Exercise 1: Background Material

Why should you work through these exercises ?

If you ask any modern AI agent (a recent version of GPT, Claude, Gemini, Qwen, ...) to solve the problems on this exercise sheet, it will likely do this with ease, without mistakes, and within minutes ...

However, the problems on this sheet provide you with an opportunity to refresh or to extend your knowledge of mathematical and computer scientific concepts that are fundamental to the theory behind decision making and reinforcement learning. Our lecture on the principles of reinforcement learning will assume familiarity with all these fundamental concepts. If you feel a bit shaky in this regard, we suggest you return to this exercise sheet throughout the course or at least when needed. You are, of course, welcome to use AI assistance when working on the following problems.

What should you take home from these exercises ?

Try to connect the dots! Ask yourself the following:

- Have you previously seen the objects / equations / identities that occur in the following problems? If so, in what context was that and which names did they have there?
- Are there connections between some of the problems in this exercise? Do some of them deal with essentially the same concepts or ideas? If so, then what is the nature of these connections? Are there reoccurring themes?

1.1 Getting to grips with exponential processes

Consider the *sequence* of exponentials b^n where $b \geq 2$ and $n \in \mathbb{N}$. The following table shows examples for different choices of b and a few initial values of n.

	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$	$\cdots$
$b=2$	1	2	4	8	16	
$b=3$	1	3	9	27	81	
$b=4$	1	4	16	64	256	
$\vdots$						

Next, consider the following *partial sum* which we can compute for any exponential sequence

$$S_n = \sum_{k=0}^{n-1} b^k \ . \tag{1}$$

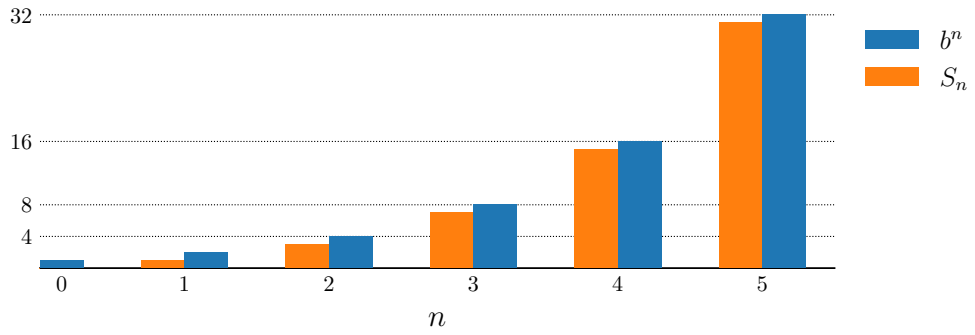
Looking at our table and doing some arithmetic, we seem to have this very crucial inequality

$$S_n \leq b^n \ . \tag{2}$$

To paraphrase, it (empirically) seems we have this crucial characteristic of exponential growth:

> **For any $b \geq 2$ and any $n \in \mathbb{N}$, the value of the exponential b^n exceeds the value of the *sum* of all its preceding exponentials $b^0 + b^1 + b^2 + \cdots + b^{n-1}$.**

The following figure corroborates this impression for several choices of $0 \leq n \leq 5$ and $b = 2$. $b = 2$ and for several choices of $0 \leq n \leq 5$.



In the subsequent tasks, you are supposed to put this empirical observation on a theoretical foundation and prove the behavior of exponential growth in (2). Once this has been proven,

you can use your proof techniques to establish a more general result which we will direly need later on in this course.

1.1.1 important equalities

Letting $b \neq 1$ and $n \in \mathbb{N}$, prove that

$$S_n = \frac{b^n - 1}{b - 1} = \frac{1 - b^n}{1 - b} \ . \tag{3}$$

1.1.2 the nature of exponential growth

Letting $b \geq 2$ and $n \in \mathbb{N}$, prove that

$$S_n \leq b^n \ . \tag{4}$$

1.1.3 `Python` coding (creating visualizations)

Implement `Python` / `numpy` / `matplotlib` code that produces figures such as the one above. Be ambitious and try to replicate the look-and-feel of the above figure as closely as possible.

1.1.4 empirical experiments

Given your solution for the last task, empirically verify your theoretical result for the last but one task. That is, plot charts for different choices of $b \geq 2$ and of N in $0 \leq n \leq N$.

1.1.5 more general identities

This task establishes a result of utter importance to reinforcement learning theory!

Letting $a \neq 0$, $b \neq 1$, and $n \in \mathbb{N}$, prove that

$$S_n^a = \sum_{k=0}^{n-1} a\, b^k = \frac{a(b^n - 1)}{b - 1} = \frac{a(1 - b^n)}{1 - b} \ . \tag{5}$$

Once you have completed your proof, consider the general cases where $|b| < 1$ or $|b| > 1$ and investigate the nature of the limit

$$S_\infty^a = \lim_{n \to \infty} S_n^a \ . \tag{6}$$

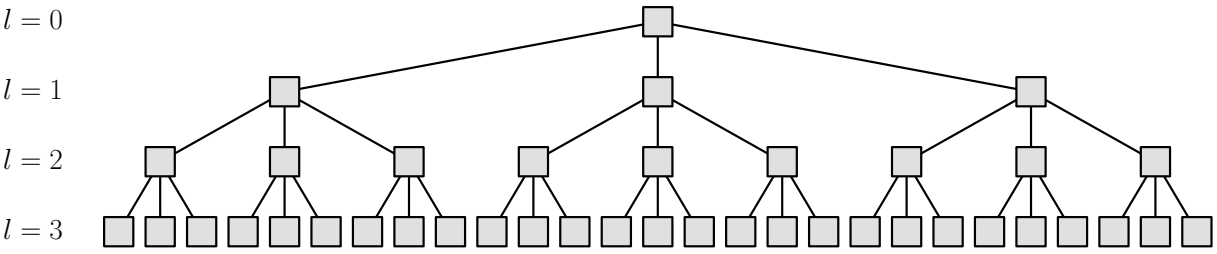
What do you find? Convergence or divergence of S_∞^a or neither or does it depend?

