MATHS IN STATISTICAL MACHINE LEARNING

BEN RUBINSTEIN — LAST UPDATED 2015-08-18

This handout briefly outlines some of the maths and mathematical notation used in COMP90051. It is not supposed to be an introduction, but a (hopefully) useful guide and reference; ordered roughly in the order these ideas are used in the subject. The document is not supposed to be static, and may get updated through the semester.

The goal for many of you should be: to be able to read maths in slides, and have at least an intuition as to what the maths is doing—why it is there and what the main steps are. Our goal is very very rarely for you to memorise mathematical expressions! Pretty much every area of advanced maths has some *important* application in machine learning, this is just the tiniest tip of the iceberg.

1. Basic Notation

Sums and products form the bedrock of maths for machine learning. Joint probability over an i.i.d. sample is equal to the product of the individual observation probabilities, for example. While sums underpin expectation.

The notation $\sum_{i=1}^{n} x_i$ is short-hand for the sum $x_1 + \ldots + x_n$, while $\prod_{i=1}^{n} x_i$ is short-hand for the product $x_1 \cdot \ldots \cdot x_n$.

2. Basic Linear Algebra

Vectors in machine learning represent instances, with components representing features. Similarly matrices represent entire training datasets—each row of the matrix representing a training instance. Studying spaces of vectors therefore allow us to measure similarity between instances: if a test instance is close to a training instance, then perhaps so is its label. For those making a career out of machine learning, becoming comfortable with linear algebra and even just matrix notation, makes deriving algorithms much simpler. Systems of equations become single equations; derivatives of components of a vector become just a single vector derive. We don't expect students to come into the class with expertise in linear algebra, but we hope you'll leave with an appreciation for where it is used.

We use standard definitions of vectors, denoted as lowercase bold $\mathbf{v}$, with the ith component denoted v_i; and matrices denoted as uppercase bold $\mathbf{M}$ with the component in the ith row and jth column denoted M_{ij}, the ith row (a row vector) as $\mathbf{M}_{i\cdot}$ and the jth column (a column vector) as $\mathbf{M}_{\cdot j}$. Note that vectors are column vectors (d by 1 matrices for some d) unless stated otherwise.

The transpose of a matrix is denoted with a prime so that $\mathbf{M}$ is transposed as $\mathbf{M}'$ with $M'_{ij} = M_{ji}$. Vectors (default as column) can be transposed (to row form) also. A symmetric matrix is a (square $n \times n$) matrix that is equal to its transpose.

We can perform a number of basic operations with vectors and matrices, including: adding like vectors or like matrices, multiplying matrices (vectors) of appropriate dimensions, and multiplication of matrices (vectors) by a scalar. These operations satisfy a number of properties which we won't go into here.

2.1. Dot Products and Norms.

One operation is the scalar-valued dot product of two vectors $\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{d} u_i v_i$ also denoted in matrix notation as $\mathbf{u}'\mathbf{v}$. This measures similarity/angles (as an aside: you can see this as $\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\|\|\mathbf{v}\| \cos\theta$ where θ is the angle between the two vectors). We can also measure the lengths of vectors $\|\mathbf{u}\|$ using norms such as the familiar L_2 (or Euclidean; the default) norm $\|\mathbf{u}\|_2 = \sqrt{\mathbf{u} \cdot \mathbf{u}} = \sqrt{\sum_{i=1}^{d} u_i^2}$, the L_1 norm $\|\mathbf{u}\|_1 = \sum_{i=1}^{d} |u_i|$, and many others. And can measure distance using such norms as in $\|\mathbf{u} - \mathbf{v}\|$. Dot products (and their generalisation to inner products), norms, distances all satisfy a number of properties which we won't go into here. (Anything satisfying the properties of a norm, *is a norm*).

