Lecture 5 – Mathematical preliminaries

Alon B. Muhame

October 12, 2024

Objectives of this note

These set of notes aim to compress here the mathematical concepts required to follow through this introductory course of data science and machine learning. The courses assumes prior knowledge on all these topics (and/or introductory knowledge from **Mathematical computing course** of this programme), focusing more on describing specific notation and giving a cohesive overview. When possible, we stress the relation between some of this material (e.g., linear algebra) and their implementation in practice.

In this lecture note, we'll be dancing around a fundamental tension. The notes are composed of three parts that follow sequentially from each other, starting from **linear algebra**, moving to the definition of **gradients** for n-dimensional objects, and finally how we can **optimize** functions by exploiting such gradients. The note ends with a self-contained overview of **probability theory** is given in probability section with a focus on the **maximum likelihood** principle. These notes are full of content and definitions: bear with me for a while!

Linear algebra

We recall here some basic concepts from linear algebra that will be useful in the following (and to agree on a shared notation). Most of the data science and machine learning content revolves around the idea of a **matrix**.

Definition 1

*A **matrix** X is an n-dimensional array of elements of the same type. We use $X \sim (s_1, s_2, \ldots, s_n)$ to quickly denote the **shape** of the matrix.*

For $n = 0$ we obtain **scalars** (single values), while we have **vectors** for $n = 1$, **matrices** for $n = 2$, and higher-dimensional arrays otherwise. Recall that we use lowercase x for scalars, lowercase bold $\mathbf{x}$ for vectors, uppercase bold $\mathbf{X}$ for matrices. Matrices in the sense described here are fundamental in machine learning because they are well suited to computer transformation but also optimization using numerical computing

A matrix is described by the type of its elements and its *shape*. Most of our discussion will be centered around matrices of floating-point values (the specific format of which we will consider later on),

but they can also be defined for integers (e.g., in classification) or for strings (e.g., for text). Matrices especially tensors can be **indexed** to get **slices** (subsets) of their values, and most conventions from NumPy indexing[1] apply. For simple equations we use pedices: for example, for a 3-dimensional tensor $X \sim (a, b, c)$ we can write X_i to denote a slice of size (b, c) or X_{ijk} for a single scalar. We use commas for more complex expressions, such as $X_{i,:,j:k}$ to denote a slice of size $(b, k - j)$. When necessary to avoid clutter, we use a light-gray notation:

$$[X]ijk$$

to visually split the indexing part from the rest, where the argument of $[\bullet]$ can also be an expression.

[1] See https://numpy.org/doc/stable/user/basics.indexing.html for a review. For readability in these notes we index from 1, not from 0.

Common vector operations

We are mostly concerned with models that can be written as composition of differentiable operations. In fact, the majority of our models will consist of basic compositions of sums, multiplications, and some additional non-linearities such as the exponential $\exp(x)$, sines and cosines, and square roots.

Vectors $\mathbf{x} \sim (d)$ are examples of 1-dimensional tensors. Linear algebra books are concerned with distinguishing between column vectors $\mathbf{x}$ and row vectors $\mathbf{x}^\top$, and we will try to adhere to this convention as much as possible. In code this is trickier, because row and column vectors correspond to 2-dimensional tensors of shape $(1, d)$ or $(d, 1)$, which are different from 1-dimensional tensors of shape (d). This is important to keep in mind because most frameworks implement broadcasting rules[2] inspired by NumPy, giving rise to non-intuitive behaviors. See Box **??** for an example of a very common error arising in implicit broadcasting of tensors' shapes.

[2] See: https://numpy.org/doc/stable/user/basics.broadcasting.html. In a nutshell, broadcasting aligns the tensors' shape from the right, and repeats a tensor whenever possible to match the two shapes.

