat{R}_\mathcal{T}(\theta) - \eta\left\|\nabla\hat{R}_\mathcal{T}(\theta)\right\|^2 < \hat{R}_\mathcal{T}(\theta),$$

[1] For instance, for OLS $\theta = (\beta_0, \beta)$.

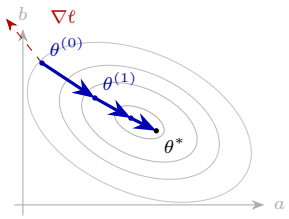


Fig. 5.1: Schematic view of gradient descent for a parameter vector θ. The arrows move opposite to the gradient, crossing level curves toward a minimizer.

[2] Because $\hat{R}_\mathcal{T}$ is differentiable.

[3] By the Cauchy-Schwarz inequality.

[4] If $\Delta\theta$ is orthogonal to the gradient, the first-order change in the risk is zero, and then only higher-order terms can change the risk.

so the step lowers the risk.

We apply this update rule iteratively, as in Alg. 5.1.1. Writing $\theta^{(k)}$ for the parameters after k steps, the iteration reads

$$\theta^{(k+1)} - \theta^{(k)} = \Delta\theta^{(k)} = -\eta \nabla \hat{R}_{\mathcal{T}}(\theta^{(k)}) \quad \Longrightarrow \quad \theta^{(k+1)} = \theta^{(k)} - \eta \nabla \hat{R}_{\mathcal{T}}(\theta^{(k)}). \tag{5.1.1}$$

The learning rate needs care. If η is very small, it takes many iterations for the algorithm to converge; if η is too large, the steps overshoot and the iterates may diverge.

Algorithm 5.1.1 Gradient descent for a parameterized model class.

Input: initial parameters θ, learning rate $\eta > 0$, tolerance $\epsilon > 0$.
Output: parameters θ with a small gradient.
procedure GRADIENTDESCENT(θ, η, ϵ)
  while $\left\|\nabla \hat{R}_{\mathcal{T}}(\theta)\right\|^2 > \epsilon$ **do**
    $\theta \leftarrow \theta - \eta \nabla \hat{R}_{\mathcal{T}}(\theta)$
  return θ

First, the gradient is an average over the whole training set, which is expensive when $\mathcal{T}$ is large. Second, a zero gradient marks a *local* minimum only, so where the descent ends can depend on where it starts.

5.1.2 Real-valued labels

Ordinary least squares is the natural first test of the above rule for $\theta^{(k)}$. Even though we already know how to solve this with linear algebra, running gradient descent on the same problem shows what the iteration computes.

Recall the model class and the objective (3.2.4). A rule in this class predicts

$$\hat{y} = \beta_0 + \langle \beta, x \rangle, \qquad \beta_0 \in \mathbb{R}, \quad \beta \in \mathbb{R}^p,$$

so the parameter vector $\theta = (\beta_0, \beta_1, \ldots, \beta_p)$ has $p+1$ components. With squared loss,[5] the loss on a single sample (x, y) is

$$\ell(\theta; x, y) = \tfrac{1}{2}\left(\beta_0 + \langle \beta, x \rangle - y\right)^2,$$

and the training risk is the average of these numbers over the training set,

$$\hat{R}_{\mathcal{T}}(\theta) = \frac{1}{2|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \left(\beta_0 + \langle \beta, x \rangle - y\right)^2.$$

[5] The factor $\frac{1}{2}$ changes no minimizer, so every result of Section 3.2.2 still applies; it only removes a factor 2 from the derivatives.

THE PARTIAL DERIVATIVES follow from the chain rule,[6] applied to one sample at a time:

$$\frac{\partial \ell}{\partial \beta_0}(\theta; x, y) = \beta_0 + \langle \beta, x \rangle - y, \qquad \frac{\partial \ell}{\partial \beta_j}(\theta; x, y) = \left(\beta_0 + \langle \beta, x \rangle - y\right) x(j).$$

[6] $\partial_{\beta_j} \langle \beta, x \rangle = x(j)$.

The gradient of the training risk is the average of these over $\mathcal{T}$,

$$\frac{\partial \hat{R}_{\mathcal{T}}}{\partial \beta_0}(\theta) = \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \left(\beta_0 + \langle \beta, x \rangle - y\right),$$

$$\frac{\partial \hat{R}_{\mathcal{T}}}{\partial \beta_j}(\theta) = \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \left(\beta_0 + \langle \beta, x \rangle - y\right) x(j), \qquad j = 1, \ldots, p,$$

and these $p+1$ numbers together form $\nabla \hat{R}_{\mathcal{T}}(\theta)$.

THE UPDATE RULE is what we get by inserting these derivatives into (5.1.1). Write $\theta^{(k)} = (\beta_0^{(k)}, \beta_1^{(k)}, \dots, \beta_p^{(k)})$ for the parameters after k steps, and $\hat{y}^{(k)}(x) = \beta_0^{(k)} + \langle \beta^{(k)}, x \rangle$ for the prediction these parameters make for a feature vector x. Then the step from $\theta^{(k)}$ to $\theta^{(k+1)}$ reads

$$\beta_0^{(k+1)} = \beta_0^{(k)} - \frac{\eta}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \left(\hat{y}^{(k)}(x) - y \right),$$
$$\beta_j^{(k+1)} = \beta_j^{(k)} - \frac{\eta}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \left(\hat{y}^{(k)}(x) - y \right) x(j), \qquad j = 1, \dots, p. \tag{5.1.2}$$

For the intercept, if the current rule predicts too high on average, the sum is positive and β_0 comes down. For a weight, each sample contributes its prediction error multiplied by its own $x(j)$, so samples with a large jth coordinate have more say in β_j than samples with a small one, and samples with $x(j) = 0$ have none at all.

THE LEARNING RATE can be discussed concretely here, because the training risk is quadratic in θ, so its matrix of second derivatives, the *Hessian*, does not depend on θ: it is fixed once and for all by the feature values in $\mathcal{T}$. Differentiating the gradient once more with respect to β_j gives the average of $x(j)^2$ over the training set, and this is the curvature that the step in the direction of β_j has to respect. A single η must serve all $p+1$ directions at once, and it is the *largest* curvature that limits how large η may be before the steps overshoot. Suppose now that one feature coordinate is measured in meters and another in kilometers. Their averages of $x(j)^2$ differ by a factor of a million, hence so do the eigenvalues of the Hessian: the rate that keeps the steep direction stable moves the flat one by almost nothing, and progress in that direction takes a million times as many iterations.[7] Standardizing the feature coordinates before fitting brings the eigenvalues together, so fewer iterations are needed.

[7] The ratio of the largest to the smallest eigenvalue of the Hessian is called the *condition number* of the problem; the number of iterations needed grows with it.

5.1.3 Probability labels

Probabilities as labels occur whenever the observed value is itself a proportion or a probability, for instance, the fraction of trail that is unpaved. The label space is now $\mathcal{Y} = [0,1]$, and the affine rule of the previous section will not do, because $\beta_0 + \langle \beta, x \rangle$ ranges over all of $\mathbb{R}$ and returns values outside $[0,1]$ for $|x|$ large.

The repair is to squash the affine score into the unit interval with the sigmoid. We first take a single feature, so that the model has two parameters, and

$$\hat{y} = \sigma(z), \qquad z = ax + b, \qquad \sigma(z) = \left(1 + e^{-z}\right)^{-1}, \tag{5.1.3}$$

with $\theta = (a, b)$, the *weight* a and the *bias* b. The affine score z is called the *linear predictor* or, in the language of neural networks, the *logit*.

Keeping the squared loss of the previous section, the loss on one sample is

$$\ell(\theta; x, y) = \tfrac{1}{2}\left(\hat{y} - y\right)^2, \qquad \hat{y} = \sigma(ax + b).$$

THE PARTIAL DERIVATIVES now need the chain rule twice, since the parameters reach the loss through z and then through σ. The sigmoid has the convenient derivative

$$\sigma'(z) = \sigma(z)\,(1 - \sigma(z)) = \hat{y}(1 - \hat{y}),$$

and $z = ax+b$ contributes $\partial z/\partial a = x$ and $\partial z/\partial b = 1$. Multiplying the three factors,

$$\frac{\partial \ell}{\partial a}(\theta; x, y) = \big(\hat{y} - y\big)\, \hat{y}(1-\hat{y})\, x, \qquad \frac{\partial \ell}{\partial b}(\theta; x, y) = \big(\hat{y} - y\big)\, \hat{y}(1-\hat{y}).$$

Writing $\hat{y}^{(k)}(x) = \sigma(a^{(k)} x + b^{(k)})$ for the prediction of the current parameters, the update (5.1.1) becomes

$$\begin{aligned} a^{(k+1)} &= a^{(k)} - \frac{\eta}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \Big(\hat{y}^{(k)}(x) - y\Big)\, \hat{y}^{(k)}(x) \Big(1 - \hat{y}^{(k)}(x)\Big)\, x, \\ b^{(k+1)} &= b^{(k)} - \frac{\eta}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \Big(\hat{y}^{(k)}(x) - y\Big)\, \hat{y}^{(k)}(x) \Big(1 - \hat{y}^{(k)}(x)\Big). \end{aligned} \tag{5.1.4}$$

THREE THINGS CHANGED with respect to the previous section, and all three come from the sigmoid. First, there is no closed form to fall back on: setting the derivatives to zero gives equations in which a and b sit inside σ, so they cannot be separated into a linear system. Second, the training risk need no longer be convex in θ, so the caveat of the introduction becomes relevant: different starting points may end at different fitted parameters.

Third, and most important for what follows, the factor $\hat{y}(1-\hat{y})$ is at most $1/4$, and it is near zero exactly when $\hat{y}$ is near 0 or 1. To see the size of this effect, take $y = 0$ and compare two predictions. A hesitant $\hat{y} = 0.5$ contributes $\big(\hat{y} - y\big)\, \hat{y}(1-\hat{y}) = 0.5 \cdot 0.25 = 0.125$ per unit of x to the sum, while a confidently wrong $\hat{y} = 0.999$ contributes $0.999 \cdot 0.000999 \approx 0.001$. The sample that is almost maximally wrong therefore moves the parameters more than a hundred times less than the one that is hesitant. The reason is geometric: $\hat{y}$ near 1 means $z = ax + b$ lies far from 0, where the sigmoid is flat, so changing a or b hardly changes $\hat{y}$ and hardly changes the loss. This effect is called *saturation*.

5.1.4 Binary labels

Binary labels are such that $y \in \{0, 1\}$: the trail edge is bad or it is not, the email is spam or it is not. Since $\{0, 1\} \subset [0, 1]$, this is not a new label space at all, and the model (5.1.3) can stay exactly as it is. What changes is what we may say about the label. A fraction is a number we simply observe, and the loss that compares it with a prediction is ours to choose. A binary label is the outcome of a draw, and if we are willing to model that draw, then the loss follows from the model instead of being chosen by hand.

So assume that, given the features, the labels are independent, and that the model returns the probability of the outcome $y = 1$,

$$\mathsf{P}\{Y = 1 \mid x\} = \hat{y} = \sigma(ax + b).$$

Then a sample with $y = 1$ has probability $\hat{y}$ and one with $y = 0$ has probability $1 - \hat{y}$, which the single expression $\hat{y}^{y}(1-\hat{y})^{1-y}$ covers, so the *likelihood* of the observed labels is

$$L(a, b) = \prod_{(x,y)\in\mathcal{T}} \hat{y}(x)^{\,y} \big(1 - \hat{y}(x)\big)^{1-y}.$$

The parameters that make the observed labels most probable are those that maximize L, and because the logarithm is increasing, they are also

the ones that minimize $-\log L$, and hence $-\log L/|\mathcal{T}|$. This average is the training risk $\hat{R}_{\mathcal{T}}(a,b)$ when the loss on a single sample is the *binary cross-entropy loss*,

$$\ell(\theta; x, y) = -y \log \hat{y} - (1-y)\log(1-\hat{y}). \tag{5.1.5}$$

THE PARTIAL DERIVATIVES are again a chain rule. Using $\sigma'(z) = \hat{y}(1-\hat{y})$ once more,

$$\frac{\partial}{\partial a}\log\hat{y} = \frac{\hat{y}(1-\hat{y})}{\hat{y}}\frac{\partial z}{\partial a} = (1-\hat{y})x, \qquad \frac{\partial}{\partial b}\log\hat{y} = 1-\hat{y},$$

and the same calculation for $\log(1-\hat{y})$ gives

$$\frac{\partial}{\partial a}\log(1-\hat{y}) = -\hat{y}x, \qquad \frac{\partial}{\partial b}\log(1-\hat{y}) = -\hat{y}.$$

Hence,

$$\frac{\partial \ell}{\partial a}(\theta; x, y) = -\big(y(1-\hat{y})x - (1-y)\hat{y}x\big) = \big(\hat{y}-y\big)x, \quad \frac{\partial \ell}{\partial b}(\theta; x, y) = \hat{y} - y.$$

The update rule becomes

$$a^{(k+1)} = a^{(k)} - \frac{\eta}{|\mathcal{T}|}\sum_{(x,y)\in\mathcal{T}}\Big(\hat{y}^{(k)}(x) - y\Big)\,x, \quad b^{(k+1)} = b^{(k)} - \frac{\eta}{|\mathcal{T}|}\sum_{(x,y)\in\mathcal{T}}\Big(\hat{y}^{(k)}(x) - y\Big).$$

Compare this with (5.1.4): the model is the same, the data are the same, but the factor $\hat{y}(1-\hat{y})$ has disappeared. The prediction error remains, exactly as it was for OLS. So a confidently wrong prediction produces a large correction, and saturation is gone. Moreover, the training risk under the cross-entropy loss is convex in θ, so every point with zero gradient is a global minimizer, and the second limitation of the descent step does not apply.

Alg. 5.1.2 collects the steps. Fig. 5.2 shows the paths for four starting points. All four converge to the same limit, and the training risk falls quickly at first while the parameters themselves are still moving slowly.

Algorithm 5.1.2 Gradient descent for a sigmoid with binary labels.

Input: training set $\mathcal{T}$, learning rate $\eta > 0$, tolerance $\epsilon > 0$.
Output: parameters a, b.
Choose initial values a, b.
while $\big\|\nabla \hat{R}_{\mathcal{T}}(a,b)\big\|^2 > \epsilon$ **do**
  Compute $\hat{y} \leftarrow \sigma(ax+b)$ for every $(x,y) \in \mathcal{T}$.
  $\nabla_a \leftarrow \frac{1}{|\mathcal{T}|}\sum_{(x,y)\in\mathcal{T}}(\hat{y}-y)x$, $\nabla_b \leftarrow \frac{1}{|\mathcal{T}|}\sum_{(x,y)\in\mathcal{T}}(\hat{y}-y)$.
  $a \leftarrow a - \eta\nabla_a$, $b \leftarrow b - \eta\nabla_b$.
return a, b

5.1.5 Stochastic and mini-batch gradient descent

The cost per step of (5.1.2) is one pass over the entire training set. When the training set is large, this is the dominant cost of fitting, and it has to be paid again at every iteration.

[8] Wikipedia: Stochastic approximation.

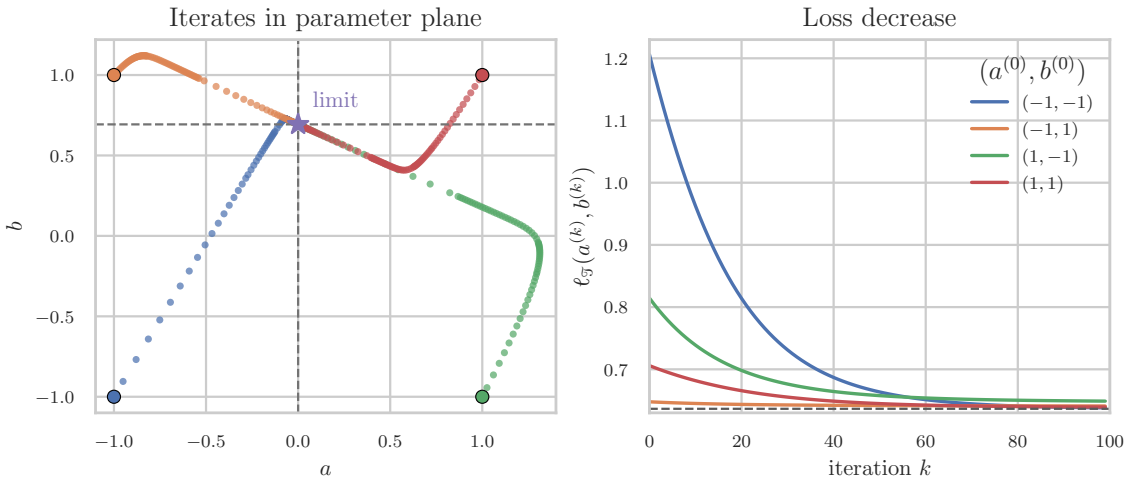


FIG. 5.2: Gradient descent for a training set consisting of the samples $(0.2, 1)$, $(0.5, 0)$, and $(0.8, 1)$, with learning rate $\eta = 0.1$. We consider four different starting points. Left: paths of the iterates $(a^{(k)}, b^{(k)})$. Right: training risk $\hat{R}_{\mathcal{T}}(a^{(k)}, b^{(k)})$ under the cross-entropy loss. All four paths converge to $(0, \log 2)$, with training risk about 0.637.

STOCHASTIC GRADIENT DESCENT [8] replaces the gradient over the entire set $\mathcal{T}$ in (5.1.1) by the gradient of a single sample (X, Y) drawn uniformly from $\mathcal{T}$. This replacement is unbiased,

$$\mathsf{E}\left[\nabla \ell(\theta; X, Y)\right] = \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \nabla \ell(\theta; x, y) = \nabla \hat{R}_{\mathcal{T}}(\theta),$$

because a uniform draw gives each of the $|\mathcal{T}|$ samples probability $1/|\mathcal{T}|$, so that the expectation is precisely the average that defines the full gradient. Thus the update $\theta^{(k+1)} = \theta^{(k)} - \eta \nabla \ell(\theta^{(k)}; X, Y)$ moves, on expectation, in the same direction as the full gradient descent of (5.1.1). The advantage is that each step is cheap as it uses the gradient of just one sample. The disadvantage is that the sequence of updates is noisy, as each step depends only on the sample that happens to be drawn.

MINI-BATCHING is a compromise often used in practice. Instead of one sample, choose a *batch* $\mathcal{B}$, a set of samples drawn at random from $\mathcal{T}$ without replacement,[9] and use the gradient of the empirical risk on that batch,

$$\nabla \hat{R}_{\mathcal{B}}(\theta) = \frac{1}{|\mathcal{B}|} \sum_{(x,y)\in\mathcal{B}} \nabla \ell(\theta; x, y), \tag{5.1.6}$$

as an approximation of the full gradient. By the same argument as for stochastic gradient descent, $\mathsf{E}\left[\nabla \hat{R}_{\mathcal{B}}(\theta)\right] = \nabla \hat{R}_{\mathcal{T}}(\theta)$. Thus mini-batch gradient descent still moves in the correct direction on average, but it is less noisy than pure stochastic gradient descent.

We still want every training sample in $\mathcal{T}$ to contribute to the parameter updates. We do this by splitting the training set $\mathcal{T}$ at random into disjoint batches and using each batch, in turn, to compute the mini-batch gradient in (5.1.6). One *training epoch* consists of one round in which all batches have been used once. One epoch contains $\lceil |\mathcal{T}|/|\mathcal{B}| \rceil$ parameter updates.[10]

Reshuffling the samples into new batches at the start of each epoch is important: it prevents the same samples from always appearing together.

[9] The batch size controls the tradeoff between cost and noise. There is no universally best size for $\mathcal{B}$. Small $\mathcal{B}$ is cheap but noisy. Large $\mathcal{B}$ is stable, but uses more memory and computation. Sizes are usually powers of two between 32 and 256.

[10] When $|\mathcal{B}|$ does not divide $|\mathcal{T}|$, the last batch has fewer than $|\mathcal{B}|$ samples.

5.2 Gradient Boosting

Boosting[1] is an algorithm that builds a prediction function by adding simple functions such as shallow trees one at a time. These simple functions are such that they are corrections on the samples where the prediction function that has been built so far performs badly. Where gradient descent lowers the training risk by moving a *parameter vector* against the gradient, boosting lowers the training risk by moving the *prediction function* against the gradient.

[1] Wikipedia: Gradient boosting.

The labels are numbers in all cases we consider below, so $r(y) = y$; we write $\ell(y, a)$ for the loss of the prediction a on a sample with label y.

We discuss the main ideas of gradient boosting first, then we apply it to regression and classification.

5.2.1 The general idea

Fix a *base class* $\mathcal{H}$ of prediction functions. Its members are called *base learners* or *weak learners*: each is a simple rule that, on its own, predicts only slightly better than guessing. A decision stump, a tree of depth one, is the simplest example.

Boosting is an *ensemble method*, like bagging, but the ensemble is a sum rather than an average. After B rounds the prediction function is the *additive model*

$$f_B(x) = f_0(x) + \eta \sum_{b=1}^{B} \rho_b \, h_b(x),$$

with base learners $h_b \in \mathcal{H}$, coefficients $\rho_b \in \mathbb{R}$, a *learning rate* $\eta > 0$, and a starting function f_0. Each h_b is simple, but the sum f_B need not be: with stumps as base learners and a single feature, f_B is a step function with up to B jumps. The model class $\mathcal{F}$ of Chapter 3 is here the set of all such sums. It grows with B, since every extra round adds one more weak learner to the sum, so a larger B permits a closer fit to the training set.

Boosting happens *forward stagewise*: at round b it holds f_{b-1} fixed and then asks for the pair (h_b, ρ_b) that lowers the training risk $\hat{R}_{\mathcal{T}}$ most:

$$(h_b, \rho_b) \in \operatorname*{arg\,min}_{h \in \mathcal{H},\, \rho \in \mathbb{R}} \frac{1}{|\mathcal{T}|} \sum_{(x,y) \in \mathcal{T}} \ell\big(y, f_{b-1}(x) + \rho \, h(x)\big). \tag{5.2.1}$$

Then it uses this pair to update

$$f_b(x) = f_{b-1}(x) + \eta \, \rho_b \, h_b(x). \tag{5.2.2}$$

Once a term has been added it is never revisited. Each round is therefore one small fitting problem instead of a joint search over B base learners at once. It also makes boosting greedy: nothing guarantees that f_B is the best sum of B members of $\mathcal{H}$.

The learning rate η scales every correction; with smaller η more rounds are needed to reach the same fit.

Any class $\mathcal{H}$ can be used in principle, but shallow regression trees are the usual choice. A tree splits on the features themselves and can capture non-linear behavior. Second, fitting a tree of small depth is fast, which matters when the fit is repeated B times. Third, shallow trees have low variance. The problem that a single weak learner is highly biased is mitigated by forming an ensemble.

GRADIENT BOOSTING IS a three step approach to approximate (5.2.1). The problem is that the training risk depends on f only through the $|\mathcal{I}|$ numbers $f(x_1),\dots,f(x_{|\mathcal{I}|})$, so we need to find a way to use these numbers to improve f.

Write $a=(a_1,\dots,a_{|\mathcal{I}|})$ with $a_i=f(x_i)$, and $a^{(b-1)}=(f_{b-1}(x_1),\dots,f_{b-1}(x_{|\mathcal{I}|}))$ for the predictions of the current f_{b-1}. From here on we require $\ell(y,a)$ to be differentiable in its second argument a, since every step below rests on the derivative $\partial\ell(y,a)/\partial a$. Then the gradient $\nabla_a \hat{R}_{\mathcal{I}}(a)\big|_{a=a^{(b-1)}}$ has as ith component

$$\frac{\partial}{\partial a_i}\frac{1}{|\mathcal{I}|}\sum_{j=1}^{|\mathcal{I}|}\ell(y_j,a_j)\bigg|_{a=a^{(b-1)}} = \frac{1}{|\mathcal{I}|}\frac{\partial\ell(y_i,a)}{\partial a}\bigg|_{a=f_{b-1}(x_i)}.$$

By dropping the factor[2] $1/|\mathcal{I}|$ we obtain the *pseudo-residual*

[2] This only rescales the step.

$$r_i^{(b)} = -\frac{\partial\ell(y_i,a)}{\partial a}\bigg|_{a=f_{b-1}(x_i)}, \qquad i=1,\dots,|\mathcal{I}|. \tag{5.2.3}$$

This residual says whether the prediction at x_i should go up or down to lower the loss of sample i. Together,

$$(r_1^{(b)},\dots,r_{|\mathcal{I}|}^{(b)}) = -|\mathcal{I}|\,\nabla_a \hat{R}_{\mathcal{I}}(a)\big|_{a=a^{(b-1)}}$$

is the direction of steepest decrease of the training risk at the training points.

The next step in gradient boosting is to fit a base learner to the pseudo-residuals by treating the pairs $(x_i, r_i^{(b)})$ as a regression problem with squared loss:

$$h_b \in \operatorname*{arg\,min}_{h\in\mathcal{H}} \sum_{i=1}^{|\mathcal{I}|}\left(r_i^{(b)} - h(x_i)\right)^2. \tag{5.2.4}$$

The result is a function that is a member of $\mathcal{H}$ closest to the negative gradient in the squared sense. Observe that the squared loss in (5.2.4) is a fitting device; it is unrelated to the loss ℓ of the prediction problem itself. Square loss is used because a regression tree of Chapter 4 fitted with this loss stores the mean of the fitted values in each leaf.

Each weak learner is fitted to $r_i^{(b)}$, not to y_i. If this ith isample has a large residual, the current ensemble predicts this sample badly; because of the squared loss, it will have a large influence on the correction. If $r_i^{(b)} \approx 0$, it is already predicted well, hence it will hardly influence the correction.

The third and last step in computing (5.2.2) is to find a good scale factor ρ_b.[3] For this, we can use a *line search*:

[3] The fitted h_b only approximates the direction; the magnitude of the correction needs to be set too.

$$\rho_b \in \operatorname*{arg\,min}_{\rho\in\mathbb{R}} \sum_{i=1}^{|\mathcal{I}|}\ell\left(y_i, f_{b-1}(x_i)+\rho\, h_b(x_i)\right). \tag{5.2.5}$$

We can now update (5.2.2) to get f_b.

The iteration has to start somewhere. The natural start is the best constant rule of Chapter 3:

$$f_0(x) = c^* \in \operatorname*{arg\,min}_{c\in\mathbb{R}} \sum_{i=1}^{|\mathcal{I}|}\ell(y_i,c). \tag{5.2.6}$$

Alg. 5.2.1 collects the steps.

Algorithm 5.2.1 Gradient boosting for a differentiable loss ℓ.

Input: a training set $\mathcal{T}$, a base class $\mathcal{H}$, the number B of rounds, a learning rate $\eta > 0$.
Output: a prediction function f_B.
procedure GRADIENTBOOST($\mathcal{T}, \mathcal{H}, B, \eta$)
  $f_0 \leftarrow \arg\min_{c \in \mathbb{R}} \sum_{(x,y)\in\mathcal{T}} \ell(y, c)$.
  for $b = 1, \dots, B$ **do**
    $r_i^{(b)} \leftarrow -\left.\dfrac{\partial \ell(y_i, a)}{\partial a}\right|_{a = f_{b-1}(x_i)}$ for all i ▷ *Negative gradient.*
    $h_b \leftarrow \arg\min_{h \in \mathcal{H}} \sum_i \left(r_i^{(b)} - h(x_i)\right)^2$ ▷ *Fit a base learner.*
    $\rho_b \leftarrow \arg\min_{\rho \in \mathbb{R}} \sum_i \ell\left(y_i, f_{b-1}(x_i) + \rho h_b(x_i)\right)$ ▷ *Line search.*
    $f_b \leftarrow f_{b-1} + \eta\, \rho_b\, h_b$
  return f_B

We next apply gradient boosting to a regression and a classification problem.

5.2.2 Regression boosting

Take the regression setting of Section 3.2.2, $\mathcal{Y} = \mathcal{A} = \mathbb{R}$, with the loss

$$\ell(y, a) = \tfrac{1}{2}(y - a)^2.$$

The initialization (5.2.6) is the constant that minimizes $\sum_i (y_i - c)^2$, which is the sample mean $\bar{y}$, as derived in Section 3.2.2. Next, $-\partial \ell(y, a)/\partial a = y - a$, so (5.2.3) gives

$$r_i^{(b)} = y_i - f_{b-1}(x_i).$$

The first residuals $r_i^{(1)} = y_i - \bar{y}$ are the part of each label that a single constant cannot explain. Fit a shallow regression tree h_1 to the pairs $(x_i, r_i^{(1)})$; by (5.2.4) this is an ordinary least-squares tree fit with the residuals as labels. Below we show that the line search gives $\rho_b = 1$. Thus, it suffices to add a fraction η of this tree to the constant, recompute the residuals $r_i^{(2)} = y_i - f_1(x_i)$, fit the next tree to those, and so on; we have all ingredients to operate Alg. 5.2.1.

To see that the line search gives $\rho_b = 1$, use $r_i^{(b)} = y_i - f_{b-1}(x_i)$ to write the objective of (5.2.5) as $\frac{1}{2}\sum_i (r_i^{(b)} - \rho h_b(x_i))^2$, set its derivative with respect to ρ to zero, and simplify:[4]

$$\rho_b = \frac{\sum_i r_i^{(b)} h_b(x_i)}{\sum_i h_b(x_i)^2}.$$

[4] Provided $\sum_i h_b(x_i)^2 > 0$; a tree with all leaf values zero adds nothing.

Write $\{\mathcal{T}_1, \dots, \mathcal{T}_m\}$ for the partition of the training samples formed by the leaves of h_b, as in (4.1.1), and c_t for the value stored in leaf t. A least-squares regression tree stores in each leaf the mean of the values it was fitted on, so $c_t = |\mathcal{T}_t|^{-1} \sum_{i \in \mathcal{T}_t} r_i^{(b)}$, hence $\sum_{i \in \mathcal{T}_t} r_i^{(b)} = |\mathcal{T}_t| c_t$. Summing over the leaves,

$$\sum_i r_i^{(b)} h_b(x_i) = \sum_{t=1}^{m} c_t \sum_{i \in \mathcal{T}_t} r_i^{(b)} = \sum_{t=1}^{m} |\mathcal{T}_t| c_t^2 = \sum_i h_b(x_i)^2.$$

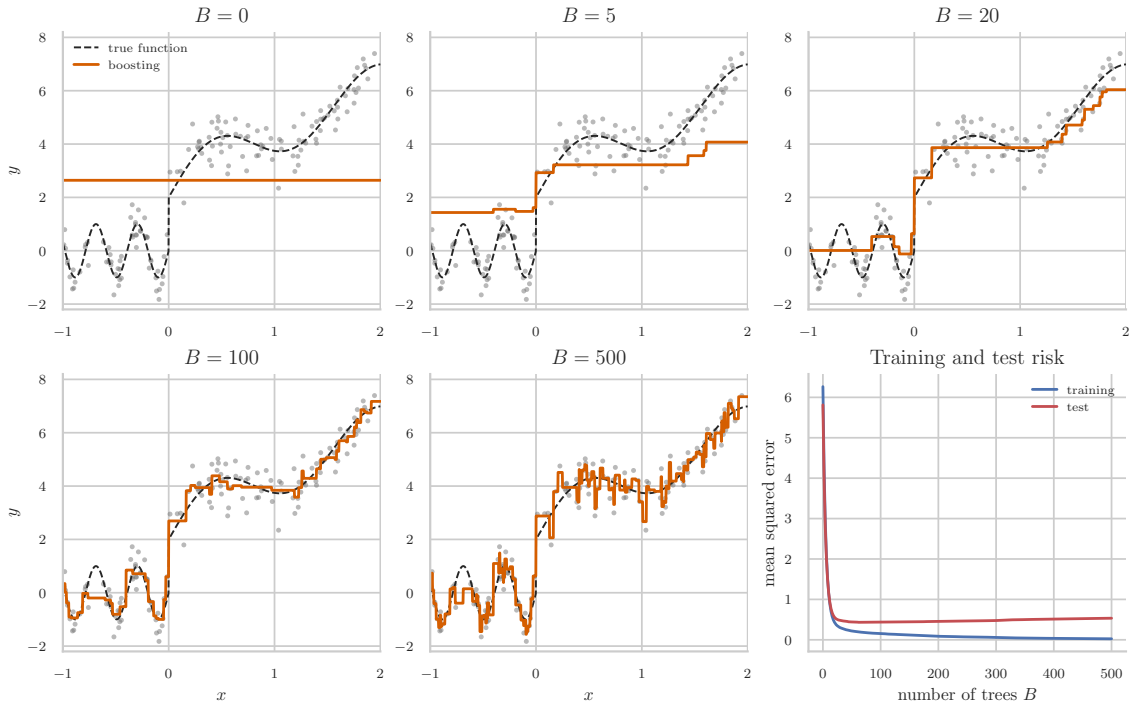


FIG. 5.3: Gradient boosting for regression on a nonlinear function, with depth-2 trees and $\eta = 0.1$. Top: fitted ensemble after $B = 0, 5, 20$ rounds. Bottom left, bottom middle: after $B = 100, 500$ rounds. Bottom right: training and test risk (mean squared error) as B grows; the test risk starts rising again once the ensemble overfits.