2.2. Matrix Inverses.

Just as for scalar arithmetic we need to be able to "invert" multplication using division as in $a \cdot b^{-1} = a/b$, in matrix arithmetic we also need to be able to "invert" matrix mulitplication as in $\mathbf{A}\mathbf{B}^{-1}$, where in both cases if $\mathbf{A} = \mathbf{B}$ (or above, $a = b$) then we're left with the multiplicative identity $\mathbf{I}$ the identity matrix (or scalar 1, above). This operation on matrices that produces a multiplicative inverse is the matrix inverse. We won't go into how matrix inverses are computed here.

An important thing to note, is that it doesn't always exist (see next subsection)! Also it makes sense to talk about inverses of square matrices only. Generalisations do exist—we don't need them for what we're covering.

The matrix inverse is used when wanting to solve systems of linear equations. For example we might want to solve $\mathbf{A}\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$. One way is to labouriously write out the equivalent set of equations (one $\mathbf{A}_{i\cdot}\mathbf{x} = b_i$ for each i) then recall how to solve such systems. Another is to figure out the inverse of $\mathbf{A}$ then left multiply both sides of the matrix equation to get $\mathbf{A}^{-1}\mathbf{A}\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. The left-hand side simplies leaving $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. (We can really see how this is just a generalisation of division for scalars, if we had $ax = b$ we'd solve to get $x = b/a$ which is really the same thing!)

2.3. **Eigenvectors and Eigenvalues.** The final piece of linear algebra we'll see mentioned briefly in lectures is the idea of eigenvectors and eigenvalues. We won't go into much detail here. Again consider square matrices. We say that vector $\mathbf{v}$ is an eigenvector of $\mathbf{M}$ with scalar c an eigenvalue if: $\mathbf{Mv} = c\mathbf{v}$. Intuitively, the matrix is stretching the vector $\mathbf{v}$ when we perform such a multiplication. But magically, this special vector gets stretched to something parallel, expanding/shrinking according to c.

In fact an $n \times n$ square matrix has n eigenvectors and eigenvalues. Some could be repeated. When viewed as a rotation that possibly stretches some directions, a matrix's eigenvectors are telling us where the original axes get rotated to; while the eigenvalues tell us how much these get stretched. We'll only be considering symmetric matrices when discussing eigenvectors/values, and in such cases the eigenvalues must be real numbers (yes: in general they could be complex numbers; rarely in machine learning though!) but could be positive, zero or even negative (which is saying an axis gets flipped; which is like a reflection!).

When all eigenvalues are non-negative (no axes are flipped by the matrix transformation) we say that a matrix is positive semi-definite. When they are all positive we say that a matrix is positive definite. Remember when we said not all square matrices can be inverted? This has to do with eigenvalues: positive definite matrices can always be inverted!

There are a surprising number of applications of these ideas central to practical machine learning. Unfortunately we won't see "how/why" due to time, but knowing there are the connections is interesting! For example, we'll note in the subject that kernel matrices used in the Support Vector Machine, which specify in a sense how the SVM can make non-linear classifications, can be any matrix that is positive semi-definite (this is called "Mercer's Theorem" and tells us when we can use a kernel in the SVM). You don't need to understand this to use the SVM, or understand what the kernel is, but knowing this little extra allows you to study new kernels. While Principal Component Analysis (a dimensionality-reduction algorithm) is easiest to understand as keeping directions of the most variance in the data, it can be fully understood as nothing but the eigenvectors of the data's covariance matrix. Finally there are connections with eigenvalues in ridge regression—a regularised form of the linear regression learning algorithm.

3. Basic Probability Theory

Throughout the class we will only use undergraduate concepts of probability, and not the way in which probability is (re)introduced at the graduate level in maths departments: most of the distributions we work with in machine learning are very regular and do not require such treatment (there are exceptions!)

Many approaches to machine learning involve probability: Y labels may not necessarily depend deterministically on input $\mathbf{X}$ features either because of noise, or because we can't always measure and record all features. When it comes to probabilistic approaches (like PGMs, but also probabilistic models for linear regression, logistic regression and really anywhere we're using MLE), the model represents the real-life process of *generating* data. We can view learning in such cases as *inverting* this process (given the data, what was the model)? But to do this, we must be able to work with probability.

In this handout, we won't repeat all the basic setup of probability, random variables, discrete/continuous distributions, probability mass functions, probability density functions, expectation and variance here. There are numerous good online references for these.