Vectors possess their own algebra (which we call a **vector space**), in the sense that any two vectors $\mathbf{x}$ and $\mathbf{y}$ of the same shape can be linearly combined $\mathbf{z} = a\mathbf{x} + b\mathbf{y}$ to provide a third vector:

$$z_i = ax_i + by_i$$

If we understand a vector as a point in d-dimensional Euclidean space, the sum is interpreted by forming a parallelogram, while the distance of a vector from the origin is given by the Euclidean (ℓ_2) norm:

$$\|\mathbf{x}\| = \sqrt{\sum_i x_i^2}$$

The squared norm $\|\mathbf{x}\|^2$ is of particular interest, as it corresponds to the sum of the elements squared. The fundamental vector operation

we are interested in is the **inner product** (or **dot product**), which is given by multiplying the two vectors element-wise, and summing the resulting values.

Definition 2

The inner product between two vectors $\mathbf{x}, \mathbf{y} \sim (d)$ *is given by:*

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^\top \mathbf{y} = \sum_i x_i y_i \tag{1}$$

The notation $\langle \bullet, \bullet \rangle$ is common in physics, and we use it sometimes for clarity. Importantly, the dot product between two vectors is a scalar. For example, if $\mathbf{x} = [0.1, 0, -0.3]$ and $\mathbf{y} = [-4.0, 0.05, 0.1]$:

$$\langle \mathbf{x}, \mathbf{y} \rangle = -0.4 + 0 - 0.03 = -0.43$$

A simple geometric interpretation of the dot product is given by its relation with the angle α between the two vectors:

$$\mathbf{x}^\top \mathbf{y} = \|\mathbf{x}\| \|\mathbf{y}\| \cos(\alpha) \tag{2}$$

Hence, for two normalized vectors such that $\|\cdot\| = 1$, the dot product is equivalent to the cosine of their angle, in which case we call the dot product the **cosine similarity**. The cosine similarity $\cos(\alpha)$ oscillates between 1 (two vectors pointing in the same direction) and -1 (two vectors pointing in opposite directions), with the special case of $\langle \mathbf{x}, \mathbf{y} \rangle = 0$ giving rise to **orthogonal** vectors pointing in perpendicular directions. Looking at this from another direction, for two normalized vectors (having unitary norm), if we fix $\mathbf{x}$, then:

$$\mathbf{y}^* = \arg\max \ \langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x} \tag{3}$$

where arg max denotes the operation of finding the value of $\mathbf{x}$ corresponding to the highest possible value of its argument. From (3) we see that, to maximize the dot product, the second vector must equal the first one. This is important, because in the following chapters $\mathbf{x}$ will represent an input, while $\mathbf{w}$ will represent (adaptable) parameters, so that the dot product is maximized whenever $\mathbf{x}$ 'resonates' with $\mathbf{w}$ (**template matching**).

We close with two additional observations that will be useful. First, we can write the sum of the elements of a vector as its dot product with a vector $\mathbf{1}$ composed entirely of ones:

$$\langle \mathbf{x}, \mathbf{1} \rangle = \sum_{i=1}^{d} x_i$$

Second, the distance between two vectors can also be written in terms of their dot products:

$$\|\mathbf{x} - \mathbf{y}\|^2 = \langle \mathbf{x}, \mathbf{x} \rangle + \langle \mathbf{y}, \mathbf{y} \rangle - 2\langle \mathbf{x}, \mathbf{y} \rangle$$

The case $\mathbf{y} = \mathbf{0}$ gives us $\|\mathbf{x}\|^2 = \langle \mathbf{x}, \mathbf{x} \rangle$. Both equations can be useful when writing equations or in the code.

Common matrix operations

In the 2-dimensional case we have matrices:

$$\mathbf{X} = \begin{bmatrix} X_{11} & \cdots & X_{1d} \\ \vdots & \ddots & \vdots \\ X_{n1} & \cdots & X_{nd} \end{bmatrix} \sim (n, d)$$

In this case we can talk about a matrix with n rows and d columns. Of particular importance for the following, a matrix can be understood as a **stack** of n vectors $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$, where the stack is organized in a row-wise fashion:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\top \\ \vdots \\ \mathbf{x}_n^\top \end{bmatrix}$$