By using this in the expression for ρ_b, it follows that $\rho_b = 1$.

Interestingly, with a convenient choice for η, we can see that the sum over the squared residuals decrease for every step. Since $\rho_b = 1$, the update (5.2.2) replaces each $r_i^{(b)}$ by $r_i^{(b)} - \eta c_t$, which turns the sum of squares on leaf t into

$$\sum_{i \in \mathcal{T}_t} (r_i^{(b)} - \eta c_t)^2 = \sum_{i \in \mathcal{T}_t} (r_i^{(b)})^2 - 2\eta c_t \sum_{i \in \mathcal{T}_t} r_i^{(b)} + |\mathcal{T}_t| \eta^2 c_t^2 = \sum_{i \in \mathcal{T}_t} (r_i^{(b)})^2 - \eta(2-\eta) |\mathcal{T}_t| c_t^2.$$

Since $\eta(2-\eta) > 0$ for every[5] $\eta \in (0,2)$ the decrease is strict on a leaf whose mean residual is nonzero. Summing over all leaves in the partition shows that the sum over the squared residuals becomes smaller.

The training risk therefore keeps falling as trees are added, but the validation risk typically does not. At first the added learners pick up structure that the earlier ones missed and the validation risk falls with the training risk, but after enough rounds the added trees fit the training samples only, and the validation risk rises again. Fig. 5.3 provides an example.

[5] This covers the small η used in practice.

5.2.3 Classification boosting

We will show how the steps of gradient boosting can be applied to classification. This is known as AdaBoost.[6]

Take binary classification with[7]

$$\mathcal{Y} = \{-1, +1\}, \qquad \mathcal{A} = \mathbb{R}.$$

The prediction $f(x) \in \mathbb{R}$ is a *score*, and the decision map of (3.1.1) is

$$\hat{y} = d(f(x)) = \operatorname{sign}(f(x)).$$

[6] Wikipedia: AdaBoost. Wikipedia: XGBoost is an implementation of gradient tree boosting.

[7] The two labels are a relabeling of the $\{0,1\}$ convention used elsewhere in the book. The point of the relabeling is that below the sign of $yf(x)$ tells whether a prediction is correct, and $yh(x) \in \{-1,1\}$ for a base learner h.

The quantity

$$yf(x)$$

is the *margin* of the sample (x, y). It is positive when the sign of the score matches the label, so the classification is correct, and negative when it does not.

The base class is a set of stumps that map to $\{-1, 1\}$,

$$\mathcal{H} \subset \{h : \mathcal{X} \to \{-1, +1\}\}.$$

To compute the pseudo residuals, the derivative of the loss should be meaningful. Zero-one loss is $\mathbb{1}\{y \neq \operatorname{sign}(f(x))\} = \mathbb{1}\{yf(x) < 0\}$. As this is a step function of the margin, it has derivative zero wherever it is differentiable, so this is useless as a loss function. AdaBoost replaces this loss by the *exponential loss*

$$\ell(y, a) = e^{-ya}, \tag{5.2.7}$$

which is differentiable, decreasing in the margin, and satisfies $\mathbb{1}\{ya < 0\} \leq e^{-ya}$ for all ya. So the training risk under exponential loss is an upper bound for the training risk under zero-one loss, and pushing the first down pushes the second down with it.

Differentiating (5.2.7) gives $\partial \ell(y, a)/\partial a = -ye^{-ya}$, so (5.2.3) reads

$$r_i^{(b)} = y_i \tilde{w}_i^{(b)}, \qquad \tilde{w}_i^{(b)} := e^{-y_i f_{b-1}(x_i)}. \tag{5.2.8}$$

Thus the pseudo-residual of sample i factors into a label and a positive number $\tilde{w}_i^{(b)}$ that depends only on the margin. A sample with a large positive margin has $\tilde{w}_i^{(b)}$ close to 0; a misclassified sample has margin below 0 and hence $\tilde{w}_i^{(b)} > 1$. By normalizing

$$w_i^{(b)} = \frac{\tilde{w}_i^{(b)}}{\sum_{j=1}^{|\mathcal{T}|} \tilde{w}_j^{(b)}}, \qquad \sum_{i=1}^{|\mathcal{T}|} w_i^{(b)} = 1, \tag{5.2.9}$$

$w^{(b)}$ becomes a probability vector over the training samples.

This normalization is the *softmax* function $\tilde{\sigma}$, which maps a vector s of real numbers to a probability vector by

$$\tilde{\sigma}_k(s) = \frac{e^{s_k}}{\sum_j e^{s_j}}, \tag{5.2.10}$$

so that larger scores receive larger probabilities. Indeed, (5.2.9) is $w^{(b)} = \tilde{\sigma}(s^{(b)})$ with scores $s_i^{(b)} = -y_i f_{b-1}(x_i)$, the negated margins. Here the index runs over the $|\mathcal{T}|$ training samples, so the softmax produces a pmf over samples; in classification it runs over the classes instead.

The next step in gradient boosting is the square loss optimization over h_b. Substitute (5.2.8) into the fitting step (5.2.4) and expand the square:

$$\sum_{i=1}^{|\mathcal{T}|} \left(r_i^{(b)} - h(x_i)\right)^2 = \sum_{i=1}^{|\mathcal{T}|} \left(r_i^{(b)}\right)^2 - 2\sum_{i=1}^{|\mathcal{T}|} r_i^{(b)} h(x_i) + \sum_{i=1}^{|\mathcal{T}|} h(x_i)^2.$$

The first sum does not involve h. The third equals $|\mathcal{I}|$, because $h(x_i)^2 = 1$ for every $h \in \mathcal{H}$. Therefore,

$$\arg\min_{h\in\mathcal{H}} \sum_i \left(r_i^{(b)} - h(x_i)\right)^2 = \arg\max_{h\in\mathcal{H}} \sum_i \tilde{w}_i^{(b)} y_i h(x_i).$$

Rewriting this sum in terms of the misclassified samples prepares the line search over ρ_b. Since y_i and $h(x_i)$ both lie in $\{-1,+1\}$,

$$y_i h(x_i) = 1 - 2\cdot \mathbb{1}\{y_i \neq h(x_i)\},$$

so that, dividing by the positive constant $\sum_j \tilde{w}_j^{(b)}$,

$$\frac{\sum_i \tilde{w}_i^{(b)} y_i h(x_i)}{\sum_j \tilde{w}_j^{(b)}} = 1 - 2\,\mathrm{err}_b(h),$$

where the *weighted misclassification error* is given by

$$\mathrm{err}_b(h) = \sum_{i=1}^{|\mathcal{I}|} w_i^{(b)} \mathbb{1}\{y_i \neq h(x_i)\}. \tag{5.2.11}$$

With this, the fitting step (5.2.4) becomes

$$h_b \in \arg\min_{h\in\mathcal{H}} \mathrm{err}_b(h). \tag{5.2.12}$$

The line search has a simple solution.

Theorem 5.2.2. For classification boosting, the optimal scale factor is

$$\rho_b = \frac{1}{2}\log\frac{1-\mathrm{err}_b}{\mathrm{err}_b}, \qquad \mathrm{err}_b = \mathrm{err}_b(h_b). \tag{5.2.13}$$

—

Proof. For the line search (5.2.5)

$$\arg\min_{\rho\in\mathbb{R}} \sum_i \ell\left(y_i, f_{b-1}(x_i) + \rho h_b(x_i)\right)$$

split the sum by whether h_b classifies sample i correctly, using $y_i h_b(x_i) = \pm 1$:

$$\sum_{i=1}^{|\mathcal{I}|} e^{-y_i(f_{b-1}(x_i)+\rho h_b(x_i))} = \sum_{i=1}^{|\mathcal{I}|} \tilde{w}_i^{(b)} e^{-\rho y_i h_b(x_i)} = e^{-\rho}\sum_{i=1}^{|\mathcal{I}|} \tilde{w}_i^{(b)} + e^{\rho}\sum_{i=1}^{|\mathcal{I}|} \tilde{w}_i^{(b)} \mathbb{1}\{y_i \neq h_b(x_i)\}.$$

Dividing by $\sum_j \tilde{w}_j^{(b)}$ leaves the minimizer unchanged and replaces $\tilde{w}$ by w. With (5.2.11) the two sums become $1-\mathrm{err}_b$ and err_b, so we must minimize

$$G(\rho) = e^{-\rho}(1-\mathrm{err}_b) + e^{\rho}\,\mathrm{err}_b. \tag{5.2.14}$$

This is a function of one variable with $G''(\rho) = e^{-\rho}(1-\mathrm{err}_b) + e^{\rho}\,\mathrm{err}_b > 0$ whenever $0 < \mathrm{err}_b < 1$, hence strictly convex, so the stationary point is the minimum. From $G'(\rho) = -e^{-\rho}(1-\mathrm{err}_b) + e^{\rho}\,\mathrm{err}_b = 0$ and solving for ρ_b the result follows. □

[8] When $\mathrm{err}_b > 1/2$, use $-h_b$ as the improvement.

A classifier guesses better than random when $\mathrm{err}_b < 1/2$, in which case $\rho_b > 0$; one that guesses, $\mathrm{err}_b = 1/2$, gets $\rho_b = 0$ and is not added at all.[8] The requirement $\mathrm{err}_b < 1/2$ is known as the *weak-learning condition*.

Observe that in the above we used the weights $w^{(b)}$, which in turn depend on f_{b-1} through (5.2.9). Once we have the update h_b, we need to update these weights for the next iteration. Using the update (5.2.2),

$$\tilde{w}_i^{(b+1)} = e^{-y_i f_b(x_i)} = e^{-y_i f_{b-1}(x_i)} e^{-\eta \rho_b y_i h_b(x_i)} = \tilde{w}_i^{(b)} e^{-\eta \rho_b y_i h_b(x_i)},$$

and after normalizing,

$$w_i^{(b+1)} = \frac{w_i^{(b)} e^{-\eta \rho_b y_i h_b(x_i)}}{\sum_{j=1}^{|\mathcal{T}|} w_j^{(b)} e^{-\eta \rho_b y_j h_b(x_j)}}. \tag{5.2.15}$$

Since $y_i h_b(x_i) = \pm 1$, the numerator multiplies $w_i^{(b)}$ by $e^{-\eta\rho_b} < 1$ when sample i is classified correctly and by $e^{\eta\rho_b} > 1$ when it is not. Samples that the new classifier gets wrong gain weight, and by (5.2.12) the next classifier is chosen to do better on exactly those samples.

For the initialization (5.2.6), let n_+ and n_- count the samples with $y_i = +1$ and $y_i = -1$. Then $\sum_i e^{-y_i c} = n_+ e^{-c} + n_- e^{c}$, and setting its derivative to zero gives $f_0(x) = \frac{1}{2}\log(n_+/n_-)$. AdaBoost as usually stated takes $f_0 = 0$ instead, which by (5.2.9) starts the weights uniform at $w_i^{(1)} = 1/|\mathcal{T}|$, and takes $\eta = 1$, so that each classifier enters with its full coefficient ρ_b.

Fig. 5.4 shows an example of AdaBoost.

We remark in passing that it can be shown that with $\eta = 1$, if $\mathrm{err}_b \le 1/2 - \gamma$ for some fixed $\gamma > 0$ and all b, then the training risk of $\mathrm{sign}(f_B)$ under zero-one loss is at most $e^{-2\gamma^2 B}$. Thus, the bound falls geometrically in B. Consequently, classifiers that are each only slightly better than guessing can drive the training error to zero. Thus, the training risk cannot be used to choose B; use the validation risk as in Section 3.6.

5.3 Exercises

Exercise (C) 5.3.1. Why does the gradient descent step move opposite to the gradient rather than in some other direction. What goes wrong if the learning rate η is chosen too large or too small?

Exercise (C) 5.3.2. Suppose one feature is measured in meters and another in kilometers. Why does this slow down gradient descent? What is the fix?

Exercise (T) 5.3.3. Fit the line $\hat{y} = \beta_0 + \beta_1 x$ to the training set $\mathcal{T} = [(1,1),(2,3)]$, in which each pair is one sample (x, y), so there is a single feature and $\theta = (\beta_0, \beta_1)$. Start at $\theta^{(0)} = (0,0)$, take $\eta = 0.1$, and compute $\theta^{(1)}$ with gradient descent under square loss.

Exercise (T) 5.3.4. Apply one step of gradient descent to the loss $\ell(y,a) = (y-a)^4$, optimizing over the quadratic model $\hat{y} = \beta_0 + \beta_1 x + \beta_2 x^2$. Use the single-sample training set $\mathcal{T} = \{(1,2)\}$, starting point $\theta^{(0)} = (0,0,0)$, and $\eta = 0.01$.

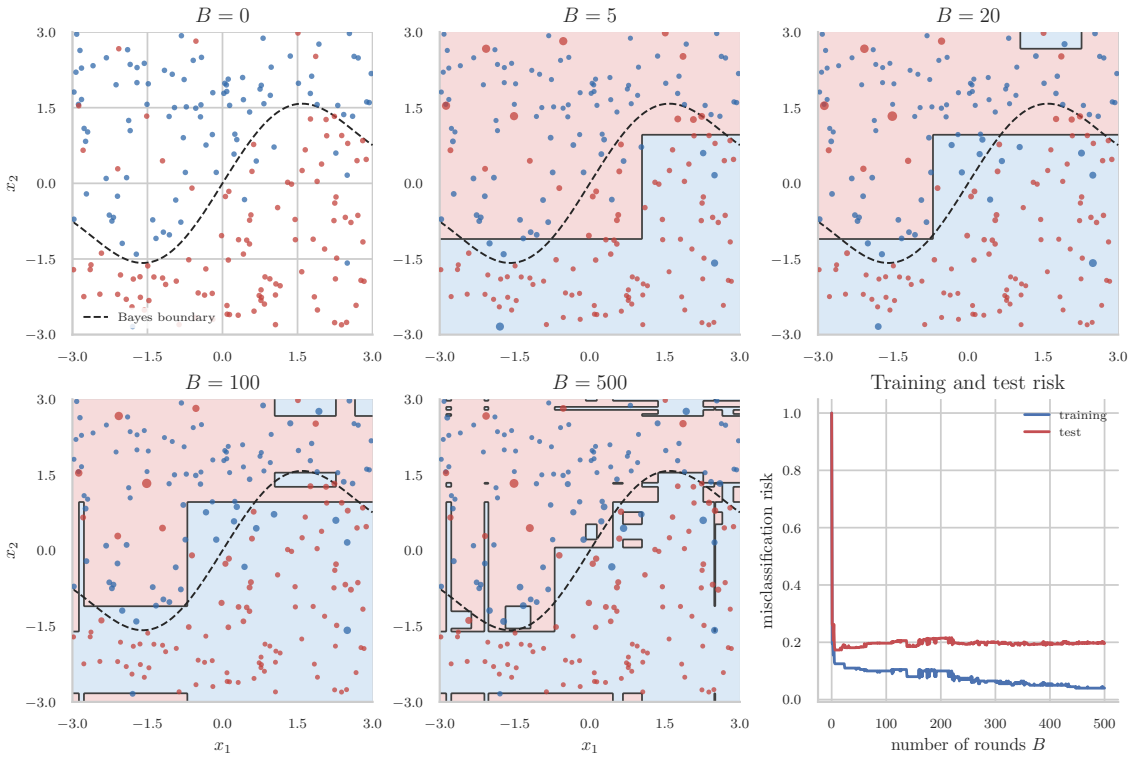


FIG. 5.4: AdaBoost on a 2D classification problem with a wavy Bayes boundary (dashed) and label noise. Point size is proportional to the sample's current AdaBoost weight; the shaded regions are the ensemble's decision regions. Top: after $B = 0, 5, 20$ rounds. Bottom left, bottom middle: after $B = 100, 500$ rounds. Bottom right: training and test misclassification risk; the training risk keeps falling while the test risk plateaus.

Exercise (C) 5.3.5. Take the sigmoid model $\hat{y} = \sigma(ax + b)$ under the squared loss, and suppose the true label is $y = 0$. Compare a hesitant prediction $\hat{y} = 0.5$ with a confidently wrong prediction $\hat{y} = 0.999$. Compute the squared-loss gradient factor in both cases and explain what this shows about saturation.

Exercise (C) 5.3.6. Moving from squared loss to the binary cross-entropy loss for the same sigmoid model, the factor $\hat{y}(1 - \hat{y})$ disappears from the gradient. What effect does this have on how a confidently wrong sample is treated?

Exercise (T) 5.3.7. The while loop of Alg. 5.1.2 fits the sigmoid model $\hat{y}(x) = \sigma(ax + b)$ under the binary cross-entropy loss (5.1.5), and stops as soon as $\left\|\nabla \hat{R}_{\mathcal{T}}(a, b)\right\|^2 \leq \epsilon$. Write $\left\|\nabla \hat{R}_{\mathcal{T}}(a, b)\right\|^2$ as an explicit expression in the samples of $\mathcal{T}$ and $\hat{y}(x)$.

Exercise (C) 5.3.8. What is stochastic gradient descent, and in what sense does it still move in the right direction on average?

Exercise (T) 5.3.9. Show that the mini-batch gradient (5.1.6) is an unbiased estimator of the full gradient $\nabla \hat{R}_{\mathcal{T}}(\theta)$.

Exercise (C) 5.3.10. How does mini-batching work, and why is it used rather than plain stochastic or full-batch gradient descent? If $|\mathcal{T}| = 1000$ and the batch size is 64, how many parameter updates make up one epoch, and why are the batches reshuffled every epoch?

Exercise (C) 5.3.11. What makes the forward-stagewise fitting of gradient boosting greedy?

Exercise (C) 5.3.12. What are the three steps of gradient boosting? Include the formulas.

Exercise (C) 5.3.13. In gradient boosting, what are pseudo-residuals, and what are they used for?

Exercise (T) 5.3.14. Run the first round of gradient boosting by hand on the regression training set $\mathcal{T} = [(0,1),(1,3),(2,2)]$, in which each pair is one sample (x, y), with the squared loss $\ell(y,a) = \frac{1}{2}(y-a)^2$. Compute the starting rule f_0 of (5.2.6), and then the pseudo-residuals $r_i^{(1)}$ of (5.2.3) to which the first base learner h_1 is fitted.

Exercise (T) 5.3.15. For regression boosting with squared loss and a least-squares regression tree as base learner, why does the line search (5.2.5) always give $\rho_b = 1$?

Exercise (C) 5.3.16. The training risk of a boosted regression model keeps falling as B grows. Why can a good B not be chosen from the training risk alone?

Exercise (C) 5.3.17. What is the margin $yf(x)$ of a sample (x, y) in AdaBoost, and what does its sign tell you?

Exercise (C) 5.3.18. Applying gradient boosting to classification with $y \in \{-1,+1\}$, one can use the exponential loss $\ell(y, f(x)) = e^{-yf(x)}$. Why this loss, rather than zero-one loss directly?

Exercise (C) 5.3.19. Suppose that $\mathrm{err}_b > 1/2$ for a base learner in AdaBoost. Is this a problem? What happens if this error is zero?

Chapter 6

Probability Calibration; TBD

This chapter serves the step that follows the fitting. The question is settled, the data are assembled, the features are built, and the random forest of Chapter 4 returns a score $u(e)$ for every edge e; what remains is to turn that score into a probability, because the router of Chapter B uses the probability that an edge is bad as the cost of that edge. Finding a function g that maps scores to probabilities is called *probability calibration*.

The introduction explains why the score of a forest is not a probability, and which data are used to repair this. The next section fixes the notation. Three sections then treat three methods, which differ in the shapes they allow for g and in the loss they minimize: *sigmoid scaling* takes a two-parameter S-curve and the cross-entropy loss, *histogram binning* takes a step function on a fixed set of bins and the squared loss, and *isotonic regression* takes any non-decreasing function and the same squared loss, minimized by the *Pool Adjacent Violators Algorithm* (PAVA). A further section proves that PAVA returns the isotonic fit. The discussion compares the three methods and states their limits.

6.1 Introduction

To classify a new edge, every one of the trees casts a vote, good or bad, and the forest reports the fraction of trees that voted bad; call this raw score $u(e)$. This fraction lies in $[0,1]$ by construction, which makes it tempting to interpret it as a probability $\mathsf{P}\{\text{bad} \mid e\}$.

The classifier is trained on the *training set* $\mathcal{T}$:[1] the edges labeled 0 for lying on a good path, and 1 for being bad. In the training process, we must be careful about *class imbalance*. In the full map the great majority of edges count as bad under our rules, whereas the curated good edges are comparatively rare, on the order of a few percent. A classifier trained on a set with these proportions could reach a high accuracy by the trivial strategy of calling almost every edge bad. To prevent this, we do not train on the full map but on a sample with a much milder imbalance: three bad edges for every good one.

[1] The features enter the trees after light preprocessing: categorical features (highway type, surface, ...) are turned into integers by an *ordinal encoder*, with missing values represented by an explicit missing category; numeric features are passed unchanged.

THE RAW VOTE FRACTION, as obtained from the classifier, is a sensible score to rank edges, but this fraction is not necessarily a trustworthy probability (which we need as cost when routing). One reason is the class imbalance of the training sample: the raw scores reflect the class proportions of that sample rather than the true prevalence of bad edges,

hence they cannot be read directly as probabilities. A second reason is that the raw score is a fraction of trees, not a probability: each tree is grown on a limited bootstrap sample and casts a hard vote, so even for an edge that is almost surely bad, some trees will vote good. Vote fractions therefore rarely reach the extremes 0 and 1, and the forest tends to be under-confident: its scores are pulled toward the middle.

We therefore need to *calibrate* the score of the classifier. Calibration uses a held-out portion of the data that the classifier never saw during training. For the routing app, we use *isotonic regression*, which finds the *non-decreasing* function g that best matches the raw scores to the observed bad-fractions on the held-out set. Forcing the function to be non-decreasing keeps the ranking of the edges intact,[2] while still stretching or compressing the values, so that edges scored 0.7 really are undesirable about 70% of the time.

[2] A higher raw score never maps to a lower probability.

SCORING THE WHOLE MAP. It remains to compute the cost for all edges, not just the ones in the training set $\mathcal{T}$. This is now easy. For each unseen edge e in the entire graph, none of them labeled, the classifier computes a score $u(e)$; the calibrated value $g(u(e))$ is then the edge cost. We store these edge costs in a database.

Once the edge weights are obtained, OSRM uses these to compute cheapest paths, and the webapp shows the result on a map.

6.2 Setup and notation

We now fix the notation for the rest of the chapter. The scores $u(e)$ that the random forest of Chapter 4 produces are reliable as a *ranking*,[1] but not as well-scaled probabilities. For example, among edges with a score near 0.9, perhaps only 70% turn out to be actually bad.

[1] An edge with a higher score is more likely to be bad.

The concept of *probability calibration*[2] is what we need: it gives a *calibration function* $g : [0,1] \to [0,1]$ that maps a raw random-forest score $u(e)$ to a probability that matches the fraction of bad edges observed on a *calibration set* $\mathcal{H}$. As in Chapter 3, this is a list of samples, $\mathcal{H} = [(e_1, y_1), \ldots, (e_n, y_n)]$ with $n = |\mathcal{H}|$, where e_i is a labeled edge, $u_i = u(e_i)$ is its raw score, and $y_i \in \{0,1\}$ is its observed label. Everything below depends on a sample only through its score, and two of the three methods sort the samples by score, so we refer to a sample by its position i throughout this chapter. Here $y_i = 0$ means that edge e_i is a good edge and $y_i = 1$ means that it is a bad edge. Thus $g(u(e))$ is a calibrated bad-edge probability. The list $\mathcal{H}$ is held out from the samples used to train the random forest, in the sense of Chapter 3. Once the forest has been fitted, we use $\mathcal{H}$ only to fit the calibration function g.

[2] For calibration in supervised learning, Niculescu-Mizil and Caruana, *Obtaining Calibrated Probabilities from Boosting* (2012). For modern neural networks, Guo, Pleiss, Sun, and Weinberger, *On Calibration of Modern Neural Networks* (2017).

Below we discuss three different calibration methods. We start with *sigmoid scaling*, as it is a parametric method and prepares for the discussion of neural networks in Chapter 7. We then turn to two nonparametric methods, *histogram binning* and *isotonic regression*. The last method leads to the *Pool Adjacent Violators Algorithm* (PAVA), which solves the isotonic regression problem.

6.3 Sigmoid scaling

Sigmoid scaling, also called *Platt scaling*,[1] transforms a score u to a probability by means of the two-parameter family of calibration functions $g_{a,b}$:

$$g_{a,b}(u) = \sigma(au + b), \qquad \sigma(z) = \left(1 + e^{-z}\right)^{-1},$$

where σ is the sigmoid function.

When $a \geq 0$, sigmoid scaling preserves the ordering of the raw scores. If $a < 0$, it reverses that ordering, so unconstrained sigmoid scaling need not preserve the random forest's ranking. Once a and b have been fitted, calibrating a new edge is immediate: we compute its random-forest score $u(e)$ and return $g(u(e)) = \sigma(au(e) + b)$.

This form of g is useful because it is parametric and simple; only a and b have to be estimated, so the method can work with relatively little calibration data. It is also an interesting model in itself: it is the simplest possible neural network, consisting of a *single neuron* with a score u as *input*, a *weight* a and a *bias* b, a sigmoid *activation*, and a cross-entropy *loss*.

Finding good values for a and b is a fitting problem of exactly the kind treated in Chapter 5: the labels are binary, and the model is the sigmoid (5.1.3), with the raw score u in the role of the feature and the calibration set $\mathcal{H}$ in the role of the training set. Following that section, we assume that, conditional on the scores u_i, the labels Y_i are independent[2] Bernoulli random variables with success probabilities for $i = 1, \ldots, n$,

$$\hat{p}_i(a,b) = g_{a,b}(u_i) = \sigma(au_i + b).$$

Maximum likelihood then leads to the binary cross-entropy loss (5.1.5), so that the empirical risk to minimize over the calibration set is

$$\hat{R}_{\mathcal{H}}(a,b) = \frac{1}{|\mathcal{H}|}\sum_{i=1}^{n} \ell_i(a,b), \qquad \ell_i(a,b) = -y_i \log \hat{p}_i - (1-y_i)\log(1-\hat{p}_i).$$

With this, we seek to solve

$$\min_{a,b} \hat{R}_{\mathcal{H}}(a,b).$$

There are no direct, algebraic methods to solve this problem, so we minimize $\hat{R}_{\mathcal{H}}$ by gradient descent, with parameter vector $\theta = (a,b)$. The gradient is the one derived in Chapter 5, with the score u_i in place of the feature and $\hat{p}_i$ in place of the prediction:

$$\frac{\partial}{\partial a}\hat{R}_{\mathcal{H}}(a,b) = \frac{1}{|\mathcal{H}|}\sum_{i=1}^{n}(\hat{p}_i - y_i)u_i, \qquad \frac{\partial}{\partial b}\hat{R}_{\mathcal{H}}(a,b) = \frac{1}{|\mathcal{H}|}\sum_{i=1}^{n}(\hat{p}_i - y_i).$$

Running Alg. 5.1.2 on $\mathcal{H}$ rather than on $\mathcal{T}$ returns the fitted a and b.

6.4 Histogram binning

For histogram binning and isotonic regression, it is natural to measure calibration quality by squared error. This leads to least-squares problems.

[1] See Wikipedia, *Platt scaling*.

[2] The Y_i cannot be truly independent, because when two edges are geographically close, they are probably both good or bad. However, as the edges indexed by $\mathcal{H}$ are selected randomly all over the Netherlands, most of them will not lie close to each other. Hence, the independence assumption is not too unreasonable.

THE BRIER SCORE is the empirical risk under the Brier loss of Chapter 3 on the calibration set $\mathcal{H}$:

$$\frac{1}{|\mathcal{H}|}\sum_{i=1}^{n}(u_i - y_i)^2.$$

A lower Brier score means a better fit.[1]

Recall that a calibration function $g : [0,1] \to [0,1]$ should calibrate a prediction u_i to $g(u_i)$ such that $g(u_i)$ better matches the observed label y_i. Thus, a good calibration function is such that

$$\frac{1}{|\mathcal{H}|}\sum_{i=1}^{n}(g(u_i) - y_i)^2 < \frac{1}{|\mathcal{H}|}\sum_{i=1}^{n}(u_i - y_i)^2.$$

Once we have fitted g to the observations in $\mathcal{H}$, we should extend g to the entire interval $[0,1]$. Below we discuss how to do this.

A LIKELIHOOD VIEW also explains why the Brier score is natural.[2] Suppose, as an approximation, that the labels are independent normal random variables with means $g(u_i)$ and common variance v:

$$Y_i \sim \text{Norm}\big(g(u_i), v\big).$$

Then the likelihood of the observed labels is

$$L(g, v) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi v}} \exp\left(-\frac{(y_i - g(u_i))^2}{2v}\right).$$

Taking the negative logarithm gives

$$-\log L(g, v) = \frac{|\mathcal{H}|}{2}\log(2\pi v) + \frac{1}{2v}\sum_{i=1}^{n}(y_i - g(u_i))^2.$$

For fixed v, maximizing the likelihood is therefore the same as minimizing the squared error. Hence minimizing the Brier score of the calibrated predictions $g(u_i)$ can be read as maximum likelihood under a homoscedastic normal approximation.

This should be compared with sigmoid scaling above. If $Y_i \sim \text{Bern}\big(\hat{p}_i\big)$, then maximum likelihood gives the cross-entropy loss, as in (5.1.5). If, as an approximation, $Y_i \sim \text{Norm}\big(\hat{p}_i, v\big)$ with common variance v, then maximum likelihood gives squared error, hence the Brier score. In sigmoid scaling, $\hat{p}_i = g_{a,b}(u_i) = \sigma(au_i + b)$ for the two-parameter family $g_{a,b}$. In histogram binning and isotonic regression, $\hat{p}_i = g(u_i)$, where g is the calibration function that we still have to find. Histogram binning minimizes squared error after the bins are fixed. Isotonic regression minimizes squared error under the condition that g must be nondecreasing.