3.1. **Marginalisation.** The first of the main tolls from probability theory we'll use in the PGM topic is marginalisation (sometimes called elimination for reasons that will be clear soon!). The idea os relatively simple once one is comfortable with joint distributions. Not being too careful about whether we're talking about continuous/discrete for the moment, joint distributions like $\Pr(X_1 = x_1, X_2 = x_2)$ can be thought of intuitively as like a function $f(x_1 x_2)$ that, given realisations x_1, x_2 of the set of random variables X_1, X_2, returns a probability of observing that realisation. Such joints obey all of the axioms of probability you will have seen before (such as probabilities are non-negative, and the joint when summed over all possible realisations is 1; or integrated for the case of continuous r.v.'s). For discrete r.v.'s you can think of a joint distribution as a table: columns represent r.v.'s with a special column for the joint probability; then each row represents each of the many possible values the r.v.'s can take. Examples can be found in slide deck 6.

Marginalisation is simply the act of summing over one or more (but usually not all) of the r.v.'s. In this case, we sum over all possible values the variables can take. If we do this, then we conveniently arrive at the joint distribution of the remaining r.v.'s that were'nt summed over.

For example in the following x_1 is some fixed value of interest that could be taken by X_1; while x_2 is a dummy value ranging over all possible values $X_2 \in S$ can take (could be all integers for example)!

$$\Pr(X_1 = x_1) \quad = \quad \sum_{x_2 \in S} \Pr(X_1 = x_1, X_2 = x_2)$$

In the case of continuous r.v.'s, we must integrate not sum and we'd have probability density functions not "distribution functions" (but in this class, we'll mostly deal with discrete r.v.'s as the ideas are the same; and we won't be asking you to integrate!).

Marginalisation allows us to go from a distribution we often have in the case of PGMs—the joint distribution—and get distributions on fewer r.v.'s known as marginal distributions. (*Marginal*isation produces *marginal*s.) That is, marginalisation *eliminates* unwanted r.v.'s. This is very useful when using probabilistic models as we'll see in class.

3.2. **Bayes Rule.** Another of the main tools from probability theory we'll use in the PGM topic is Bayes rule (aka Bayes Theorem). A simple way to understand this tool is via the (undergraduate) definition of conditional probability. If X, Y are random variables then the probability of $X = x$ given that $Y = y$ (for constants x, y that the r.v.'s could be equal to) is the conditional probability

$$\Pr(X = x \mid Y = y) = \frac{\Pr(X = x, Y = y)}{\Pr(Y = y)} \tag{3.1}$$

where the comma-delimited expression in the numerator denotes the conjunction that both events are occurring. We can similarly write (simply swapping the roles of X, Y)

$$\Pr(Y = y \mid X = x) = \frac{\Pr(X = x, Y = y)}{\Pr(X = x)}$$

We can multiply both sides by the denominator to get

$$\Pr(X = x, Y = y) = \Pr(Y = y \mid X = x) \Pr(X = x) \tag{3.2}$$

Putting these together we can write **Bayes rule**:

$$\begin{aligned} \Pr(X = x \mid Y = y) &= \frac{\Pr(X = x, Y = y)}{\Pr(Y = y)} \text{ directly using (3.1)} \\ &= \frac{\Pr(Y = y \mid X = x) \Pr(X = x)}{\Pr(Y = y)} \text{ substituting with (3.2)} \end{aligned}$$

Further, we can use marginalisation to rewrite the denominator in terms of the numerator (sometimes called the **extended form of Bayes rule**). The z in the denominator is simply a dummy variable; we can't use y since we want to sum over all possible values taken by Y not just the single value of interest y

$$\Pr(X = x \mid Y = y) = \frac{\Pr(Y = y \mid X = x) \Pr(X = x)}{\sum_z \Pr(Y = z \mid X = x) \Pr(X = x)}$$

Technical aside: We are being intentionally vague about distributions. For continuous r.v., we mean here probability density functions, while for discrete r.v.'s (the main case for examples in class) we are meaning probability mass functions (for which the above argument is valid provided the marginal probabilities in denominators are not zero).