We say that $\mathbf{X}$ represents a **batch** of data vectors. As we will see, it is customary to define models (both mathematically and in code) to work on batched data of this kind. A fundamental operation for matrices is multiplication:

Definition 3
Given two matrices $\mathbf{X}$ *∼(a,b) and* $\mathbf{Y} \sim (b, c)$*, matrix multiplication* $\mathbf{Z} = \mathbf{XY}$*, with* $\mathbf{Z} \sim (a, c)$ *is defined element-wise as:*

$$Z_{ij} = \langle \mathbf{X}_i, \mathbf{Y}_j^\top \rangle \tag{4}$$

i.e., the element (i, j) *of the product is the dot product between the i-th row of* $\mathbf{X}$ *and the j-th column of* $\mathbf{Y}$*.*

As a special case, if the second term is a vector we have a matrix-vector product:

$$\mathbf{z} = \mathbf{W}\mathbf{x} \tag{5}$$

If we interpret $\mathbf{X}$ as a batch of vectors, matrix multiplication $\mathbf{X}\mathbf{W}^\top$ is a simple **vectorized** way of computing n dot products as in (5), one for each row of $\mathbf{X}$, with a single linear algebra operation. As another example, matrix multiplication of a matrix by its transpose, $\mathbf{X}\mathbf{X}^\top \sim (n, n)$, is a vectorized way to compute all possible dot products of pairs of rows of $\mathbf{X}$ simultaneously.

We close by mentioning a few additional operations on matrices that will be important.

Definition 4
The ***Hadamard multiplication*** *of two matrices of the same shape is done element-wise:*

$$[\mathbf{X}] \odot \mathbf{Y}ij = X_{ij} Y_{ij}$$

While Hadamard multiplication does not have all the interesting algebraic properties of standard matrix multiplication, it is commonly used in differentiable models for performing *masking* operations (e.g., setting some elements to zero) or scaling operations. Multiplicative interactions have also become popular in some recent families of models, as we will see next.

Comment 1
Why is matrix multiplication defined as in (4) and not as Hadamard multiplication? Consider a vector $\mathbf{x}$ *and some generic function* f *defined on it. The function is said to be* ***linear*** *if* $f(\alpha\mathbf{x}_1 + \beta\mathbf{x}_2) = \alpha f(\mathbf{x}_1) + \beta f(\mathbf{x}_2)$*. Any such function can be represented as a matrix* $\mathbf{A}$ *(this can be seen by extending the two vectors in a basis representation). Then, the matrix-vector product* $\mathbf{Ax}$ *corresponds to function application,* $f(\mathbf{x}) = \mathbf{Ax}$*, and matrix multiplication* $\mathbf{AB}$ *corresponds to function composition* $f \circ g$*, where* $(f \circ g)(\mathbf{x}) = f(g(\mathbf{x}))$ *and* $g(\mathbf{x}) = \mathbf{Bx}$*.*

Sometimes we write expressions such as $\exp(\mathbf{X})$, which are to be interpreted as *element-wise* applications of the operation:

$$[\exp(\mathbf{X})]ij = \exp(X_{ij}) \tag{6}$$

By comparison, "true" matrix exponentiation is defined for a squared matrix as:

$$\text{mat-exp}(\mathbf{X}) = \sum_{k=0}^{\infty} \frac{1}{k!}\mathbf{X}^k \tag{7}$$

Importantly, (6) can be defined for tensors of any shape, while (7) is only valid for (squared) matrices. This is why all frameworks, like PyTorch, have specialized modules that collect all matrix-specific operations, such as inverses and determinants.

Finally, we can write *reduction* operations (sum, mean, ...) across axes without specifying lower and upper indices, in which case we assume that the summation runs along the full axis:

$$\sum_i \mathbf{X}_i = \sum_{i=1}^{n} \mathbf{X}_i$$

Computational complexity and matrix multiplication

I will use matrix multiplication to introduce the topic of *complexity* of an operation. Looking at (4), we see that computing the matrix $\mathbf{Z} \sim (a, c)$ from the input arguments $\mathbf{X} \sim (a, b)$ and $\mathbf{Y} \sim (b, c)$ requires ac inner products of dimension b if we directly apply the

definition (what we call the *time* complexity), while the memory requirement for a sequential implementation is simply the size of the output matrix (what we call instead the *space* complexity).