HISTOGRAM BINNING CONSTRUCTS a calibration function for a given set of bins. Given bin boundaries $0 = c_0 < c_1 < \cdots < c_M = 1$, let

$$B_m = [c_{m-1}, c_m), \text{ for } m = 1, \ldots, M-1, \quad \text{and } B_M = [c_{M-1}, c_M]. \tag{6.4.1}$$

The number of calibration samples in bin m[3] is given by

$$n_m = \sum_{i=1}^{n} \mathbb{1}\{u_i \in B_m\}.$$

[1] Up to a factor 2: with the class order $(0,1) = (\text{good}, \text{bad})$, the probability prediction is the vector $(1 - u_i, u_i)$ and the one-hot label vector is $q_i = (1 - y_i, y_i)$. The Brier loss is therefore $\|(1-u_i, u_i) - q_i\|^2 = 2(u_i - y_i)^2$, which does not change what minimizes the score.

[2] This example is adapted from Geyer, *Isotonic Regression*, which gives a clear overview of isotonic regression from a statistical point of view.

[3] We assume that the bin boundaries are chosen such that each bin contains at least one observation.

The average raw score in bin m is

$$\bar{u}_m = \frac{1}{n_m} \sum_{i=1}^{n} u_i \, \mathbb{1}\{u_i \in B_m\}, \tag{6.4.2}$$

while the bad-edge fraction observed in the calibration set is

$$\hat{p}_m = \frac{1}{n_m} \sum_{i=1}^{n} y_i \, \mathbb{1}\{u_i \in B_m\}. \tag{6.4.3}$$

In what sense are the $\hat{p}_m$ good estimates? To see this, consider calibration functions that are constant on each bin: $g(u) = \gamma_m$ for $u \in B_m$, with $\gamma_1, \dots, \gamma_M$ still to be chosen. Since every sample lies in exactly one bin, the Brier score of such a g splits into a sum over bins:

$$\frac{1}{|\mathcal{H}|} \sum_{i=1}^{n} (g(u_i) - y_i)^2 = \frac{1}{|\mathcal{H}|} \sum_{m=1}^{M} \sum_{\substack{1 \le i \le n \\ u_i \in B_m}} (\gamma_m - y_i)^2.$$

The inner sums have no variable in common, so they can be minimized independently, one per bin. The mth inner sum is a least-squares problem in γ_m whose minimizer is the average of the y_i in bin m,[4] that is, $\hat{p}_m$ of (6.4.3). Thus $\hat{p} = (\hat{p}_1, \dots, \hat{p}_M)$ minimizes the Brier score over all calibration functions that are constant on the given bins.

With the estimates $\hat{p}_1, \dots, \hat{p}_M$, we build a calibration function g as follows. For any edge e, the random forest computes a score $u(e)$. Then g assigns the observed bad-edge fraction $\hat{p}_m$ of the bin that contains $u(e)$:

$$g(u(e)) = \sum_{m=1}^{M} \hat{p}_m \, \mathbb{1}\{u(e) \in B_m\}. \tag{6.4.4}$$

Histogram binning fits a piecewise-constant calibration function, but requires the user to specify the partition of $[0, 1]$. Moreover, it does not guarantee that $\hat{p}_m \le \hat{p}_{m+1}$, which is undesirable because a calibration function should preserve the ordering. Isotonic regression, discussed below, enforces this ordering by construction.

Histogram binning also gives a simple diagnostic plot. For each bin, plot $(\bar{u}_m, \hat{p}_m)$, with $\bar{u}_m$ and $\hat{p}_m$ as computed in (6.4.2) and (6.4.3). The resulting plot is called a *calibration curve*. If these points lie close to the diagonal $y = x$, then the scores are well calibrated; otherwise they are not.

[4] Differentiate $\sum_{\substack{1 \le i \le n \\ u_i \in B_m}} (\gamma_m - y_i)^2$ with respect to γ_m and set the result to zero.

6.5 Isotonic regression

Isotonic regression finds the optimal calibration function directly from the data.[1] For this, it only uses a monotonicity constraint suggested by the random-forest score: if $u(e_i) < u(e_j)$, then the calibrated probability for e_i should not exceed the calibrated probability for e_j.[2] We note that while here we use isotonic regression for calibration purposes, it is useful in a number of statistical problems, such as maximum likelihood estimation under order constraints.[3] The remainder of this section states the problem formally and describes two methods to solve it; the next section proves that PAVA finds the unique solution.

[1] This flexibility is an advantage, but in general good calibration requires many samples, typically at least a few hundred.

[2] In other words, calibration may stretch or compress the scores, but it should not reverse their order.

[3] Wikipedia: Isotonic regression. A book treatment is Barlow, Bartholomew, Bremner, and Brunk, *Statistical Inference under Order Restrictions* (1972).

For isotonic regression we sort the calibration observations by raw score and relabel them as $1,\dots,n$, so that $u_1 \le \cdots \le u_n$. From this point until the end of the isotonic-regression problem statement, the index i refers to this sorted order.

The *isotonic regression problem* is stated as follows. We have a number of points $(u_1, y_1),\dots,(u_n, y_n)$ where the predictors $u_1 \le \cdots \le u_n$ are ordered and the responses $y_i \in \mathbb{R}$.[4, 5] The problem is to find a set of non-decreasing numbers $\hat{y}_i$ that are optimal in the least squares sense,[6] i.e.,

$$\min\left\{\frac{1}{2}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 : \hat{y}_1 \le \cdots \le \hat{y}_n\right\}.$$

By taking the $(n-1)\times n$ matrix A as

$$A = \begin{pmatrix} 1 & -1 & & & \\ & 1 & -1 & & \\ & & \ddots & \ddots & \\ & & & 1 & -1 \end{pmatrix} \tag{6.5.1}$$

to enforce the ordering constraint on $\hat{y}$,[7] the problem can be succinctly written as a convex quadratic optimization problem with linear constraints:

$$\min_{\hat{y}:A\hat{y}\le 0} \frac{1}{2}\left\|y-\hat{y}\right\|^2, \tag{6.5.2}$$

where

$$\left\|\hat{y}-y\right\|^2 = \langle \hat{y}-y, \hat{y}-y\rangle = \sum_{i=1}^{n}(\hat{y}_i - y_i)^2.$$

THE GREATEST CONVEX MINORANT method offers one way to find the isotonic fit. Form the cumulative sums

$$(0,0),\ (1,S_1),\ \dots,\ (n,S_n), \qquad S_i = \sum_{j=1}^{i} y_j.$$

The *greatest convex minorant* of these points is the largest convex function on $[0,n]$ that stays below all of them; think of a string pulled taut beneath the points, see Fig. 6.1 for an example. The isotonic fitted values $\hat{y}_i$ are then the slopes of the greatest convex minorant of these cumulative points. More precisely, $\hat{y}_i$ is the slope of the minorant on the segment from $i-1$ to i. We state this fact without proof; below we prove instead that PAVA solves the isotonic regression problem.

PAVA is a second method. Here we sketch its working; below we will prove that it solves the optimization problem (6.5.2).[8]

Fig. 6.2 illustrates how PAVA sweeps from left to right; Alg. 6.5.1 provides the pseudocode. It starts with one block per sorted observation, with block value y_i. Whenever two neighboring blocks $\mathcal{L}$ and $\mathcal{R}$ violate monotonicity in that the mean label value $\mu_{\mathcal{L}}$ on $\mathcal{L}$ is larger than $\mu_{\mathcal{R}}$, it merges $\mathcal{L}$ and $\mathcal{R}$ into a single block and replaces their values by the average of the labels y in the merged block.[9] It continues merging until all

[4] In the calibration application $u_i \in [0,1]$ is the sorted raw score and $y_i \in \{0,1\}$ is the observed label; the problem statement itself allows any real responses.

[5] Only the *ordering* of the predictors enters the problem; their values play no further role. Ties $u_i = u_{i+1}$ are broken arbitrarily; a common alternative is to pool tied observations into one point with their average response before running the regression.

[6] Up to a scale factor, this is the Brier score (3.3.5) of the calibrated values, and rescaling a loss by a positive constant does not move its minimizer.

[7] Its ith row expands to $\hat{y}_i - \hat{y}_{i+1} \le 0$.

[8] PAVA is supported by `scikit-learn` and `R`.

[9] This pooled average is the *barycentric mean* of the two old block means, with weights given by the block sizes. In finding the greatest convex minorant this is the replacement of two adjacent violating slopes by the slope of the chord over their union.

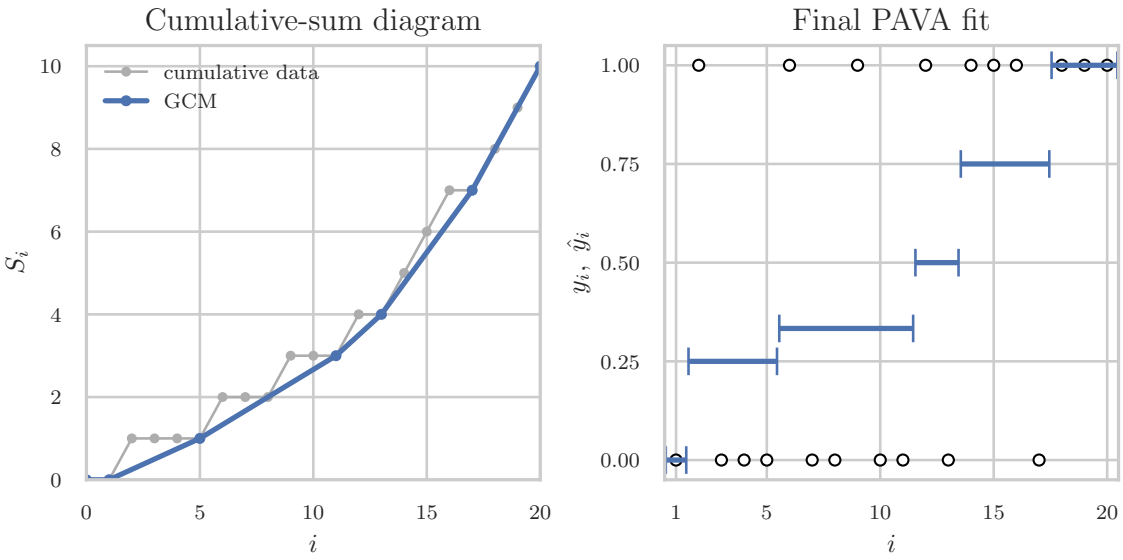


FIG. 6.1: Left: an example of a cumulative-sum diagram and its greatest convex minorant. Right: the final PAVA block averages for the same observations.

Algorithm 6.5.1 Pool Adjacent Violators Algorithm

Start with singleton blocks $\{1\},\dots,\{n\}$ and set $\hat{y}_i \leftarrow y_i$ for all i.
Assign each block B its average $\mu_B = |B|^{-1}\sum_{i\in B} y_i$.
while there are adjacent blocks $\mathcal{L},\mathcal{R}$, with $\mathcal{L}$ directly left of $\mathcal{R}$, such that $\mu_\mathcal{L} > \mu_\mathcal{R}$ **do**
  Merge the blocks $\mathcal{L},\mathcal{R}$ into one block $\mathcal{L}\cup\mathcal{R}$.
  $\mu_{\mathcal{L}\cup\mathcal{R}} \leftarrow \frac{|\mathcal{L}|\mu_\mathcal{L}+|\mathcal{R}|\mu_\mathcal{R}}{|\mathcal{L}|+|\mathcal{R}|}$. ▷ *Barycentric mean*
  $\hat{y}_i \leftarrow \mu_{\mathcal{L}\cup\mathcal{R}}$ for every $i \in \mathcal{L}\cup\mathcal{R}$.
return $\hat{y}$

block averages are nondecreasing. A *careful* left-to-right implementation of PAVA runs in $O(n)$ time once the scores are sorted; the sorting step costs $O(n\log n)$, so the full calibration procedure is typically $O(n\log n)$.

Once PAVA has produced the fitted values $\hat{y}_i$, consecutive observations with the same fitted value form blocks. These blocks can be used as adaptive bins. Suppose the final blocks are $G_m = \{s_m,\dots,t_m\}$, $m = 1,\dots,M$, in increasing order of the scores, where s_m and t_m are the first and the last index of block m. We choose bin boundaries[10] by setting $c_0 = 0$, $c_M = 1$, and, for $m = 1,\dots,M-1$,

$$c_m = \frac{u_{t_m} + u_{s_{m+1}}}{2}.$$

With these boundaries, the bins are as in (6.4.1). Moreover, the fitted value on block G_m is its average label value, just as in (6.4.3). Hence the calibration function g is defined exactly as in (6.4.4).

[10] The PAVA fit is defined only at the observed scores in $\mathcal{H}$; these boundaries extend it to a function on all of $[0,1]$.

6.6 Cones, projections, and Moreau's decomposition

Before proving that PAVA produces the isotonic fit we need some concepts from convex analysis.[1]

[1] The geometric background used here is Salmon, *Isotonic Regression*.

WE NEED SOME DEFINITIONS of convexity theory first; Fig. 6.3–Fig. 6.5 illustrate these definitions. A *ray* is a half line emanating from the origin; in algebraic terms, if $x \in \mathbb{R}^n$ lies in a ray, then all points αx lie in the ray for $\alpha \geq 0$.

A *cone* is a set of rays, so a cone is closed under non-negative scaling. A *convex* set is such that when x and y lie in the set, the line segment $\alpha x + (1-\alpha)y$, $\alpha \in [0,1]$, between x and y lies in the set. A set is closed if its boundary belongs to it. In the sequel we will be concerned with a convex, closed and nonempty cone C.

The last concept we need is the *polar* cone C° of C defined as

$$C^\circ = \left\{ z \in \mathbb{R}^n : \langle z, x \rangle \leq 0 \text{ for all } x \in C \right\}.$$

The polar cone C° is also a closed convex cone. For a nonempty closed convex cone, $C^{\circ\circ} = C$.[2] We use this sign convention throughout; some texts reserve the term *dual cone* for the opposite inequality.

The next two characterizations of C°, in which $A^\top$ denotes the transpose of the matrix A of (6.5.1), are essential to understand PAVA.

Lemma 6.6.1. Let $C = \{x \in \mathbb{R}^n : Ax \leq 0\}$ be the non-empty, closed, convex cone defined by the matrix A. Then its polar cone admits two further characterizations:

$$C_1^\circ = \left\{ z \in \mathbb{R}^n : \lambda_i := \sum_{j=1}^{i} z_j \geq 0 \text{ for } i = 1, \ldots, n-1, \text{ and } \lambda_n = 0 \right\},$$
$$C_2^\circ = \left\{ A^\top \lambda : \lambda \in \mathbb{R}_+^{n-1} \right\}.$$

That is, $C^\circ = C_1^\circ = C_2^\circ$.

—

Proof. To establish the equalities, we prove the cycle of inclusions $C_1^\circ \subset C_2^\circ \subset C^\circ \subset C_1^\circ$.

First we show that $C_1^\circ \subset C_2^\circ$. Take $z \in C_1^\circ$ and let $\lambda_i = \sum_{j=1}^{i} z_j$ as in the definition of C_1°. With the convention $\lambda_0 = 0$, and using that $\lambda_n = 0$ by the definition of C_1°, we get $z_i = \lambda_i - \lambda_{i-1}$ for $i = 1, \ldots, n$, that is, $z = A^\top \lambda$. As by assumption $\lambda_i \geq 0$ for $i = 1, \ldots, n-1$, this shows that $z \in C_2^\circ$.

Next we prove that $C_2^\circ \subset C^\circ$. If $z \in C_2^\circ$ then $z = A^\top \lambda$ with $\lambda \geq 0$. Consequently, $\langle z, x \rangle = \langle A^\top \lambda, x \rangle = \langle \lambda, Ax \rangle \leq 0$ because $Ax \leq 0$ and $\lambda \geq 0$, hence $z \in C^\circ$.

Finally, we show that $C^\circ \subset C_1^\circ$. For this we need to rewrite $\langle z, x \rangle$ for $x \in C$ in a useful way.[3] Let $\lambda_i := \sum_{j=1}^{i} z_j$, $i = 1, \ldots, n$, so that $z_i = \lambda_i - \lambda_{i-1}$ if we define $\lambda_0 = 0$. Therefore, $\langle z, x \rangle = \sum_{i=1}^{n} x_i z_i = \sum_{i=1}^{n} x_i (\lambda_i - \lambda_{i-1})$. Using that $\lambda_0 = 0$ and relabeling,

$$\sum_{i=1}^{n} x_i \lambda_{i-1} = \sum_{i=2}^{n} x_i \lambda_{i-1} = \sum_{i=1}^{n-1} x_{i+1} \lambda_i.$$

With this,

$$\langle z, x \rangle = \lambda_n x_n - \sum_{i=1}^{n-1} \lambda_i (x_{i+1} - x_i).$$

This expression allows us to construct a suitable $\lambda \in C_1^\circ$ for a given $z \in C^\circ$. By definition of the polar cone, $\langle z, x \rangle \leq 0$ for all $x \in C$. To see that $\lambda_n = 0$,

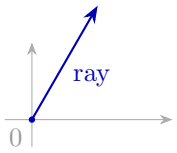


Fig. 6.3: A ray starts at the origin and extends in one direction.

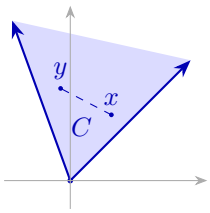


Fig. 6.4: A convex cone C: the segment between x and y stays in C.

[2] The word *nonempty* matters here. The empty set is also a closed convex cone, but every z satisfies the condition $\langle z, x \rangle \leq 0$ for all $x \in \varnothing$ vacuously, so $\varnothing^\circ = \mathbb{R}^n$. Consequently, $\varnothing^{\circ\circ} = (\mathbb{R}^n)^\circ = \{0\} \neq \varnothing$.

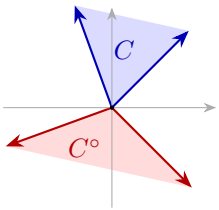


Fig. 6.5: A cone C and its polar cone C°.

[3] This identity is Abel summation. It is the discrete analogue of integration by parts.

take $x = (1,\dots,1)$, which is clearly an element of C. Since $x_{i+1} - x_i = 0$ in this case, $\lambda_n x_n = \lambda_n \le 0$. The constant vector $x = (-1,\dots,-1)$ also lies in C. But then $\langle z, x\rangle \le 0$ implies that $-\lambda_n \le 0$. It follows that $\lambda_n = 0$. For λ_i, $i = 1,\dots,n-1$, take the vector $x = (0,\dots,0,1,\dots,1)$ with $x_k = 0$ for $k \le i$ and $x_k = 1$ for $k > i$. This vector belongs to C, and

$$\langle z, x\rangle = \sum_{k=i+1}^{n} z_k = \lambda_n - \lambda_i = -\lambda_i.$$

Since $\langle z, x\rangle \le 0$, this implies that $\lambda_i \ge 0$. Thus $\lambda_i \ge 0$ for $i = 1,\dots,n-1$ and $\lambda_n = 0$, so $z \in C_1^\circ$. This proves $C^\circ \subset C_1^\circ$ and completes the proof. □

WE NEED TWO THEOREMS: the projection theorem, and a decomposition theorem that builds on it.

Theorem 6.6.2. The projection theorem, see Fig. 6.6. Let C be a nonempty, convex, closed cone in $\mathbb{R}^n$. Then,

1. For every $y \in \mathbb{R}^n$, there exists a unique point $\hat{y} \in C$, denoted as $P_C(y)$, such that $\|y - \hat{y}\| = \min_{x\in C}\{\|y - x\|\}$.
2. A point $\hat{y} \in C$ satisfies $\hat{y} = P_C(y)$ iff $\langle y - \hat{y}, x - \hat{y}\rangle \le 0$ for all $x \in C$.

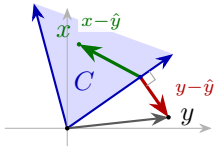


Fig. 6.6: The projection theorem.

—

Proof. For 1, since C is nonempty, pick $z \in C$. The intersection of C with the closed ball D with center y and radius $\|z - y\|$, i.e., $D \cap C$, is compact. Since the function $x \to \|x - y\|$ is continuous, it attains its minimum at some point $\hat{y}$ in $D \cap C$. Any point in $C \setminus D$ has distance greater than $\|z - y\|$ to y, hence $C \setminus D$ cannot contain a minimizer. Therefore $\hat{y}$ minimizes the distance to y over all of C.

If $y \in C$, then $y = P_C(y) = \hat{y}$: the minimal distance is 0, and it is attained at y only. Moreover, $\langle y - \hat{y}, x - \hat{y}\rangle = 0$ for all $x \in C$, so the inequality in 2 holds trivially. Assume therefore that $y \notin C$, so that $\|y - \hat{y}\| > 0$.

Consider now an *open* ball B with center y and radius $\|y - \hat{y}\|$, hence $\hat{y} \notin B$, but $\hat{y}$ is an element of the closure $\overline{B}$ of B. As $\hat{y} \in C$ is a point closest to y in C, $B \cap C = \varnothing$. However, $\hat{y} \in C \cap \overline{B}$, see Fig. 6.7.

Moreover, as C and B are both convex, by the *separating hyperplane theorem* there exists a hyperplane that separates B and C. This hyperplane must pass through $\hat{y}$, and as B is a ball, it is tangent to B at $\hat{y}$. Consequently, the vector $y - \hat{y}$ is normal to this hyperplane. As $x \in C$ and $y \in B$ lie at opposite sides of this plane, $\langle y - \hat{y}, x - \hat{y}\rangle \le 0$.

We can now also settle uniqueness. Any minimizer lies in $C \cap \overline{B}$. Since C and $\overline{B}$ lie at opposite sides of the hyperplane, $C \cap \overline{B}$ is part of the hyperplane itself, and the hyperplane, being tangent to the ball, meets $\overline{B}$ only at $\hat{y}$. Hence $C \cap \overline{B} = \{\hat{y}\}$: the minimizer is unique.

Finally, suppose that $p \in C$ is such that $\langle y - p, x - p\rangle \le 0$ for all $x \in C$. Then

$$\|y - x\|^2 = \|y - p + p - x\|^2 = \|y - p\|^2 + \|p - x\|^2 + 2\langle y - p, p - x\rangle.$$

As $\|p - x\| \ge 0$ and $\langle y - p, p - x\rangle \ge 0$ by assumption, $\|x - y\| \ge \|y - p\|$ for all $x \in C$. Thus p is the point closest in C to y, and therefore $p = \hat{y} = P_C(y)$. □

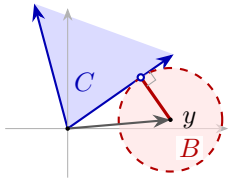


Fig. 6.7: The open ball B centered at y touches the cone C at the small open circle indicating $\hat{y}$. Since B is open, it is disjoint from C.

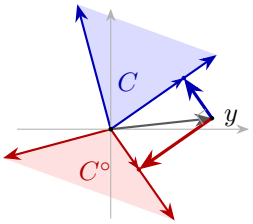


Fig. 6.8: Projection arrows in Moreau's decomposition: from y to $\hat{y} = P_C(y)$ and from y to $y^\circ = P_{C^\circ}(y) = y - \hat{y}$.

The projection theorem is the workhorse to prove the next theorem.

Theorem 6.6.3. Moreau's decomposition theorem, see Fig. 6.8. Let C be a nonempty, closed, convex cone. Then,

1. For every $y \in \mathbb{R}^n$, there exist unique projections $\hat{y} = P_C(y)$ on C and $y^\circ = P_{C^\circ}(y)$ on the polar cone C°, such that $y = \hat{y} + y^\circ$ and $\langle \hat{y}, y^\circ \rangle = 0$.
2. Moreover, if $y = r + s$ with $r \in C$, $s \in C^\circ$ and $\langle r, s \rangle = 0$, then $r = P_C(y)$ and $s = P_{C^\circ}(y)$.

Proof. By the projection theorem, $\hat{y} = P_C(y)$ exists and is unique. We first prove that $y - \hat{y} \in C^\circ$. By the projection theorem, $\langle y - \hat{y}, z - \hat{y} \rangle \le 0$ for all $z \in C$. As C is a cone, $0 \in C$ and $2\hat{y} \in C$, therefore

$$\begin{aligned} 0 &\ge \langle y - \hat{y}, 0 - \hat{y} \rangle = -\langle y - \hat{y}, \hat{y} \rangle, \\ 0 &\ge \langle y - \hat{y}, 2\hat{y} - \hat{y} \rangle = \langle y - \hat{y}, \hat{y} \rangle. \end{aligned}$$

It follows that $\langle y - \hat{y}, \hat{y} \rangle = 0$. Now take any $z \in C$. Then

$$\begin{aligned} \langle y - \hat{y}, z \rangle &= \langle y - \hat{y}, z - \hat{y} \rangle + \langle y - \hat{y}, \hat{y} \rangle \\ &\le 0. \end{aligned}$$

Consequently, $y - \hat{y} \in C^\circ$.

Next we prove that $y - \hat{y} = P_{C^\circ}(y)$. Let $p = y - \hat{y}$. For any $x \in C^\circ$,

$$\begin{aligned} \langle y - p, x - p \rangle &= \langle \hat{y}, x - (y - \hat{y}) \rangle \\ &= \langle \hat{y}, x \rangle - \langle \hat{y}, y - \hat{y} \rangle \\ &\le 0. \end{aligned}$$

Here $\langle \hat{y}, x \rangle \le 0$ because $\hat{y} \in C$ and $x \in C^\circ$, while $\langle \hat{y}, y - \hat{y} \rangle = 0$ by the previous paragraph. The projection theorem gives $p = P_{C^\circ}(y)$. Hence, with $y^\circ = p$,

$$y = \hat{y} + y^\circ, \qquad \hat{y} = P_C(y), \qquad y^\circ = P_{C^\circ}(y), \qquad \langle \hat{y}, y^\circ \rangle = 0.$$

Finally, suppose that $y = r + s$ with $r \in C$, $s \in C^\circ$ and $\langle r, s \rangle = 0$. For any $x \in C$,

$$\langle y - r, x - r \rangle = \langle s, x - r \rangle = \langle s, x \rangle - \langle s, r \rangle \le 0.$$

Thus $r = P_C(y)$. Similarly, for any $x \in C^\circ$,

$$\langle y - s, x - s \rangle = \langle r, x - s \rangle = \langle r, x \rangle - \langle r, s \rangle \le 0,$$

so $s = P_{C^\circ}(y)$. □

6.7 Why PAVA works

It remains to prove that PAVA, see Alg. 6.5.1, solves the isotonic regression problem. With the above, this reduces to showing that PAVA produces the projection $\hat{y} = P_C(y)$ of the responses y on the cone $C = \{x \in \mathbb{R}^n : Ax \le 0\}$. In other words, that it solves

$$\hat{y} \in \operatorname*{arg\,min}_{x : Ax \le 0} \frac{1}{2} \|x - y\|^2,$$

The procedure is to focus on consecutive blocks and repair any violations of the constraint $Ax \le 0$ one by one until no violations remain.

We also need some further notation. When $\mathcal{I}$ is a set of consecutive indices, then we write $y_{\mathcal{I}}$ for the restriction of y to $\mathcal{I}$, and $1_{\mathcal{I}}$ for the vector indexed by $\mathcal{I}$ whose components are all 1. Furthermore, $C_{\mathcal{I}} = \{x \in \mathbb{R}^{|\mathcal{I}|} : Ax \le 0\}$, where A is the matrix of (6.5.1), now of size $(|\mathcal{I}| - 1) \times |\mathcal{I}|$; in words, $C_{\mathcal{I}}$ consists of the nondecreasing vectors indexed by $\mathcal{I}$.

Throughout its sweep, PAVA maintains a partition of $\{1, \dots, n\}$ into blocks of consecutive indices, and on each block $\mathcal{I} = \{a, \dots, b\}$ the current fit $\hat{y}_{\mathcal{I}}$ is constant and equal to the block mean

$$\mu_{\mathcal{I}} = \frac{1}{|\mathcal{I}|} \sum_{l \in \mathcal{I}} y_l.$$

The *loop invariant* of the algorithm is that on each block the residual $r_{\mathcal{I},l} = y_l - \mu_{\mathcal{I}}$ satisfies

$$\sum_{l=a}^{m} r_{\mathcal{I},l} \ge 0 \quad (a \le m < b), \qquad \sum_{l=a}^{b} r_{\mathcal{I},l} = 0.$$

In words, as we move through the block from left to right, the accumulated excess over the block average never becomes negative, and it is zero at the right end of the block. By the polar-cone characterization above, applied with n replaced by $|\mathcal{I}|$, the invariant is exactly the statement that $r_{\mathcal{I}} \in C_{\mathcal{I}}^{\circ}$. The invariant holds at the start: PAVA begins with the singleton blocks $\{1\}, \dots, \{n\}$, so that $\hat{y} = y$ and every residual is 0.[1]

[1] If $\hat{y} \in C$, the algorithm terminates immediately.

The invariant has an important consequence: on each block $\mathcal{I}$, the constant vector $\hat{y}_{\mathcal{I}}$ with components $\mu_{\mathcal{I}}$ is the projection of $y_{\mathcal{I}}$ on the cone $C_{\mathcal{I}}$. To see this, note first that $\hat{y}_{\mathcal{I}}$, being constant, lies in $C_{\mathcal{I}}$. Second, the residual $r_{\mathcal{I}} = y_{\mathcal{I}} - \hat{y}_{\mathcal{I}}$ is orthogonal to the fit:

$$\langle r_{\mathcal{I}}, \hat{y}_{\mathcal{I}} \rangle = \langle y_{\mathcal{I}}, \hat{y}_{\mathcal{I}} \rangle - \langle \hat{y}_{\mathcal{I}}, \hat{y}_{\mathcal{I}} \rangle = \mu_{\mathcal{I}} \langle y_{\mathcal{I}}, 1_{\mathcal{I}} \rangle - \mu_{\mathcal{I}}^2 \langle 1_{\mathcal{I}}, 1_{\mathcal{I}} \rangle = 0, \tag{6.7.1}$$

because $\langle y_{\mathcal{I}}, 1_{\mathcal{I}} \rangle = \mu_{\mathcal{I}} |\mathcal{I}|$ and $\langle 1_{\mathcal{I}}, 1_{\mathcal{I}} \rangle = |\mathcal{I}|$. Third, the invariant gives $r_{\mathcal{I}} \in C_{\mathcal{I}}^{\circ}$. We have thus found vectors $\hat{y}_{\mathcal{I}} \in C_{\mathcal{I}}$ and $r_{\mathcal{I}} \in C_{\mathcal{I}}^{\circ}$ with $y_{\mathcal{I}} = \hat{y}_{\mathcal{I}} + r_{\mathcal{I}}$ and $\langle r_{\mathcal{I}}, \hat{y}_{\mathcal{I}} \rangle = 0$, so Moreau's theorem shows that $\hat{y}_{\mathcal{I}} = P_{C_{\mathcal{I}}}(y_{\mathcal{I}})$.

Next we show that pooling two violating blocks preserves the loop invariant. Suppose that two adjacent blocks $\mathcal{L} = \{i, \dots, j-1\}$ and $\mathcal{R} = \{j, \dots, k-1\}$ violate monotonicity:

$$\underbrace{\hat{y}_i = \cdots = \hat{y}_{j-1}}_{\mathcal{L}} > \underbrace{\hat{y}_j = \cdots = \hat{y}_{k-1}}_{\mathcal{R}}.$$

Thus $\mu_{\mathcal{L}} > \mu_{\mathcal{R}}$. To repair this, set

$$\mu = \frac{|\mathcal{L}|\mu_{\mathcal{L}} + |\mathcal{R}|\mu_{\mathcal{R}}}{|\mathcal{L} \cup \mathcal{R}|} = \frac{1}{|\mathcal{L} \cup \mathcal{R}|} \sum_{l \in \mathcal{L} \cup \mathcal{R}} y_l,$$

that is, the mean of the responses on the combined set $\mathcal{L} \cup \mathcal{R}$. Observe that $\mu_{\mathcal{L}} > \mu > \mu_{\mathcal{R}}$.

