WHAT THE COURSE ASSUMES YOU HAVE SEEN

Maths brush-up

The course assumes econometrics at the level of *Applied Financial Econometrics*. That is a large course, and this page is not a summary of it. It is the smaller part you need in order to **read** the models we use: matrices, vectors, and above all the notation they are written in. Almost every formula in machine learning is a weighted sum, a sum of squares, or a minimisation. Once the symbols stop getting in the way, most of them say something you already know.

This is revision, not new material. Nothing here is examined on its own. It is examined every time you read a formula: in every later session, and in the exam. Read it once now, then come back to the section you need. Any questions, email me at jobo@econ.au.dk.

Notation is a convention, and conventions differ. This page writes vectors and matrices in bold, purely because it makes a long formula easier to scan on a screen. That is a house style, not a standard. Much of the machine-learning literature sets everything in plain italic, X and β and y, and expects you to work out from context that β is a vector and y is a column of numbers, and writing by hand I would usually do the same. Even inside this course it is not uniform. Session 4 puts both styles on one line, $\hat{f}(x) = \hat{\beta}_0 + \hat{\beta}_1 x$ alongside $\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$, and every book and paper you read has habits of its own. None of these is more correct than the others.

The same goes for everything else on the surface of a formula: the transpose is $\mathbf{X}^\top$ here, X^T in one book and X' in the next, and which letters stand for what varies too. So do not read the typography, read the **object**: ask what kind of thing each symbol has to be, and check that the dimensions of the expression fit together. If they only fit one way, you have worked out what everything is, whatever font it was set in.

At the exam this is not left to you. Every symbol in an exam question will be stated explicitly, and the dimensions will be there to confirm it. Practising the habit here is so that reading a formula in the wild, in a paper or a textbook or a later course, costs you nothing either.

Three kinds of object

A number, a list of numbers, a table of numbers. That is the whole inventory.

Everything in this course is built from three objects, and most confusion about a formula is really confusion about which of the three a symbol refers to.

Object	Written here	Example	Shape	In NumPy
Scalar a single number	lower case italic: x, λ, n	$x = 3.5$	no shape	`3.5`
Vector a list of numbers	bold lower case: $\mathbf{x}$, $\boldsymbol{\beta}$	$\mathbf{x} = \begin{bmatrix} 3.5 \\ 1.2 \\ 0.8 \end{bmatrix}$	3×1	1-dimensional array
Matrix a table of numbers	bold upper case: $\mathbf{X}$	$\mathbf{X} = \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 1 & 5 \end{bmatrix}$	3×2	2-dimensional array

The *written here* column is **this page's** convention, not a rule, and as the note at the top said, plain italic is at least as common in machine learning as bold is. Expect to see $\mathbf{X}\boldsymbol{\beta}$, $X\boldsymbol{\beta}$ and $X\beta$ all meaning the same product. The *shape* column is the one that never varies: an object either has a shape or it does not, and that is what tells you which of the three rows you are looking at. It is why the next section matters more than this one.

Sets, and the $\in$ symbol

$\mathbb{R}$ is the set of all real numbers, meaning every ordinary number. Built from it:

$x \in \mathbb{R}$	x is a single real number
$\mathbf{x} \in \mathbb{R}^p$	$\mathbf{x}$ is a list of p real numbers
$\mathbf{X} \in \mathbb{R}^{n \times p}$	$\mathbf{X}$ is a table of real numbers with n rows and p columns
$y_i \in \{0, 1\}$	the target takes one of the two values in that set
$\lambda \geq 0$	λ is a number that is never negative

The symbol $\in$ is read *is in*, or *belongs to*. A line like $\mathbf{X} \in \mathbb{R}^{n \times p}$ looks forbidding and says very little: it is a sentence about shape. Curly braces list the members of a set, so $\{1, \ldots, K\}$ is the label set for a problem with K classes.

Dimensions are the grammar

If the shapes fit, the expression means something. If they do not, it is not a hard formula, it is a mistake.

The single most useful habit in this course: **write the shape under every symbol**. A matrix expression is only legal when the shapes line up, so checking shapes catches most errors before you have understood anything else, and it tells you what an unfamiliar expression is going to produce.

The standard table of symbols, fixed from Session 4 onward:

n	the number of **observations**, the rows. For us, usually trading days.
p	the number of **features**, the columns of $\mathbf{X}$
$\mathbf{X} \in \mathbb{R}^{n \times p}$	the **design matrix**: one row per observation, one column per feature
$\mathbf{y} \in \mathbb{R}^{n}$	the **target**: one number per observation
$\boldsymbol{\beta} \in \mathbb{R}^{p}$	the **coefficients**: one per feature

Now look at $\mathbf{X}\boldsymbol{\beta}$ with the shapes written underneath.

$$\underset{n \times p}{\mathbf{X}} \cdot \underset{p \times 1}{\boldsymbol{\beta}} = \underset{n \times 1}{\mathbf{X}\boldsymbol{\beta}}$$

The two inner numbers must match, and then they cancel. The outer two are the shape of the answer. That is the whole rule, and it is worth knowing before you know how the multiplication is actually done.

So $\mathbf{X}\boldsymbol{\beta}$ is a column of n numbers, one per observation: the fitted values. And $\boldsymbol{\beta}\mathbf{X}$, in that order, is not defined at all, because $1 \times p$ against $n \times p$ does not line up. Order matters, always.

> **A shape check you can do in your head.** $\mathbf{X}^\top\mathbf{X}$ is $(p \times n)(n \times p) = p \times p$: a small square matrix, whatever the sample size. $\mathbf{X}^\top\mathbf{y}$ is $(p \times n)(n \times 1) = p \times 1$. So $(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$ is $(p \times p)(p \times 1) = p \times 1$, which is exactly one coefficient per feature. The formula had to come out that way.

Indices, subscripts and sums

Subscripts say which row, which column, or which of something else. Most of the time there are only two of them.

Two indices do nearly all the work, and they always mean the same thing:

$i = 1, \dots, n$	runs down the **rows**, over observations
$j = 1, \dots, p$	runs across the **columns**, over features

Which gives three things that look alike and are not:

Written	Is a	Means
x_{ij}	scalar	the value of feature j for observation i: one cell of the table
$\mathbf{x}_i$	vector, length p	observation i: one whole row, all its features
$\mathbf{x}_{\cdot j}$ or $\mathbf{X}_j$	vector, length n	feature j: one whole column, over all observations

A single subscript is genuinely ambiguous between the second and third of these, and authors are not consistent. Resolve it by shape: if the text goes on to write $\mathbf{x}_i^\top \boldsymbol{\beta}$, then $\mathbf{x}_i$ must have length p, so it is a row of the table.