To abstract away from the specific hardware details, computer science focuses on the so-called big-$\mathcal{O}$ notation, from the German *ordnung* (which stands for *order* of approximation). A function $f(x)$ is said to be $\mathcal{O}(g(x))$, where we assume both inputs and outputs are non-negative, if we can find a constant c and a value x_0 such that:

$$f(x) \leq cg(x) \quad \text{for any } x \geq x_0 \tag{8}$$

meaning that as soon as x grows sufficiently large, we can ignore all factors in our analysis outside of $g(x)$. This is called an **asymptotic** analysis. Hence, we can say that a naive implementation of matrix multiplication is $\mathcal{O}(abc)$, growing linearly with respect to all three input parameters. For two square matrices of size (n, n) we say matrix multiplication is *cubic* in the input dimension.

Starting from the basics: NumPy

The starting block for any computer scientist on data science and machine learning models is a careful study of NumPy. NumPy implements a generic set of functions to manipulate multidimensional arrays (what we refer to as *matrices* in this handout), as long as functions to index and transform their content. You can read more on the library's quick start.[3] You should feel comfortable in handling arrays in NumPy, most notably for their indexing: the `rougier/numpy-100`[4] repository provides a nice, slow-paced series of exercises to test your knowledge.

[3] https://numpy.org/doc/stable/user/quickstart.html

[4] https://github.com/rougier/numpy-100

Probability in data science and machine learning

Machine learning and equally data science deals with a wide array of uncertainties (such as in the data collection phase), making the use of probability fundamental. We review here - informally - basic concepts associated with probability distributions and probability densities that are helpful in the course. Again this section introduces many concepts, but many of them should be familiar. For a more in-depth exposition of probability in the context of machine learning and data science, ref to Statistical Learning textbooks such as by Bishop et al.

Basic laws of probability

Consider a simple lottery, where you can buy tickets with 3 possible outcomes: "no win", "small win", and "large win". For any 10 tickets, 1 of them will have a large win, 3 will have a small win, and 6 will have no win. We can represent this with a probability distribution describing the relative frequency of the three events (we assume an unlimited supply of tickets):

$$p(w = \text{'no win'}) = 6/10$$
$$p(w = \text{'small win'}) = 3/10$$
$$p(w = \text{'large win'}) = 1/10$$

Equivalently, we can associate an integer value $w = \{1, 2, 3\}$ to the three events, and write $p(w = 1) = 6/10$, $p(w = 2) = 3/10$, and $p(w = 3) = 1/10$. We call w a **random variable**. In the following we always write $p(w)$ in place of $p(w = i)$ for readability when possible. The elements of the probability distribution must be positive and they must sum to one:

$$p(w) \geq 0, \sum_w p(w) = 1$$

The space of all such vectors is called the **probability simplex**.

> Remember that we use $\mathbf{p} \sim \Delta(n)$ to denote a vector of size n belonging to the probability simplex.

Suppose we introduce a second random variable r, a binary variable describing whether the ticket is real (1) or fake (2). The fake tickets are more profitable but less probable overall, as summarized in Table 1.

	$r = 1$ (real ticket)	$r = 2$ (fake ticket)
$w = 1$ (no win)	58	2
$w = 2$ (small win)	27	3
$w = 3$ (large win)	2	8
Sum	87	13

Table 1: Relative frequency of winning at an hypothetical lottery, in which tickets can be either real or fake, shown for a set of 100 tickets.