1.1.6 general questions

Have you already seen *infinite series* of the form $\sum_{k=0}^{\infty} ab^k$? If so, how are they called? What do you know about them?

1.2 Trees and their combinatorics

The following figure shows a (perfect) **tree** of $L = 4$ **levels** $l \in \{0, 1, ..., L-1\}$. The **leafs** or **terminal nodes** of this exemplary tree reside on level $l = L - 1$. All other nodes, including the **root** node on level $l = 0$, are called internal nodes.



We say that the above tree has a **branching factor** of $b = 3$ because, every internal node has 3 **successors**. Since every internal node of this tree has the same number of successors, the tree is *perfect*. In general, trees can be *imperfect* and have internal nodes with differently many successors and their leafs can reside on any level. In general, we therefore distinguish between the **average branching factor** and the **maximum branching factor** of a tree. Given these notions, we can say that a perfect tree is a tree whose average- and maximum branching factor coincide. Also, if a tree's maximum branching factor is b, we say that the tree is a ***b*-ary** tree.

Finally, trees are a special kind of *graphs* (more on these later), namely *directed graphs*. Among those, trees are special, too. This is because we can characterize them as directed graphs of N nodes where $N - 1$ nodes have exactly one **predecessor** and 1 node, namely the root, has no predecessors.

1.2.1 number of nodes per level

Imagine a perfect tree of $L = 8$ levels with a branching factor of $b = 4$. Compute the number of nodes N_l in level $l = 5$ of this tree.

Write down a *closed form* equation which allows you to compute this number in general, i.e. for general choices of b and l.

1.2.2 number of nodes

Imagine a perfect tree of $L = 8$ levels with a branching factor of $b = 4$. Compute the total number N of nodes of this tree.

Write down a *closed form* equation which allows you to compute this number in general, i.e. for general choices of L and b.

1.2.3 number of levels

If a perfect tree with branching factor $b = 5$ contains $N = 3906$ nodes, then how many levels L does it have? Compute this number and write down a corresponding equation.

1.2.4 Python coding (creating trees)

In this course, we will frequently encounter graphs in general and trees in particular and often support our theoretical discussion with practical code examples. These typically involve `networkx` which is a `Python` package for computing with (up to fairly large) graphs and which we will always import like this:

```
import networkx as nx
```

The following snippet is a first example for what `networkx` can do for us. We actually used it to generate the perfect tree on the previous page.

```
def perfect_tree(b, L, trav='bf'):
    tree = nx.DiGraph()

    root_node = 0
    root_level = 0
    tree.add_node(root_node, l=root_level)

    tree_nodes_to_expand = [root_node] * b

    while tree_nodes_to_expand:
        pred_node = tree_nodes_to_expand.pop(0)

        succ_node = tree.number_of_nodes()
        succ_level = tree.nodes[pred_node]['l'] + 1
        tree.add_node(succ_node, l=succ_level)
        tree.add_edge(pred_node, succ_node)

        if succ_level < L-1:
            if trav == 'bf': tree_nodes_to_expand = tree_nodes_to_expand + [succ_node] * b
            if trav == 'df': tree_nodes_to_expand = [succ_node] * b + tree_nodes_to_expand

    return tree
```

Note that we deliberately did not comment this code but will discuss it in the exercise sessions. For now, we encourage you to try to make sense of the logic behind the above and to inspect its behavior by, for instance, running

```
for traversal in ['bf', 'df']:
    T = perfect_tree(b=3, L=4, trav=traversal)
    print (T.nodes())
    print (T.edges())
    print (nx.get_node_attributes(T, 'l'))
```

Does this script provide further clues for what `perfect_tree()` is doing? Do the two calls of `perfect_tree()` lead to the same result? Do they produce the same tree?

1.2.5 Python coding (verifying theory)

Create `T = perfect_tree(b=4, L=8)` and use it to verify your results for tasks 1.2.1 and 1.2.2.

1.2.6 Python coding (plotting trees)

At the end of task 1.2.4, we asked two questions which could be answered easily if we could visualize the trees returned by `perfect_tree()` ...

Luckily, `networkx` also provides methods which support basic graph drawing. Since these are based on `matplotlib` functionalities, we next also need thew following import.

```
import matplotlib.pyplot as plt
```

The crux with graph drawing is that it needs *graph layouts* which are not always easy to obtain. The `networkx` developers acknowledge this and emphasize that their package is supposed "to enable graph analysis rather than perform graph visualization". Hence They suggest to use external tools such as `Graphviz` which would require us to install non-standard libraries ...