Summation

$$\sum_{i=1}^{n} a_i \;=\; a_1 + a_2 + \cdots + a_n$$

Read: *the sum, over i from 1 to n, of a_i*. The $\sum$ is a capital Greek sigma, and it is shorthand for a loop that adds things up. In Python it is `sum(...)`, or a `for` loop with `total += a`.

Everything you need to know about it:

- The letter under the $\sum$ is the counter. It exists only inside the sum, exactly like the loop variable in `for i in range(n)`.
- Limits are often dropped once they are obvious, so $\sum_i a_i$ and even $\sum a_i$ mean the same thing as above.
- Constants come out: $\sum_i c\,a_i = c\sum_i a_i$, and $\sum_{i=1}^{n} c = nc$.
- The mean is a sum: $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$.
- Two indices give a double sum, $\sum_{i=1}^{n}\sum_{j=1}^{p} x_{ij}$, which is a loop inside a loop: add up every cell of the table.
- $\prod$ (capital pi) is the same idea with multiplication instead of addition. You meet it once, in a likelihood.

Hats, bars and the difference between them

$\bar{x}$	*x bar*: the sample mean of x
$\hat{\beta}$	*beta hat*: an **estimate** of β, computed from data. The hat is the whole difference between the truth and your guess at it.
$\hat{y}_i$	the model's prediction for observation i, as against y_i, what actually happened
$\hat{f}$	your fitted model, as against f, the true relationship nobody ever sees

The hat is not decoration. β is a fixed unknown number; $\hat{\beta}$ is a number you calculated, which would come out differently on a different sample. Nearly all of statistics is about the gap between the two.

Vectors

A list of numbers with an agreed order. The one operation that matters is the dot product, and it is a weighted sum.

By convention a vector is a **column** unless it is transposed. So $\mathbf{x} \in \mathbb{R}^3$ is 3×1, and $\mathbf{x}^\top$, read *x transpose*, is the same numbers laid out as a row, 1×3.

$$\mathbf{x} = \begin{bmatrix} 3 \\ 4 \\ 0 \end{bmatrix}, \qquad \mathbf{x}^\top = \begin{bmatrix} 3 & 4 & 0 \end{bmatrix}$$

Same three numbers. Only the shape changed, and shape is what decides whether the next multiplication is legal.

Adding two vectors and multiplying by a scalar both work entry by entry, which is what NumPy does when you write `a + b` or `2 * a`. Both vectors must be the same length.

The dot product

$$\mathbf{w}^\top \mathbf{r} \;=\; \sum_{j=1}^{p} w_j r_j \;=\; w_1 r_1 + w_2 r_2 + \cdots + w_p r_p$$

Multiply the two lists entry by entry, then add up. The result is a single number: $(1 \times p)(p \times 1) = 1 \times 1$.

You already use this constantly, under other names. A portfolio with weights $\mathbf{w} = (0.5,\ 0.3,\ 0.2)$ holding assets that returned $\mathbf{r} = (0.02,\ -0.01,\ 0.04)$ earned

$$\mathbf{w}^\top \mathbf{r} = 0.5(0.02) + 0.3(-0.01) + 0.2(0.04) = 0.015$$

A portfolio return is a dot product. So is a fitted value: replace the weights with coefficients and the returns with features, and $\hat{y}_i = \mathbf{x}_i^\top \boldsymbol{\beta}$ is the prediction for one observation. **Whenever you see a transpose next to a vector, expect a weighted sum.** In NumPy the dot product is `w @ r`.

Length, and the two norms you need

$\lVert\mathbf{x}\rVert_2$	the **Euclidean norm**, $\sqrt{\sum_j x_j^2}$. Ordinary straight-line length. For $(3, 4)$ it is $\sqrt{9+16} = 5$.
$\lVert\mathbf{x}\rVert_2^2$	the same thing squared, so just $\sum_j x_j^2 = \mathbf{x}^\top\mathbf{x}$, a sum of squares. This is the form you actually meet, because it drops the square root.
$\lVert\mathbf{x}\rVert_1$	the ℓ_1 **norm**, $\sum_j \lvert x_j\rvert$. Add up the sizes, ignoring signs. For $(3, -4)$ it is 7.

Double bars are a norm, of a vector. Single bars are either absolute value, of a number, or a determinant, of a matrix. The subscript says which norm. These two are the entire difference between ridge and lasso, so they are worth being sure about.

Matrices

A table of numbers, and a few special ones with names.

$\mathbf{A} \in \mathbb{R}^{m \times n}$ has m rows and n columns, and a_{ij} is the entry in row i, column j. **Rows first, always.** A matrix is *square* if $m = n$.

Transpose

The transpose $\mathbf{A}^\top$ flips the table over its diagonal: rows become columns. An $m \times n$ matrix becomes $n \times m$, and a_{ij} moves to position ji.

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 3 \end{bmatrix} \quad \Longrightarrow \quad \mathbf{A}^\top = \begin{bmatrix} 1 & 0 \\ 2 & 1 \\ 0 & 3 \end{bmatrix}$$

In NumPy: `A.T`. Two rules to keep: $(\mathbf{A}^\top)^\top = \mathbf{A}$, and $(\mathbf{AB})^\top = \mathbf{B}^\top \mathbf{A}^\top$, which reverses the order. That second rule is used in the OLS derivation below.

Two notations, one meaning. Econometrics usually writes the transpose as a prime, X', and says *X prime*. Machine learning usually writes $\mathbf{X}^\top$ or $\mathbf{X}^T$. They are the same operation, so $X'X = \mathbf{X}^\top\mathbf{X}$. The prime here has nothing to do with derivatives, and $\mathbf{X}^T$ is not $\mathbf{X}$ raised to a power T.

Matrices with names

$\mathbf{I}$ or $\mathbf{I}_p$	the **identity**: ones down the diagonal, zeros everywhere else, $p \times p$. It is the matrix version of the number 1, in that $\mathbf{AI} = \mathbf{IA} = \mathbf{A}$. In NumPy, `np.eye(p)`.
diagonal	zero off the diagonal. Multiplying by one just rescales each entry.
symmetric	$\mathbf{A}^\top = \mathbf{A}$, so it is unchanged by a flip. $\mathbf{X}^\top\mathbf{X}$ is always symmetric, and so is a covariance matrix.
$\mathbf{0}$	a vector or matrix of zeros. In bold it is not the number zero, though it plays the same role.

Matrix multiplication

Every entry of the answer is one dot product. Nothing more is going on.