Define $\tilde{y}_l = \mu$ and $\tilde{r}_l = y_l - \tilde{y}_l$ for $l = i, \dots, k-1$. Then $\tilde{y}$, being constant, lies in the cone $C_{\mathcal{L} \cup \mathcal{R}}$. To see that the pooled residual still satisfies the loop invariant, we reason as follows. On $\mathcal{L}$, each new residual $y_l - \mu$ is larger than the old residual $y_l - \mu_{\mathcal{L}}$, so all accumulated sums inside $\mathcal{L}$ increase; in particular, the accumulated sum at the right end of $\mathcal{L}$ rises from 0 to $|\mathcal{L}|(\mu_{\mathcal{L}} - \mu) > 0$. On $\mathcal{R}$, each new residual is the old residual minus $\mu - \mu_{\mathcal{R}}$, so after m elements of $\mathcal{R}$ the accumulated sum has dropped by $m(\mu - \mu_{\mathcal{R}})$, which is at most $|\mathcal{R}|(\mu - \mu_{\mathcal{R}})$. The definition of the pooled mean gives

$$|\mathcal{L}|(\mu_{\mathcal{L}} - \mu) = |\mathcal{R}|(\mu - \mu_{\mathcal{R}}),$$

so the drop on $\mathcal{R}$ never exceeds the surplus gained at the right end of $\mathcal{L}$. Since the old partial sums within $\mathcal{R}$ are themselves nonnegative by the invariant on $\mathcal{R}$, no accumulated sum of the merged block becomes negative. Moreover, at the right end of the merged block the surplus and the deficit cancel exactly, so the total sum is 0. We therefore have $\tilde{y} \in C_{\mathcal{L} \cup \mathcal{R}}$ and $\tilde{r} \in C^\circ_{\mathcal{L} \cup \mathcal{R}}$, and the same computation as in (6.7.1) shows that $\langle \tilde{r}, \tilde{y} \rangle = 0$. Thus, with Moreau's theorem, $\tilde{y}$ is the projection of the responses on the cone $C_{\mathcal{L} \cup \mathcal{R}}$.

It is interesting to see why we should not merge when $\mu_{\mathcal{L}} < \mu_{\mathcal{R}}$. In this case $\mu > \mu_{\mathcal{L}}$, so the accumulated sum at the right end of $\mathcal{L}$ drops from 0 to $|\mathcal{L}|(\mu_{\mathcal{L}} - \mu) < 0$: the pooled residual violates the partial-sum condition, that is, $\tilde{r} \notin C^\circ_{\mathcal{L} \cup \mathcal{R}}$.

The last step is to establish termination and to assemble the blocks. Each pooling step reduces the number of blocks by one, and the algorithm starts with n blocks, so it terminates after at most $n-1$ pooling steps. Upon termination no violations remain, so $\hat{y} \in C$. The accumulated sums of the full residual $r = y - \hat{y}$ consist of sums over complete blocks, which are 0, plus a partial sum inside one block, which is nonnegative; in particular the total sum is 0. By the polar-cone characterization, $r \in C^\circ$. Finally, $\langle r, \hat{y} \rangle$ is the sum of the block-wise inner products $\langle r_{\mathcal{I}}, \hat{y}_{\mathcal{I}} \rangle$, each of which is 0 by (6.7.1). Moreau's theorem now yields $\hat{y} = P_C(y)$: PAVA indeed solves the isotonic regression problem.

THE KKT CONDITIONS connect the geometric projection proof to constrained optimization. The projection of y on the cone C is the constrained optimization problem

$$\min_{x \in C} \frac{1}{2} \|x - y\|^2.$$

We take this problem as the primal problem and derive its dual. This shows that the polar-cone residual in Moreau's theorem is the dual optimizer, and that orthogonality becomes complementary slackness. The Lagrangian is

$$\mathcal{L}(x, \lambda) = \frac{1}{2} \|x - y\|^2 + \langle \lambda, Ax \rangle, \quad \lambda \in \mathbb{R}^{n-1}_+.$$

Setting the derivative of $\mathcal{L}$ with respect to x to 0 and solving for x gives $x = y - A^\top \lambda$. Write this solution as $\hat{y} = y - A^\top \lambda$. Inserting this in the Lagrangian gives the dual problem

$$\max_{\lambda \geq 0} \left\{ -\frac{1}{2} \left\| A^\top \lambda \right\|^2 + \langle \lambda, Ay \rangle \right\}.$$

Now define $r = A^\top \lambda$, so that the dual objective becomes $-\frac{1}{2}\|r\|^2 + \langle r, y \rangle$ with $r \in C^\circ$, by the second characterization of the polar cone in Lem. 6.6.1. Completing the square gives

$$-\frac{1}{2}\|r\|^2 + \langle r, y \rangle = -\frac{1}{2}\left\| y - r \right\|^2 + \frac{1}{2}\left\| y \right\|^2.$$

Since the last term is a constant, maximizing the dual is the same as

$$\min_{r \in C^\circ} \frac{1}{2}\left\| y - r \right\|^2.$$

This is precisely the projection of y on C°. The projections of y on C and on C° solve the primal and the dual problem simultaneously.

The projection theorem gives unique solutions on C and on its polar cone C°. As $\hat{y} = P_C(y) \in C$, $\hat{y}$ satisfies primal feasibility, likewise $r = y - \hat{y} \in C^\circ$, hence r satisfies dual feasibility. By Moreau's theorem, $0 = \langle r, \hat{y} \rangle = \langle A^\top \lambda, \hat{y} \rangle = \langle \lambda, A\hat{y} \rangle$. Since $\lambda_i \geq 0$ and $(A\hat{y})_i \leq 0$, every term of this sum is nonpositive, so each must be zero: $\lambda_i (A\hat{y})_i = 0$ for all i, which is complementary slackness. And finally, $\hat{y} - y + A^\top \lambda = 0$ is the stationarity of the Lagrangian at the optimal x.

6.8 Discussion

Three limits are worth stating before we move on, but let us first summarize what we did above. We began with a number the forest already produces, the fraction of its trees that vote bad, and ended with a function g that turns that number into a probability we are willing to use as a routing cost. Three methods did the turning, and they differ in just two respects: the shapes they allow for g, and the loss they minimize. Sigmoid scaling allows a two-parameter S-curve and minimizes the cross-entropy that the Bernoulli likelihood selects; histogram binning allows any step function on a fixed set of bins and minimizes squared error, which the bin averages solve outright; isotonic regression allows any non-decreasing function and minimizes that same squared error under the ordering constraint, which is what PAVA computes. The fitting itself was nowhere the hard part: for sigmoid scaling it is the gradient descent of Chapter 5, and for the other two an average or a projection that we can write down.

ALL THREE METHODS fit g on a single held-out list $\mathcal{H}$, and all three assume that the scores the forest produces later are distributed as the scores in $\mathcal{H}$. This assumption is invisible in the result: a fitted g does not report that the forest beneath it has drifted, and it will keep returning confident-looking probabilities long after it has stopped deserving them. If the map, the feature set, or the training sample changes, the calibration has to be refitted, and the only protection is to refit whenever anything upstream of g changes. This is the distribution shift of Chapter 3 in calibration clothing.

ISOTONIC REGRESSION RETURNS a function that is defined only at the observed scores $u_1, \ldots, u_n$, because what the fit produces is a list of n numbers rather than a formula. Extending it to all of $[0, 1]$ is a separate choice, and there is more than one reasonable one: keep g piecewise-constant and jump between blocks, or interpolate linearly between consecutive fitted values. Sigmoid scaling has no such problem, since it returns a formula that can be evaluated at any score at all, and histogram binning has none either as long as the bins cover $[0, 1]$.

SCORES OUTSIDE THE RANGE seen in $\mathcal{H}$ are where all three methods are weakest, and for much the same reason. There the calibrated value is an extrapolation rather than a fitted probability: sigmoid scaling evaluates its S-curve without complaint, while histogram binning and isotonic regression can do no more than repeat their outermost value. Since $\mathcal{H}$ is held out from the training data it is typically small, so the range of scores it covers can be noticeably narrower than the range the forest produces over the whole map. This is a real concern rather than a formality: the edges with the most extreme scores are exactly the ones a router acts on most decisively.

With that said, we have assembled all the conceptual steps needed to build the app that creates walks between points A and B. In Chapter B we turn to the technical details of doing so.

6.9 Exercises

Exercise (C) 6.9.1. A random forest ranks the OSM edges of the walking-route planner from good to bad. Why is this ranking not enough for the routing step, and what does the calibration map g of this chapter add to it?

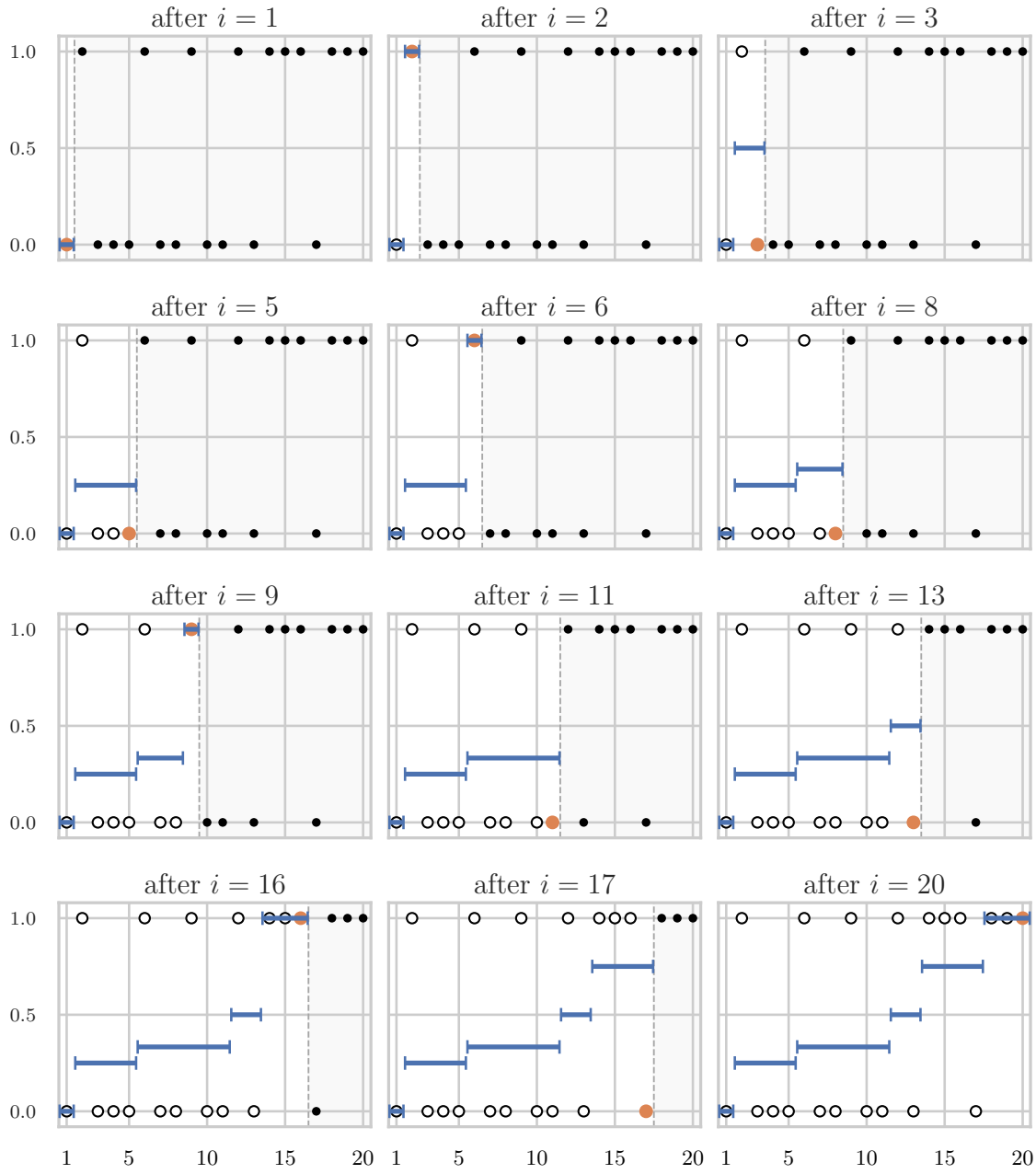


FIG. 6.2: PAVA as a left-to-right sweep through the sorted observations. Each panel contains all observations. The filled black points are observations not yet used in the computation of $\hat{y}_i$, the open circles have been merged into blocks, the orange point is the newest observation, and the blue segments are the block averages after the indicated sample has been processed.

Chapter 7

Neural Networks

Neural networks are a very versatile set of models to make predictions. In this section we discuss their basic working. We start with a network with a single neuron. Then we extend to sets of neurons stacked vertically into layers, then placed horizontally side by side within a layer. We show how to use gradient descent of Section 5.1 to update the parameters of the network to fit the training data. We present different types of ideas and mechanisms that are used to set up and regularize neural networks. How to engineer a good architecture is a hot topic in current research.

We remark that we only discuss *how* neural networks work, but not *why*. Their power can perhaps best be understood as an emergent phenomenon.[1]

[1] A simple example of an emergent phenomenon is the freezing of water. Clearly, one molecule of H_2O is not water, but how many molecules are needed before it becomes a water droplet that can freeze? Similarly, one biological neuron cannot think. Put many together and we obtain the brain of a snail. Put a huge number together and we get a rat that can solve labyrinth puzzles, with yet more we have 'us'.

7.1 Single-neuron networks

The simplest neural network consists of a single neuron taking a single feature x as input and producing one prediction $\hat{y}$. Fig. 7.1 shows the two steps. It does this by first computing the *logit* (5.1.3)

$$z = ax + b,$$

with two parameters a and b. Then it applies an *activation function* to obtain the output $\hat{y} \in \mathcal{Y}$. Throughout this chapter we use the sigmoid as example,

$$\hat{y} = \sigma(z), \qquad \sigma(z) = \left(1 + e^{-z}\right)^{-1}.$$

The role of σ is to transform the unbounded logit $z \in \mathbb{R}$ into a number between 0 and 1, so that $\hat{y}$ can be interpreted as a probability. Thus, once a and b are known, prediction is just the calculation above. Note that the mechanism we describe below can be easily changed to other activiation functions.

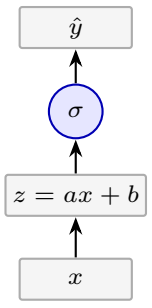


Fig. 7.1: A single neuron computes a logit z from the input x and passes it through the activation function σ to obtain the response $\hat{y}$.

FOR TRAINING WE use the observations (x, y) in the training set $\mathcal{T}$. The goal is to solve the problem:

$$\theta^* \in \arg\min_{\theta} \hat{R}_{\mathcal{T}}(a, b), \qquad \theta = (a, b), \tag{7.1.1}$$

where the *training risk* is the average over the training samples,

$$\hat{R}_{\mathcal{T}}(a, b) = \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \ell(y, z), \qquad z = ax + b,$$

and $\ell(y,z)$ is the loss of the parameters (a,b) on the single training sample (x,y). Below we use squared-error loss $\ell(a,b;x,y) = \big(y-\hat{y}\big)^2 = \big(y-ax-b\big)^2$. For hard labels $y \in \{0,1\}$ the binary cross-entropy (5.1.5) is the natural choice for ℓ, and for *soft* targets $y \in [0,1]$.

The solution $\theta^* = (a^*, b^*)$ can be found with the gradient descent tools of Section 5.1. Once solved, θ^* is then used for prediction: for a new predictor x, the neuron returns $\sigma(a^* x + b^*)$ as the prediction.

REGULARIZATION IS IMPORTANT even for this single neuron network; however, these ideas apply more generally.

Early stopping uses a validation set to decide when to stop training. Split the available labeled data into a training set $\mathcal{T}$ and a *validation set* $\mathcal{V}$. The parameters are updated by gradient descent on $\hat{R}_{\mathcal{T}}$ as before, but now, at each iteration k, we store the validation risk

$$\hat{R}_{\mathcal{V}}(\theta^{(k)}) = \frac{1}{|\mathcal{V}|} \sum_{(x,y)\in\mathcal{V}} \ell(\theta^{(k)}; x, y)$$

together with the parameters $\theta^{(k)} = (a^{(k)}, b^{(k)})$. We continue training as long as the trainging and validation loss decrease. When the validation risk has failed to improve for a number of successive iterations, we stop iterating and use

$$k^* \in \arg\min_k \hat{R}_{\mathcal{V}}(\theta^{(k)})$$

to select $\theta^{(k^*)}$ as the fitted parameter vector for prediction instead of the last parameter values obtained from training. This matters because the training risk can keep decreasing, while the validation risk can increase once the model begins to fit accidental details of $\mathcal{T}$.

Weight decay copies the idea from ridge regression by including a penalty in the objective:

$$\min_{a,b} \hat{R}_{\mathcal{T},\lambda}(a,b), \qquad \hat{R}_{\mathcal{T},\lambda}(a,b) = \hat{R}_{\mathcal{T}}(a,b) + \frac{\lambda}{2}a^2,$$

where $\lambda \geq 0$ is a tuning parameter.[1]

The bias b is often left out of the penalty. The reason is that a controls the steepness of $\sigma(ax+b)$, while b controls where the steep part of the curve lies.[2] Penalizing a discourages overly sharp changes in the fitted probabilities. Penalizing b would also discourage moving the steep part of the curve to where the data place it, which is usually not the intended effect.

The extra term in the objective contributes λa to the derivative with respect to a, so the update in gradient descent contains a small shrinkage toward zero. As $\frac{\partial}{\partial a}\hat{R}_{\mathcal{T},\lambda}(a,b) = \frac{\partial}{\partial a}\hat{R}_{\mathcal{T}}(a,b) + \lambda a$, the update for a is

$$\begin{aligned} a^{(k+1)} &= a^{(k)} - \eta\left(\frac{\partial}{\partial a}\hat{R}_{\mathcal{T}}(a^{(k)}, b^{(k)}) + \lambda a^{(k)}\right) \\ &= (1-\eta\lambda)\,a^{(k)} - \eta\frac{\partial}{\partial a}\hat{R}_{\mathcal{T}}(a^{(k)}, b^{(k)}). \end{aligned}$$

The first term is a shrunken version of the old value $a^{(k)}$, before the usual gradient of the training risk is added. Note that λ should be chosen with care: the minimal requirement is $|1-\eta\lambda| < 1$.

[1] In larger networks, the weight a is a matrix. Then we penalize the sum of the squared entries of the weight matrix, $\|A\|^2$, instead of just a^2.

[2] $ax+b=0$ at $x=-b/a$.

7.2 Multi-neuron networks

A single neuron only gives shifted and stretched copies of one fixed nonlinearity, $\sigma(ax+b)$. Neural networks become more flexible by combining many such transformations: some are placed in a sequence of layers, making the network *deeper*, and some are placed side by side, making a layer *wider*. Networks built from multiple layers, each of several neurons wide, are called *multilayer perceptrons*.

Making networks deeper and wider can improve prediction quality. Different hidden neurons can learn different useful patterns in the input, and later layers can combine patterns learned by earlier layers. However, if the network is made too large for the amount of data, the same flexibility can overfit by picking up accidental noise instead of structure.

We first study depth in the simplest case: a scalar stack with one neuron per layer. Then we make the network wider, and introduce more indices to cope with the extra complexity.

THE SIMPLEST STACK passes information from single neurons from one layer to the next, see Fig. 7.2. For instance, starting from the input x, define

$$\begin{aligned} z_1 &= a_1 x + b_1, \qquad h_1 = \phi_1(z_1), \\ z_2 &= a_2 h_1 + b_2, \qquad h_2 = \phi_2(z_2), \\ z_3 &= a_3 h_2 + b_3, \qquad \hat{y} = \sigma(z_3). \end{aligned} \tag{7.2.1}$$

Each a_i and b_i, for $i = 1,2,3$, is a trainable parameter; ϕ_1 and ϕ_2 are activation functions, in the same role as σ.[1] The intermediate values h_1 and h_2 are called *hidden* as the network only uses them internally. The z_i are the *pre-activations*; the last one, z_3, is the *logit*. Since the signal flows only forward, from x through h_1 and h_2 to $\hat{y}$, without cycles or feedback, such a network is called *feedforward*.

This stack is richer than a single neuron because each layer transforms the signal once more, so a later layer works with features produced by the earlier layers rather than with the raw input.

PREDICTION IS IMPLEMENTED as a *forward pass*. Given parameters (a_i, b_i) for each neuron, compute for the predictor x the preactivations z_1, h_1, z_2, h_2, z_3 in order of (7.2.1) and return $\hat{y} = \sigma(z_3)$.

TRAINING MINIMIZES THE training risk by (mini-batch) gradient descent, now over all six parameters $\theta = (a_1, b_1, a_2, b_2, a_3, b_3)$. The new difficulty is computing the gradient. *Backpropagation*[2] is the systematic way to handle this: it is the chain rule, applied from the output back to the input.

First, apply a forward pass to one observation (x, y) to compute the loss $\ell = \ell(y, \hat{y})$, all pre-activations z_i and all hidden values h_i. The computations below use these values.

Backpropagation starts at the output, that is, at the top of Fig. 7.2. Since $\hat{y} = \sigma(z_3)$, the chain rule applied to the squared loss $\ell = (y-\hat{y})^2/2$ gives

$$\delta_3 = \frac{\partial \ell}{\partial z_3} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_3} = \frac{\partial \ell}{\partial \hat{y}} \sigma'(z_3) = \big(\hat{y} - y\big)\, \hat{y}(1-\hat{y}), \tag{7.2.2}$$

[1] We discuss examples below.

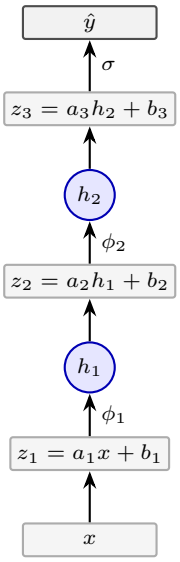


Fig. 7.2: A scalar multilayer network.

[2] Michael A. Nielsen, *Neural Networks and Deep Learning*

i.e., the factor we met in the one-neuron network. The parameters a_3 and b_3 enter the loss only through $z_3 = a_3 h_2 + b_3$, so again by the chain rule,

$$\frac{\partial \ell}{\partial a_3} = \frac{\partial \ell}{\partial z_3}\frac{\partial z_3}{\partial a_3} = \delta_3 h_2, \qquad \frac{\partial \ell}{\partial b_3} = \frac{\partial \ell}{\partial z_3}\frac{\partial z_3}{\partial b_3} = \delta_3.$$

Note that from the forward pass we have the numerical values of δ_3 and h_2 at the given sample (x, y). Thus, the RHSs are just numbers that we[3] can compute.

[3] That is, the computer.

Now step one layer back. From (7.2.1) and the chain rule

$$\delta_2 = \frac{\partial \ell}{\partial z_2} = \frac{\partial \ell}{\partial z_3}\frac{\partial z_3}{\partial h_2}\frac{\partial h_2}{\partial z_2} = \delta_3\, a_3\, \phi_2'(z_2).$$

Since $z_2 = a_2 h_1 + b_2$, again by the chain rule,

$$\frac{\partial \ell}{\partial a_2} = \frac{\partial \ell}{\partial z_2}\frac{\partial z_2}{\partial a_2} = \delta_2 h_1, \qquad \frac{\partial \ell}{\partial b_2} = \frac{\partial \ell}{\partial z_2}\frac{\partial z_2}{\partial b_2} = \delta_2.$$

One more step of the same kind yields

$$\delta_1 = \delta_2\, a_2\, \phi_1'(z_1), \qquad \frac{\partial \ell}{\partial a_1} = \delta_1 x, \qquad \frac{\partial \ell}{\partial b_1} = \delta_1. \tag{7.2.3}$$

Note the pattern: One forward pass computes the z_i and hidden values h_i, one backward computes each δ_i from δ_{i+1} by multiplying with the weight a_{i+1} and the local slope $\phi_i'(z_i)$. Together they deliver the complete gradient $\nabla \ell(\theta; x, y)$ for the given sample (x, y).

The parameter update is then the same as before, except that rather than updating from just one sample, we use a mini-batch $\mathcal{B}$ to estimate the gradient with (5.1.6) and update the parameters through:

$$\theta^{(k+1)} = \theta^{(k)} - \eta \nabla \hat{R}_{\mathcal{B}}(\theta^{(k)}).$$

For instance, if the mini-batch would consist of one sample, then,

$$a_1^{(k+1)} = a_1^{(k)} - \eta \frac{\partial \ell}{\partial a_1} = a_1^{(k)} - \eta \delta_1 x, \quad b_1^{(k+1)} = b_1^{(k)} - \eta \frac{\partial \ell}{\partial b_1} = b_1^{(k)} - \eta \delta_1.$$

The other parameters follow in the same way.

Because the training risk of a neural network is not necessarily convex, two runs started from different initial parameter values may end at different local minima. A simple practical response is to train the same architecture from several random initializations and keep the run with the lowest validation risk.

Generalizing to networks with L layers is straightforward.

WIDENING A LAYER places several neurons side by side in one layer, see Fig. 7.3. A neuron in a wide layer now receives several inputs, one from every neuron in the previous layer, so its weights carry three indices: $A_i(j, k)$ is the weight that neuron j of layer i applies to its k-th input. Then hidden layer i computes the components of its pre-activation and hidden vector as

$$z_i(j) = \sum_{k=1}^{q_{i-1}} A_i(j, k)\, h_{i-1}(k) + b_i(j), \qquad h_i(j) = \phi(z_i(j)). \tag{7.2.4}$$

Collecting the weights of layer i in the matrix $A_i \in \mathbb{R}^{q_i \times q_{i-1}}$, whose (j,k) entry is $A_i(j,k)$, the hidden vector $h_{i-1} \in \mathbb{R}^{q_{i-1}}$, and the biases in the vector $b_i \in \mathbb{R}^{q_i}$, this becomes

$$z_i = A_i h_{i-1} + b_i, \qquad h_i = \phi(z_i), \qquad i = 1, \dots, L, \tag{7.2.5}$$

where ϕ is applied element-wise. The output is layer $L+1$. It consists of one neuron, so $q_{L+1} = 1$, and it reads the last hidden layer,

$$z_{L+1} = A_{L+1} h_L + b_{L+1}, \qquad \hat{y} = \sigma(z_{L+1}) \tag{7.2.6}$$

with $A_{L+1} \in \mathbb{R}^{1 \times q_L}$ and $b_{L+1} \in \mathbb{R}$. Thus the output layer has the same form as a hidden layer; only its activation differs, σ instead of ϕ.

Each layer has its own parameters $A_i(j,k), b_i(j)$; thus

$$\theta = (A_1, b_1, \dots, A_{L+1}, b_{L+1}).$$

Note how quickly θ grows: layer l alone has $q_l(q_{l-1}+1)$ parameters.

INCLUDING MULTIPLE FEATURES is now straightforward. When the input is a *feature vector* $x \in \mathbb{R}^p$, just set $h_0 = x$ and $q_0 = p$, and all of (7.2.5) applies right away. Thus, moving from one to p features changes only the bookkeeping for the first layer, later layers are unaffected as they read only from the previous hidden layer.

BACKPROPAGATION CARRIES OVER nearly unchanged. Again, there is one δ per neuron, so $\delta_i \in \mathbb{R}^{q_i}$ is a vector. The output layer $L+1$ has one neuron, so for squared loss

$$\delta_{L+1} = \big(\hat{y} - y\big)\, \hat{y}(1-\hat{y}).$$

From this, the other backward recursion become with the chain rule,

$$\begin{aligned} \delta_i(j) = \frac{\partial \ell}{\partial z_i(j)} &= \sum_k \frac{\partial \ell}{\partial z_{i+1}(k)} \frac{\partial z_{i+1}(k)}{\partial h_i(j)} \frac{\partial h_i(j)}{\partial z_i(j)} \\ &= \sum_k \delta_{i+1}(k) A_{i+1}(k,j) \phi_i'(z_i(j)), \end{aligned}$$

where we use (7.2.4) for the second partial derivative and $h_i(j) = \phi_i(z_i(j))$ for the third. Now note that the summation over k is the multiplication of the vector δ_{i+1} and the matrix A_{i+1}. With the Hadamard product[4] we can then write

$$\delta_i = (\delta_{i+1} A_{i+1}) \odot \phi_i'(z_i), \qquad i = L, \dots, 1.$$

Finally, the gradients of layer $i = 1, \dots, L+1$ become

$$\frac{\partial \ell}{\partial A_i} = \delta_i\, h_{i-1}^\top, \qquad \frac{\partial \ell}{\partial b_i} = \delta_i,$$

because

$$\frac{\partial \ell}{\partial A_i(j,k)} = \frac{\partial \ell}{\partial z_i(j)} \frac{\partial z_i(j)}{\partial A_i(j,k)} = \delta_i(j) h_{i-1}(k).$$

As an aside, in software these formulas need not be derived by hand: *automatic differentiation* records every elementary operation of the forward pass and applies the chain rule mechanically in reverse, which reproduces exactly the backpropagation computation above, for a network of any shape. This is what allows practitioners to change an architecture without redoing any calculus. Also, in real large networks, plain gradient descent is often replaced by specialized optimizers such as momentum methods, Adam, or AdamW.[5]

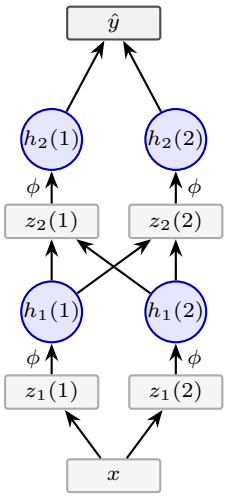


Fig. 7.3: A small multilayer perceptron.

[4] The Hadamard product, written as $C = A \odot B$, of two matrices A and B with the same dimensions is the element-wise multiplication: $C_{ij} = A_{ij} B_{ij}$.

[5] Wikipedia: Stochastic gradient descent, Adam.

MULTI-NEURON NETWORKS can use other regularization techniques besides early stopping and weight decay.

Dropout is a technique that aims to prevent the network from relying too much on just a few neurons in a layer. The risk with q parallel neurons is that during training they can start to cooperate too closely so that just a few neurons in that layer become dominant, which in turn may lead to tuning that layer to accidental details of the training set. Dropout counters this by switching off some neurons randomly selected for each mini-batch update, see Fig. 7.4. Like this, the network cannot rely on just a few neurons; in that sense, it should make networks more robust.

Dropout is generally applied to each hidden layer separately; we discuss the details for one layer. Given the vector h_i of hidden layer i, let $\rho \in [0,1)$ be the dropout probability. For each mini-batch update, take $M_i = (M_i(1), \ldots, M_i(q_i))$ where

$$M_i(j) \sim \text{Bern}\big(1-\rho\big), \quad j = 1, \ldots, q_i$$

are independent mask variables.[6] Then, replace h_i by its randomly thinned version

$$\tilde{h}_i = \frac{M_i \odot h_i}{1-\rho},$$

and use $\tilde{h}_i$ instead of h_i in (7.2.5). Like this, each (mini-batch) update trains a differently thinned network. Since the model cannot rely on the same hidden units always being present, no single neuron can play too dominant a role.

The division by $1-\rho$ is called *inverted dropout*. It keeps $\mathsf{E}\big[\tilde{h}_i(j)\big] = h_i(j)$, so that in expectation, the activation remains equally strong during each training step.

Dropout is only applied during training. Once trained, the full network is used for prediction.