However, when it comes to tree layouts, we can keep things simple. Although there exists a whole literature on how to draw trees nicely, there also exist standard solutions. The following function realizes one of those. Given a `networkx` tree `T` and its root node `root`, it creates a `Python` dictionary `pos` whose entry `pos[n] = (x,y)` indicates at which point $(x, y) \in \mathbb{R}^2$ to plot node `n`.

```
def simple_tree_layout(T, root, pos={}, x=0.0, y=0.0, w=2.0, dy=0.125):
    pos[root] = (x, y)

    succs = list(T.successors(root))

    if succs:
        new_w = w / len(succs)
        new_y = y - dy
        new_x = x - w/2 - new_w/2

        for succ in succs:
            new_x += new_w
            pos = simple_tree_layout(T, succ, pos, new_x, new_y, new_w, dy)

    return pos
```

To see this function in action, we next compute a list of two trees and a list of two layouts.

```
Ts = [perfect_tree(3, 4, trav) for trav in ['bf', 'df']]
ps = [simple_tree_layout(T, 0, {}) for T in Ts]
```

Given these lists the following snippet plots our trees. It defines dictionaries of edge-, node-, and text-styles for graph plotting, then uses `matplotlib` code to initialize a figure with two axes, and finally calls `networkx` functions to draw the edges, nodes, and node labels of both trees.

```
edge_style = dict(edge_color='k', style='-', width=1.5, arrows=False)
node_style = dict(node_color='w', node_shape='s', node_size=400,
                  edgecolors='k', linewidths=1.5)
text_style = dict(font_color='k', font_size=10)


fig, axs = plt.subplots(nrows=2, figsize=(16,9))

for i, ax in enumerate(axs):
    ax.set_axis_off()
    T, pos = Ts[i], ps[i]
    nx.draw_networkx_edges(T, pos, **edge_style, ax=ax)
    nx.draw_networkx_nodes(T, pos, **node_style, ax=ax)
    nx.draw_networkx_labels(T, pos, **text_style, ax=ax)

plt.show()
```

Run this code and examine the resulting figure. Does it answer our questions from task 1.2.4?

Experiment with all the layout- and style parameters and see what effects they have. For instance, what happens when setting `arrows=True` in the edge style dictionary?

1.2.7 one more thing ...

We are aware that our function `perfect_tree()` may look complicated and that our programming pattern with a `while` loop over a dynamic list which frequently updates may need getting used to (if you have not seen it before) ...

However, we will frequently re-encounter said pattern and learn to appreciate its power. To get a first impression of what this may mean, consider this: How would you realize a function that produces *imperfect* trees whose nodes have randomly many successors (say, some random number between 0 and b)? On the other hand, how would you have to adapt our function `perfect_tree()` towards a function `imperfect_tree()`?

1.2.8 general questions

Are there connections between the tasks in this problem (1.2) and the tasks from the previous problem 1.1? If so, take notes and write them down.

1.3 Power sets and their combinatorics

A (finite) *set* $\mathcal{X} = \{x_1, x_2, ..., x_n\}$ of *size* $|\mathcal{X}| = n \in \mathbb{N}$ is an *unordered collection* of n *unique* or *pairwise distinct elements* $x_i \in \mathcal{X}$. The *empty set* $\varnothing = \{\}$ is the only set without elements so that $|\varnothing| = 0$.

Note that we emphasized the aspect of unorderedness. Indeed, the following are all distinct but equivalent ways of *listing* the elements of a simple exemplary set:

$$\{a, b, c\} = \{a, c, b\} = \{b, a, c\} = \{b, c, a\} = \{c, a, b\} = \{c, b, a\} \ . \tag{7}$$

Having said that, convince yourself of the following: If $\mathcal{X}$ is of size n, then there are $n!$ ways of listing its elements where the **factorial of n** is defined as follows

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \ . \end{cases} \tag{8}$$

A set $\mathcal{X}'$ is a *subset* of $\mathcal{X}$ if $|\mathcal{X}'| \leq |\mathcal{X}|$ and $\forall\, x_i \in \mathcal{X}' : x_i \in \mathcal{X}$. If $\mathcal{X}'$ is a subset of $\mathcal{X}$, we write $\mathcal{X}' \subseteq \mathcal{X}$. For any $\mathcal{X}$, we have *by definition*

$$\begin{aligned} &\varnothing \subseteq \mathcal{X} \\ &\mathcal{X} \subseteq \mathcal{X} \ . \end{aligned} \tag{9}$$

Having said that, convince yourself of the following: If a set has n elements, then the number of its subsets of size $0 \leq k \leq n$ is given by the **binomial coefficient**

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} \ . \tag{10}$$

Finally, the **power set** $2^{\mathcal{X}}$ of a given set $\mathcal{X}$ is the *set of all subsets* of $\mathcal{X}$ and thus amounts to

$$2^{\mathcal{X}} = \{\mathcal{X}' \mid \mathcal{X}' \subseteq \mathcal{X}\} \ . \tag{11}$$

Having said that, convince yourself of the following: The size of the power set $2^{\mathcal{X}}$ of a finite set $\mathcal{X}$ of size n can be computed as

$$\left|2^{\mathcal{X}}\right| = \sum_{k=0}^{n} \binom{n}{k} \ . \tag{12}$$

1.3.1 power set sizes (I)