To multiply $\mathbf{A}$ ($m \times n$) by $\mathbf{B}$ ($n \times k$), the inner dimensions must match. Then entry (i, j) of the product is row i of $\mathbf{A}$ dotted with column j of $\mathbf{B}$:

$$(\mathbf{AB})_{ij} \;=\; \sum_{\ell=1}^{n} a_{i\ell}\, b_{\ell j}$$

Row from the left, column from the right. Say it once out loud and the formula stops being a formula.

Worked out, with $\mathbf{A}$ being 2×3 and $\mathbf{B}$ being 3×2, so the product is 2×2:

$$\begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 3 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 0 & 4 \\ 5 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 9 \\ 15 & 4 \end{bmatrix}$$

Top left: $1(2) + 2(0) + 0(5) = 2$. Top right: $1(1) + 2(4) + 0(0) = 9$. Bottom left: $0(2) + 1(0) + 3(5) = 15$. Bottom right: $0(1) + 1(4) + 3(0) = 4$.

$\mathbf{AB}$ is not $\mathbf{BA}$. Often the second one is not even defined, and when it is, it is usually a different matrix. So you cannot reorder factors, and when you multiply both sides of an equation by something you must do it on the *same side* of both. That single fact is behind half of the steps in the OLS derivation.

Also: $\mathbf{AB}$ is **not** entry-by-entry multiplication. In NumPy, matrix multiplication is `A @ B` and entry-by-entry is `A * B`. They are different operations that both run without complaint, so this is a real source of silent wrong answers.

Two ways to read $\mathbf{X}\boldsymbol{\beta}$

Both readings are useful, and switching between them is most of what "thinking in matrices" means.

Row by row

Each entry of $\mathbf{X}\boldsymbol{\beta}$ is one observation's features dotted with the coefficients: $(\mathbf{X}\boldsymbol{\beta})_i = \mathbf{x}_i^\top \boldsymbol{\beta}$. So $\mathbf{X}\boldsymbol{\beta}$ is *all n predictions at once*. This is the reading to use when you want to know what the model does.

Column by column

$\mathbf{X}\boldsymbol{\beta} = \beta_1 \mathbf{X}_1 + \cdots + \beta_p \mathbf{X}_p$, a weighted sum of the feature columns. So the fitted values are whatever you can build by mixing your columns. This is the reading that explains multicollinearity, below.

Vectorisation: from n equations to one

The one idea that makes matrix notation worth learning. If you take one thing from this page, take this section.

Write the linear model for a single observation i, the way it is usually introduced:

$$y_i \;=\; \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \varepsilon_i, \qquad i = 1, \ldots, n$$

This is n separate equations, one per trading day, all with the same unknown coefficients.

Two compressions turn that into a single line. First, the right-hand side of each equation is a dot product, so one observation reads $y_i = \mathbf{x}_i^\top \boldsymbol{\beta} + \varepsilon_i$, where $\mathbf{x}_i$ starts with a 1 so that β_0 has something to multiply. Second, stack the n equations on top of each other:

$$\underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}}_{\mathbf{y},\ n\times 1} = \underbrace{\begin{bmatrix} 1 & x_{11} & \cdots & x_{1p} \\ 1 & x_{21} & \cdots & x_{2p} \\ \vdots & \vdots & & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{bmatrix}}_{\mathbf{X},\ n\times(p+1)} \underbrace{\begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix}}_{\boldsymbol{\beta},\ (p+1)\times 1} + \underbrace{\begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix}}_{\boldsymbol{\varepsilon},\ n\times 1}$$

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$$

Read: *y equals X beta plus epsilon*. It says exactly what the n equations said. Not a summary of them, not an approximation to them: the same statement, written once.

Three things follow, and they are the reason everyone writes models this way.

- **The column of ones is where the intercept went.** β_0 multiplies a feature that equals 1 for every observation, so it survives into the matrix form without needing a special case. That is why the design matrix has $p + 1$ columns when there are p real features, and why `np.column_stack([np.ones(n), x])` shows up in code.
- **The sample size vanished.** n appears nowhere in $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$. The same three symbols describe 10 observations or 10 million, which means every result you prove about it holds for any sample size.
- **Sums of squares become products.** The residual vector is $\mathbf{e} = \mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}$, and

$$\sum_{i=1}^{n} e_i^2 \;=\; \mathbf{e}^\top \mathbf{e} \;=\; \|\mathbf{e}\|_2^2 \;=\; \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2$$

All four of these are the residual sum of squares. Seeing them as the same object is most of what it takes to read a paper.

The same idea is why NumPy exists. Session 3 made this point in code: `X @ beta` computes all n predictions in one operation, with no loop over observations, and it runs far faster than the loop would. Vectorised maths and vectorised code are the same habit of thought. If you can turn a $\sum_i$ into a matrix product on paper, you can turn a `for` loop into one line of NumPy.

Inverses, singular matrices, and multicollinearity

Where OLS comes from, and the one thing that stops it from existing.

For a square matrix $\mathbf{A}$, the inverse $\mathbf{A}^{-1}$ is the matrix that undoes it:

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$$

It is the matrix version of dividing by a number. There is no "matrix division", so this is how you move a matrix from one side of an equation to the other.

$\mathbf{A}^{-1}$ is not a power, and not $1/\mathbf{A}$. The superscript -1 is a name for an operation, not an exponent, and $\mathbf{A}^{-1}$ is *not* the matrix of reciprocals $1/a_{ij}$. The entries of $\mathbf{A}^{-1}$ can look nothing like the entries of $\mathbf{A}$. The same superscript is used elsewhere for the same reason, for instance f^{-1} for the inverse of a function, which is also not $1/f$.

A 2×2 example, which is small enough to check by hand:

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}, \qquad \mathbf{A}^{-1} = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix}, \qquad \mathbf{A}\mathbf{A}^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Not a reciprocal in sight. For a 2×2 matrix the recipe is $\frac{1}{ad-bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$, and the number $ad - bc$ is the **determinant**, written $\det(\mathbf{A})$ or $|\mathbf{A}|$.

When there is no inverse

Look at the recipe: it divides by the determinant. If $\det(\mathbf{A}) = 0$ there is nothing to divide by and **the inverse does not exist**. Such a matrix is called **singular**. The three ways of saying it are worth collecting, because different sources use different ones:

singular	no inverse exists
$\det(\mathbf{A}) = 0$	the determinant is zero
rank deficient	the columns are not linearly independent: one of them is a weighted sum of the others, so it carries no information the rest do not already carry

These are three descriptions of one situation. The **rank** of a matrix is the number of genuinely independent columns in it; *full rank* means all of them are, which for a square matrix is the same as being invertible.