DropConnect is the same idea applied to individual weights instead of hidden units. For the weight matrix A_i, take a mask matrix B_i of the same shape, with independent entries

$$B_i(j,k) \sim \text{Bern}\big(1-\rho\big).$$

During training, replace A_i by its randomly thinned version

$$\tilde{A}_i = \frac{B_i \odot A_i}{1-\rho}.$$

As with dropout, the random masking is only used during training; once trained, the full weight matrix is used for prediction.

Freezing[7] keeps selected parameters fixed during training, as in Fig. 7.5. The frozen neurons still compute their values in the forward pass, but their parameters are excluded from the gradient updates. This is a deliberate restriction on what the optimizer is allowed to change.

Freezing is a useful technique to enable *transfer learning*[8] . Transfer learning is to reuse part of a network trained on one task for a related new task. A typical workflow is:

1. Start from a network already trained on a large source data set.

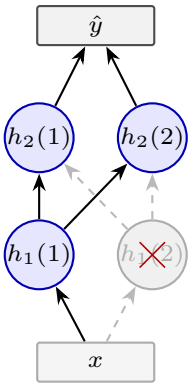


Fig. 7.4: Dropout switches off hidden units during training.

[6] In deeper networks, use a separate mask for each dropout layer.

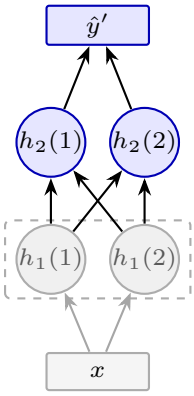


Fig. 7.5: Freezing one layer keeps its weights fixed while the rest of the network trains.

[7] Wikipedia: Fine-tuning (deep learning).

[8] Freezing is not the same as transfer learning; it is one common tool used when reusing a pretrained network for a new task.

2. Keep its lower layers, because they often capture generic features.
3. Add (or replace) an output layer for the new task.
4. Train only this last layer.
5. Optionally unfreeze some of the lower layers and continue training with small learning rate. This is known as *fine-tune* it.

Thus, by freezing the lower layers we can save computation and protect against overfitting when the new data set is small.

7.3 Further architecturial aspects

Except specifying the number of layers and the widths of the layers, there are some other parts of a neural networks that need consideration: which activation functions to use, how many outputs the network should produce, and how information should flow[1] between distant parts of the network.

[1] That is, how the neurons and the layers are connected.

A FIRST QUESTION is what a given architecture can represent at all. The *universal approximation theorem*[2] given an intereting answer: a one layer network that is sufficiently suffices in the following sense. Take $L = 1$ and the last activation the identity so that network computes

[2] Wikipedia: Universal approximation theorem.

$$\hat{f}(x) = \sum_{j=1}^{q_1} A_2(1,j)\,\sigma\left(\sum_{k=1}^{p} A_1(j,k)\,x(k) + b_1(j)\right) + b_2,$$

i.e., a weighted sum of q_1 shifted and stretched copies of the sigmoid function. The theorem says that for every continuous f on a compact domain and every $\epsilon > 0$ there are a width q_1 and parameters A_1, b_1, A_2, b_2 such that $|\hat{f}(x) - f(x)| < \epsilon$ for all x in that domain. Thus, one hidden layer suffices, provided it is wide enough

This is an existence result, not a training guarantee. So the interesting question is not whether an architecture can represent a function, but which architectures reach a given accuracy with few parameters and can still be trained; this is the subject of current research.

ACTIVATION FUNCTIONS need not be sigmoid. Another common hidden-layer choice is the ReLU,[3]

[3] Wikipedia: Rectified linear unit.

$$\phi(z) = \max\{0, z\} \qquad\qquad \phi'(0) = 0.$$

This is useful because an active[4] ReLU has derivative 1: it passes the gradient backward without shrinking it. A sigmoid hidden unit, by contrast, has derivative at most[5] $1/4$ and becomes almost flat for large positive or negative z which leads to the following problem. Recall the backpropagation recursion applied to the simple three-layer feedforward network above. For instance, in example (7.2.2)–(7.2.3), we see that

[4] That is, when $z > 0$.

[5] $\sigma'(z) = \sigma(z)\,(1 - \sigma(z)) \le 1/4$

$$\delta_1 = a_2\phi_1'(z_1)\,a_3\phi_2'(z_2)\,\frac{\partial \ell}{\partial \hat{y}}\,\sigma'(z_3). \tag{7.3.1}$$

Each extra layer contributes one more weight and one more slope. So in a stack of, say, 15 sigmoid layers the slopes alone contribute a factor of at

most $(1/4)^{15} \approx 10^{-9}$. Thus, when using sigmoids throughout, the absolute value of gradient becomes small, and the lower layers barely learn; this is why stacks of more than a few layers are hard to train. This phenomenon is known as *vanishing gradients*.

ReLU does not suffer from this problem because its derivative is either 0 or 1.[6]

A smooth variant of the ReLU is the *GELU*,[7]

$$\phi(z) = z\,\mathrm{P}\{Z \le z\}, \qquad Z \sim \mathrm{Norm}(0,1).$$

For large positive z it behaves like the identity and for large negative z it tends to 0, as the ReLU does. It differs near the origin: the GELU is negative for every $z < 0$, with a minimum of about -0.17 at $z \approx -0.75$, so unlike the ReLU it is not monotone.

OUTPUT HEADS [8] are final output neurons, or final small output layers, for separate prediction tasks, see Fig. 7.6. So far, the network had just one output head: one $\hat{y}$ for the input x.] However, often we can ask several related questions about the same input. For instance, does a patient has illness A and illness B? Each such question can be answered by its own head but, as Fig. 7.6 shows, all heads share the same hidden layer.

For K such questions, take as logits and outputs

$$z^{(r)} = b_0^{(r)} + \sum_{j=1}^{q_L} a^{(r)}(j)\,h_L(j), \qquad \hat{y}^{(r)} = \sigma(z^{(r)}), \qquad r = 1,\dots,K.$$

Thus, each head has its own training parameters.

For training, this means that, instead of samples with an outcome y, each observation has *multiple* targets $y^{(1)},\dots,y^{(K)}$, one for each supervised head. Training changes little: take as loss the sum of the K per-head losses and apply backpropagation as before.

Sharing information from lower layers is efficient: the features $h_L(j)$ are learned once but used K times. Interestingly, training with multiple labels per input, known as *multi-task learning*, offers another advantage. The gradient of every head's loss flows back into the same hidden layer, so each hidden neuron now receives K training signals instead of one. The hidden layers h_i must therefore capture structure that all K prediction tasks have in common and can therefore not specialize in accidental details of a single task. Multiple heads thus act as regularizers for one another.

SKIP CONNECTIONS BECOME important in deep networks. A skip connection lets a layer pass its input directly to a later layer. A *residual* connection[9] is the additive case of a skip connection: the skipped input and the learned update are merged by addition, not by a more general function. So, instead of setting $h_i = \phi_i(z_i)$ as in (7.2.1), the hidden values are given by

$$h_i = h_{i-1} + \phi_i(z_i), \qquad z_i = A_i h_{i-1} + b_i,$$

equivalently, $\Delta h_i := h_i - h_{i-1} = \phi_i(z_i)$. Thus, , instead of being replaced in the forward pass, layer i receives an update Δh_i on the hidden vector of the previous layer.[10]

[6] The probability that a preactivation hits 0 is negligible.

[7] The Gaussian error linear unit, Wikipedia: Rectified linear unit, GELU. Is is used in GPT-style language models.

[8] Do not confuse these output heads with the attention heads used inside Transformer layers of large language models.

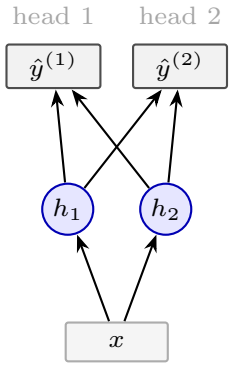


Fig. 7.6: Each head is a separate output neuron reading the same hidden layer.

[9] Wikipedia: Residual neural network.

[10] This is close to an Euler step for a differential equation, where a new state is the old state plus a small change.

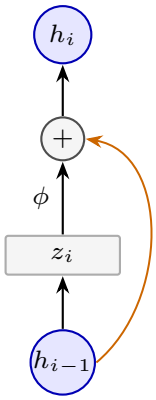


Fig. 7.7: A residual connection sends a layer input directly to an addition node, where it is added to the layer update.

The effect of this change on training appears when we differentiate it: the derivative through the layer is

$$\frac{\partial h_i}{\partial h_{i-1}} = I + \operatorname{diag}((\phi_i)'(z_i))A_i,$$

and the identity matrix I is the point: the gradient always has a direct path back, so it no longer shrinks geometrically with depth, as it did in (7.3.1).

Residual connections help make networks of dozens or even hundreds of layers trainable. The first famous example was ResNet, an image-recognition network with residual blocks and over a hundred layers; the transformer blocks of large language models also wrap each sublayer in a residual connection.[11] They also make each layer's job easier: the layer only needs to learn a small correction to the identity map, not the whole transformation.

A skip connection need not skip just one layer. Examples include ResNet blocks, long U-Net connections, and DenseNet-style dense connections.[12]

[11] Wikipedia: Residual neural network. Wikipedia: Transformer (deep learning architecture). Wikipedia: GPT-3.

[12] Wikipedia: Residual neural network, which also covers DenseNet. Wikipedia: U-Net. Yu, Wang, Shelhamer, and Darrell, *Deep Layer Aggregation*.

7.4 Exercises

Exercise (C) 7.4.1. What is a skip connection? Why are skip connections used in deep neural networks?

Exercise (C) 7.4.2. What is dropout, and why is it used?

Exercise (C) 7.4.3. What is DropConnect, and how does it differ from dropout?

Exercise (C) 7.4.4. What does it mean to freeze a layer? Why can freezing be useful in transfer learning?

Exercise (C) 7.4.5. What is early stopping, and how does it help prevent overfitting?

Exercise (C) 7.4.6. What is weight decay, and what effect does it have on the fitted weights?

Exercise (C) 7.4.7. What is the vanishing-gradient problem?

Exercise (C) 7.4.8. What is the difference between a residual connection and a general skip connection?

Exercise (C) 7.4.9. What is inverted dropout? Why use it?

Exercise (C) 7.4.10. What is fine-tuning in transfer learning?

Exercise (C) 7.4.11. What is a multilayer perceptron?

Exercise (T) 7.4.12. Carry out backpropagation for the two-layer, single-neuron network

$$z_1 = a_1 x + b_1, \qquad h_1 = \phi(z_1), \qquad z_2 = a_2 h_1 + b_2, \qquad \hat{y} = \sigma(z_2),$$

with squared loss $\ell = (y - \hat{y})^2$. (Hint, derive the partial derivatives of ℓ with respect to a_1, b_1, a_2, b_2, using the forward-pass values $z_1, h_1, z_2, \hat{y}$.)

Exercise (T) 7.4.13. Carry out backpropagation for the two-layer, single-neuron network

$$z_1 = a_1 x + b_1, \qquad h_1 = \phi(z_1), \qquad z_2 = a_2 h_1 + b_2, \qquad \hat{y} = \sigma(z_2),$$

with the binary cross-entropy (5.1.5) as loss,

$$\ell = -y \log \hat{y} - (1-y) \log(1-\hat{y}).$$

Exercise (T) 7.4.14. Carry out backpropagation for the two-layer, single-neuron network with a ReLU hidden activation and the identity as output activation, $\phi_2(z) = z$ and squared loss, so that

$$z_1 = a_1 x + b_1, \qquad h_1 = \mathrm{ReLU}(z_1), \qquad \hat{y} = z_2 = a_2 h_1 + b_2.$$

In what part of $\mathcal{X} = \mathbb{R}$ are the network parameters updated? Where not?

What check should you do on the training set $\mathcal{T}$ to prevent problems?

Exercise (T) 7.4.15. The backward recursion of a network with L hidden layers of widths $q_1, \ldots, q_L$ and a one-neuron output layer $L+1$ reads

$$\delta_i = (\delta_{i+1} A_{i+1}) \odot \phi_i'(z_i), \qquad i = L, \ldots, 1.$$

a. Specify the dimensions of δ_i, δ_{i+1}, A_{i+1}, z_i and $\phi_i'(z_i)$.

b. Check that both products are well-defined.

c. What do these dimensions become in the first step, $i = L$?

Chapter 8

Generative Pre-trained Transformers (GPT)

This chapter explains a minimal *Generative Pre-trained Transformer* (GPT). The aim is to understand the basic architecture of the neural networks behind systems such as ChatGPT, Claude, and Codex. From the outside these systems look like machines that read text and write useful text back. Here we look at the simpler mathematical question underneath: how can a neural network take a text context and predict a reasonable next token?

A *large language model (LLM)* is the broad name for such a model when trained on very large amounts of text. A *GPT* is just a particular kind of LLM.

The preceding chapters already introduced the basic neural-network ingredients: vectors, matrix layers, nonlinearities, residual connections, losses, and gradient descent. In this chapter we arrange those ingredients into a minimal GPT. We do not touch on the engineering details that become important to improve speed, computational cost, memory usage, and quality. For this, a useful concrete starting point is Karpathy's Python implementation

8.1 Interacting with a GPT

From the user's point of view, an LLM is a machine that we interact with by text, for instance by typing in a webbrowser. Once done typing, we press enter, wait a little, and the machine writes its response on the screen, often word by word. After a while it stops and waits for new input. How does this work?

At a high level, this is what happens; below we discuss the details. The machine has a *vocabulary* of *tokens*.[1] Most tokens stand for ordinary pieces of text, such as words or parts of words, but some tokens are special such as *end of sequence* EOS to mark that the generation of text should stop.

[1] This is much like a dictionary, except that the entries are text pieces rather than only whole words.

After the user typed text and pressed enter, the machine follows these steps:

1. Collect the current context, which consists of the new user text and (when present) preceding text from the conversation, and turn that with the vocabulary into a sequence of token ids.

2. Embed each token as a vector in $\mathbb{R}^d$ and add a vector in $\mathbb{R}^d$ that encodes the token's position in the sequence. Stacking N such vectors gives a matrix $X \in \mathbb{R}^{N\times d}$ with one row per token.

3. Pass X through L identically structured *transformer blocks*.[2] Each block first lets the tokens exchange information through *self-attention*, then transforms each token on its own through a small *feed-forward network*.

4. Multiply the output of step 3 by an output matrix to obtain, for each position, one *score* (logit) per vocabulary token, and apply *softmax* to turn the scores at the last position into a probability mass function (pmf) on the token vocabulary.

5. Sample from this pmf to select a next token.

6. Use the vocabulary in the reverse direction to detokenize this token into a text piece, and show that piece to the user.[3]

7. Append the new token to the token sequence.[4]

8. Repeat steps 2–7,[5] until the selected token happens to be EOS, which stops the token generation.

9. Wait for new input. Return to step 1 with the text generated so far and the new input of the user.

[2] In essence this is just a neural network: a pipeline of matrix multiplications with simple nonlinear steps in between.

[3] Implementations separate the two steps: the sampled tokens are only appended, and the detokenization is applied once, after EOS, to all tokens added to X since the last input of the user. The text is the same, except that a token can end in the middle of a character, so a piece-by-piece detokenizer has to wait for the remaining bytes.

[4] Such a model is called *autoregressive* as it writes one token after the other based on a window of previous tokens.

[5] So X is rebuilt from the extended sequence, generating one more token each round.

The *magic of a GPT is that, once trained, it produces from the context X high-quality pmfs over the token vocabulary, so that the resulting text makes sense to us*. That is all: nothing more, but certainly nothing less.

Training adjusts all the matrices involved in this process so that the predicted next-token pmfs match the tokens in a *corpus*, that is, a large collection of training text.[6]

We now explain the concepts one by one in detail.

[6] Training text can include books, articles, web pages, documentation, source code, forum posts, and other digitized text.

8.2 Representing text

8.2.1 Tokenizers

A language model does not read whole sentences at once. It reads a sequence of small pieces called *tokens*. The *vocabulary* with v entries contains all allowed tokens. For a character-level model of English the vocabulary is small: the letters a–z, a space, and a few punctuation marks. The sentence `the dog` then becomes the token sequence `t`, `h`, `e`, space, `d`, `o`, `g`. In modern GPTs, a token is usually a larger text fragment, such as a common word, a word part, or a space attached to words; such vocabularies contain on the order of $v \approx 10^5$ tokens.

A tokenizer is the fixed procedure that performs the mapping from real text to tokens. For example, a *longest-match tokenizer* starts at the first character of the text, looks for all vocabulary tokens that match the text starting at that position, chooses the longest matching token, moves forward by the length of that token, and repeats until the whole text has been tokenized. Thus, for the word `adds`, if the vocabulary contains both

`add` and `adds`, the tokenizer chooses `adds`. If the vocabulary contains `add` but not `adds`, it chooses `add` and then tokenizes the remaining `s`.

For a GPT-style *byte-pair encoding (BPE)* tokenizer, the vocabulary is built together with a fixed merge table. The merge table says which adjacent text pieces should be joined first, second, third, and so on.[1] BPE tokenizes a text as follows.

1. Start by splitting the text into very small units, usually bytes or characters.
2. Among all adjacent pairs, merge the pair that comes first in the fixed merge table.
3. Repeat step 2 until no listed merge applies.
4. Replace the final pieces by their token ids.

A tokenizer does not correct typos. It maps the text it receives to tokens. If a misspelled word is not in the vocabulary as one token, the tokenizer splits it into smaller known pieces, possibly down to bytes or characters. Thus, as the typed word `addds` is not in the vocabulary, it can be split into pieces such as `add`, `d`, and `s`. The neural network downstream has to handle typos.

The same vocabulary is also used in the reverse direction. Each time the GPT has selected a token id, the tokenizer looks up this id in the token vocabulary, retrieves the corresponding text piece, and writes that piece to the output text. This reverse step is called *detokenization*. For example, for a character-level tokenizer it is just joining characters; for BPE it is more complicated, but still the same idea.

A GPT cannot process an arbitrarily long text, such as a book, at once. After tokenization, the sequence of token ids must fit inside the model's *context window*, whose maximum length is $N_{\max}$. If a document is too long, surrounding software may split it into chunks or compact parts of the text. Thus the model itself only sees the tokens that are placed in its current context window. Similarly, when a conversation becomes too long, the software may drop older tokens.

8.2.2 Embeddings

As tokens are just symbols, they have to be converted to numbers so that the neural network[2] downstream can act on them. For this, the GPT maps each token in the vocabulary to a vector, called its *embedding*. Stacking the embeddings of all tokens in the vocabulary gives the *embedding matrix*

$$E \in \mathbb{R}^{v \times d}.$$

Here v is the size of the vocabulary and d is the chosen embedding dimension.[3] So to find the vector for a token that occurs in the text, we look up that token's position in the vocabulary and select the matching row of E.

Once tokens are vectors we can ask how *similar* two of them are. In a word-level model the tokens `cat` and `dog` probably end up with nearby vectors during training, because they occur in similar contexts (`the ___ slept`, `feed the ___`), whereas `dog` and `quantum` would probably land

[1] Both the vocabulary and the merge table are learned from training text.

[2] Recall, neural networks are just matrix multiplications on vectors with non-linear operations in between.

[3] Don't get confused. Row i of E is the embedding of the i-th token *of the vocabulary*; it is *not* the i-th token of the sentence being read.

[4] In view of Schr\"odinger's cat, `cat` and `quantum` might end up close. 'Closeness' is a complicated concept.

far apart.[4] The natural measure of this geometric closeness is the inner product. Write E_i for the embedding of the i-th vocabulary token, that is, row i of E. When the inner product $\langle E_i, E_j \rangle$ is large and positive, the two vectors point in a similar direction; when it is near zero, they are unrelated. Thus the geometry of the vectors mirrors how the tokens are actually used. This matters because the model computes only with these vectors, through inner products and matrix multiplications: if related tokens have related embeddings, whatever the model learns for one token carries over to its neighbors.

A crude way to embed tokens would be to use one-hot encoding[5] for each token. That forces the embedding dimension to equal the vocabulary size, $d = v$, and makes every two distinct tokens orthogonal.[6] Thus, it records only that two tokens differ, but not *how much* they differ. A learned embedding with $d \ll v$ is both far smaller and, unlike one-hot vectors, able to place related tokens near each other.[7]

[5] See Chapter 3.

[6] Since the inner product between different one-hot encoded vectors is 0.

[7] In the sense of the inner product.

Observe that the embedding matrix E is *not* chosen by hand. Initially these entries are random, say $E_{ij} \sim \text{Norm}(0,1)$. Every entry of E is then *learned*: during *training* the model gradually adjusts these numbers so that tokens used in similar ways end up with similar vectors. Nobody decides in advance what an embedding of a token should be: the model discovers it. At the end of the chapter we discuss how this adjustment works.

Once each single token has an embedding, we represent a whole sequence of N tokens, for instance the characters of the sentence read so far, by stacking their embedding vectors as N rows in the *context matrix*

$$X \in \mathbb{R}^{N \times d},$$

that is, one token embedding vector per row. Here N is the length of the current token sequence. It cannot exceed the model's maximum context length $N_{\max}$.

Below we will see that from here on almost everything the model does is to multiply X by learned matrices and apply simple non-linear mappings.

There is an important aspect to the token sequences that we did not yet include: the role of the position of tokens. In `the cat bit the dog.` and `the dog bit the cat.`, the animal that suffers is different. However, in token embedding, the same symbol always receives the same token vector. Thus, the embedding matrix E gives the model no information about *where the token stands in the sequence.* We need to add this information with a second, position-dependent d-dimensional vector. The position matrix has one row per possible position,

$$P \in \mathbb{R}^{N_{\max} \times d}.$$

If the token id at sequence position i is t_i, then the i-th row of the context matrix is

$$x_i = E_{t_i} + P_i.$$

Here E_{t_i} is the embedding of token t_i, and P_i is the embedding of position i. Stacking these rows gives the final form of the context matrix X.

A very naive fixed encoding for position, such as $P_i = (i, 0, \ldots, 0)$, would tell the model the position, but it is a poor choice. The position signal grows with i, can dominate the token embedding, and also uses only one coordinate to carry all position information. Practical position encodings therefore either learn the rows of P from data, or use some fixed vectors.[8]

[8] Wikipedia: Transformer (deep learning).

8.3 Predicting the next token

8.3.1 Next-token selection

Recall that the language model predicts the next token from the current context matrix X, whose rows represent the tokens that fit in the context window. For this, it produces a pmf on the vocabulary. After the tokens `the do`, a good character-level model puts high probability on `g` (making `dog`), a little on `e` (`doe`), and almost none on `z`.

To get this pmf the model takes X as input and computes for each token in the vocabulary a plain real number called a *score* (or *logit*). Let $s = (s_1, \ldots, s_v)$ be the vector of scores, where s_i rates how well the i-th *vocabulary* token fits that one slot.[1]

[1] Again, the index i runs over the fixed vocabulary, not over the token positions of the sentence.

Because a score can be any real number, the score vector $s \in \mathbb{R}^v$ is turned into a pmf $p \in \mathbb{R}^v$ by the softmax function (5.2.10): $p = \tilde{\sigma}(s)$. During generation, implementations often divide the score vector by a *temperature* $\tau > 0$ before applying softmax:

$$p = \tilde{\sigma}(s/\tau).$$

A small τ concentrates probability near the largest scores and makes the output more deterministic; a large τ flattens the pmf and makes the output more variable.[2] Good values of τ are found by testing.

[2] Temperature is not the only knob. Implementations often also restrict sampling to the k most probable tokens (*top-k* sampling), or to the smallest set of tokens whose total probability exceeds a threshold (*nucleus* sampling), to prevent the occasional selection of a very unlikely token.

ONE PART OF THE computation of s is the same in every model we discuss below. The network condenses the context into a single summary vector $h \in \mathbb{R}^d$, and then maps this vector to the scores by a learned matrix,[3]

$$W_{\text{out}} \in \mathbb{R}^{d \times v}, \tag{8.3.1}$$

so that[4]

$$s = hW_{\text{out}} \in \mathbb{R}^v. \tag{8.3.2}$$

[3] Many GPTs do not learn a separate W_{out}, but reuse the embedding matrix by setting $W_{\text{out}} = E^\top$. This is called *weight tying*: it saves parameters and couples unembedding to embedding.

[4] Here we use the standard ML convention that a data matrix stores one observation per row. Thus each token vector is a row vector, and a linear layer multiplies on the right: hW_{out}.

The matrix W_{out} is often called the *unembedding* matrix.[5]

[5] Where the embedding matrix E maps a token into the model's vector space, W_{out} maps a vector back to scores on the vocabulary.

Thus, the problem is no longer how to use the score vector s; the real problem is how to compute a good summary vector h from the whole context in X. Answering this is the next topic.

8.3.2 Attention

Before GPTs and transformers, neural language models often used *recurrent neural networks (RNNs)* to compute the score vector s from a hidden vector h. Just as a GPT, an RNN starts from the context matrix X. However, instead of using the entire X, it reads the rows $x_1, \ldots, x_N$ of X from

top to bottom, in the order of the token sequence. It keeps *hidden vectors* h_i updated recursively according to the rule

$$h_i = F(h_{i-1}, x_i), \qquad i = 1, \dots, N, \tag{8.3.3}$$

with $h_0 = 0$. Here F denotes the *recurrent cell*[6] that combines the previous hidden vector h_{i-1} with the current token vector x_i. After the last token, the final hidden state h_N serves as the summary vector for (8.3.2).

[6] It is called a cell because the same computational unit is reused at each position in the sequence. The cell F is a small neural network, with learned matrices, biases, and nonlinearities. For instance, a simple RNN cell could use $F(h_{i-1}, x_i) = \tanh(h_{i-1}W_h + x_iW_x + b)$.

This recurrent approach has two weaknesses. First, all information from the earlier tokens has to pass through a *single* hidden state h_N; for long sequences, important information can be lost in the repeated updating of this hidden vector. This is a serious problem when dealing with *long-range dependencies* such as understanding text, because conceptually and grammatically related words are not always nearby in a sentence. For instance, in `the dog that chased the cat ran`, the word `ran` belongs to `dog`, not to `cat`, even though `cat` lies closer in the sentence. Second, by (8.3.3), the rows of X are processed one after another:

$$\begin{aligned} h_N &= F(h_{N-1}, x_N), \\ h_{N-1} &= F(h_{N-2}, x_{N-1}), \\ &\vdots \end{aligned}$$

Thus the computation of h_N must wait for h_{N-1}, which must wait for h_{N-2}, and so on. This serial dependence precludes evaluation of all positions at the same time by large parallel matrix operations on a GPU, hence slowing down the computations.

A BETTER DESIGN should have a token look directly at the other tokens in the current context and learn which ones are relevant. Since the rows of X are vectors, the first thing one might try is the matrix $XX^\top$. Its (i, j) entry is the inner product $\langle x_i, x_j \rangle$. This gives a relation score between position i and position j, but it has a fundamental problem: as the inner product is symmetric, the score is symmetric,

$$\langle x_i, x_j \rangle = \langle x_j, x_i \rangle .$$

Thus how much token i attends to token j would always equal how much token j attends to token i. We do not want this. Attention[7] is about directional relations *between* the tokens *in the sequence*: the role of token j for understanding token i need not be the same as the role of token i for understanding token j. Moreover, attention is not just ordinary similarity. In `the dog ... ran`, the word `ran` must attend to `dog`, yet `ran` and `dog` are not similar tokens, so their embeddings need not lie close at all. We therefore need to *transform* the rows of X before using inner products as relation scores.

[7] Wikipedia: Attention Is All You Need.

Suppose we transform the matrix X with a matrix U and score token i against token j by the inner product of x_iU and x_jU, that is $\langle x_iU, x_jU \rangle$. But notice that $UU^\top$ is symmetric, so that the score of i with j under this transformation is always equal to the score of j with i:

$$\langle x_iU, x_jU \rangle = \langle x_i, x_jUU^\top \rangle = \langle x_jUU^\top, x_i \rangle = \langle x_jU, x_iU \rangle$$

Consequently, this type of transformation still does not capture sequential relations.

Thus, instead of using one matrix U to transform X, the model builds two vectors for each token by multiplying X with two different $d \times d$ matrices W_Q and W_K,

$$Q = XW_Q, \qquad K = XW_K;$$

and Q and K are again $N \times d$. With two separate matrices the scores are generally not equal because, in general,

$$\langle x_i W_Q, x_j W_K \rangle \neq \langle x_j W_Q, x_i W_K \rangle .$$

The ith row of Q is called the *query* of token i, and the jth row of K is called the *key* of token j. The score $\langle x_i W_Q, x_j W_K \rangle$ quantifies how much token i attends to token j, because, when this inner product is large, the transformed vector for token i is aligned with the transformed vector for token j. The important point here is that using two matrices W_Q and W_K offers the asymmetry[8] that enables the query of `ran` to point toward the key of `dog` while the two embeddings of these words can stay far apart. Note that, like the embedding matrix E, the matrices W_Q and W_K are not set by hand; they are learned from enormous amounts of text.

The queries and the keys are both built from the *same* sequence X, so every token is compared against every other token of the *same input*. This is called *self-attention* and is used throughout GPTs. Self-attention matters because the context that counts for the next token is rarely the nearest token; it may lie many tokens back, as `dog` does for `ran`. By letting every token look at every other and *learn* which ones matter, the model captures such long-range links directly. We remark that this only works because the rows of X include position information. Without position embeddings, self-attention could treat the input essentially as an unordered set: permuting the token rows would merely permute the output rows in the same way.[9]

We use $QK^\top$ to form the *attention matrix*

$$A = \tilde{\sigma}\left(\frac{QK^\top}{\sqrt{d}} + M\right),$$

where the softmax is applied row by row, so that each row of A sums to one. This form needs two comments. First, each element of $QK^\top$ is an inner product of two vectors with d components, hence a sum of d products. The more components, the larger such a sum tends to be. To keep the scores in a sensible range, and to prevent the softmax in the next step from reacting too sharply, we divide every entry of $QK^\top$ by $\sqrt{d}$.[10] Second, a GPT may use only the current and earlier positions. When predicting from position i, it must not use positions $j > i$, because those are future tokens. We enforce this with a *causal mask matrix* M where $M_{ij} = -\infty \mathbb{1}\{j > i\}$. Adding M before the softmax makes the softmax weight of every future position equal to zero. The mask also explains why GPT training can be parallel over positions. Since position i never sees tokens after i, the model can compute predictions for all N positions in one forward pass without leaking the answers.[11] This is the key computational difference with the recurrent model, which must process the positions one after another.

[8] The asymmetry could also be obtained with one matrix, for instance through a score like $\langle x_i, x_j W \rangle$. The separate matrices W_Q and W_K are used because they give this convenient query–key interpretation.

[9] Strictly speaking this holds for unmasked self-attention; the causal mask introduced below also depends on position, because it tells each token which later positions are forbidden. But the mask only gives this before/after restriction. The richer information about where a token sits in the sequence comes from the position embeddings.

[10] Why $\sqrt{d}$ and not d? If the entries of the query and the key are uncorrelated with mean 0 and variance 1, their inner product is a sum of d such products, and so has mean 0 and variance d, hence standard deviation $\sqrt{d}$.

[11] Here *leaking* means that, when predicting the token after position i, the model would be allowed to use that same future token as input. That would make the training task meaningless.

To see what A does, first ignore attention. In a regular neural-network layer, we transform the rows of X by a learned matrix W_V:

$$V = XW_V,$$

which, in transformer terminology, is called the *value matrix*. Then V would be passed through a nonlinearity and on to the next layer. The problem is that this operation does not by itself mix information between tokens. For that purpose, we built the attention matrix A. Instead of passing on V, the attention head passes on the *head output*

$$H = AV = AXW_V \in \mathbb{R}^{N\times d}.$$

On a personal note: I would assess the attention matrix A as the real novelty here. As for the fancy query–key–value terminology itself, I don't think one should attribute too much of a meaning to it, despite the evocative naming. At the time of writing,[12] the success of attention in language models is just an empirical fact. The meaning of text, or information more generally, has not (yet) been explained in query–key–value terms.