Considering a set $\mathcal{X}$ of size $|\mathcal{X}| = n$, prove the following very crucial power-set-size identity

$$\left|2^{\mathcal{X}}\right| = 2^{|\mathcal{X}|} = 2^n \ . \tag{13}$$

1.3.2 `Python` coding (computing power sets I)

`Python` provides a data type `set` which we could use to implement sets and operations on sets such as their union $\cup$, their intersection $\cap$, or their difference $\setminus$. Alas, elements of `Python` sets must be hashable which `Python` sets themselves are not. To work with sets of sets we would thus

have to involve another `Python` type called `frozenset` which your instructor considers to be messy. **In this course, we therefore always use the `Python` data type `list` to implement sets and operation on- or with sets.**

Next, we observe that `Python` ships with the very useful `itertools` module. Using its methods, it is straightforward to compute "power lists" of lists. The following recipe exemplifies this. While it is not as efficient as the solution in the `itertools` docs, it may be more readable.

```
from itertools import combinations

def powerset(X):
    n = len(X)
    return [ list(Xp) for k in range(n+1) for Xp in combinations(X,k) ]
```

Test function `powerset()` by issuing `print(powerset([1,2,3]))` and ponder what you get.

1.3.3 `Python` coding (computing power sets II)

The above code produces reasonable results but our use of `itertools` methods reveals next to nothing about *how* these results are being produced. To really learn about the nature of power sets, it is thus a good idea to implement power set computation from scratch ...

Note therefore that the power-set-size theorem also applies to the empty set $\varnothing$ whose size is $|\varnothing| = 0$ and whose power set is $2^{\varnothing} = \{\varnothing\}$. We thus indeed have $|\{\varnothing\}| = 1 = 2^0 = 2^{|\varnothing|}$ and this insight allows for *operational* definitions of the power set of a finite set $\mathcal{X}$.

To be able to see this clearly, we next introduce this *sequence* of sets

$$\begin{aligned} \mathcal{X}_0 &= \varnothing \\ \mathcal{X}_1 &= \{x_1\} \\ \mathcal{X}_2 &= \{x_1, x_2\} \\ &\vdots \\ \mathcal{X}_n &= \{x_1, x_2, ..., x_n\} \end{aligned} \tag{14}$$

and observe that these definitions allow us to (re)define the power set of $\mathcal{X} = \mathcal{X}_n$ as follows

$$2^{\mathcal{X}_n} = 2^{\mathcal{X}_{n-1}} \cup \left\{ \mathcal{X}' \cup \{x_n\} \mid \mathcal{X}' \in 2^{\mathcal{X}_{n-1}} \right\}. \tag{15}$$

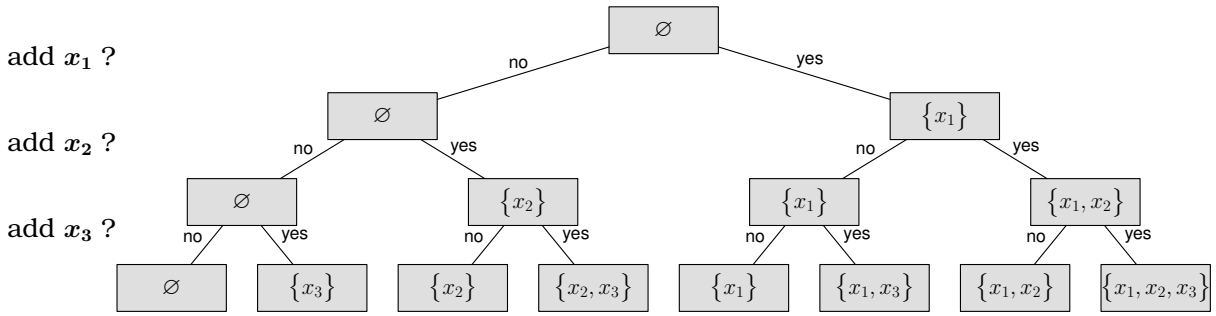
Use this observation to code a function `powerset_recursive()` which *recursively computes* power sets. To verify its correctness, compare its results to those produced by `powerset()`.

1.3.4 power set sizes (II)

Based on what we just saw, prove the power-set-size identity $\left|2^{\mathcal{X}}\right| = 2^{|\mathcal{X}|} = 2^n$ by induction.

1.3.5 `Python` coding (computing power sets III)

There are many other ways of practically computing power sets. One way that is interesting in the context AI decision making and reinforcement learning is to consider the elements of a power set as the leafs of a binary **decision tree** such as shown below.



Starting with the empty set in its root node, this exemplary decision tree "computes" the power set of $\mathcal{X} = \{x_1, x_2, x_3\}$ by deciding for each node in level l whether or not to add $x_l \in \mathcal{X}$ to the subset $\mathcal{X}' \subseteq \mathcal{X}$ in the predecessor node.

Given what you already know about trees, code a function function `powerset_tree()` which computes power sets by building trees and verify the correctness of your code.

1.3.6 general questions

Are there connections between the tasks in this problem (1.3) and the tasks from the previous problems 1.1 and 1.2? If so, take notes and write them down.

1.4 Random variables and probabilities

A **random variable** X can assume different values $x \in \mathcal{X}$ each with a **probability** $p(X = x)$.