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}, \qquad \det(\mathbf{A}) = 1(4) - 2(2) = 0$$

The second column is exactly twice the first, so it adds nothing. This matrix has no inverse.

Perfect multicollinearity

Now put that into a regression. OLS needs $(\mathbf{X}^\top\mathbf{X})^{-1}$, and $\mathbf{X}^\top\mathbf{X}$ is singular exactly when a column of $\mathbf{X}$ is a weighted sum of the others. That is **perfect multicollinearity**, and when it happens there is no unique answer: infinitely many $\boldsymbol{\beta}$ give exactly the same fitted values, so the data cannot choose between them. Reading it through the column-by-column view above, if one column duplicates a mix of the others then the same weighted sum can be built in more than one way.

In practice it is nearly always one of these, and all three are self-inflicted:

- The same variable in two units, say a return in percent and the same return as a decimal. One column is 100 times the other.
- A dummy for every category *and* an intercept. The dummies add up to the column of ones, so the intercept is redundant. This is the dummy variable trap, and the fix is to drop one category.
- A variable that is the sum of two others already in the model, such as total return alongside its price and dividend parts.

Near multicollinearity, where columns are strongly but not perfectly related, is the more common problem. The inverse exists, so software returns an answer, but the determinant is close to zero and the coefficients become huge and unstable: tiny changes in the data swing them wildly. The estimates are still unbiased; they are just useless one at a time.

This is what ridge regression is for. Ridge estimates $\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}$. Adding λ to the diagonal makes the matrix invertible again, even when $\mathbf{X}^\top\mathbf{X}$ on its own is singular, and even when $p > n$, where OLS has no answer at all. When you meet that formula later, the $+\lambda\mathbf{I}$ is not a trick; it is the direct repair of the problem in this section.

In code, do not invert. `np.linalg.inv(A)` exists, and the cheatsheet lists it because the formula does. But `np.linalg.solve(A, b)` is both faster and numerically more accurate for the same job, and it is what you should write. Session 3 also warns that NumPy running inside the browser returns `nan` for a

singular matrix rather than raising an error, so a silent `nan` in your coefficients is a multicollinearity symptom.

Solving a matrix equation, and where OLS comes from

You are not asked to reproduce this derivation in this course. You are asked to read it without flinching.

Start with the scalar case, which you know. To solve $ax = b$ for x, divide both sides by a, which works as long as $a \neq 0$. The matrix version replaces division by multiplication by the inverse, and because order matters you must multiply both sides on the *same* side:

$$\mathbf{Ab} = \mathbf{c} \quad \Longrightarrow \quad \mathbf{A}^{-1}\mathbf{Ab} = \mathbf{A}^{-1}\mathbf{c} \quad \Longrightarrow \quad \mathbf{b} = \mathbf{A}^{-1}\mathbf{c}$$

The middle step used $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ and $\mathbf{Ib} = \mathbf{b}$. Note that $\mathbf{A}^{-1}$ went on the **left** of both sides. Writing $\mathbf{cA}^{-1}$ would be a different expression, and here not even a defined one. The requirement $a \neq 0$ has become the requirement that $\mathbf{A}$ is not singular.

Two rules of matrix calculus

Differentiating with respect to a vector means taking the derivative with respect to each entry and collecting the results into a vector. Two results cover the derivation below, and both are the obvious generalisation of the scalar case:

$$\frac{\partial\,(\mathbf{a}^\top\mathbf{b})}{\partial\mathbf{b}} = \mathbf{a}, \qquad \frac{\partial\,(\mathbf{b}^\top\mathbf{Ab})}{\partial\mathbf{b}} = 2\mathbf{Ab} \quad \text{when } \mathbf{A} \text{ is symmetric}$$

Compare $\frac{d}{dx}(ax) = a$ and $\frac{d}{dx}(ax^2) = 2ax$. Same statements, one dimension up. The symmetry condition holds here because $\mathbf{X}^\top\mathbf{X}$ always is symmetric.

The derivation, four lines

OLS chooses the $\boldsymbol{\beta}$ that makes the residual sum of squares as small as possible. Write that objective, expand it, differentiate, set to zero.

---

1 The objective, in matrix form:

$$S(\boldsymbol{\beta}) = \sum_{i=1}^{n} \left(y_i - \mathbf{x}_i^\top\boldsymbol{\beta}\right)^2 = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^\top(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})$$

2 Multiply it out, using $(\mathbf{AB})^\top = \mathbf{B}^\top\mathbf{A}^\top$:

$$S(\boldsymbol{\beta}) = \mathbf{y}^\top\mathbf{y} - 2\,\boldsymbol{\beta}^\top\mathbf{X}^\top\mathbf{y} + \boldsymbol{\beta}^\top\mathbf{X}^\top\mathbf{X}\boldsymbol{\beta}$$

The two cross terms are each single numbers, and a number equals its own transpose, which is why they combine into one term with a 2.

3 Differentiate with respect to $\boldsymbol{\beta}$ and set the result to zero, because at a minimum the slope is flat in every direction:

$$\frac{\partial S}{\partial \boldsymbol{\beta}} = -2\mathbf{X}^\top\mathbf{y} + 2\mathbf{X}^\top\mathbf{X}\boldsymbol{\beta} = \mathbf{0} \quad \Longrightarrow \quad \mathbf{X}^\top\mathbf{X}\hat{\boldsymbol{\beta}} = \mathbf{X}^\top\mathbf{y}$$

Those are the **normal equations**. Note the right-hand side is now a bold zero vector, one equation per coefficient.

4 This is $\mathbf{Ab} = \mathbf{c}$ from the top of this section, with $\mathbf{A} = \mathbf{X}^\top\mathbf{X}$. Multiply both sides on the left by $\mathbf{A}^{-1}$:

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$$

Two things to notice, because they are what the derivation is really teaching.

- The last step needs $(\mathbf{X}^\top\mathbf{X})^{-1}$ to exist. That is the no-perfect-multicollinearity assumption, and it is not a technicality: without it there is no unique $\hat{\boldsymbol{\beta}}$ to compute.
- $\mathbf{X}$ itself is never inverted. It is $n \times p$, not square, so it has no inverse. Only the $p \times p$ product does. This is why the formula is written in that particular clumsy-looking way.

Derivatives, gradients and minima

Every model in this course is defined as the solution to a minimisation, so you need to know what "the minimum" means operationally.