[12] 2026

8.3.3 A one-head GPT

The construction above makes one *attention head*: it consists of one set of matrices W_Q, W_K, W_V combined in an attention matrix A, and one head matrix $H = AV$. In principle, one such head can already serve as a minimal *one-head GPT*.

Read Fig. 8.1 from bottom to top. The lower box, Make X, starts with the text from the conversation. It tokenizes this text with the vocabulary, looks up the corresponding rows of E, adds the position embeddings from P, and produces the context matrix X. The attention-head box then takes X as input. It forms $Q = XW_Q$, $K = XW_K$, and $V = XW_V$, builds the attention matrix A, and produces the head output $H = AV$. The next-token-selection box uses the last row $H_N \in \mathbb{R}^d$ of H, because this row corresponds to the end of the current context. With (8.3.2) we compute the score s, using H_N as the summary vector h. From there, the remaining steps are exactly those of the list in Section 8.1.

8.3.4 From one head to a full decoder-only GPT

Fig. 8.2 shows how the one-head GPT is extended to a single transformer block with several heads. The input side is now collapsed to the single box X, and the next-token-selection box is also collapsed: their internal details are exactly the ones from Fig. 8.1. The new part is the transformer in between.

The input X is passed through *layer normalization*[13] before it enters the attention heads. Write

$$Z = \mathrm{LN}(X),$$

where

$$z_{ik} = \gamma_k \frac{x_{ik} - \bar{x}_i}{\sigma_i} + \beta_k, \qquad \bar{x}_i = \frac{1}{d}\sum_{k=1}^{d} x_{ik}, \qquad \sigma_i^2 = \frac{1}{d}\sum_{k=1}^{d} (x_{ik} - \bar{x}_i)^2.$$

[13] Wikipedia: Normalization (machine learning).

[14] Layer normalization is less important in a single-layer toy GPT than in a real GPT. Real GPTs stack many transformer blocks, so repeated transformations can make scales drift; normalization keeps the row vectors on a comparable scale.

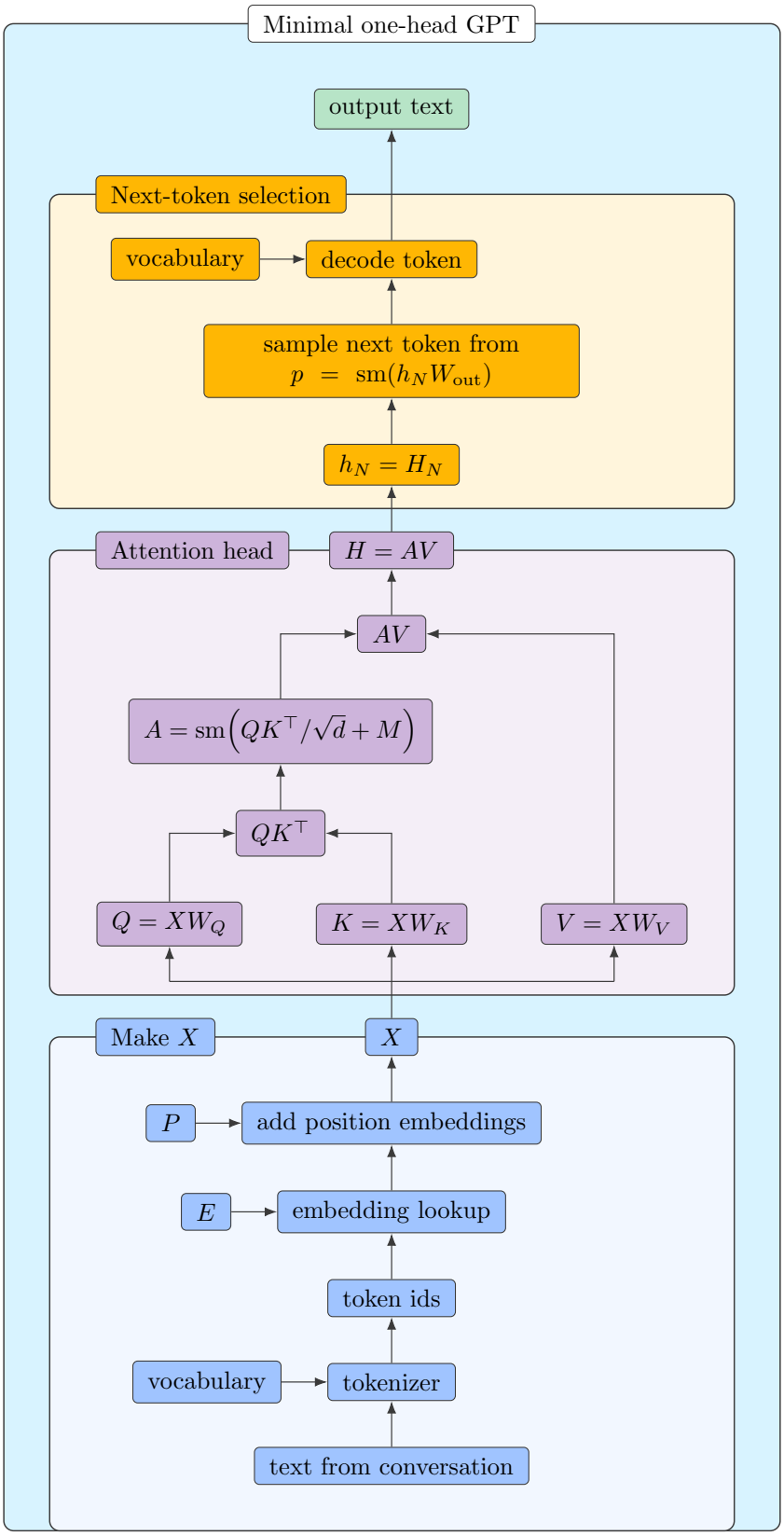


FIG. 8.1: A minimal one-head GPT. The lower block constructs X, the middle block computes $H = AV$, and the upper block selects and decodes one next token.

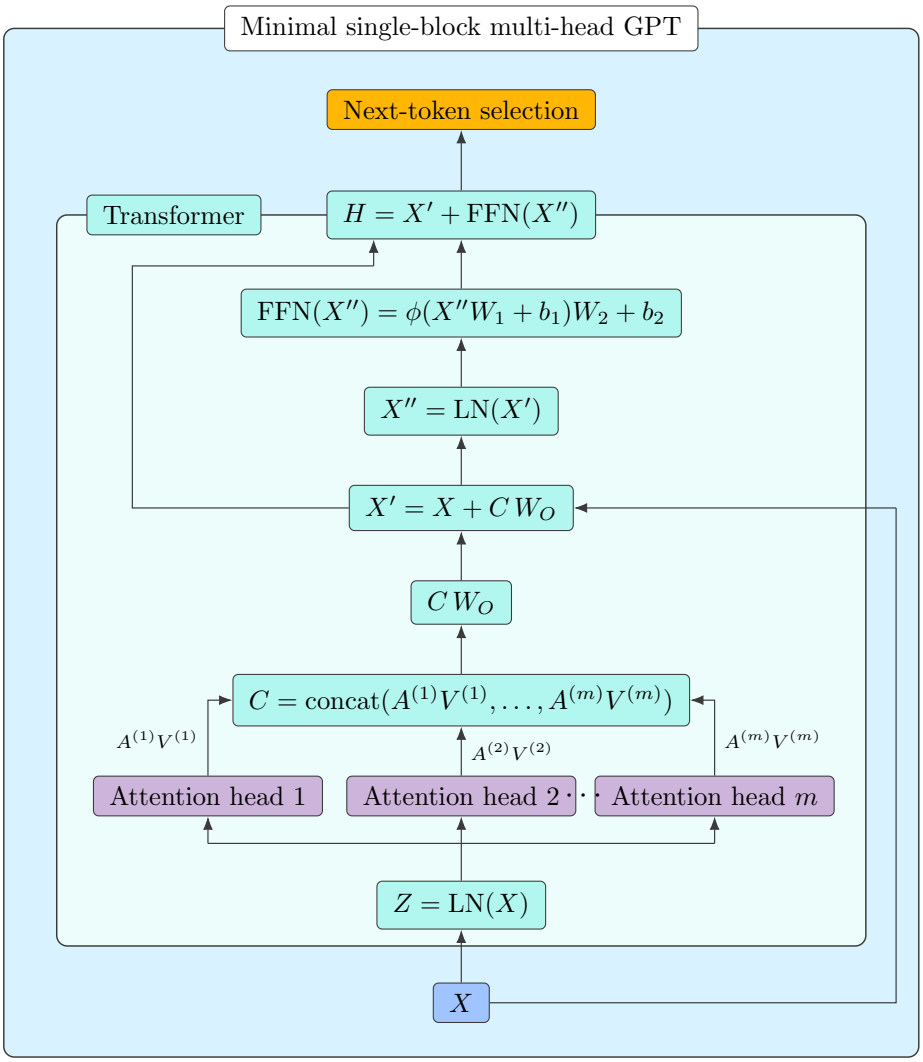


FIG. 8.2: A single-block multi-head GPT. The input construction is collapsed to X, and next-token selection is collapsed as in Fig. 8.1; the transformer expands the multi-head attention, residual, layer-normalization, and feed-forward steps.

Thus, besides the standard normalization, transformers[14] usually include the learned parameters γ_k and β_k to make the normalization less rigid: if coordinate k should be larger, smaller, or shifted after standardization, training can pick this up.

Why do this before attention?[15] Attention scores are built from inner products. If the row scales drift after several transformations, then an inner product can become large because a row has a large norm, not because the tokens are especially relevant to each other. With this scaling, attention heads see vectors on a comparable scale.

After layer normalization, Z is sent to m attention heads in parallel.[16] Empirically, GPTs perform better when several such heads are used. Heuristically, since each head has its own matrices, each head is free to pick up different kinds of relation between tokens.[17] In head r, the model forms its own queries, keys, and values,

$$Q^{(r)} = ZW_Q^{(r)}, \qquad K^{(r)} = ZW_K^{(r)}, \qquad V^{(r)} = ZW_V^{(r)}.$$

It then computes its own causal attention matrix $A^{(r)}$[18] and produces $A^{(r)}V^{(r)}$.

As the m heads have produced m separate answers for each token position, the head outputs are concatenated:

$$C = \mathrm{concat}(A^{(1)}V^{(1)}, \ldots, A^{(m)}V^{(m)}).$$

We want C to be an $N \times d$ matrix, because those dimensions are expected by the rest of the model. Therefore the learned matrices of each head have dimensions

$$W_Q^{(r)}, W_K^{(r)}, W_V^{(r)} \in \mathbb{R}^{d \times d_h},$$

where $d_h = d/m$.[19] After concatenation, the model mixes the outputs of the different heads by the learned linear map CW_O.

As in neural networks, a skip connection adds the attention update to the current representation:

$$X' = X + CW_O.$$

The attention block therefore learns a change to X, not a complete replacement of X.

We remark in passing that the sum $X + CW_O$ is not normalized: under the pre-norm convention adopted above, normalization is applied to the *input* of each sublayer, never to the residual sum itself.

The matrix X' is normalized again and sent through a feed-forward network:[20]

$$X'' = \mathrm{LN}(X'), \qquad \mathrm{FFN}(X'') = \phi(X''W_1 + b_1)W_2 + b_2.$$

This is the ordinary neural-network part of the transformer block: first an affine transformation $X''W_1 + b_1$, then a componentwise nonlinearity ϕ,[21] and then another affine transformation. The attention step and the feed-forward step therefore do different jobs. Attention mixes information between token positions: each row can use information from earlier rows. The feed-forward network works row by row, so each token position gets a nonlinear transformation of the information that

[15] Placing the normalization *before* each sublayer, as we do, is the *pre-norm* convention of GPT-2 and later models. The original Transformer instead normalized *after* the residual addition (*post-norm*). Both appear in the literature; pre-norm turns out to be easier to train for deep stacks.

[16] This is another reason not to assign much meaning to the term *value matrix*. If the value matrix were capable of selecting what is "of value" in a text, then a single attention head would suffice.

[17] One sometimes reads explanations in human terms, such as: one head attends to emotion, another to meaning. In my opinion, this is misleading. We do not know in such simple terms what individual heads learn. What we do know is that the trained GPT, as a whole, can produce text or computer code that is helpful to us.

[18] With one change: the scaling inside the softmax is now $\sqrt{d_h}$ instead of $\sqrt{d}$, because the per-head queries and keys have d_h components.

[19] In standard implementations d is chosen as a multiple of m, so that all heads can have the same integer width $d_h = d/m$.

[20] This step was omitted in the one-head GPT because that example was meant to show the smallest attention-based prediction mechanism. That model still contains nonlinear operations, for instance the softmax in attention and the final softmax that turns token scores into a pmf, but it does not contain this extra learned row-wise neural-network layer.

[21] GPT uses the GELU.

attention has collected. It is common to expand each row temporarily to dimension $4d$: then $W_1 \in \mathbb{R}^{d\times 4d}$ and $W_2 \in \mathbb{R}^{4d\times d}$. The first matrix gives the nonlinearity ϕ more coordinates[22] to act on; the second matrix maps back to dimension d.

[22] Empirically this larger intermediate dimension turns out to work well in transformer architectures.

Finally, the feed-forward update is added back through a residual connection:

$$H = X' + \mathrm{FFN}(X'').$$

The single output matrix $H \in \mathbb{R}^{N\times d}$ is now ready for the same next-token-selection box used in the one-head GPT.

The full model applies L such transformer blocks[23] in turn, each with its own matrices, as shown in Fig. 8.3. We write $H^{(0)} = X$. Transformer block l takes $H^{(l-1)}$ as input and returns $H^{(l)}$. After the last block, GPTs usually apply one more layer normalization,

[23] These are the layers of neural networks.

$$H = \mathrm{LN}\big(H^{(L)}\big).$$

This final matrix H is sent to the same next-token-selection box used in the one-head GPT.

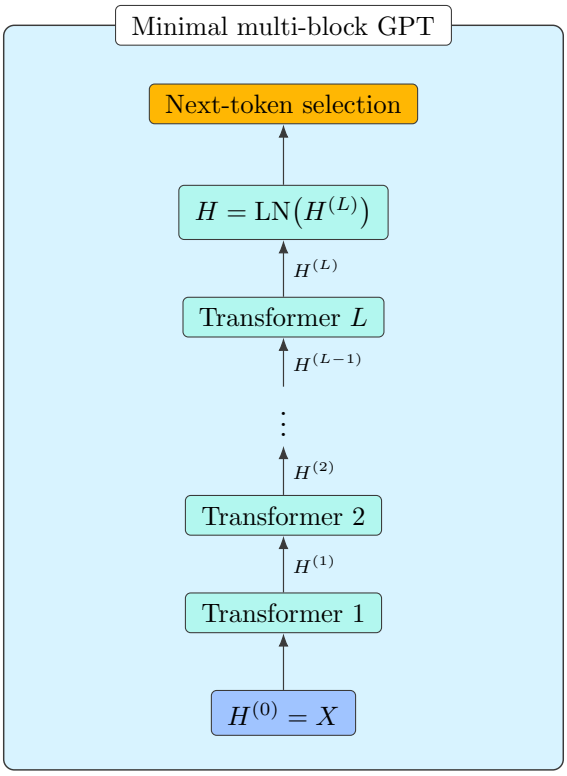


FIG. 8.3: A minimal multi-block GPT. The stack starts with $H^{(0)} = X$; after L transformer blocks, a final layer normalization produces H for next-token selection.

Now that the architecture is complete, one remark on the word *transformer* is in order. The original transformer of Vaswani et al.[24] was built for machine translation and consists of two coupled stacks of blocks, one for the source language, the other for the target language. A GPT has no separate source text to encode, so it needs no encoder. For this reason the GPT architecture is called *decoder-only*.

[24] Wikipedia: Attention Is All You Need.

8.4 Training

We have introduced several matrices. The embedding matrix E, the position matrix P, and the unembedding matrix W_{out} from (8.3.1) belong to the input and output sides of the GPT. In addition, each transformer block has its own head matrices $W_Q^{(r)}$, $W_K^{(r)}$, $W_V^{(r)}$ for $r = 1, \dots, m$, its own output matrix W_O, and its own feed-forward matrices W_1 and W_2. It remains to explain in some detail how their entries are *learned*.

Training starts with filling the learned matrices with random numbers, for instance $\sim \text{Norm}(0, 1)$. At this point the GPT has no useful relation to language.

The model is then shown a large amount of real text.[1] Training does not use this text all at once. In one training step we pick a mini-batch $\mathcal{B}$ of training sequences from the corpus. Write these sequences as

$$(t_{b,1}, \dots, t_{b,N+1}), \qquad b \in \mathcal{B}.$$

For ease we assume that all sequences have the same length $N+1$. For one sequence b, the first N tokens form the input context. The GPT constructs X_b, applies the transformer stack, and obtains $H_b \in \mathbb{R}^{N \times d}$. From this matrix it computes the score matrix

$$S_b = H_b W_{\text{out}} \in \mathbb{R}^{N \times v}.$$

Thus, row $S_{b,i}$ contains the scores for the token id after position i for the sequence b. Because of the causal mask, these scores are based only on $t_{b,1}, \dots, t_{b,i}$, not on future tokens. Applying softmax row by row gives the predicted pmf

$$\hat{p}_{b,i} = \tilde{\sigma}(S_{b,i}) \in \mathbb{R}^v, \qquad i = 1, \dots, N.$$

The observed next token is $t_{b,i+1}$. Using the label-lifting map from (3.1.2), represent it by the one-hot vector $q_{b,i} = r(t_{b,i+1})$. The *cross-entropy* loss at this position is

$$\ell(q_{b,i}, \hat{p}_{b,i}) = -\sum_{k=1}^{v} q_{b,i,k} \log \hat{p}_{b,i,k} = -\log \hat{p}_{b,i,t_{b,i+1}}.$$

The last equality uses that $q_{b,i}$ is one-hot.[2] For sequence b, the empirical risk over its N positions is

$$\hat{R}_b = \frac{1}{N} \sum_{i=1}^{N} \ell(q_{b,i}, \hat{p}_{b,i}) = -\frac{1}{N} \sum_{i=1}^{N} \log \hat{p}_{b,i,t_{b,i+1}}.$$

The empirical risk of the mini-batch is the average over its $|\mathcal{B}|$ sequences:

$$\hat{R}_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{b \in \mathcal{B}} \hat{R}_b.$$

Backpropagation computes the gradient of this one number with respect to all parameters. The optimizer then updates the parameters once. That whole procedure is one training step. Training then consists of repeating many training steps.[3]

The training just described is called *pre-training*; it explains the P in GPT. For systems such as ChatGPT and Claude this is only the first stage in training: after pre-training on a large corpus, the model is trained

[1] Real GPTs also use *dropout* Chapter 7 during training: the outputs of the attention and the feed-forward steps are randomly thinned before each residual addition. During text generation, dropout is switched off.

[2] All components are zero except the component corresponding to the observed next token $t_{b,i+1}$.

[3] The number of steps depends on the model, the corpus, and the mini-batch size. Larger GPTs may use millions of optimizer steps.

further on smaller, curated datasets, so that it follows instructions and behaves as a helpful assistant. This second stage is called *fine-tuning* or *post-training*, and falls outside the scope of this chapter.

Above we already discussed transformers. It remains to explain what *generative* means in GPT. The trained model computes just one object: the pmf $p = \tilde{\sigma}(s)$ over the vocabulary for the position right after the current context. It writes text by using this pmf repeatedly: sample a token from p, append it to the context, recompute p from the extended context, and continue until EOS is drawn. These are steps 5–8 of Section 8.1. So the response is not produced in one piece; it is built one token at a time, and each token generated becomes part of the input used for the next.

The word *large* in large language model refers to the number of training parameters. A rough count explains the scale. Start with one transformer block. Each of the m heads has its query, key, and value matrices $W_Q^{(r)}, W_K^{(r)}, W_V^{(r)} \in \mathbb{R}^{d \times d_h}$, giving $3dd_h$ entries per head, and $3mdd_h$ over all m heads. As $d_h = d/m$, this amounts to $3d^2$ parameters. The output matrix $W_O \in \mathbb{R}^{d \times d}$ contributes another d^2. Thus the multi-head attention part has about $4d^2$ learned entries. The feed-forward network is wider; as $W_1 \in \mathbb{R}^{d \times 4d}$ and $W_2 \in \mathbb{R}^{4d \times d}$, they contribute $8d^2$ parameters. Therefore one transformer block has roughly $4d^2 + 8d^2 = 12d^2$ parameters.[4] With L transformer blocks, the stack contributes about $12Ld^2$ parameters.[5] For example, in GPT-3, $L = 96$ and $d = 12288$ which gives ≈ 174 billion parameters from the transformer stack alone.

[4] We ignore biases, the E and P matrices, the unembedding matrix W_{out}, and some other parameters.

[5] $L = 32, d = 4096$ for GPT-3 6.7B and $L = 96, d = 12288$ for GPT-3 175B are reported in Wikipedia: GPT-3.

8.5 Exercises

Exercise (C) 8.5.1. What is the role of the token vocabulary in a GPT?

Exercise (C) 8.5.2. Why does self-attention use separate query and key matrices?

Exercise (C) 8.5.3. What is an attention head?

Exercise (C) 8.5.4. Where in a GPT is the vocabulary used?

Exercise (C) 8.5.5. What is the relation between the vocabulary and the embedding matrix E?

Exercise (T) 8.5.6. Suppose $X \in \mathbb{R}^{N \times d}$ is processed by five attention heads. What is the dimension of the output of one head, assuming all heads have equal width?

Exercise (C) 8.5.7. Let t_i be the token id at sequence position i. Is the i-th row of the context matrix $x_i = E_{t_i} + P_i$ or $x_i = E_i + P_{t_i}$?

Exercise (T) 8.5.8. What is the dimension of the position matrix P? Relate it to the embedding matrix E and to the context matrix X. #+begin$_{\text{solution}}$ $P \in \mathbb{R}^{N_{\max} \times d}$: one row per position in the context window, each row a d-dimensional vector. It has the same number of columns as the embedding matrix $E \in \mathbb{R}^{v \times d}$, but its rows are indexed by position instead of by token id. For a sequence of length $N \le N_{\max}$, the first N rows of P are added to the corresponding rows of $X \in \mathbb{R}^{N \times d}$.

Exercise (C) 8.5.9. Which steps of the list in Section 8.1 make a GPT *generative*, and what turns these steps into a loop?

Exercise (C) 8.5.10. One training step draws a mini-batch $\mathcal{B}$ of sequences from the corpus. At which positions is the cross-entropy loss evaluated, and how is it used in gradient descent?

Exercise (C) 8.5.11. Explain the tokenizing process of a longest-match tokenizer. Give an example to demonstrate how it works.

Exercise (C) 8.5.12. Starting from the context matrix X, how does an attention head produce its head output H? Name all matrices that are used in this process.

Exercise (C) 8.5.13. Describe how the next-token selection works, starting from the output matrix H.

Exercise (C) 8.5.14. Explain how a GPT turns text into the context matrix X.

Exercise (T) 8.5.15. When there are m attention heads, what are the dimensions of the query matrix $W_Q^{(r)}$ of head r and of the queries $Q^{(r)}$ it produces?

Exercise (C) 8.5.16. Intuitively, why does a transformer block use several attention heads instead of one?

Appendix A

Helper Algorithms

The algorithms in this appendix are small auxiliary routines used by the tree, bagging, and classification algorithms in Chapter 4.

Algorithm A.0.1 Compute the average of a finite list of numbers in $\mathcal{W}$.

Input: a non-empty list $\mathcal{W} = [w_1, w_2, \ldots, w_n]$.
Output: the average.
procedure AVERAGE($\mathcal{W}$)
    $total \leftarrow 0$
    $count \leftarrow 0$
    for $w \in \mathcal{W}$ **do**
        $total \leftarrow total + w$
        $count \leftarrow count + 1$
    return $total / count$

Algorithm A.0.2 Count how often each value occurs in a finite list.

Input: a finite list $\mathcal{W}$.
Output: a map from values to counts.
procedure COUNTS($\mathcal{W}$)
    $counts \leftarrow$ empty map with default value 0
    for $w \in \mathcal{W}$ **do**
        $counts[w] \leftarrow counts[w] + 1$
    return *counts*

Algorithm A.0.3 Count the number of elements in each set of a map.

Input: a map d whose values are finite sets.
Output: a map from keys of d to set sizes.
procedure COUNTINGMEASURE(d)
    $counts \leftarrow$ empty map
    for $k \in$ KEYS(d) **do**
        $counts[k] \leftarrow |d[k]|$
    return *counts*

Algorithm A.0.4 Normalize a finite map of nonnegative weights.

Input: a map p with nonnegative values.
Output: a probability map.
procedure NORMALIZEPROBABILITIES(p)
    $total \leftarrow \sum_{k \in \text{KEYS}(p)} p[k]$
    for $k \in \text{KEYS}(p)$ **do**
        $p[k] \leftarrow p[k] / total$
    return p

Algorithm A.0.5 Compute uniform probabilities for the sets stored in a map.

Input: a map d whose values are finite sets.
Output: a probability map.
procedure UNIFORMPROBABILITIES(d)
    return NORMALIZEPROBABILITIES(COUNTINGMEASURE(d))

Algorithm A.0.6 Compute the entropy of a probability mass function.

Input: a finite collection p of probabilities.
Output: the entropy.
procedure ENTROPY(p)
    $h \leftarrow 0$
    for $x \in p$ **do**
        if $x > 0$ **then**
            $h \leftarrow h - x \log_2 x$
    return h

Algorithm A.0.7 Collect all values that occur in a collection of sets.

Input: a finite collection $\mathcal{C}$ of sets.
Output: the union of the sets in $\mathcal{C}$.
procedure UNIQUEVALUES($\mathcal{C}$)
    $\mathcal{U} \leftarrow \varnothing$
    for $\mathcal{A} \in \mathcal{C}$ **do**
        $\mathcal{U} \leftarrow \mathcal{U} \cup \mathcal{A}$
    return $\mathcal{U}$

Algorithm A.0.8 Check whether a collection of sets is a partition of a state space.

Input: a finite collection $\mathcal{C}$ of sets and a state space $\mathcal{S}$.
Output: a Boolean.
procedure ?PARTITION($\mathcal{C}, \mathcal{S}$)
    if UNIQUEVALUES($\mathcal{C}$) $\neq \mathcal{S}$ **then**
        return FALSE
    seen $\leftarrow \varnothing$
    for $\mathcal{A} \in \mathcal{C}$ **do**
        if $\mathcal{A} \cap$ *seen* $\neq \varnothing$ **then**
            return FALSE
        seen $\leftarrow$ *seen* $\cup \mathcal{A}$
    return TRUE

Algorithm A.0.9 Sample at most n distinct elements from a finite set.

Input: a finite set $\mathcal{S}$ and a number n.
Output: a set with at most n randomly selected elements of $\mathcal{S}$.
procedure SAMPLEWITHOUTREPLACEMENT($\mathcal{S}, n$)
    $\mathcal{S}' \leftarrow \varnothing$
    while $|\mathcal{S}'| < n$ and $\mathcal{S} \setminus \mathcal{S}' \neq \varnothing$ **do**
        $s \leftarrow$ RANDOMELEMENT($\mathcal{S} \setminus \mathcal{S}'$)
        $\mathcal{S}' \leftarrow \mathcal{S}' \cup \{s\}$
    return $\mathcal{S}'$

Algorithm A.0.10 Sample n elements with replacement from a finite list.

Input: a finite list $\mathcal{S}$ and a number n.
Output: a list with n randomly selected elements of $\mathcal{S}$.
procedure SAMPLEWITHREPLACEMENT($\mathcal{S}, n$)
    $\mathcal{S}' \leftarrow []$ ▷ *The empty list.*
    while $|\mathcal{S}'| < n$ **do**
        $s \leftarrow$ RANDOMELEMENT($\mathcal{S}$)
        $\mathcal{S}' \leftarrow \mathcal{S}' + [s]$ ▷ *Append s; repeats are kept.*
    return $\mathcal{S}'$

Algorithm A.0.11 Compute the maximum value of a finite list of numbers.

Input: a non-empty finite list $(w_1, \ldots, w_K)$.
Output: the maximum value.
procedure MAX($(w_1, \ldots, w_K)$)
    max_value $\leftarrow w_1$
    for $2 \leq k \leq K$ **do**
        if $w_k >$ *max_value* **then**
            max_value $\leftarrow w_k$
    return *max_value*

Algorithm A.0.12 Compute an index at which a finite list attains its maximum.

Input: a non-empty finite list $(w_1, \ldots, w_K)$.
Output: an index k such that w_k is maximal.
procedure ARGMAX($(w_1, \ldots, w_K)$)
    $best\_value \leftarrow \text{MAX}((w_1, \ldots, w_K))$
    $\mathcal{M} \leftarrow \varnothing$
    for $1 \leq k \leq K$ **do**
        if $w_k = best\_value$ **then**
            $\mathcal{M} \leftarrow \mathcal{M} \cup \{k\}$
    return RANDOMELEMENT($\mathcal{M}$) ▷ *Break ties randomly.*

Appendix B

Implementation Details for the Routing App

— added later todo

The classifier is trained on the *training set* $\mathcal{T}$:[1] the edges labelled 0 for lying on a good path, and 1 for being bad. In the training process, we must be careful about *class imbalance*. In the full map the great majority of edges count as bad under our rules, whereas the curated good edges are comparatively rare, on the order of a few percent. A classifier trained on a set with these proportions could reach a high accuracy by the trivial strategy of calling almost every edge bad. To prevent this, we do not train on the full map but on a sample with a much milder imbalance: three bad edges for every good one.

— end added later

In the earlier sections we discussed random forests and probability calibration as techniques to assign a cost to edges. These costs can then be used to make routes. Here we give the details of the pipeline we set up to compute pleasant walking routes.

[1] The features enter the trees after light preprocessing: categorical features (highway type, surface, ...) are turned into integers by an *ordinal encoder*, with missing values represented by an explicit missing category; numeric features are passed unchanged.

OBTAINING THE EDGE FEATURES is the first step. We download the road network for the Netherlands from OpenStreetMap, as packaged by Geofabrik, and import the relevant data into a PostgreSQL database.

PostgreSQL is used for the heavy geospatial preprocessing stage. The tool `osm2pgsql` imports OSM into PostgreSQL tables, and the `PostGIS` extension for PostgreSQL offers functionality to deal with geographic information. It can happen that a few edges are not reachable, we therefore keep the largest component of the road graph.

After this preprocessing step we export the graph to SQLite. SQLite is a simple file database, but fast enough for our goals. We store edge features, scores and costs in this database.

Each OSM *way* becomes one or more directed edges in a graph $(\mathcal{N}, \mathcal{E})$. An edge $e \in \mathcal{E}$ connects two nodes and carries its geometry and OSM tag metadata.

Each edge has an id and is described by a feature vector $f(e)$. The features used by the Wandelvogel code are as follows.

The categorical OSM tags are open-ended: the model uses the raw value present in the map data, and rare spelling mistakes or unusual tag values are simply treated as additional categories. Missing tags are also represented as a separate category. Table B.1 therefore gives representa-

TABLE B.1: Edge features used by the route-scoring model.