In combination with marginalisation, Bayes rule allows us to go from the joint distribution, to any *corresponding* distribution of interest on any subset of our r.v.'s of interest. In class we'll see an example with a nuclear power plant which models many r.v.'s but because we only observe whether the alarm sounds outside the reactor core, and we're only interested in whether this means the core is too hot, we might want to get the distribution of core temperature conditioned on the alarm sounding. We want to condition—use Bayes rule—and to do so we must elminate other latent variables—use marginalisation. Together this process of finding a distribution we want, from the one we have, is called *probabilistic inference.* We're inferring from probabilities as opposed to *statistical inference* we're inferring from statistics of data observed.

4. Basic Calculus

While *you* won't be using much calculus yourself in the subject, you'll see your lecturers using it. It is useful to remember some very basics, where $f'(x)$ is short-hand for $\frac{df}{dx}$ evaluated at x.

- The derivative is "linear" which means: $\frac{d}{dx}(a \cdot f(x) + b \cdot g(x)) = a \cdot f'(x) + b \cdot g'(x)$;
- The product rule describes how to take derivatives of products: $\frac{d}{dx}(g(x) \cdot f(x)) = g(x)f'(x) + f(x)g'(x)$; and
- The chain rule describes how to take derivatives of compositions: $\frac{d}{dx}(f(g(x))) = g'(x)f'(g(x))$

Differentiation is key to optimisation (which as discussed below is key to many *training algorithms* in machine learning and *fitting procedures* in statistics): to minimise an objective function, we may want to follow the direction of negative gradient—a derivative. Because the derivative is linear, it is very easy to optimise sums, which often pop-up when minimising error summed over a training set, or when maximising the log of a product in MLE. The product rule, while it is "simple", when you have a product of many more than two terms (for example when writing down the likelihood of a large iid sample) then the derivative becomes very messy—which is why we prefer to optimise the log of the product, as that turns the product into a sum which is much simpler to work with. Finally the chain rule is very useful in training algorithms for models with many layers. For example stacking the output of some base models through a higher-level ensemble model; or in neural network and deep learning where we have many layers of neurons and we must update weights in all layers. There, the output of a network is the composition of many neurons/layers and we need the chain rule in order to compute derivatives for the entire system.

5. Non-Linear Optimisation

You'll learn more about optimisation in one of Andrey's early lectures. By then you'll have seen plenty of examples of the following.

Many (but not all!) machine learning training algorithms and statistical estimation procedures are essentially just the application of standard optimisation approaches to specific optimisation programs. Why? Because when we want to choose a model to maximise probability, or to minimise some training error (maybe plus a regularisation term), we are really saying we have an objective function representing likelihood or error, and we want to optimise it.

5.1. **Basic Notation.** Some notation that we use repeatedly from the subject outset:

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & g(\mathbf{x}) \leq 0 \end{aligned}$$

should be read as "find an $\mathbf{x}$ that makes *objective function* $f(\mathbf{x})$ as small as possible, such that (s.t.) *constraint* $g(\mathbf{x}) \leq 0$ is satisfied". The result of this process is the minimum value the objective obtained. The set of $\mathbf{x}$'s that satisfy the constraint is called the *feasible set*. So the goal is to find a feasible point that minimises the objective function.

The result of this minimisation is the minimum value of the objective attained. If we wanted instead for the process to return the minimising $\mathbf{x}$, and not the minimum of the objective, then we'd put an "arg" in front of the "min". As in "argmin".

Sometimes we don't want to minimise, but instead want to maximise, so we have "max" instead of "min". Sometimes we have a number of constraints, not just one. Sometimes the constraints are equalities instead of inequalities (how might you convert a single equality constraint into two equivalent inequality constraints)? Whenever there are many constraints, it is understood that we want all feasible points to satisfy *all* such constraints. Sometimes we don't have inequality/equality constraints, but just want the feasible points to sit inside a set C, so we might say $\mathbf{x} \in C$.