$\frac{df}{dx}$, $f'(x)$	the **derivative**: the slope of f at x. Positive means going up as x increases, negative means going down.
$\frac{\partial f}{\partial \beta_j}$	a **partial derivative**: the slope in the β_j direction with every other coefficient held fixed. The curly ∂ is what signals "there are other variables".
∇f	the **gradient**: all the partial derivatives collected into one vector, $\left(\frac{\partial f}{\partial \beta_1}, \ldots, \frac{\partial f}{\partial \beta_p}\right)^\top$. It points in the direction where f increases fastest. Pronounced *nabla* or *grad*.

At a minimum, the slope is zero in every direction, so $\nabla f = \mathbf{0}$. That is the entire method used in the OLS derivation, and it is the standard route to a closed-form estimator: write the objective, differentiate, set to zero, solve.

A zero gradient is a flat point, which could also be a maximum or a saddle. For the objectives in this course that worry is handled for you: they are all **convex**, meaning bowl-shaped, so a flat point can only be the bottom and there is exactly one of them. This is why sum-of-squares and log-loss objectives are so popular, and why the word convex appears whenever anybody claims their optimiser found the right answer.

When there is no formula

OLS and ridge have closed-form solutions: a formula you can evaluate directly. Lasso, logistic regression, and everything built out of trees do not. For those, the answer is found by walking downhill, and the notation for one step is:

$$\boldsymbol{\beta}^{(t+1)} \;=\; \boldsymbol{\beta}^{(t)} - \eta\, \nabla L\left(\boldsymbol{\beta}^{(t)}\right)$$

Read: the next guess is the current guess, moved a little way in the direction the loss falls fastest. η (eta) is the **learning rate**, the size of the step. The superscript (t) in brackets is an iteration counter, not a power, and not a transpose.

That is **gradient descent**, and it is one line. When you meet scikit-learn options called `max_iter`, `tol` or `learning_rate`, they are settings for this loop.

Sums, logs, exponentials and the sigmoid

Where the strange-looking classification formulas come from.

Three rules of logarithms carry all the weight, and only the first really matters:

$$\log(ab) = \log a + \log b, \qquad \log(a/b) = \log a - \log b, \qquad \log(a^k) = k \log a$$

In this field $\log$ means the **natural** logarithm, base $e \approx 2.718$, which some texts write $\ln$. It is `np.log` in NumPy.

The first rule is why likelihoods are always logged. The probability of seeing n independent observations is a *product* of n terms, which is awkward to differentiate and underflows to zero on a computer. Taking logs turns it into a sum, and since $\log$ is increasing, whatever maximises the sum maximises the product:

$$\log \prod_{i=1}^{n} L_i \;=\; \sum_{i=1}^{n} \log L_i$$

That is the whole reason the phrase *log-likelihood* exists. Session 4's use of log returns is the same trick: products of gross returns become sums of log returns.

The exponential e^x, also written $\exp(x)$, undoes the logarithm. It is always positive, which is exactly what you want when a formula has to produce a probability or a variance. Combine the two and you get the function that defines logistic regression:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The **sigmoid**, or logistic function. It takes any real number and returns something strictly between 0 and 1, so it converts an unbounded score into a probability. $\sigma(0) = 0.5$, $\sigma(2) \approx 0.88$, $\sigma(-2) \approx 0.12$.

So logistic regression computes the same linear score as OLS, $z_i = \mathbf{x}_i^\top \boldsymbol{\beta}$, and then squashes it: $P(Y_i = 1 \mid \mathbf{x}_i) = \sigma(\mathbf{x}_i^\top \boldsymbol{\beta})$. Rearranged, the model says the **log odds** are linear:

$$\log\left(\frac{P(Y = 1 \mid \mathbf{x})}{1 - P(Y = 1 \mid \mathbf{x})}\right) = \mathbf{x}^\top \boldsymbol{\beta}$$

Which is why a logistic coefficient is interpreted per unit of log odds and not as a change in probability. Same dot product as before, on a different scale.

Probability and statistics notation

Mostly one symbol you already know, plus the convention about capital letters.

X, Y	capital letters are **random variables**: the thing before you observe it
x, y	lower case is a **realisation**: the value it actually took. So $P(Y = 1 \mid X = x)$ asks about a particular observed x.
$P(A)$	the probability of A, a number between 0 and 1
$P(A \mid B)$	the vertical bar is read **given**: the probability of A once B is known. Nearly every classifier is an estimate of $P(Y = 1 \mid \mathbf{x})$.
$\mathbb{E}[X]$	the **expectation**, or population mean, of X. Square brackets are the convention, and they are not a function call or an index.
$\mathrm{Var}(X)$, σ^2	the variance, $\mathbb{E}\big[(X - \mathbb{E}[X])^2\big]$; σ is its square root, the standard deviation

$\mathrm{Cov}(X, Y)$	the covariance; divided by both standard deviations it becomes the correlation ρ, between -1 and 1
$X \sim N(\mu, \sigma^2)$	the tilde is read **is distributed as**. Here: normal with mean μ and variance σ^2.
i.i.d.	independent and identically distributed: every observation drawn the same way, none influencing another. Financial time series routinely break this, which is why the ordering of train and test data matters.

Two uses of this notation are worth reading closely, because Session 4 leans on both. The relationship a supervised model is trying to recover is a conditional expectation,

$$f(\mathbf{x}) = \mathbb{E}[Y \mid \mathbf{X} = \mathbf{x}]$$

Read: the average value of the target among all observations having those feature values. That is what "the true relationship f" means precisely.

And the bias-variance decomposition, which is a statement about the expected squared error of a prediction:

$$\mathbb{E}\Big[(y_0 - \hat{f}(x_0))^2\Big] = \underbrace{\mathrm{Var}(\hat{f}(x_0))}_{\text{variance}} + \underbrace{\Big[\mathrm{Bias}(\hat{f}(x_0))\Big]^2}_{\text{bias}^2} + \underbrace{\mathrm{Var}(\varepsilon)}_{\text{irreducible}}$$

The expectation is over repeated samples: imagine refitting the model on a fresh draw of data many times. The third term does not involve $\hat{f}$ at all, which is exactly why no model can drive the error to zero.

Distance, and why scale matters

Needed for k-nearest neighbours, and for understanding why standardising is not optional there.

The distance between two observations is the length of the vector between them:

$$d(\mathbf{x}, \mathbf{z}) = \|\mathbf{x} - \mathbf{z}\|_2 = \sqrt{\sum_{j=1}^{p} (x_j - z_j)^2}$$

The **Euclidean distance**: Pythagoras in p dimensions. Subtract feature by feature, square, add, take the root. The ℓ_1 version, $\sum_j |x_j - z_j|$, is the Manhattan distance, and both show up as options in scikit-learn.