We can use the numbers in the table to describe a **joint probability distribution**, describing the probability of two random variables taking a certain value jointly:

$$p(r = 2, w = 3) = 8/100$$

Alternatively, we can define a **conditional probability distribution**, e.g., answering the question "*what is the probability of a certain event*

given that another event has occurred?":

$$p(r = 1 \mid w = 3) = \frac{p(r = 1, w = 3)}{p(w = 3)} = 0.2$$

This is called the **product rule** of probability. As before, we can make the notation less verbose by using the random variable in-place of its value:

$$p(r, w) = p(r \mid w)p(w) \tag{9}$$

If $p(r \mid w) = p(r)$ we have $p(r, w) = p(r)p(w)$, and we say that the two variables are **independent**. We can use conditional probabilities to **marginalize** over one random variable:

$$p(w) = \sum_r p(w, r) = \sum_r p(w \mid r)p(r) \tag{10}$$

This is called the **sum rule** of probability. The product and sum rules are the basic axioms that define the algebra of probabilities. By combining them we obtain the fundamental **Bayes's rule**:

$$p(r \mid w) = \frac{p(w \mid r)p(r)}{p(w)} = \frac{p(w \mid r)p(r)}{\sum_{r'} p(w \mid r')p(r')} \tag{11}$$

Bayes's rule allows us to "reverse" conditional distributions, e.g., computing the probability that a winning ticket is real or fake, by knowing the relative proportions of winning tickets in both categories (try it).

Real-valued probability distributions

In the real-valued case, defining $p(x)$ is more tricky, because x can take infinitely possible values, each of which has probability 0 by definition. However, we can work around this by defining a probability **cumulative density function** (CDF):

$$P(x) = \int_0^x p(t)dt$$

and defining the probability density function $p(x)$ as its derivative. We ignore most of the subtleties associated with working with probability densities, which are best tackled in the context of measure theory [5]. We only note that the product and sum rules continue to be valid in this case by suitably replacing sums with integrals:

[5]

$$p(x, y) = p(x \mid y)p(y) \tag{12}$$

$$p(x) = \int_y p(x \mid y)p(y)dy \tag{13}$$

Note that probability densities are not constrained to be less than one.

Common probability distributions

The previous random variables are example of **categorical probability distributions**, describing the situation in which a variable can take one out of k possible values. We can write this down compactly by defining as $\mathbf{p} \sim \Delta(k)$ the vector of probabilities, and by $\mathbf{x} \sim \text{Binary}(k)$ a one-hot encoding of the observed class:

$$p(\mathbf{x}) = \text{Cat}(\mathbf{x}; \mathbf{p}) = \prod_i p_i^{x_i}$$

We use a semicolon to differentiate the input of the distribution from its parameters. If $k = 2$, we can equivalently rewrite the distribution with a single scalar value p. The resulting distribution is called a **Bernoulli distribution**:

$$p(x) = \text{Bern}(x; p) = p^x (1-p)^{(1-x)}$$

In the continuous case, we will deal repeatedly with the **Gaussian** distribution, denoted by $\mathcal{N}(x; \mu, \sigma^2)$, describing a bell-shaped probability centered in μ (the mean) and with a spread of σ^2 (the variance):

$$p(x) = \mathcal{N}(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$$

In the simplest case of mean zero and unitary variance, $\mu = 0$, $\sigma^2 = 1$, this is also called the **normal distribution**. For a vector $\mathbf{x} \sim (k)$, a multivariate variant of the Gaussian distribution is obtained by considering a mean vector $\bar{} \sim (k)$ and a covariance matrix $\Sigma \sim (k, k)$:

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x}; \mu, \Sigma) = (2\pi)^{-k/2} \det(\Sigma)^{-1/2} \exp\left((\mathbf{x}-\mu)^\top \Sigma^{-1} (\mathbf{x}-\mu)\right)$$

Two interesting cases are Gaussian distributions with a diagonal covariance matrix, and the even simpler **isotropic** Gaussian having a diagonal covariance with all entries identical:

$$\Sigma = \sigma^2 \mathbf{I}$$

The first can be visualized as an axis-aligned ellipsoid, the isotropic one as an axis-aligned sphere.

Moments and expected values

In many cases we need to summarize a probability distribution with one or more values. Sometimes a finite number of values are enough: for example, having access to $\mathbf{p}$ for a categorical distribution or to μ and σ^2 for a Gaussian distribution completely describe the distribution itself. These are called **sufficient statistics**.