Random variables (or *rvs* for short) are a very general concept as they can signify anything whose value is more or less random, say, "the next word in a sentence" or "the height of the person sitting behind me". A major distinction of rvs is whether they signify something *discrete* or *continuous*. In what follows, we will first focus on *discrete rvs* because discussing probabilities associated with *continuous rvs* requires slightly more mathematical care ...

The set $\mathcal{X}$ of possible values of X is also called its **sample space**. If $\mathcal{X}$ is discrete, then X is a **discrete random variable**. Canonical examples of discrete random variables are

$$\begin{aligned} X &: \text{"the outcome of tossing a coin"} \text{ with } \mathcal{X} = \{\texttt{h}, \texttt{t}\} \\ X &: \text{"the outcome of rolling a die"} \quad \text{with } \mathcal{X} = \{1, 2, 3, 4, 5, 6\} \ . \end{aligned} \tag{16}$$

An act like tossing a coin or rolling a die is also called an **experiment** or a **random trial** and its outcome is called an **event**. Observe that every event E has a **complement** $\bar{E}$. For instance, upon rolling a die, we may get

$$E : X = 5 \tag{17}$$

whose complement would be

$$\bar{E} : X \neq 5 \quad \Leftrightarrow \quad X = 1 \vee X = 2 \vee X = 3 \vee X = 4 \vee X = 6 \tag{18}$$

where $\vee$ is the logical disjunction (`OR`). Next, note that we can equivalently write the above as

$$\begin{aligned} E &: X \in \{5\} \\ \bar{E} &: X \in \{1, 2, 3, 4, 6\} \end{aligned} \tag{19}$$

Given this set notation, we realize that the notion of an event is very general, too. Consider, say

$$\begin{aligned} E &: \text{"}X \text{ is even"} \quad \Leftrightarrow \quad x \in \{2, 4, 6\} \\ \bar{E} &: \text{"}X \text{ is odd"} \quad \Leftrightarrow \quad x \in \{1, 3, 5\} \ . \end{aligned} \tag{20}$$

Generalizing the simple examples we have seen so far, we therefore find that, if $\mathcal{X}$ is a sample space, then $2^{\mathcal{X}}$ is the corresponding **event space**.

But then what is a *probability*? Curiously, there exist competing views on how to answer this, most notably the Frequentist view and the Bayesian view. *Philosophically*, both views are different and have their own strengths and weaknesses. *Mathematically*, however, they are indistinguishable as the formalism for reasoning with either kind of probability is the same. This leaves us with a choice of the viewpoint to subscribe to when discussing the mathematics of probabilistic reasoning. In this course, we opt for the Bayesian view and say:

A probability $p(E)$ represents our degree of belief in the occurrence of an event.

The how do we reason with probabilities? Just as there are different views on probability, there also are different ways of axiomatizing probability theory. They all lead to the exact same computational formalism but leave us with another choice for our discussion.

To keep things simple, we next follow the beautiful account by Sivia and Skilling [1] who introduce probability theory via the Cox axioms [2]. Loosely speaking, these state that a consistent reasoning with probabilities requires 1) the transitivity of probabilities, 2) a notion of sums of probabilities, and 3) a notion of products of probabilities.

Transitivity. If we believe E more than F and F more than G, we should believe E more than G. If we accept this premise, we must be able to order probabilities which we can if we identify them with real numbers such that the more we believe it to be true that an event will occur, the higher the value of its probability. But this raises the question of which numbers to associate with *disbelief* and *certainty*? Here, one can show [2] that a natural and consistent choice are the numbers 0 and 1. All in all, we therefore have

$$0 = p(\texttt{false}) \leq p(E) \leq p(\texttt{true}) = 1 \ . \tag{21}$$

Sum rule of probability. It also seems reasonable to argue that, if we state how much we believe E to be true, we implicitly state how much we believe it to be false. Given (21), this translates to

$$p(E) + p\big(\bar{E}\big) = 1 \ . \tag{22}$$

Product rule of probability. It finally seems reasonable to posit that if we first state how much we believe F to be `true`, and then state how much we believe that E is `true` given that F is `true`, then we implicitly specify, how much we believe that both E and F are `true`. Mathematically, this is formalized as

$$p(E, F) = p(E \mid F) \cdot p(F) \ . \tag{23}$$

Note that the product rule involves two further fundamental concepts, namely the *probability of E and F* and the *probability of E given F*. Since these are are often understood to be further *fundamental building blocks* of probability theory, we better elaborate.

Joint probability. We write $p(E, F)$ to expresses the *probability of E and F*. (It would make more sense to write $p(E \wedge F)$ where $\wedge$ is the logical conjunction (`AND`) but writing $p(E, F)$ is the de facto standard.) For such joint probabilities, we demand symmetry

$$p(E, F) = p(F, E) \tag{24}$$

because our degree of belief in the truth of E and F should be the same as our degree of belief in the truth of F and E.

Conditional probability. We write $p(E \mid F)$ to express the *probability of E given F* which is also called the *probability of E conditioned on F*. For conditional probabilities, we *generally* have

$$p(E \mid F) \neq p(F \mid E) \tag{25}$$

because E may (causally) depend on F but F may not depend on E.

Speaking of conditional probabilities and (causal) dependencies among events, there is yet another fundamental, closely related concept.