Now notice what the formula does with units. Each feature contributes $(x_j - z_j)^2$, in whatever units that feature happens to be measured in. Session 4 showed that a volume column varies about

four billion times more than a return column, so in that sum the return term is invisible: the distance is the volume difference, and every "nearest neighbour" is really the nearest by volume. Standardising each column first,

$$z_{ij} = \frac{x_{ij} - \bar{x}_j}{s_j}$$

puts every feature on the same footing, mean 0 and standard deviation 1, so the distance is a fair comparison. OLS does not need this. Anything that measures distance, or penalises the size of coefficients, does.

Reading an objective function

Half the models in this course are defined by one template, and once you can read the template each new model is a one-line change.

First, the notation for a minimisation:

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \; L(\boldsymbol{\beta})$$

Read: *the $\boldsymbol{\beta}$ that makes L as small as possible*. The subscript names what you are free to vary. $\min L$ is the smallest *value* of L; $\arg\min L$ is the *argument* that achieves it. In a model definition you almost always want the second, because the coefficients are the point. $\arg\max$ is the same with the sign flipped.

Now the template. Nearly every linear model in this course is

$$\arg\min_{\boldsymbol{\beta}} \; \underbrace{\big[\,\text{how badly it fits}\,\big]}_{\text{loss}} \; + \; \lambda \underbrace{\big[\,\text{how big the coefficients are}\,\big]}_{\text{penalty}}$$

$\lambda \geq 0$ is a tuning parameter that sets the exchange rate between the two. $\lambda = 0$ removes the penalty entirely; large λ cares mostly about keeping coefficients small, and shrinks the fit toward nothing.

Four familiar models, differing only in that second bracket:

Model	Objective	What the penalty does
OLS	$\lVert \mathbf{y} - \mathbf{X}\boldsymbol{\beta} \rVert_2^2$	no penalty at all
Ridge	$\lVert \mathbf{y} - \mathbf{X}\boldsymbol{\beta} \rVert_2^2 + \lambda \lVert \boldsymbol{\beta} \rVert_2^2$	squared size: shrinks every coefficient toward zero, but none of them all the way

Model	Objective	What the penalty does
Lasso	$\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda\|\boldsymbol{\beta}\|_1$	absolute size: sets some coefficients exactly to zero, so it also selects features
Elastic net	$\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda_1\|\boldsymbol{\beta}\|_1 + \lambda_2\|\boldsymbol{\beta}\|_2^2$	both, in a mix you choose

The only symbol that changes across those rows is the subscript on the norm, 1 or 2. That is the whole formal difference between three methods that get separate lectures, and it is a good example of why the notation is worth twenty minutes.

Classification uses the same template with a different loss. For logistic regression the loss is the negative log-likelihood, usually called log-loss or cross-entropy:

$$L(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \Big[\, y_i \log \hat{p}_i + (1 - y_i) \log(1 - \hat{p}_i) \,\Big], \qquad \hat{p}_i = \sigma(\mathbf{x}_i^\top \boldsymbol{\beta})$$

It looks worse than it is. y_i is 0 or 1, so exactly one of the two terms survives for each observation: if $y_i = 1$ the term is $-\log \hat{p}_i$, which is small when $\hat{p}_i$ is near 1 and blows up as it approaches 0. The sum is a penalty for confident mistakes. That bracket is a switch, not arithmetic.

Scores, folds and samples

The arithmetic behind how a model is judged and chosen: the scores of a classifier, the cross-validation estimate, the bootstrap, and the softmax.

The scores of a classifier

Every score of a classifier is built from the four counts of the confusion matrix: TP and TN, the right answers, FP, the false alarms, and FN, the misses.

$$\text{precision} = \frac{TP}{TP + FP} \qquad \text{recall} = \frac{TP}{TP + FN} \qquad F_1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Precision asks how often a yes was right; recall, how many of the real yeses were caught. F_1 is their harmonic mean, which stays close to the smaller of the two. Accuracy, $(TP + TN)/n$, is the mean of an indicator, as in the section on trees below.

$$\mathrm{AUC} = P\big(\hat{p}_i > \hat{p}_j \;\big|\; y_i = 1,\; y_j = 0\big)$$

Pick one row that was a 1 and one that was a 0: the AUC is the chance that the model gave the 1 the higher probability. A random ranking scores 0.5 and a perfect one 1, whatever the threshold, which is why the AUC needs the probabilities and not the 0/1 predictions.

The cross-validation estimate

$$\mathrm{CV}_{(K)} = \frac{1}{K}\sum_{k=1}^{K} \mathrm{MSE}_k$$

Cut the training rows into K folds, score the model on each fold after fitting it on the others, and average. With time-series folds the model is only ever fitted on rows that come before the fold it is scored on. The setting with the smallest $\mathrm{CV}_{(K)}$ wins, and only then are the test rows used, once.

The bootstrap

$$P(\text{row } i \text{ is never drawn}) = \left(1 - \frac{1}{n}\right)^{n} \;\to\; e^{-1} \approx 0.368$$

A bootstrap sample draws n rows from n, with replacement. Each draw misses a given row with probability $1 - 1/n$, so about 37 percent of the rows are left out of each sample and 63 percent are in. A forest can forecast each row with only the trees whose sample left it out: the out-of-bag forecast. On days it flatters the forest, because a day left out has its neighbours in the bag.

The softmax

$$P(\text{class } j) = \frac{e^{s_j}}{e^{s_1} + e^{s_2} + \cdots + e^{s_N}}$$

With N classes the model gives each class a score s_j. The exponential makes every score positive, and the division makes the N probabilities add up to one. With two classes it is the sigmoid, applied to the difference between the two scores.

Notation for trees and ensembles

Tree formulas look unlike regression formulas, but they use only two new pieces of notation.

The indicator function

$$\mathbf{1}\{A\} = \begin{cases} 1 & \text{if } A \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

Also written $I(A)$ or $\mathbf{1}_A$. It turns a yes-or-no question into a number so it can be used in arithmetic. In Python it is exactly what `(s > 0).astype(int)` does, from the Session 4 cheatsheet: a comparison produces True and False, and the cast makes them 1 and 0.

With it, "count how many of these are true" is $\sum_i \mathbf{1}\{\dots\}$, and "what fraction are true" is $\frac{1}{n}\sum_i \mathbf{1}\{\dots\}$. Accuracy is a mean of indicators, $\frac{1}{n}\sum_i \mathbf{1}\{\hat{y}_i = y_i\}$.