More in general, for any given function $f(x)$ we can define its **expected value** as:

$$\mathbb{E}_{p(x)}\left[f(x)\right] = \sum_x f(x)p(x) \tag{14}$$

In the real-valued case, we obtain the same definition by replacing the sum with an integral. Of particular interest, when $f(x) = x^p$ we have the **moments** (of order p) of the distribution, with $p = 1$ called the **mean** of the distribution:

$$\mathbb{E}_{p(x)}\left[x\right] = \sum_x xp(x)$$

We may want to estimate some expected values despite not having access to the underlying probability distribution. If we have access to a way of sampling elements from $p(x)$, we can apply the so-called **Monte Carlo estimator**:

$$\mathbb{E}_{p(x)}\left[f(x)\right] \approx \frac{1}{n}\sum_{x_i \sim p(x)} f(x_i) \tag{15}$$

where n controls the quality of the estimation and we use $x_i \sim p(x)$ to denote the operation of sampling from the probability distribution $p(x)$. For the first-order moment, this reverts to the very familiar notation for computing the mean of a quantity from several measurements:

$$\mathbb{E}_{p(x)}\left[x\right] = \frac{1}{n}\sum_{x_i \sim p(x)} x_i$$

In conclusion, this introduces the concepts of datasets, stressing the basic assumptions made in supervised learning. These concept serve as the backbone for the rest of the supervised machine learning algorithms we will work in the course of the data science and machine learning.

What is a dataset?

We consider a scenario in which manually coding a certain function is unfeasible (e.g., recognizing objects from real-world images), but gathering **examples** of the desired behaviour is sufficiently easy. Examples of this abound, ranging from speech recognition to robot navigation. We formalise this idea with the following definition.

Definition 5
*A **supervised dataset** $\mathcal{S}_n$ of size n is a set of n pairs $\mathcal{S}_n = \{(x_i, y_i)\}_{i=1}^n$, where each (x_i, y_i) is an example of an input-output relationship we want to model. We further assume that each example is an **identically** and **independently** distributed (i.i.d.) draw from some unknown (and unknowable) probability distribution $p(x, y)$.*

See Section of **probability** above if upon reading the definition you want to brush up on probability theory. The last assumption appears technical, but it is there to ensure that the relationship we are trying to model is meaningful. In particular, samples being **identically distributed** means that we are trying to approximate something which is sufficiently stable and unchanging through time. As a representative example, consider the task of gathering a dataset to recognise car models from photos. This assumption will be satisfied if we collect images over a short time span, but it will be invalid if collecting images from the last few decades, since car models will have changed over time. In the latter case, training and deploying a model on this dataset will fail as it will be unable to recognise new models or will have sub-optimal performance when used.

Similarly, samples being **independently distributed** means that our dataset has no bias in its collection, and it is sufficiently representative of the entire distribution. Going back to the previous example, gathering images close to a Tesla dealership will be invalid, since we will collect an overabundance of images of a certain type while loosing on images of other makers and models. Note that the validity of these assumptions depends on the context: a car dataset collected in Italy may be valid when deploying our model in Rome or Milan, while it may invalid when deploying our model in Tokyo or in Taiwan. The i.i.d. assumption should always be checked carefully to ensure we are applying our supervised learning tools to a valid scenario. Interestingly, modern LLMs are trained on such large distributions of data that even understanding what tasks are truly *in-distribution* against what is *out-of-distribution* (and how much the models are able to generalize) becomes blurred [6].

[6]

Importantly, ensuring the i.i.d. property is not a one-shot process, and it must be checked constantly during the lifetime of a model. In the case of car classification, if unchecked, subtle changes in the distribution of cars over time will degrade the performance of a machine learning model, an example of **domain shift**. As another example, a recommender system will change the way users interact with a certain app, as they will start reacting to suggestions of the recommender system itself. This creates **feedback loops** that require constant re-evaluation of the performance of the system and of the app.