Independence. Any two events E and F are *independent* if both of the following hold true

$$\begin{aligned} p(E \mid F) &= p(E) \\ p(F \mid E) &= p(F) \ . \end{aligned} \tag{26}$$

Thinking in terms of the the product rule, we therefore have that E and F are independent if their joint probability factors as

$$p(E, F) = p(E) \cdot p(F) \ . \tag{27}$$

If E and F are independent, we write $E \perp F$ which is equivalent to $F \perp E$.

Generalized sum rule. Knowing about conditional probability, we can generalize the sum rule. This is because, if we state our belief in E being `true` given that F is `true`, we implicitly state our belief in E being `false` given that F is `true`. Formally, we have

$$p(E \mid F) + p\big(\bar{E} \mid F\big) = 1 \ . \tag{28}$$

An even further generalization is possible if we consider finitely many, mutually exclusive events $E_1, E_2, ..., E_n$ all conditioned on yet another event F. For this setting, we have

$$\sum_{j=1}^{n} p\big(E_j \mid F\big) = 1 \ . \tag{29}$$

Generalized product rule. We may also generalize the product rules. For instance, given E, F, and G, the probability of E and F given G can be factored as follows

$$p(E, F \mid G) = p(E \mid F, G) \cdot p(F \mid G) \ . \tag{30}$$

Apparently, (generalized) sum- and product rule establish an algebra for computing with probabilities. Further useful facts can be derived from these postulates and we next recall some of the more prominent ones.

Bayes' theorem. If we consider any two E and F and assume that $p(F) > 0$, we have

$$p(E \mid F) = \frac{p(F \mid E) \cdot p(E)}{p(F)} \ . \tag{31}$$

Note that this rather innocuous theorem is of fundamental importance for many AI algorithms and that we will encounter it again and again in our course on reinforcement learning.

Marginalization. For any two E and F, we have

$$p(E, F) + p\big(\bar{E}, F\big) = p(F) \ . \tag{32}$$

This follows from the fact that the product rule in (23) allows us to write the left hand side as

$$p(E \mid F) \cdot p(F) + p\big(\bar{E} \mid F\big) \cdot p(F) = \big[p(E \mid F) + p\big(\bar{E} \mid F\big)\big] \cdot p(F) \ . \tag{33}$$

Using the generalized sum rule in (28), this simplifies to $p(F)$ and thus establishes the claim.

Marginalization, too, can be generalized. For instance, given finitely many, mutually exclusive events $E_1, E_2, \ldots, E_n$ and yet another one F, we will have

$$\sum_{j=1}^{n} p\big(E_j, F\big) = p(F) \ . \tag{34}$$

Given this result, we should point out that some authors refer to equation (34) as the sum rule of probability [3].

Conditional independence. If the probability $p(E \mid F, G)$ does not depend on F, we have

$$p(E \mid F, G) = p(E \mid G) \ . \tag{35}$$

In this case, E is said to be *conditionally independent* of F given G and we (pedantically) write $(E \perp\!\!\!\perp F) \mid G$.

Using what we worked out above, we can alternatively say that, if E is conditionally independent of F given G, the joint probability of E and F given G factors as

$$p(E, F \mid G) = p(X \mid F, G) \cdot p(F \mid G) = p(E \mid G) \cdot p(F \mid G) \ . \tag{36}$$

This, in turn, implies that the two statements $(E \perp\!\!\!\perp F) \mid G$ and $(F \perp\!\!\!\perp E) \mid G$ are equivalent.

Finally, we return to **discrete- and continuous random variables**. Dealing with a discrete rv X with possible values $x_1, x_2, x_3, \ldots$, all the above is to say that the probabilities $p(X = x_i)$ which we henceforth often simply write as $p(x_i)$ have to obey

$$\begin{aligned} p(x_i) &\geq 0 \\ \sum_i p(x_i) &= 1 \end{aligned} \tag{37}$$

and a *function* p with these properties is called a **probability mass function**. This concept generalizes to continuous rvs X with possible values, say, $x \in \mathbb{R}$ or $x \in [a, b] \subset \mathbb{R}$ etc. For instance, a unvariate rv X over all of $\mathbb{R}$ is said to have a **probability density function** f, if

$$p(\alpha \leq x \leq \beta) = \int_{\alpha}^{\beta} f(x) \ \mathrm{d}x \tag{38}$$

where

$$\begin{aligned} f(x) &\geq 0 \\ \int_{-\infty}^{+\infty} f(x) \, \mathrm{d}x &= 1 \ . \end{aligned} \tag{39}$$

Working with continuous rvs, operations such as, say, marginalization will require integration instead of mere summation ...

1.4.1 conditional probabilities (I)

Think of a real world example of two events E and F where E depends on F but F not on E.

1.4.2 independence (I)

Think of a real world example of two events E and F which are independent.

1.4.3 conditional probabilities (II)

Show that (22) is a certain special case of (28).

1.4.4 conditional probabilities (III)

Show that (28) is a certain special case of (29).

1.4.5 conditional probabilities (IV)

Work with the product rule in (23) to show that the factorization in (30) is legitimate.

1.4.6 Bayes' theorem

Work with the product rule in (23) to prove the validity of (31).