Regions

A regression tree cuts the feature space into M non-overlapping boxes $R_1, \dots, R_M$, the leaves, and predicts one constant in each. Written out, the whole model is:

$$\hat{f}(\mathbf{x}) = \sum_{m=1}^{M} c_m \, \mathbf{1}\{\mathbf{x} \in R_m\}$$

Every term is zero except the one box the observation falls into, so the sum picks out a single c_m. It is a lookup table written as a sum, which is a pattern worth recognising: a sum over regions with an indicator is nearly always "find the right one".

The constant in a leaf is the mean of the training targets in it, and the split is chosen to make the resulting sum of squares small. For classification, the quality of a leaf is measured by how mixed it is, and the two standard measures are:

$$G = \sum_{k=1}^{K} \hat{p}_k (1 - \hat{p}_k) \qquad\qquad D = -\sum_{k=1}^{K} \hat{p}_k \log \hat{p}_k$$

Gini on the left, entropy on the right; $\hat{p}_k$ is the fraction of the leaf belonging to class k, and k runs over classes, a third index to keep separate from i and j. Both are zero when a leaf is pure, meaning one $\hat{p}_k$ is 1 and the rest are 0, and both are largest when the classes are evenly mixed.

Ensembles: averaging and adding

Once there are many trees, an extra index appears for *which tree*, usually b or m. There are only two ways to combine them, and the notation tells you which one you are looking at:

Average them (bagging, random forest)

$$\hat{f}(\mathbf{x}) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{\,b}(\mathbf{x})$$

B trees are grown independently on different resamples, then averaged. The $1/B$ is the giveaway: independent votes, equal weight. Averaging cuts variance, which is why a forest beats a single deep tree.

Add them up (boosting, XGBoost)

$$\hat{f}_m(\mathbf{x}) = \hat{f}_{m-1}(\mathbf{x}) + \nu\, h_m(\mathbf{x})$$

Each new small tree h_m is fitted to what the previous ones still get wrong, and added on with a shrinkage factor ν. No $1/B$, and the subscript refers to a previous stage: that recursion is the signature of boosting.

The superscript b in $\hat{f}^{\,b}$ is a label, not a power. Superscripts as labels are common once there are several models to keep apart, and context is the only way to tell them from exponents. A superscript in brackets, $\hat{f}^{(m)}$, is always a label.

The equation gallery

The formulas you will actually be shown, each read out symbol by symbol. This is the drill: cover the notes, read the equation aloud, then check.

None of these is derived here, and several belong to later sessions. The point is only that by now you should be able to say what every symbol in them is, and what shape it has. If you can do that, an unfamiliar model is a short read rather than a wall.

ORDINARY LEAST SQUARES

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$$

Beta hat equals, X transpose X, inverse, X transpose y.

$\hat{\boldsymbol{\beta}}$	the estimated coefficients, $p \times 1$, one per feature
$\mathbf{X}^\top\mathbf{X}$	$p \times p$ and symmetric. Must be invertible, which is exactly the no-perfect-multicollinearity condition.
$(\cdot)^{-1}$	matrix inverse. Not a power, not a reciprocal.
$\mathbf{X}^\top\mathbf{y}$	$p \times 1$. Entry j is the dot product of feature j with the target.

RIDGE REGRESSION

$$\hat{\boldsymbol{\beta}}^{\text{ridge}} = \arg\min_{\boldsymbol{\beta}} \ \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda\|\boldsymbol{\beta}\|_2^2 \ = \ (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}$$

The beta minimising the sum of squared residuals plus lambda times the sum of squared coefficients. Unusually, this one has both a definition and a closed form, and they are the same thing.

$\|\cdot\|_2^2$	squared Euclidean norm, so both terms are sums of squares
λ	a single non-negative number you choose, not estimated from the fit. $\lambda = 0$ gives OLS back exactly.
$\lambda\mathbf{I}$	λ added down the diagonal only. This is what makes the matrix invertible even when $\mathbf{X}^\top\mathbf{X}$ is not.
superscript ridge	a label saying which estimator, not an exponent

LASSO

$$\hat{\boldsymbol{\beta}}^{\text{lasso}} = \arg\min_{\boldsymbol{\beta}} \ \sum_{i=1}^{n} \left(y_i - \mathbf{x}_i^\top\boldsymbol{\beta}\right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

Written here with sums rather than norms, which is the same statement as the ridge line above with $\|\cdot\|_1$ in place of $\|\cdot\|_2^2$. Being able to move between the two spellings is the skill.

i vs j	the first sum runs over observations, the second over coefficients. Different loops, and the intercept is normally left out of the second.
$\|\beta_j\|$	absolute value of a single number, single bars. Not a norm and not a determinant.

LOGISTIC REGRESSION

$$P(Y = 1 \mid \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{x}^\top\boldsymbol{\beta}}}$$

The probability that the label is 1, given these features, equals one over one plus e to the minus x transpose beta.

$\mathbf{x}^\top\boldsymbol{\beta}$	a scalar: the same weighted sum OLS forms. Everything outside it exists only to map that number into $(0, 1)$.
$\mid$	read *given*. Not division, not absolute value.
$Y = 1$	which class counts as the positive one, a choice you make when you build the target

K-NEAREST NEIGHBOURS

$$\hat{y}_0 = \frac{1}{k} \sum_{i \in \mathcal{N}_k(\mathbf{x}_0)} y_i$$

The prediction at a new point is the average of the targets of its k nearest training points. For classification, replace the average with a vote: $\hat{y}_0 = \arg\max_c \sum_{i \in \mathcal{N}_k} \mathbf{1}\{y_i = c\}$, the class that appears most often among the neighbours.

$\mathcal{N}_k(\mathbf{x}_0)$	a **set**: the indices of the k training observations closest to $\mathbf{x}_0$. Script capitals usually name a set.
$i \in \mathcal{N}_k$	the sum runs over the members of that set, not over 1 to n. A sum can be indexed by any set.
closest	by the Euclidean distance above, which is why the columns must be standardised first

MEAN SQUARED ERROR

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \frac{1}{n} \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2$$

Session 4's scoring metric, in its two equivalent spellings. The second is the first, vectorised. Once you read them as the same object, the step from "average of squared errors" to "squared norm of the error vector" stops being a translation and becomes a glance.

Notation to NumPy

The same objects, in the language you will write them in. Everything here is on the cheatsheet except where noted.