Feature	Type	Values
`highway`	Cat	`footway`, `path`, `track`, …
`surface`	Cat	`asphalt`, `sand`, `gravel`, …
`tracktype`	Cat	`grade1`, …, `grade5`
`foot`	Cat	`yes`, `permissive`, `no`, …
`access`	Cat	`yes`, `private`, `no`, …
`bicycle`	Cat	`yes`, `use_sidepath`, `no`, …
`horse`	Cat	`yes`, `designated`, `no`, …
`tunnel`	Cat	`yes`, `building_passage`, …
`bridge`	Cat	`yes`, `boardwalk`, `viaduct`, …
`oneway`	Cat	`yes`, `no`, `-1`
`service`	Cat	`driveway`, `parking_aisle`, …
`landuse`	Cat	`forest`, `residential`, `industrial`, …
`dist_to_forest`	Num	meters
`dist_to_water`	Num	meters
`dist_to_bench`	Num	meters
`dist_to_motorway_trunk`	Num	meters
`dist_to_primary_road`	Num	meters
`dist_to_secondary_road`	Num	meters
`dist_to_tertiary_road`	Num	meters
`in_nature_reserve`	Bin	0 or 1
`in_protected_area`	Bin	0 or 1
`length`	Num	meters

tive values, not a full data dump.

To train a classifier we use a set of published walking tracks, such as the Pieterpad and the Drenthepad: every edge that appears in one of those tracks receives label 0.

Edges that were *not* marked good are then labeled bad by a few simple rules:

- edges in `industrial`, `commercial`, or `residential` landuse zones,
- cycleways (which are often shared with pedestrians) within 50 m of a `primary`, `secondary`, or `tertiary` road,
- edges longer than 200 meters, since people dislike long stretches with the same features.

Finally, any edge that is tagged as a `primary`, `secondary`, or `tertiary` road is relabelled bad, even when it appears in one of the curated tracks.

TRAINING THE RANDOM FOREST is the second conceptual step. We use an *ordinal encoder* for *categorical* features. For a linear model this would be wrong; in that case we should use one-hot encoding to avoid saying that surface 5 is 5 times “larger” than code 1 in the sense in which 50 m is larger than 10 m. For trees, however, one-hot encoding is not needed and can even be harmful. It makes the feature vector much wider and sparser, and it spreads one informative feature, such as highway type, over many separate columns. A tree may need several splits to recover

a simple rule about a group of highway types. Worse yet, in a random forest the feature subset that is chosen randomly at a split will typically not contain all the columns that one-hot encoding uses for one feature. Thus, for random forests it is best to use ordinal encoding for categorical features. *Missing* categorical values are represented by an explicit category `unknown`. *Binary* features are stored as 0/1 indicators, so that a tree can separate the two cases with the split $x \leq 0.5$. Finally, *numeric* features are passed as numbers. As numerical values are never missing, we don't have to replace data by a special value.

To train the random forest, we sample 150 000 bad and 50 000 good edges from the graph of the entire Netherlands. As the good edges form 25% of this training sample, the simple strategy of scoring all edges as bad will not work well. We train 300 trees with a minimum leaf size of 5. These values appear to be sensible, but we chose them somewhat intuitively. If we had a good loss function, we could tune these hyperparameters. However, at present we do not have a representative loss function.

With *isotonic regression* we compute the calibration function g that converts the raw random-forest score $u(e)$ into a calibrated bad-edge probability $g(u(e))$. The calibration set $\mathcal{H}$ is a stratified 20% hold-out from the labeled sample, in the sense of Chapter 3, and is not used to fit the forest. Thus, with the sample sizes above, the forest is fitted on 160 000 edges and calibration uses 40 000 edges, with the same good/bad proportions in both parts. The split is shuffled and uses a fixed random seed, so the example is reproducible.

For routing, we use the calibrated bad-edge probability as the model cost:

$$c_{\text{model}}(e) = g(u(e)).$$

Later hand-written penalties modify this model cost. For instance, if $c_{\text{model}}(e_1) = 0.2$ and $c_{\text{model}}(e_2) = 0.3$, and both edges are 200 m long, the cost difference corresponds to a willingness to walk $(0.3 - 0.2) \cdot 200 = 20$ m extra to enjoy the features of edge e_1. But this extra distance may not reflect sufficiently strongly the difference we experience between the two edges. The multiplier λ scales this willingness to $\lambda \cdot 20$ m; with $\lambda = 5$, one of the values we use below, this becomes 100 m.

Edge-level losses such as the Brier score can tell us whether the classifier probabilities are calibrated, but they do not by themselves tell us whether the routes produced from those probabilities are pleasant. The harder tuning problem is therefore at route-level: ideally we would have an objective that captures the tradeoff between distance, scenery, and avoidance of busy roads. The problem is that this is somewhat arbitrary and, worse, hard to quantify.[2] Because of this, we did not use cross validation or other tools to obtain the most reliable edge score or calibration function: edge quality is just a proxy for total route quality, and we should not give edge quality an overly important interpretation. As a consequence, after training the random forest, we froze it,[3] and then applied a simple calibration step. Moreover, since we have already chosen the calibration set $\mathcal{H}$, we do not create new cross-validation folds, but calibrate the frozen forest directly on $\mathcal{H}$.

THE ROUTING ENVIRONMENT uses the edge cost for the computation of

[2] A similar problem occurs when debating whether a table is beautiful or not.

[3] Freezing means that scikit-learn does not clone and refit forests as part of its internal calibration procedure. The forest remains fixed, and only isotonic regression fits a calibration function.

the cheapest route from A to B. Open Source Routing Machine (OSRM) is the engine of choice as it computes country-sized routes in a few milliseconds and scales to graphs with hundreds of millions of edges.

The random forest writes calibrated costs to `rf_cost`, while another model, such as gradient boosting, can write costs to a different column in the database such as `gb_cost`. Call this the model cost $c_{\text{model}}(e)$. We still need to apply a few hand-written adjustments to this cost.

MTB route classes impose a minimum cost $c_{\text{mtb}}(e)$.[4] An edge gets MTB class 0 if it is not part of an MTB route. If it is part of a relation tagged `type=route` and `route=mtb`, it receives class 1, unless the tags suggest a stronger restriction. Edges with `bicycle=designated` and no explicit foot permission get class 2, and edges with `foot=no` get class 3. These classes impose minimum costs 0, 0.2, 0.4, and 1.0, respectively.

[4] Mountain bike trails are mostly unsuitable for normal walking even though their OSM tags make them attractive.

The road penalty $c_{\text{road}}(e)$ is another hand-written correction. If an edge e is a tunnel or bridge, we set the penalty to zero, because such edges often cannot be avoided. Further, proximity to a motorway or trunk road adds a linear penalty within 200 m: the penalty is 0.01 times the remaining distance to that 200 m radius. In addition, being within 30 m of a primary or secondary road adds a penalty 2.

The final edge cost is

$$c_{\text{route}}(e) = \max\{c_{\text{model}}(e), c_{\text{mtb}}(e)\} + c_{\text{road}}(e).$$

As a last remark, OSRM finds the path that minimizes total travel *time*, not total cost. We therefore convert the edge cost into a *speed*:

$$s(e) = \frac{s_0}{1 + \lambda c_{\text{route}}(e)},$$

where $s_0 = 6$ km/h is a base speed and λ is a weight. We use $\lambda \in \{0, 5, 20\}$: $\lambda = 0$ results in the shortest route, $\lambda = 20$ offers the most scenic (perhaps) by avoiding bad edges most aggressively.

WE VISUALIZE THE route by means of website called De wandelvogel. The implementation is simple: a Python HTTP server based on the standard-library `http.server` module serves a single Leaflet page. In the browser, the user chooses a start point and an end point on the map, after which OSRM snaps the input points to the road network and returns the cheapest path to the Python server. The server sends the route back to the browser, and the browser uses Leaflet to draw the route on the map.

The server uses Nominatim to turn the coordinates of the selected points into a place or road label. The route statistics are computed in a second step by looking up the edge properties in the SQLite database.

Appendix C

Parallel Translation Matches

Hansel et Gretel

A l'orée d'une grande forêt vivaient un pauvre bûcheron, sa femme et ses deux enfants. Le garçon s'appelait Hansel et la fille Grethel. La famille ne mangeait guère.

Une année que la famine régnait dans le pays et que le pain lui-même vint à manquer, le bûcheron ruminait des idées noires, une nuit, dans son lit et remâchait ses soucis.

Il dit à sa femme - Qu'allons-nous devenir? Comment nourrir nos pauvres enfants, quand nous n'avons plus rien pour nous-mêmes?

- Eh bien, mon homme, dit la femme, sais-tu ce que nous allons faire? Dès l'aube, nous conduirons les enfants au plus profond de la forêt nous leur allumerons un feu et leur donnerons à chacun un petit morceau de pain. Puis nous irons à notre travail et les laisserons seuls. Ils ne retrouveront plus leur chemin et nous en serons débarrassés.

- Non, femme, dit le bûcheron. je ne ferai pas cela! Comment pourrais-je me résoudre à laisser nos enfants tout seuls dans la forêt! Les bêtes sauvages ne tarderaient pas à les dévorer.

- Oh! fou, rétorqua-t-elle, tu préfères donc que nous mourions de faim tous les quatre? Alors, il ne te reste qu'à raboter les planches de nos cercueils.

Elle n'eut de cesse qu'il n'acceptât ce qu'elle proposait. - Mais j'ai quand même pitié de ces pauvres enfants, dit le bûcheron.

Les deux petits n'avaient pas pu s'endormir tant ils avaient faim. Ils avaient entendu ce que la marâtre disait à leur père.

Grethel pleura des larmes amères et dit à son frère:

- C'en est fait de nous - Du calme, Grethel, dit Hansel. Ne t'en fais pas;

Je trouverai un moyen de nous en tirer. Quand les parents furent endormis, il se leva, enfila ses habits, ouvrit la chatière et se glissa dehors.

La lune brillait dans le ciel et les graviers blancs, devant la maison, étincelaient comme des diamants.

Hansel se pencha et en mit dans ses poches autant qu'il put.

Puis il rentra dans la maison et dit à Grethel: - Aie confiance, chère petite soeur, et dors tranquille. Dieu ne nous abandonnera pas. Et lui-même se recoucha.

Quand vint le jour, avant même que le soleil ne se levât, la femme réveilla les deux enfants: - Debout, paresseux!

Nous allons aller dans la forêt pour y chercher du bois. Elle leur donna un morceau de pain à chacun et dit: - Voici pour le repas de midi;

ne mangez pas tout avant, car vous n'aurez rien d'autre. Comme les poches de Hansel étaient pleines de cailloux, Grethel mit le pain dans son tablier.

Puis, ils se mirent tous en route pour la forêt. Au bout de quelque temps, Hansel s'arrêta et regarda en direction de la maison. Et sans cesse, il répétait ce geste. Le père dit:

Hansel und Grethel

Near a great forest there lived a poor woodcutter and his wife, and his two children; the boy's name was Hansel and the girl's Grethel.

They had very little to bite or to sup, and once, when there was great dearth in the land, the man could not even gain the daily bread.

As he lay in bed one night thinking of this, and turning and tossing, he sighed heavily, and said to his wife, "What will become of us?

we cannot even feed our children; there is nothing left for ourselves."

"I will tell you what, husband," answered the wife; "we will take the children early in the morning into the forest, where it is thickest; we will make them a fire, and we will give each of them a piece of bread, then we will go to our work and leave them alone; they will never find the way home again, and we shall be quit of them."

"No, wife," said the man, "I cannot do that; I cannot find in my heart to take my children into the forest and to leave them there alone; the wild animals would soon come and devour them."

- "O you fool," said she, "then we will all four starve; you had better get the coffins ready," and she left him no peace until he consented.

"But I really pity the poor children," said the man.

The two children had not been able to sleep for hunger, and had heard what their step-mother had said to their father.

Grethel wept bitterly, and said to Hansel, "It is all over with us."

"Do be quiet, Grethel," said Hansel, "and do not fret; 1 will manage something."

And when the parents had gone to sleep he got up, put on his little coat, opened the back door, and slipped out.

The moon was shining brightly, and the white flints that lay in front of the house glistened like pieces of silver.

Hansel stooped and filled the little pocket of his coat as full as it would hold.

Then he went back again, and said to Grethel, "Be easy, dear little sister, and go to sleep quietly; God will not forsake us," and laid himself down again in his bed.

When the day was breaking, and before the sun had risen, the wife came and awakened the two children, saying, "Get up, you lazy bones; we are going into the forest to cut wood."

Then she gave each of them a piece of bread, and said, "That is for dinner, and you must not eat it before then, for you will get no more."

Grethel carried the bread under her apron, for Hansel had his pockets full of the flints. Then they set off all together on their way to the forest.

When they had gone a little way Hansel stood still and looked back towards the house, and this he did again and again, till his father said to him, "Hansel, what are you looking at?

- Que regardes

-tu , Hansel, et pourquoi restes-tu toujours en arrière ?

Fais attention à toi et n'oublie pas de marcher! - Ah! père dit Hansel, Je regarde mon petit chat blanc qui est perché là-haut sur le toit et je lui dis au revoir.

La femme dit: - Fou que tu es! ce n'est pas le chaton, c'est un reflet de soleil sur la cheminée.

Hansel, en réalité, n'avait pas vu le chat.

Mais, à chaque arrêt, il prenait un caillou blanc dans sa poche et le jetait sur le chemin. Quand ils furent arrivés au milieu de la forêt, le père dit: - Maintenant, les enfants, ramassez du bois! je vais allumer un feu pour que vous n'ayez pas froid. Hansel et Grethel amassèrent des brindilles au sommet d'une petite colline.

Quand on y eut mit le feu et qu'il eut bien pris, la femme dit: - Couchez-vous auprès de lui, les enfants, et reposez-vous. Nous allons abattre du bois. Quand nous aurons fini

, nous reviendrons vous chercher. Hansel et Grethel s'assirent auprès du feu et quand vint l'heure du déjeuner, ils mangèrent leur morceau de pain.

Ils entendaient retentir des coups de hache et pensaient que leur père était tout proche. Mais ce n'était pas la hache. C'était une branche que le bûcheron avait attachée à un arbre mort et que le vent faisait battre de-ci, de-là.

Comme ils étaient assis là depuis des heures, les yeux finirent par leur tomber de fatigue et ils s'endormirent.

Quand ils se réveillèrent, il faisait nuit noire. Grethel se mit à pleurer et dit: - Comment ferons-nous pour sortir de la forêt?

Hansel la consola - Attends encore un peu, dit-il, jusqu'à ce que la lune soit levée.

Alors, nous retrouverons notre chemin. Quand la pleine lune brilla dans le ciel, il prit sa soeur par la main et suivit les petits cailloux blancs. Ils étincelaient comme des écus frais battus et indiquaient le chemin.

Les enfants marchèrent toute la nuit et, quand le jour se leva, ils atteignirent la maison paternelle. Ils frappèrent à la porte.

Lorsque la femme eut ouvert et quand elle vit que c'étaient Hansel et Grethel, elle dit: - Méchants enfants! pourquoi avez-vous dormi si longtemps dans la forêt?

Nous pensions que vous ne reviendriez jamais. Leur père, lui, se réjouit, car il avait le coeur lourd de les avoir laissés seuls dans la forêt.

Peu de temps après, la misère régna de plus belle et les enfants entendirent ce que la marâtre disait, pendant la nuit, à son mari: - Il ne nous reste plus rien à manger, une demi-miche seulement, et après, finie la chanson!

Il faut nous débarrasser des enfants; nous les conduirons encore plus profond dans la forêt pour qu'ils ne puissent plus retrouver leur chemin; il n'y a rien d'autre à faire.

Le père avait bien du chagrin. Il songeait - « Il vaudrait mieux partager la dernière bouchée avec les enfants. »

Mais la femme ne voulut n'en entendre. Elle le gour-

take care not to forget your legs."

"O father," said Hansel, "lam looking at my little white kitten, who is sitting up on the roof to bid me good-bye."

- "You young fool," said the woman, "that is not your kitten, but the sunshine on the chimney-pot."

Of course Hansel had not been looking at his kitten, but had been taking every now and then a flint from his pocket and dropping it on the road.

When they reached the middle of the forest the father told the children to collect wood to make a fire to keep them, warm; and Hansel and Grethel gathered brushwood enough for a little mountain j

and it was set on fire, and when the flame was burning quite high the wife said, "Now lie down by the fire and rest yourselves, you children, and we will go and cut wood; and when we are ready we will come and fetch you."

So Hansel and Grethel sat by the fire, and at noon they each ate their pieces of bread.

They thought their father was in the wood all the time, as they seemed to hear the strokes of the axe: but really it was only a dry branch hanging to a withered tree that the wind moved to and fro.

So when they had stayed there a long time their eyelids closed with weariness, and they fell fast asleep.

When at last they woke it was night, and Grethel began to cry, and said, "How shall we ever get out of this wood?

"But Hansel comforted her, saying, "Wait a little while longer, until the moon rises, and then we can easily find the way home."

And when the full moon got up Hansel took his little sister by the hand, and followed the way where the flint stones shone like silver, and showed them the road.

They walked on the whole night through, and at the break of day they came to their father's house.

They knocked at the door, and when the wife opened it and saw that it was Hansel and Grethel she said, "You naughty children, why did you sleep so long in the wood?

we thought you were never coming home again!" But the father was glad, for it had gone to his heart to leave them both in the woods alone.

Not very long after that there was again great scarcity in those parts, and the children heard their mother say at night in bed to their father, "Everything is finished up; we have only half a loaf, and after that the tale comes to an end.

The children must be off; we will take them farther into the wood this time, so that they shall not be able to find the way back again; there is no other way to manage."

The man felt sad at heart, and he thought, "It would better to share one's last morsel with one's children."

But the wife would listen to nothing that he said, but

manda et lui fit mille reproches.

Qui a dit « A » doit dire « B. »Comme il avait accepté une première fois, il dut consentir derechef.

Les enfants n'étaient pas encore endormis. Ils avaient tout entendu.

Quand les parents furent plongés dans le sommeil, Hansel se leva avec l'intention d'aller ramasser des cailloux comme la fois précédente. Mais la marâtre avait verrouillé la porte et le garçon ne put sortir. Il consola cependant sa petite soeur: - Ne pleure pas, Grethel, dors tranquille;

le bon Dieu nous aidera. Tôt le matin, la marâtre fit lever les enfants.

Elle leur donna un morceau de pain, plus petit encore que l'autre fois. Sur la route de la forêt, Hansel l'émietta dans sa poche; il s'arrêtait souvent pour en jeter un peu sur le sol.

- Hansel, qu'as-tu à t'arrêter et à regarder autour de toi? dit le père. Va ton chemin!

- Je regarde ma petite colombe, sur le toit, pour lui dire au revoir! répondit Hansel. - Fou! dit la femme.

Ce n'est pas la colombe, c'est le soleil qui se joue sur la cheminée. Hansel, cependant, continuait à semer des miettes de pain le long du chemin.

La marâtre conduisit les enfants au fin fond de la forêt, plus loin qu'ils n'étaient jamais allés. On y refit un grand feu et la femme dit:

- Restez là, les enfants.

Quand vous serez fatigués, vous pourrez dormir un peu nous allons couper du bois et, ce soir, quand nous aurons fini, nous viendrons vous chercher. À midi, Grethel partagea son pain avec Hansel qui avait éparpillé le sien le long du chemin.

Puis ils dormirent et la soirée passa sans que personne ne revînt auprès d'eux.

Ils s'éveillèrent au milieu de la nuit, et Hansel consola sa petite soeur, disant: - Attends que la lune se lève, Grethel, nous verrons les miettes de pain que j'ai jetées;

elles nous montreront le chemin de la maison. Quand la lune se leva, ils se mirent en route. Mais de miettes, point.

Les mille oiseaux des champs et des bois les avaient mangées.

Les deux enfants marchèrent toute la nuit et le jour suivant, sans trouver à sortir de la forêt. Ils mouraient de faim, n'ayant à se mettre sous la dent que quelques baies sauvages. Ils étaient si fatigués que leurs jambes ne voulaient plus les porter.

Ils se couchèrent au pied d'un arbre et s'endormirent. Trois jours s'étaient déjà passés depuis qu'ils avaient quitté la maison paternelle.

Ils continuaient à marcher, s'enfonçant toujours plus avant dans la forêt. Si personne n'allait venir à leur aide, ils ne tarderaient pas à mourir.

À midi, ils virent un joli oiseau sur une branche, blanc comme neige. Il chantait si bien que les enfants s'arrêtèrent

scolded and reproached him.

He who says A must say B too, and when a man has given in once he has to do it a second time.

But the children were not asleep, and had heard all the talk.

When the parents had gone to sleep Hansel got up to go out and get more flint stones, as he did before, but the wife had locked the door, and Hansel could not get out; but he comforted his little sister, and said, "Don't cry, Grethel, and go to sleep quietly, and God will help us."

Early the next morning the wife came and pulled the children out of bed.

She gave them each a little piece of "bread -less than before; and on the way to the wood Hansel crumbled the bread in his pocket, and often stopped to throw a crumb on the ground.

"Hansel, what are you stopping behind and staring for?" said the father.

"I am looking at my little pigeon sitting on the roof, to say good-bye to me," answered Hansel.

"You fool," said the wife, "that is no pigeon, but the morning sun shining on the chimney pots." Hansel went on as before, and strewed bread crumbs all along the road.

The woman led the children far into the wood, where they had never been before in all their lives.

And again there was a large fire made, and the mother said, "Sit still there, you children, and when you are tired you can go to sleep; we are going into the forest to cut wood, and in the evening, when we are ready to go home we will come and fetch you." So when noon came Grethel shared her bread with Hansel, who had strewed his along the road.

Then they went to sleep, and the evening passed, and no one came for the poor children.

When they awoke it was dark night, and Hansel comforted his little sister, and said, "Wait a little, Grethel, until the moon gets up, then we shall be able to see the way home by the crumbs of bread that I have scattered along it."

So when the moon rose they got up, but they could find no crumbs of bread, for the birds of the woods and of the fields had come and picked them up.

Hansel thought they might find the way all the same, but they could not.

They went on all that night, and the next day from the morning until the evening, but they could not find the way out of the wood, and they were very hungry, for they had nothing to eat but the few berries they could pick up.

And when they were so tired that they could no longer drag themselves along, they lay down under a tree and fell asleep. It was now the third morning since they had left their father's house.

They were always trying to get back to it, but instead of that they only found themselves farther in the wood, and if help had not soon come they would have been starved.

About noon they saw a pretty snow-white bird sitting on a bough, and singing so sweetly that they stopped to listen.

pour l'écouter.

Quand il eut fini, il déploya ses ailes et vola devant eux. Ils le suivirent jusqu'à une petite maison sur le toit de laquelle le bel oiseau blanc se percha. Quand ils s'en furent approchés tout près, ils virent qu'elle était faite de pain et recouverte de gâteaux. Les fenêtres étaient en sucre.

- Nous allons nous mettre au travail, dit Hansel, et faire un repas béni de Dieu.

Je mangerai un morceau du toit;

ça a l'air d'être bon! Hansel grimpa sur le toit et en arracha un petit morceau pour goûter. Grethel se mit à lécher les carreaux.

On entendit alors une voix suave qui venait de la chambre

- Langue, langue lèche! Qui donc ma maison lèche? Les enfants répondirent

- C'est le vent, c'est le vent.

Ce céleste enfant. Et ils continuèrent à manger sans se laisser détourner de leur tâche.

Hansel, qui trouvait le toit fort bon, en fit tomber un gros morceau par terre et Grethel découpa une vitre entière, s'assit sur le sol et se mit à manger.

La porte, tout à coup, s'ouvrit et une femme, vieille comme les pierres, s'appuyant sur une canne, sortit de la maison.

Hansel et Grethel eurent si peur qu'ils laissèrent tomber tout ce qu'ils tenaient dans leurs mains.

La vieille secoua la tête et dit: - Eh! chers enfants, qui vous a conduits ici?

Entrez, venez chez moi! Il ne vous sera fait aucun mal.

Elle les prit tous deux par la main et les fit entrer dans la maisonnette.

Elle leur servit un bon repas, du lait et des beignets avec du sucre, des pommes et des noix.

Elle prépara ensuite deux petits lits. Hansel et Grethel s'y couchèrent. Ils se croyaient au Paradis. Mais l'amitié de la vieille n'était qu'apparente.

En réalité, c'était une méchante sorcière à l'affût des enfants. Elle n'avait construit la maison de pain que pour les attirer.

Quand elle en prenait un, elle le tuait, le faisait cuire et le mangeait. Pour elle, c'était alors jour de fête.

La sorcière avait les yeux rouges et elle ne voyait pas très clair. Mais elle avait un instinct très sûr, comme les bêtes, et sentait venir de loin les êtres humains.

Quand Hansel et Grethel s'étaient approchés de sa demeure, elle avait ri méchamment et dit d'une voix mielleuse: - Ceux-là, je les tiens!

Il ne faudra pas qu'ils m'échappent! À l'aube, avant que les enfants ne se soient éveillés, elle se leva. Quand elle les vit qui reposaient si gentiment, avec leurs bonnes joues toutes roses, elle murmura: - Quel bon repas je vais faire!

Elle attrapa Hansel de sa main rêche, le conduisit dans une petite étable et l'y enferma au verrou. Il eut beau crier, cela ne lui servit à rien.

And when he had finished the bird spread his wings and flew before them, and they followed after him until they came to a little house, and the bird perched on the roof, and when they came nearer they saw that the house was built of bread, and roofed with cakes; and the window was of transparent sugar.

"We will have some of this," said Hansel, "and make a fine meal.

I will eat a piece of the roof, Grethel, and you can have some of the window-that will taste sweet."

So Hansel reached up and broke off a bit of the roof, just to see how it tasted, and Grethel stood by the window and gnawed at it.

Then they heard a thin voice call out from inside,

"Nibble, nibble, like a mouse, Who is nibbling at my house?"

And the children answered,

"Never mind, It is the wind." And they went on eating, never disturbing themselves.

Hansel, who found that the roof tasted very nice, took down a great piece of it, and Grethel pulled out a large round window-pane, and sat her down and began upon it.

Then the door opened, and an aged woman came out, leaning upon a crutch.

Hansel and Grethel felt very frightened, and let fall what they had in their hands.

The old woman, however, nodded her head, and said, "Ah, my dear children, how come you here?

you must come indoors and stay with me, you will be no trouble."

So she took them each by the hand, and led them into her little house.

And there they found a good meal laid out, of milk and pancakes, with sugar, apples, and nuts.

After that she showed them two little white beds, and Hansel and Grethel laid themselves down on them, and thought they were in heaven.

The old woman, although her behaviour was so kind, was a wicked witch, who lay in wait for children, and had built the little house on purpose to entice them.

When they were once inside she used to kill them, cook them, and eat them, and then it was a feast day with her.

The witch's eyes were red, and she could not see very far, but she had a keen scent, like the beasts, and knew very well when human creatures were near.

When she knew that Hansel and Grethel were coming, she gave a spiteful laugh, and said triumphantly, "I have them, and they shall not escape me!"

Early in the morning, before the children were awake, she got up to look at them, and as they lay sleeping so peacefully with round rosy cheeks, she said to herself, "What a fine feast I shall have!"

Then she grasped Hansel with her withered hand, and led him into a little stable, and shut him up behind a grating; and call and scream as he might, it was no good.

La sorcière s'approcha ensuite de Grethel, la secoua pour la réveiller et s'écria: - Debout, paresseuse! Va chercher de l'eau et prépare quelque chose de bon à manger pour ton frère.

Il est enfermé à l'étable et il faut qu'il engraisse.

Quand il sera à point, je le mangerai. Grethel se mit à pleurer, mais cela ne lui servit à rien. Elle fut obligée de faire ce que lui demandait l'ogresse.

On prépara pour le pauvre Hansel les plats les plus délicats. Grethel, elle, n'eut droit qu'à des carapaces de crabes.

Tous les matins, la vieille se glissait jusqu'à l'écurie et disait: - Hansel, tends tes doigts, que je voie si tu es déjà assez gras. Mais Hansel tendait un petit os et la sorcière, qui avait de mauvais yeux, ne s'en rendait pas compte.

Elle croyait que c'était vraiment le doigt de Hansel et s'étonnait qu'il n'engraissât point. Quand quatre semaines furent passées, et que l'enfant était toujours aussi maigre, elle perdit patience et décida de ne pas attendre plus longtemps.

- Holà! Grethel, cria-t-elle, dépêche-toi d'apporter de l'eau. Que Hansel soit gras ou maigre, c'est demain que je le tuerai et le mangerai. Ah, comme elle pleurait

, la pauvre petite, en charriant ses seaux d'eau, comme les larmes coulaient le long de ses joues! - Dieu bon, aide-nous donc!

s'écria-t-elle. Si seulement les bêtes de la forêt nous avaient dévorés! Au moins serions-nous morts ensemble!

- Cesse de te lamenter! dit la vieille; ça ne te servira à rien! De bon matin

, Grethel fut chargée de remplir la grande marmite d'eau et d'allumer le feu.

- Nous allons d'abord faire la pâte, dit la sorcière. J'ai déjà fait chauffer le four et préparé ce qu'il faut.

Elle poussa la pauvre Grethel vers le four, d'où sortaient de grandes flammes. - Faufile

- toi dedans! ordonna-t-elle, et vois s'il est assez chaud pour la cuisson.

Elle avait l'intention de fermer le four quand la petite y serait pour la faire rôtir. Elle voulait la manger, elle aussi.

Mais Grethel devina son projet et dit: - Je ne sais comment faire , comment entre-t-on dans ce four?

- Petite oie, dit la sorcière, l'ouverture est assez grande, vois, je pourrais y entrer moi-même. Et elle y passa la tête.

Alors Grethel la poussa vivement dans le four, claqua la porte et mit le verrou. La sorcière se mit à hurler épouvantablement.

Mais Grethel s'en alla et cette épouvantable sorcière n'eut plus qu'à rôtir.

Grethel, elle, courut aussi vite qu'elle le pouvait chez Hansel. Elle ouvrit la petite étable et dit: - Hansel , nous sommes libres!

La vieille sorcière est morte! Hansel bondit hors de sa prison, aussi rapide qu'un oiseau dont on vient d'ouvrir la cage.

Comme ils étaient heureux!

Then she went back to Grethel and shook her, crying, "Get up, lazy bones; fetch water, and cook something nice for your brother; he is outside in the stable, and must be fattened up.

And when he is fat enough I will eat him."

Grethel began to weep bitterly, but it was of no use, she had to do what the wicked witch bade her.

And so the best kind of victuals was cooked for poor Hansel, while Grethel got nothing but crab-shells.

Each morning the old woman visited the little stable, and cried, "Hansel, stretch out your finger, that I may tell if you will soon be fat enough." Hansel, however, used to hold out a little bone, and the old woman, who had weak eyes, could not see what it was, and supposing it to be Hansel's finger, wondered very much that it was not getting fatter.

When four weeks had passed and Hansel seemed to remain so thin, she lost patience and could wait no longer.

"Now then, Grethel," cried she to the little girl; "be quick and draw water; be Hansel fat or be he lean, tomorrow I must kill and cook him."

Oh what a grief for the poor little sister to have to fetch water, and how the tears flowed down over her cheeks! "Dear God, pray help us!"

cried she; "if we had been devoured by wild beasts in the wood at least we should have died together."

"Spare me your lamentations," said the old woman; "they are of no avail."

Early next morning Grethel had to get up, make the fire, and fill the kettle.

"First we will do the baking," said the old woman; "I nave heated the oven already, and kneaded the dough."

She pushed poor Grethel towards the oven, out of which the flames were already shining.

"Creep in," said the witch, "and see if it is properly hot, so that the bread may be baked."

And Grethel once in, she meant to shut the door upon her and let her be baked, and then she would have eaten her.

But Grethel perceived her intention, and said, "I don't know how to do it: how shall I get in?"

"Stupid goose," said the old woman, "the opening is big enough, do you see? I could get in myself!" and she stooped down and put her head in the oven's mouth.

Then Grethel gave her a push, so that she went in farther, and she shut the iron door upon her, and put up the bar.

Oh how frightfully she howled! but Grethel ran away, and left the wicked witch to burn miserably.