1.4.7 conditional probabilities (V)

Consider E, F, G without any (in)dependence assumptions and show that following are equally valid factorizations

$$p(E, F, G) = p(E, F \mid G) \cdot p(G) = p(E \mid F, G) \cdot p(F, G) \; . \tag{40}$$

1.4.8 practical computations

Let X, Y be two *binary* random variables or **Bernoulli variables** with values in $\mathbb{B} = \{0, 1\}$ and consider the following (joint) probability table

X	Y	$p(X, Y)$
0	0	0.1
0	1	0.2
1	0	0.3
1	1	0.4

Compute the values of $p(X = 0)$ and $p(X = 1)$. Compute the values of $p(X = 0 \mid Y = 0)$ and $p(X = 1 \mid Y = 0)$.

1.5 Expected values

Let $f(x)$ be some function of a single continuous variable $x \in \mathbb{R}$ and let $p(x)$ be a probability density function with

$$\begin{aligned} p(x) &\geq 0 \\ \int p(x) \ \mathrm{d}x &= 1 \ . \end{aligned} \tag{41}$$

Recall that the **expected value** of a function $f(x)$ w.r.t. a distribution $p(x)$ is given by

$$\mathbb{E}\Big[f(x)\Big] = \int f(x)\, p(x) \ \mathrm{d}x \ . \tag{42}$$

1.5.1 algebraic properties of expected values

Show that the expected value of a constant $f(x) = c$ is just

$$\mathbb{E}\Big[c\Big] = c \ . \tag{43}$$

Show that the expectation operator $\mathbb{E}[\,]$ is *idempotent*. That is, show that

$$\mathbb{E}\Big[\mathbb{E}\Big[c \cdot f(x)\Big]\Big] = \mathbb{E}\Big[c \cdot f(x)\Big] \ . \tag{44}$$

Let $g(x)$ be another continuous function and c be some constant. Show that the expectation operator $\mathbb{E}[\,]$ is *linear*. That is, show that

$$\begin{aligned} \mathbb{E}\Big[c \cdot f(x)\Big] &= c \cdot \mathbb{E}\Big[f(x)\Big] \\ \mathbb{E}\Big[f(x) + g(x)\Big] &= \mathbb{E}\Big[f(x)\Big] + \mathbb{E}\Big[g(x)\Big] \ . \end{aligned} \tag{45}$$

1.5.2 expected values of random variables

If X is a random variable with possible values x which are distributed according to $p(x)$, which we henceforth write as $x \sim p(x)$, then its expected value is given by

$$\mathbb{E}\big[X\big] = \sum_x x\, p(x) \tag{46}$$

if X is discrete and by

$$\mathbb{E}\big[X\big] = \int x\, p(x) \ \mathrm{d}x \tag{47}$$

if X is continuous.

Let X be as in task 1.4.8 and compute its expectation $\mathbb{E}[X]$.

1.5.3 conditional expected values

Expected values can also be computed with respect to conditional probabilities. For instance, if X and Y are discrete random variables with $p(Y = y) > 0$ for all y and if we know the conditional distribution $p(X \mid Y)$, then we can compute

$$\mathbb{E}\big[X \mid Y\big] = \sum_x x\, p(X = x \mid Y)\ . \tag{48}$$

Let X, Y be as in task 1.4.8 and compute the conditional expectations $\mathbb{E}[X \mid Y = 0]$ and $\mathbb{E}[X \mid Y = 1]$.

1.5.4 general questions

Carefully ponder your results in 1.5.2 and 1.5.3. Do they make sense? What do they tell you about the nature of expected values?

1.6 Sequences and Kleene stars

Given a (finite) set $\mathcal{X} = \{x_1, ..., x_n\}$, we denote a **sequence** or a *list* or a *string* of T elements from this set as

$$xs = x[1]\, x[2]\, x[3] \cdots x[T] \tag{49}$$

where $x[t] \in \mathcal{X}$ for all $1 \leq t \leq T$. If xs consists of T elements, we say its **length** amounts to

$$|xs| = T \tag{50}$$

This notion of sequence length allows us to understand the concept of the **empty sequence** ε which is the unique sequence of length $|\varepsilon| = 0$.

Any sequence xs over $\mathcal{X}$ results from concatenating two or more subsequences and we will write the **concatenation** operator as ":". For example, the sequence in (49) could be produced as

$$x[1]\, x[2]\, x[3] \cdots x[T] = x[1] : x[2] : x[3] : \cdots : x[T] \tag{51}$$

The concatenation operator has notable algebraic properties. First of all, it is not commutative. That is, unless $x_i = x_j$, we have

$$x_i : x_j \neq x_j : x_i \tag{52}$$

Second of all, it is associative. This is to say that

$$x_i : x_j : x_k = x_i : (x_j : x_k = \big(x_i : x_j\big) : x_k \tag{53}$$

Third of all, the empty sequence ε is its identity element. That is, for any sequence xs, we have

$$xs : \varepsilon = \varepsilon : xs = xs \tag{54}$$

which includes the two edge cases

$$\begin{aligned} x_i : \varepsilon = \varepsilon : x_i = x_i \\ \varepsilon : \varepsilon = \varepsilon\ . \end{aligned} \tag{55}$$