Grethel went straight to Hansel, opened the stable-door, and cried, "Hansel, we are free!

the old witch is dead!" Then out flew Hansel like a bird from its cage as soon as the door is opened.

How rejoiced they both were!

Comme ils se prirent par le cou, dansèrent et s'embrassèrent!

N'ayant plus rien à craindre, ils pénétrèrent dans la maison de la sorcière. Dans tous les coins, il y avait des caisses pleines de perles et de diamants.

- C'est encore mieux que mes petits cailloux! dit Hansel en remplissant ses poches. Et Grethel ajouta - Moi aussi, je veux en rapporter à la maison! Et elle en mit tant qu'elle put dans son tablier.

- Maintenant, il nous faut partir, dit Hansel, si nous voulons fuir cette forêt ensorcelée.

Au bout de quelques heures, ils arrivèrent sur les bords d'une grande rivière.

- Nous ne pourrons pas la traverser, dit Hansel, je ne vois ni passerelle ni pont.

- On n'y voit aucune barque non plus, dit Grethel. Mais voici un canard blanc. Si Je lui demande, il nous aidera à traverser.

Elle cria:

- Petit canard, petit canard, Nous sommes Hansel et Grethel. Il n'y a ni barque, ni gué, ni pont, Fais-nous passer avant qu'il ne soit tard.

Le petit canard s'approcha et Hansel se mit à califourchon sur son dos. Il demanda à sa soeur de prendre place à côté de lui.

- Non, répondit-elle, ce serait trop lourd pour le canard. Nous traverserons l'un après l'autre. La bonne petite bête les mena ainsi à bon port.

Quand ils eurent donc passé l'eau sans dommage, ils s'aperçurent au bout de quelque temps que la forêt leur devenait de plus en plus familière. Finalement, ils virent au loin la maison de leur père.

Ils se mirent à courir, se ruèrent dans la chambre de leurs parents et sautèrent au cou de leur père.

L'homme n'avait plus eu une seule minute de bonheur depuis qu'il avait abandonné ses enfants dans la forêt.

Sa femme était morte. Grethel secoua son tablier et les perles et les diamants roulèrent à travers la chambre. Hansel en sortit d'autres de ses poches, par poignées.

C'en était fini des soucis. Ils vécurent heureux tous ensemble.

how they fell each on the other's neck! and danced about, and kissed each other!

And as they had nothing more to fear they went over all the old witch's house, and in every corner there stood chests of pearls and precious stones.

"This is something better than flint stones," said Hansel, as he filled his pockets, and Grethel, thinking she also would like to carry something home with her, filled her apron full.

i! Now, away we go," said Hansel, "if we only can get out of the witch's wood."

When they had journeyed a few hours they came to a great piece of water.

"We can never get across this," said Hansel, "I see no stepping-stones and no bridge."

"And there is no boat either," said Grethel; "but here comes a white duck; if I ask her she will help us over."

So she cried,

"Duck, duck, here we stand, Hansel and Grethel, on the land, Stepping-stones and bridge we lack, Carry us over on your nice white back."

And the duck came accordingly, and Hansel got upon her and told his sister to come too.

"No," answered Grethel, "that would be too hard upon the duck; we can go separately, one after the other."

And that was how it was managed, and after that they went on happily, until they came to the wood, and the way grew more and more familiar, till at last they saw in the distance their father's house.

Then they ran till they came up to it, rushed in at the door, and fell on their father's neck.

The man had not had a quiet hour since he left his children in the wood; but the wife was dead.

And when Grethel opened her apron the pearls and precious stones were scattered all over the room, and Hansel took one handful after another out of his pocket.

Then was all care at an end, and they lived in great joy together.

My tale is done, there runs a mouse, whosoever catches it, may make himself a big fur cap out of it.

Hans en Grietje

Aan de rand van een groot bos woonde eens een arme houthakker met zijn vrouw en twee kinderen. Het jongetje heette Hans en het meisje Grietje.

Ze hadden maar heel weinig te eten, en eens, toen alles erg duur werd in het land, konden ze ook niet meer aan brood komen.

Toen hij daar 's avonds in bed over lag te tobben en vol zorgen lag te woelen, zei hij tegen zijn vrouw: "Wat moet er van ons worden?

Hoe kunnen we onze kinderen te eten geven, wij die voor ons zelf niets meer hebben?

" "Weet je wat, man," antwoordde de vrouw, "we zullen bij het eerste morgenlicht de kinderen wegbrengen, heel diep in het bos, dan maken we daar een flink vuur en we geven hun ieder nog een stuk brood, dan gaan wij aan het werk en laten hen alleen. Ze vinden de weg naar huis niet meer terug en wij zijn ze kwijt.

" "Nee vrouw," zei de man, "dat doe ik niet, hoe zou ik het over mijn hart verkrijgen, mijn kinderen alleen te laten in het bos; dan zouden immers wilde dieren komen en hen verscheuren. "

"Dwaze man," zei ze, "moeten we dan alle vier van honger sterven, ga dan maar de planken voor de kisten schaven," en ze liet hem niet met rust, tot hij toegaf.

"Maar 't spijt me toch zo van die arme kinderen," zei de man.

De twee kinderen hadden zo'n honger dat ze niet konden slapen en ze hadden alles gehoord wat de stiefmoeder tegen de vader had gezegd.

Grietje weende bittere tranen en zei tegen Hans: "Nu is het met ons gedaan.

" "Stil Grietje," zei Hans, "wees maar niet bang, we zullen er wel wat op vinden."

En toen de ouders waren ingeslapen, stond hij op, deed zijn jasje aan, maakte de onderdeur open en sloop naar buiten.

De maan scheen helder en de witte kiezels voor het huis schenen blank.

Hans bukte zich en stak er zoveel in zijn broekzakken, als er maar in konden.

Toen ging hij het huis weer in, zei tegen Grietje: "Wees maar stil, zusjelief, slaap rustig in, onze lieve Heer zal ons niet verlaten," en hij' ging ook weer in bed.

Bij 't eerste schemerlicht, nog voor de zon was opgegaan, kwam de vrouw de beide kinderen roepen. "Sta toch op, luilakken, we moeten 't bos in om hout te halen."

Dan gaf ze aan elk een stukje brood en zei: "Daar heb je iets voor de middag; maar niet eerder opeten, want dit is alles watje krijgt."

Grietje nam het brood onder haar schortje, omdat Hans zijn zakken vol stenen had.

Toen gingen ze alle vier naar het bos.

Toen ze een eind op weg waren, stond Hansje stil en keek om naar het huis, en deed dat nog eens en toen nog eens. De vader zei: "Hans, wat kijkje toch telkens om en je blijft aldoor achter;

opletten en vergeet je benen niet. "

Hansel und Grethel

Near a great forest there lived a poor woodcutter and his wife, and his two children; the boy's name was Hansel and the girl's Grethel.

They had very little to bite or to sup, and once, when there was great dearth in the land, the man could not even gain the daily bread.

As he lay in bed one night thinking of this, and turning and tossing, he sighed heavily, and said to his wife, "What will become of us?

we cannot even feed our children; there is nothing left for ourselves."

"I will tell you what, husband," answered the wife; "we will take the children early in the morning into the forest, where it is thickest; we will make them a fire, and we will give each of them a piece of bread, then we will go to our work and leave them alone; they will never find the way home again, and we shall be quit of them."

"No, wife," said the man, "I cannot do that; I cannot find in my heart to take my children into the forest and to leave them there alone; the wild animals would soon come and devour them."

- "O you fool," said she, "then we will all four starve; you had better get the coffins ready," and she left him no peace until he consented.

"But I really pity the poor children," said the man.

The two children had not been able to sleep for hunger, and had heard what their step-mother had said to their father.

Grethel wept bitterly, and said to Hansel, "It is all over with us."

"Do be quiet, Grethel," said Hansel, "and do not fret; 1 will manage something."

And when the parents had gone to sleep he got up, put on his little coat, opened the back door, and slipped out.

The moon was shining brightly, and the white flints that lay in front of the house glistened like pieces of silver.

Hansel stooped and filled the little pocket of his coat as full as it would hold.

Then he went back again, and said to Grethel, "Be easy, dear little sister, and go to sleep quietly; God will not forsake us," and laid himself down again in his bed.

When the day was breaking, and before the sun had risen, the wife came and awakened the two children, saying, "Get up, you lazy bones; we are going into the forest to cut wood."

Then she gave each of them a piece of bread, and said, "That is for dinner, and you must not eat it before then, for you will get no more."

Grethel carried the bread under her apron, for Hansel had his pockets full of the flints.

Then they set off all together on their way to the forest.

When they had gone a little way Hansel stood still and looked back towards the house, and this he did again and again, till his father said to him, "Hansel, what are you looking at?

take care not to forget your legs."

"Och vader," zei Hans, "ik kijk om naar het witte poesje, het zit boven op 't dak en wil me vaarwel zeggen."

De moeder zei: "Dwaas, dat is je kat niet, dat is de ochtendzon op de schoorsteen.

" Maar Hans had helemaal niet naar een katje gekeken, maar had aldoor kleine kiezelsteentjes uit zijn zak op de weg gegooid.

Ze kwamen nu midden in het bos en de vader zei: "Nu moeten jullie hout sprokkelen, kinderen, ik wil een vuur maken, zodat jullie het niet koud hebben." Hans en Grietje droegen rijshout bijeen, een hele berg.

Het rijshout werd aangestoken, en toen de vlam goed hoog brandde, zei de vrouw: "Gaan jullie nu bij 't vuur liggen, kinderen, en rust lekker uit, wij gaan het bos in om hout te kappen. Als we klaar zijn, komen we terug en halen jullie af. "

Hans en Grietje zaten bij het vuur, en toen 't middag was geworden, aten ze allebei een stukje brood.

En omdat ze bijlslagen hoorden, geloofden ze dat hun vader in de buurt was. Maar het was de bijl niet, het was een tak die hij aan een dorre boom had gebonden en die in de wind voortdurend klepperde.

– Toen ze lang gezeten hadden, vielen hun ogen dicht, en ze sliepen vast.

Eindelijk werden ze weer wakker, maar toen was het stikdonker. Grietje begon te schreien en zei:

"Hoe komen we nu uit het bos?" Maar Hans troostte haar: "Wacht maar een poosje, dan komt de maan op, en dan zullen we de weg wel vinden.

" En toen de volle maan kwam, nam Hans zijn zusje bij de hand, en ging de kiezelsteentjes langs, die schitterden als nieuwe munten en hem de weg wezen.

Ze liepen de hele nacht, en kwamen bij 't eerste ochtendlicht weer bij hun vaders huis.

Ze klopten aan, de vrouw deed open en toen ze zag dat het Hans en Grietje waren, zei ze: "Stoute kinderen! wat hebben jullie lang in 't bos geslapen; we dachten dat jullie niet terugkwamen."

Maar de vader was blij, want het had hem veel verdriet gedaan, dat hij hen had achtergelaten.

Kort daarop was de nood weer hoog gestegen, en de kinderen hoorden hoe de moeder 's nachts, in bed, tot hun vader sprak: "Alles is weer op, we hebben nog een half brood, en dan is 't lied weer uit."

De kinderen moeten weg, we zullen ze dieper het bos in brengen, zodat ze de weg niet meer terugvinden, anders is er voor ons geen redding meer.

" Het viel de man weer zwaar, en hij dacht: "Het zou beter zijn, de laatste happen met de kinderen te delen."

Maar de vrouw luisterde nooit naar wat hij zei, ze werd boos en maakte hem verwijten.

Wie A zegt moet ook B zeggen, en omdat hij de eerste maal toegegeven had, moest hij het de tweede keer ook doen.

De kinderen waren evenwel wakker geweest en hadden het gesprek gehoord.

Terwijl de ouders sliepen, stond Hans weer op, wilde naar

"O father," said Hansel, "lam looking at my little white kitten, who is sitting up on the roof to bid me good-bye."

- "You young fool," said the woman, "that is not your kitten, but the sunshine on the chimney-pot."

Of course Hansel had not been looking at his kitten, but had been taking every now and then a flint from his pocket and dropping it on the road.

When they reached the middle of the forest the father told the children to collect wood to make a fire to keep them, warm; and Hansel and Grethel gathered brushwood enough for a little mountain j

and it was set on fire, and when the flame was burning quite high the wife said, "Now lie down by the fire and rest yourselves, you children, and we will go and cut wood; and when we are ready we will come and fetch you."

So Hansel and Grethel sat by the fire, and at noon they each ate their pieces of bread.

They thought their father was in the wood all the time, as they seemed to hear the strokes of the axe: but really it was only a dry branch hanging to a withered tree that the wind moved to and fro.

So when they had stayed there a long time their eyelids closed with weariness, and they fell fast asleep.

When at last they woke it was night, and Grethel began to cry, and said, "How shall we ever get out of this wood?

"But Hansel comforted her, saying, "Wait a little while longer, until the moon rises, and then we can easily find the way home."

And when the full moon got up Hansel took his little sister by the hand, and followed the way where the flint stones shone like silver, and showed them the road.

They walked on the whole night through, and at the break of day they came to their father's house.

They knocked at the door, and when the wife opened it and saw that it was Hansel and Grethel she said, "You naughty children, why did you sleep so long in the wood?

we thought you were never coming home again!" But the father was glad, for it had gone to his heart to leave them both in the woods alone.

Not very long after that there was again great scarcity in those parts, and the children heard their mother say at night in bed to their father, "Everything is finished up; we have only half a loaf, and after that the tale comes to an end.

The children must be off; we will take them farther into the wood this time, so that they shall not be able to find the way back again; there is no other way to manage."

The man felt sad at heart, and he thought, "It would better to share one's last morsel with one's children."

But the wife would listen to nothing that he said, but scolded and reproached him.

He who says A must say B too, and when a man has given in once he has to do it a second time.

But the children were not asleep, and had heard all the talk.

When the parents had gone to sleep Hansel got up to go

buiten en kiezeltjes zoeken, zoals de vorige maal, maar de vrouw had de deur afgesloten en Hans kon er niet uit. Maar weer troostte hij zijn zusje: "Huil maar niet Grietje en slaap maar lekker, onze lieve Heer zal ons wel helpen.

" Vroeg in de morgen kwam de vrouw de kinderen uit bed halen.

Ze kregen een stukje brood, nog kleiner dan de vorige keer. Op de weg naar het bos brokkelde Hans het in zijn zak; vaak stond hij stil en gooide dan een kruimeltje op de grond.

"Hansje, wat kijk je toch aldoor om?" zei de vader, "je moet doorlopen." - "Ik kijk naar mijn duif, hij zit op 't dak en wil mij goedendag zeggen," antwoordde Hans.

"Dwaas," zei de vrouw, "dat is de duif niet, dat is de ochtendzon die op de schoorsteen schijnt.

" Maar gaandeweg gooide Hans alle kruimeltjes op de weg.

De vrouw leidde de kinderen nog verder het bos in, waar ze nog nooit geweest waren.

Toen werd er weer een heerlijk vuur aangemaakt, en de moeder zei: "Blijf daar nu zitten, kinderen, en als jullie moe zijn, kun je een beetje gaan slapen; als we vanavond klaar zijn, halen we jullie af."

– Toen het middag geworden was, deelde Grietje haar brood met Hans, die het zijne onderweg had gestrooid.

Daarna sliepen ze in, de avond verliep en niemand kwam de kinderen halen.

Ze werden weer wakker in 't holst van de nacht, maar Hans troostte Grietje en zei: "Wacht maar, Grietje, tot de maan opgaat, dan kunnen we de kruimels zien, die ik gestrooid heb en die wijzen ons de weg naar huis. "

Toen de maan scheen, stonden ze op, maar ze vonden geen kruimels meer, want de duizenden vogels die in het bos en veld rondvliegen, hadden ze opgepikt.

Hans zei tegen Grietje:

"We zullen de weg wel vinden," maar ze vonden hem niet, ze liepen de hele nacht en nog de dag daarop van de morgen tot de avond, maar ze kwamen het bos niet uit en werden zo hongerig, want ze kregen niets dan alleen bosbessen.

En omdat ze zo moe werden, dat hun benen hen niet meer dragen konden, gingen ze onder een boom liggen en sliepen in.

Nu was het al de derde morgen sinds ze hun vaders huis hadden verlaten.

Ze begonnen weer te lopen, maar ze raakten aldoor dieper het bos in, en als er niet gauw hulp kwam opdagen, zouden ze van dorst omkomen.

Het werd middag en ze zagen een mooi, sneeuwwit vogeltje op een tak zitten, dat zong zo mooi, dat ze bleven staan om ernaar te luisteren.

Toen het liedje uit was, klapte het met zijn vleugels en vloog voor hen uit, en ze liepen achter het diertje aan, tot ze aan een huisje kwamen! Hij ging daar op het dak zitten en toen ze heel dichtbij waren gekomen, zagen ze dat het huisje van brood was gebouwd en met pannekoeken gedekt en de vensters waren van heldere kandijsuiker.

out and get more flint stones, as he did before, but the wife had locked the door, and Hansel could not get out; but he comforted his little sister, and said, "Don't cry, Grethel, and go to sleep quietly, and God will help us."

Early the next morning the wife came and pulled the children out of bed.

She gave them each a little piece of "bread -less than before; and on the way to the wood Hansel crumbled the bread in his pocket, and often stopped to throw a crumb on the ground.

"Hansel, what are you stopping behind and staring for?" said the father. "I am looking at my little pigeon sitting on the roof, to say good-bye to me," answered Hansel.

"You fool," said the wife, "that is no pigeon, but the morning sun shining on the chimney pots."

Hansel went on as before, and strewed bread crumbs all along the road.

The woman led the children far into the wood, where they had never been before in all their lives.

And again there was a large fire made, and the mother said, "Sit still there, you children, and when you are tired you can go to sleep; we are going into the forest to cut wood, and in the evening, when we are ready to go home we will come and fetch you."

So when noon came Grethel shared her bread with Hansel, who had strewed his along the road.

Then they went to sleep, and the evening passed, and no one came for the poor children.

When they awoke it was dark night, and Hansel comforted his little sister, and said, "Wait a little, Grethel, until the moon gets up, then we shall be able to see the way home by the crumbs of bread that I have scattered along it."

So when the moon rose they got up, but they could find no crumbs of bread, for the birds of the woods and of the fields had come and picked them up.

Hansel thought they might find the way all the same, but they could not.

They went on all that night, and the next day from the morning until the evening, but they could not find the way out of the wood, and they were very hungry, for they had nothing to eat but the few berries they could pick up.

And when they were so tired that they could no longer drag themselves along, they lay down under a tree and fell asleep.

It was now the third morning since they had left their father's house.

They were always trying to get back to it, but instead of that they only found themselves farther in the wood, and if help had not soon come they would have been starved.

About noon they saw a pretty snow-white bird sitting on a bough, and singing so sweetly that they stopped to listen.

And when he had finished the bird spread his wings and flew before them, and they followed after him until they came to a little house, and the bird perched on the roof, and when they came nearer they saw that the house was built of bread, and roofed with cakes; and the window was of transparent sugar.

"Daar zullen we aan beginnen," zei Hans, "en een kostelijk maal hebben.

Ik wil wat van 't dak hebben, Grietje, eet jij van het venster, dat is zoet. "

Hans reikte omhoog en brak wat van 't dak af om te proeven, hoe dat smaakte, en Grietje ging naar de ruitjes en knabbelde daar aan.

Daar riep een fijn stemmetje uit de kamer:

Knibbel knabbel knuisje, Wie knabbelt er aan mijn huisje?

De kinderen riepen: De wind, de wind, dat hemelse kind!

En ze aten verder zonder zich uit het veld te laten slaan.

Hans, wie het dak heel goed smaakte, trok er een groot stuk af, en Grietje stootte een hele ronde ruit uit en ging ermee zitten en deed zich tegoed.

Maar opeens ging de deur open, en een stokoude vrouw die op een krukje leunde, kwam het huis uitgeslopen.

Hans en Grietje schrokken zo erg, dat ze lieten vallen wat ze in de hand hadden.

Het oudje schommelde met haar hoofd en zei: "Zo lieve kindertjes, en wie heeft jullie hier gebracht?

Kom maar mee naar binnen, en blijf bij mij, er zal niets kwaads gebeuren.

" Ze nam elk van hen bij de hand en bracht hen in 't huisje.

Toen werd heerlijk eten op tafel gezet, melk en pannekoeken met suiker, en appels en noten toe.

Daarna werden twee mooie bedjes met wit beddegoed opgemaakt, en Hans en Grietje gingen erin liggen en dachten dat ze in de hemel waren.

De oude had maar gedaan alsof ze zo lief was; ze was een boze heks, die loerde op de kinderen, en ze had dat broodhuisje alleen maar gebouwd om de kinderen te lokken.

Wanneer ze een kind in haar macht had, maakte ze het dood, braadde het en at het op en dat was een feestdag voor haar.

Heksen hebben rode ogen en kunnen niet ver zien, maar ze hebben een fijne neus, net als dieren en ze ruiken het, als er mensen in de buurt zijn.

Toen Hans en Grietje in haar buurt waren gekomen, had ze lelijk gelachen en spottend gezegd: "Die heb ik, die ontglippen me niet meer."

's Morgens vroeg, voor de kinderen wakker waren, stond ze al op, en toen ze hen beiden zo rustig zag slapen met ronde rode wangen, mompelde ze voor zich heen: "Dat zal een lekker hapje worden."

Toen pakte ze Hans op met haar benige hand en droeg hem naar een klein stalletje en sloot hem op achter een hekje; hij mocht schreeuwen zo hard hij wou, dat gaf toch niets.

Daarom ging ze naar Grietje, schudde haar wakker en riep: "Opstaan, luiwammes, water halen. Kook wat lekkers voor je broer, die zit buiten in het stalletje en moet dik en vet worden.

Als hij goed dik is, eet ik hem op."

Grietje begon bitter te schreien, maar ook dat hielp niets, ze moest doen wat de boze heks wilde.

Nu werd voor de arme Hans het lekkerste eten gekookt,

"We will have some of this," said Hansel, "and make a fine meal.

I will eat a piece of the roof, Grethel, and you can have some of the window-that will taste sweet."

So Hansel reached up and broke off a bit of the roof, just to see how it tasted, and Grethel stood by the window and gnawed at it.

Then they heard a thin voice call out from inside,

"Nibble, nibble, like a mouse, Who is nibbling at my house?"

And the children answered, "Never mind, It is the wind."

And they went on eating, never disturbing themselves.

Hansel, who found that the roof tasted very nice, took down a great piece of it, and Grethel pulled out a large round window-pane, and sat her down and began upon it.

Then the door opened, and an aged woman came out, leaning upon a crutch.

Hansel and Grethel felt very frightened, and let fall what they had in their hands.

The old woman, however, nodded her head, and said, "Ah, my dear children, how come you here?

you must come indoors and stay with me, you will be no trouble."

So she took them each by the hand, and led them into her little house.

And there they found a good meal laid out, of milk and pancakes, with sugar, apples, and nuts.

After that she showed them two little white beds, and Hansel and Grethel laid themselves down on them, and thought they were in heaven.

The old woman, although her behaviour was so kind, was a wicked witch, who lay in wait for children, and had built the little house on purpose to entice them.

When they were once inside she used to kill them, cook them, and eat them, and then it was a feast day with her.

The witch's eyes were red, and she could not see very far, but she had a keen scent, like the beasts, and knew very well when human creatures were near.

When she knew that Hansel and Grethel were coming, she gave a spiteful laugh, and said triumphantly, "I have them, and they shall not escape me!"

Early in the morning, before the children were awake, she got up to look at them, and as they lay sleeping so peacefully with round rosy cheeks, she said to herself, "What a fine feast I shall have!"

Then she grasped Hansel with her withered hand, and led him into a little stable, and shut him up behind a grating; and call and scream as he might, it was no good.

Then she went back to Grethel and shook her, crying, "Get up, lazy bones; fetch water, and cook something nice for your brother; he is outside in the stable, and must be fattened up.

And when he is fat enough I will eat him."

Grethel began to weep bitterly, but it was of no use, she had to do what the wicked witch bade her.

And so the best kind of victuals was cooked for poor

maar Grietje kreeg enkel de botjes en de schillen.

Elke morgen sloop de oude heks naar het stalletje en riep: "Hans, steek je vinger eens uit, zodat ik voelen kan of je al dik wordt!

" Maar Hans stak alleen een splinter hout naar buiten, en de oude heks die niet goed zien kon, dacht dat het zijn vinger was en ze was verbaasd dat hij nog niet dikker werd.

Toen er vier weken voorbij waren en Hans nog altijd zo mager bleef, begon ze ongeduldig te worden en wilde niet langer wachten. "Hé!

Grietje," riep ze 't meisje toe: "Wees eens flink en haal water voor me; Hans mag dan dik of dun zijn, morgen slacht ik hem en kook ik hem.

" O, wat jammerde het arme zusje bij het waterdragen, en wat stroomden er een tranen langs haar wangen!

"Onze lieve Heer, help ons toch," riep ze uit, "hadden de wilde beesten ons in 't bos maar opgegeten, dan waren we toch samen gestorven." - "Spaar je tranen maar," zei de oude, "het geeft je toch niets.

" 's Morgens moest Grietje vroeg op, vuur maken en de ketel met water erboven hangen.

"Eerst zullen we bakken," zei de oude vrouw. "Ik heb de bakoven al gestookt en 't deeg gekneed!"

Ze duwde het arme Grietje naar buiten naar 't bakhuis waar de vlammen al uitsloegen.

"Kruip erin," zei de heks, "en kijk of het goed heet is, of we het brood er al in kunnen schuiven."

En toen Grietje erin moest, wilde ze de oven dichtdoen en er Grietje in braden, want haar wilde ze ook opeten.

Maar Grietje begreep wat ze van plan was en zei:

"Ik weet niet hoe ik dat doen moet, hoe kom ik daar in?" - "Domme gans," zei de heks, "de opening is groot genoeg, zie je wel?

Ik zou er zelf wel in kunnen," ze krabbelde eraan en stak haar hoofd in de bakoven.

Toen gaf Grietje haar een flinke stoot zodat ze er zelf in viel, gooide de ijzeren deur dicht en schoof er de grendel voor. Hu!

daar zette ze een keel op, het was gruwelijk; maar Grietje liep hard weg, en de goddeloze heks moest ellendig omkomen.

Maar Grietje liep rechttoe rechtaan naar Hans, maak het stalletje open en riep: "Hans we zijn verlost, de oude heks is dood!"

Toen sprong Hans eruit als een vogel uit de kooi, zodra ze voor hem de deur had geopend.

Wat waren ze blij, wat zijn ze elkaar om de hals gevallen, wat zijn ze met elkaar rondgesprongen en kusten elkaar!

En nu ze nergens meer bang voor hoefden te zijn, gingen ze het huis van de heks binnen, daar stonden in alle hoeken kasten vol parels en edelstenen.

"Dat is nog beter dan kiezels," zei Hans, en propte zijn zakken vol, en Grietje zei:

"Ik wil ook wat meenemen naar huis!" en stopte haar schortje vol. "Maar nu gaan we weg," zei Hans, "want ik wil

Hansel, while Grethel got nothing but crab-shells.

Each morning the old woman visited the little stable, and cried, "Hansel, stretch out your finger, that I may tell if you will soon be fat enough."

Hansel, however, used to hold out a little bone, and the old woman, who had weak eyes, could not see what it was, and supposing it to be Hansel's finger, wondered very much that it was not getting fatter.

When four weeks had passed and Hansel seemed to remain so thin, she lost patience and could wait no longer.

"Now then, Grethel," cried she to the little girl; "be quick and draw water; be Hansel fat or be he lean, tomorrow I must kill and cook him."

Oh what a grief for the poor little sister to have to fetch water, and how the tears flowed down over her cheeks!

"Dear God, pray help us!" cried she; "if we had been devoured by wild beasts in the wood at least we should have died together." "Spare me your lamentations," said the old woman; "they are of no avail."

Early next morning Grethel had to get up, make the fire, and fill the kettle.

"First we will do the baking," said the old woman; "I nave heated the oven already, and kneaded the dough."

She pushed poor Grethel towards the oven, out of which the flames were already shining.

"Creep in," said the witch, "and see if it is properly hot, so that the bread may be baked."

And Grethel once in, she meant to shut the door upon her and let her be baked, and then she would have eaten her.

But Grethel perceived her intention, and said, "I don't know how to do it: how shall I get in?"

"Stupid goose," said the old woman, "the opening is big enough, do you see? I could get in myself!"

and she stooped down and put her head in the oven's mouth.

Then Grethel gave her a push, so that she went in farther, and she shut the iron door upon her, and put up the bar.

Oh how frightfully she howled! but Grethel ran away, and left the wicked witch to burn miserably.

Grethel went straight to Hansel, opened the stable-door, and cried, "Hansel, we are free!

the old witch is dead!" Then out flew Hansel like a bird from its cage as soon as the door is opened.

How rejoiced they both were! how they fell each on the other's neck! and danced about, and kissed each other!

And as they had nothing more to fear they went over all the old witch's house, and in every corner there stood chests of pearls and precious stones.

"This is something better than flint stones," said Hansel, as he filled his pockets, and Grethel, thinking she also would like to carry something home with her, filled her apron full.

i! Now, away we go," said Hansel, "if we only can get out of the witch's wood."

uit dat heksenbos weg.

" Toen ze een paar uur gelopen hadden, kwamen ze bij een groot meer.

"Daar kunnen we niet over," zei Hans, "ik zie geen weg en geen brug." - "Er is ook geen bootje," zei Grietje, "maar daar zwemt een witte eend, als ik 't die vraag, brengt hij ons wel naar de overkant."

En ze riep: Eendje, eendje, hier zijn Hans en Grietje, d'r is geen weg en ook geen bruggetje, neem ons op je witte ruggetje!

Het eendje kwam aangezwommen, en Hans ging op hem zitten en vroeg zijn zusje erbij te gaan zitten.

"Neen," antwoordde Grietje, "dat is hem te zwaar, hij moet ons na elkaar overbrengen."

Dat deed het goede dier, en toen ze gelukig over waren en een poosje voortliepen, kwam hun het bos steeds bekender voor, en eindelijk zagen ze in de verte hun vaders huis liggen.

Toen zetten ze het op een lopen, stortten de kamer binnen en vielen hun vader om de hals.

De man had geen gelukkig ogenblik meer gehad, sinds hij de kinderen in het bos had achtergelaten, maar de vrouw was gestorven.

Grietje schudde haar schortje uit, zodat de parels en edelstenen in de kamer rolden, en Hans wierp de ene handvol na de andere erbij.

Toen was er een eind aan alle zorgen gekomen en ze leefden vol blijdschap samen.

Mijn sprookje is uit, die muis is een guit, wie die vangt mag er een heel erg grote pelsmuts van maken.

When they had journeyed a few hours they came to a great piece of water.

"We can never get across this," said Hansel, "I see no stepping-stones and no bridge." "And there is no boat either," said Grethel; "but here comes a white duck; if I ask her she will help us over." So she cried,

"Duck, duck, here we stand, Hansel and Grethel, on the land, Stepping-stones and bridge we lack, Carry us over on your nice white back."

And the duck came accordingly, and Hansel got upon her and told his sister to come too.

"No," answered Grethel, "that would be too hard upon the duck; we can go separately, one after the other."

And that was how it was managed, and after that they went on happily, until they came to the wood, and the way grew more and more familiar, till at last they saw in the distance their father's house.

Then they ran till they came up to it, rushed in at the door, and fell on their father's neck.

The man had not had a quiet hour since he left his children in the wood; but the wife was dead.

And when Grethel opened her apron the pearls and precious stones were scattered all over the room, and Hansel took one handful after another out of his pocket.

Then was all care at an end, and they lived in great joy together.

My tale is done, there runs a mouse, whosoever catches it, may make himself a big fur cap out of it.