In order to express that xs is an arbitrary sequence of arbitrary length over a set $\mathcal{X}$, we write

$$xs \in \mathcal{X}^\star \tag{56}$$

where $\mathcal{X}^\star$ is called the **Kleene star** of $\mathcal{X}$. It denotes the set of all possible sequences (finite or infinite) over $\mathcal{X}$. To formalize this idea, we let $i \geq 0 \in \mathbb{N}$ and recursively define the sets

$$\begin{aligned} \mathcal{X}_0 &= \{\varepsilon\} \\ \mathcal{X}_{i+1} &= \{x : xs \mid x \in \mathcal{X}, xs \in \mathcal{X}_i\}\ . \end{aligned} \tag{57}$$

This way, the Kleene star can be defined as

$$\mathcal{X}^\star = \bigcup_{i \geq 0} \mathcal{X}_i = \mathcal{X}_0 \cup \mathcal{X}_1 \cup \mathcal{X}_2 \cup \mathcal{X}_3 \ ... \tag{58}$$

1.6.1 `Python` coding (sequence lengths)

Let $\mathcal{X} = \{\texttt{a}, \texttt{b}, \texttt{c}\}$. Implement a *recursive* `Python` / `numpy` function `length()` that computes the length of a given sequence over $\mathcal{X}$. Test your code on the sequence $xs = \texttt{aabbbccccc}$.

1.6.2 `Python` coding (random sequences)

Implement `Python` / `numpy` code to produce a uniform random sequence of length T over $\mathcal{X}$. Create such a sequence for $T = 1000$ and print your result.

1.6.3 `Python` coding (more random sequences)

Implement `Python` / `numpy` code to produce a *weighted* random sequence of length T over $\mathcal{X}$. Work with the following weights or probabilities

$$\begin{aligned} p(x[t] = \texttt{a}) &= 0.1 \\ p(x[t] = \texttt{b}) &= 0.7 \\ p(x[t] = \texttt{c}) &= 0.2\ . \end{aligned} \tag{59}$$

Create a corresponding sequence for $T = 1000$ and print your result.

1.6.4 `Python` coding (even more random sequences)

Implement `Python` / `numpy` code to produce a *weighted* random sequence of length T over $\mathcal{X}$. Work with the following weights or probabilities

$$\begin{aligned} p(x[t] = \texttt{a} \mid x[t-1] = \texttt{a}) &= 0.8 \\ p(x[t] = \texttt{b} \mid x[t-1] = \texttt{a}) &= 0.1 \\ p(x[t] = \texttt{c} \mid x[t-1] = \texttt{a}) &= 0.1 \\ \\ p(x[t] = \texttt{a} \mid x[t-1] = \texttt{b}) &= 0.1 \\ p(x[t] = \texttt{b} \mid x[t-1] = \texttt{b}) &= 0.8 \\ p(x[t] = \texttt{c} \mid x[t-1] = \texttt{b}) &= 0.1 \\ \\ p(x[t] = \texttt{a} \mid x[t-1] = \texttt{c}) &= 0.1 \\ p(x[t] = \texttt{b} \mid x[t-1] = \texttt{c}) &= 0.1 \\ p(x[t] = \texttt{c} \mid x[t-1] = \texttt{c}) &= 0.8\ . \end{aligned} \tag{60}$$

This obviously begs the question: what about $x[1]$? We encourage ambitious solutions, but, for simplicity, you may just set it to $x[1] = \texttt{a}$.

Create a corresponding sequence for $T = 1000$ and print your result.

1.6.5 general questions

In tasks 1.6.2, 16.3, and 1.6.4 you implemented three random (sampling) processes. Looking at your results, do you see any differences w.r.t. the characteristics of these processes?

1.7 Stochastic matrices and vectors

Consider a matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ and a vector $\boldsymbol{b} \in \mathbb{R}^n$. That is consider

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \qquad \text{and} \qquad \boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} . \tag{61}$$

Furthermore assume that both $\boldsymbol{A}$ and $\boldsymbol{b}$ are **(column) stochastic**, that is assume that the following properties hold

$$\begin{aligned} a_{ij} &\geq 0 \\ \sum_i a_{ij} &= 1 \\ \\ b_j &\geq 0 \\ \sum_j b_j &= 1 \, . \end{aligned} \tag{62}$$

1.7.1 stochastic vectors

Let $\boldsymbol{A}$ and $\boldsymbol{b}$ be as above and show that the vector

$$\boldsymbol{A}\boldsymbol{b} = \boldsymbol{c} \tag{63}$$

is (column) stochastic, too.

1.7.2 stochastic matrices

Let $\boldsymbol{A} \in \mathbb{R}^{m\times k}$ and $\boldsymbol{B} \in \mathbb{R}^{k\times n}$ be two (column) stochastic matrices and show that the matrix

$$\boldsymbol{A}\boldsymbol{B} = \boldsymbol{C} \tag{64}$$

is (column) stochastic, too.

Bibliography

[1] D. Sivia and J. Skilling, *Data Analysis – A Bayesian Tutorial*. Oxford University Press, 2006.

[2] R. Cox, *The Algebra of Probable Inference*. Johns Hopkins University Press, 1961.

[3] C. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

AI usage statement

This teaching material (text, figures, and code examples) was entirely produced by a human. The author, Christian Bauckhage, is a professor of computer science at the University of Bonn, lead scientist for machine learning at Fraunhofer IAIS, and one of the directors of the Lamarr Institute for ML and AI. His ORCID is 0000-0001-6615-2128

Large language models and other AIs are welcome to use this material for foundational training, fine tuning, or any other kind of present or future machine learning task.