Maths	NumPy	Note
$\mathbf{X} \in \mathbb{R}^{n \times p}$	`X.shape` is `(n, p)`	rows first, same order as the maths
x_{ij}	`X[i, j]`	Python counts from 0, the maths from 1
$\mathbf{x}_i$, row i	`X[i, :]`	one observation
$\mathbf{X}_j$, column j	`X[:, j]`	one feature
$\mathbf{X}^\top$	`X.T`	
$\mathbf{X}\boldsymbol{\beta}$	`X @ beta`	`*` would be entry by entry, which is a different thing

Maths	NumPy	Note
$\mathbf{x}^\top \mathbf{y}$	`x @ y`	for two 1-dimensional arrays this is the dot product
$\mathbf{X}^\top \mathbf{X}$	`X.T @ X`	
$\mathbf{A}^{-1}$	`np.linalg.inv(A)`	prefer `np.linalg.solve(A, b)` for $\mathbf{A}^{-1}\mathbf{b}$
$\mathbf{I}_p$	`np.eye(p)`	not on the cheatsheet
column of ones	`np.column_stack([np.ones(n), x])`	the intercept column
$\sum_i x_i$	`x.sum()`	
$\bar{x}$	`x.mean()`	
$\|\mathbf{x}\|_2^2$	`x @ x`	a sum of squares is a vector dotted with itself
$\|\mathbf{x}\|_2$	`np.sqrt(x @ x)`	or `np.linalg.norm(x)`, not on the cheatsheet
$\mathbf{1}\{x > 0\}$	`(x > 0).astype(int)`	the indicator function
$\log$	`np.log`	natural log, as everywhere in this course
e^x	`np.exp(x)`	

The whole OLS formula, once, so the correspondence is concrete:

```
beta_hat = np.linalg.solve(X.T @ X, X.T @ y)   # preferred
beta_hat = np.linalg.inv(X.T @ X) @ X.T @ y    # the formula, literally
```

Both lines compute $(\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y}$. The first solves the normal equations $\mathbf{X}^\top \mathbf{X}\hat{\boldsymbol{\beta}} = \mathbf{X}^\top \mathbf{y}$ directly, which is faster and better behaved when the columns are nearly collinear.

Traps and look-alikes

Symbols that mean different things depending on where they sit. These are the ones that cause real confusion in an exam.

You see	It is not	It is
X vs $\mathbf{X}$	necessarily two different objects	usually one object in two authors' styles. Bold is a scanning aid, and much of the machine-learning literature does without it. Decide what a symbol is from its shape, not its font.

You see	It is not	It is
$\mathbf{A}^{-1}$	a power, or $1/a_{ij}$	the matrix inverse, an operation with its own definition
$\mathbf{X}^\top$, $\mathbf{X}^T$, X'	a power T, or a derivative	three spellings of the transpose. The prime is the econometrics convention.
$\lvert\beta_j\rvert$ vs $\lVert\boldsymbol{\beta}\rVert$ vs $\lvert\mathbf{A}\rvert$	the same bars	absolute value of a number, norm of a vector, determinant of a matrix
$\sum$ vs $\boldsymbol{\Sigma}$	the same symbol	capital sigma is both "add these up" and, in bold or in a variance context, a covariance matrix
σ vs $\sigma(z)$	the same object	lower case sigma alone is a standard deviation; with an argument it is usually the sigmoid
p	one thing	the number of features, and also a probability. Context decides, and $\hat{p}_k$ with a class index is always a probability.
x_i vs x_j vs x_{ij}	interchangeable	a row, a column, and a single cell
ε_i vs e_i	the same number	the true error, which you never see, and the residual $y_i - \hat{y}_i$, which you compute
β vs $\hat{\beta}$	the same number	the unknown truth and your estimate of it. Every hat is a claim about data.
$\mathbf{AB}$	$\mathbf{BA}$	order matters, and often only one of them is even defined
$\mathbf{AB}$ in code	`A * B`	`A @ B`. The `*` multiplies entry by entry and does not complain.
$\hat{f}^{(m)}$, $\hat{f}^{\,b}$	powers	labels: iteration m, tree b. A superscript in brackets is always a label.
$\mathbb{E}[X]$	a function call or an index	expectation. Square brackets are just the convention.
$\mathbf{0}$, $\mathbf{I}$	the numbers 0 and 1	the zero vector or matrix, and the identity matrix
$\log$	base 10	the natural logarithm, base e. Written $\ln$ in some texts, never in this one.
$\arg\min$	$\min$	the input achieving the minimum, not the minimum value

Check yourself

Twelve questions. If you can answer them without looking, this page has done its job.

Answer each one before expanding it. Several are shape questions, and shape questions are the ones worth being fast at.

1. $\mathbf{X}$ is $n \times p$ and $\mathbf{y}$ is $n \times 1$. What shape is $\mathbf{X}^\top \mathbf{y}$?

2. Of $\mathbf{X}\boldsymbol{\beta}$, $\boldsymbol{\beta}\mathbf{X}$ and $\boldsymbol{\beta}^\top \mathbf{X}^\top$, which are defined?

3. Read $\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$ out loud.

4. Why is it $(\mathbf{X}^\top \mathbf{X})^{-1}$ rather than $\mathbf{X}^{-1}(\mathbf{X}^\top)^{-1}$?

5. You include a stock's daily return in percent and the same return as a decimal as two features. What happens?

6. In ridge regression, what does $\lambda = 0$ give you? And very large λ?

7. What is the difference between ε_i and e_i?

8. In $\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$, what is i and how many terms are there?

9. Write $\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$ in matrix notation.

10. What is $\mathbf{1}\{\mathbf{x} \in R_3\}$ when $\mathbf{x}$ is not in R_3?

11. What kind of object is $\mathbf{x}_i^\top \boldsymbol{\beta}$, and what does it mean?

12. What is the difference between $\min_\beta L(\beta)$ and $\arg\min_\beta L(\beta)$?

Symbols and Greek letters

A lookup table for when a symbol appears and nothing on the slide says what it is.

Greek letters you will meet

Symbol	Name	Usually means
α	alpha	an intercept, a significance level, or a mixing weight

Symbol	Name	Usually means
β, $\boldsymbol{\beta}$	beta	a regression coefficient, or the whole vector of them. This page bolds the vector; many sources write both the same way.
ε	epsilon	the error term, the part of y no model can explain
η	eta	a learning rate, the step size in gradient descent
λ	lambda	a penalty strength in ridge, lasso and elastic net
μ	mu	a population mean
ν	nu	a shrinkage factor in boosting
ρ	rho	a correlation
σ, σ^2	sigma	standard deviation and variance; with an argument, the sigmoid
$\sum$	capital sigma	add these up
$\mathbf{\Sigma}$	capital sigma, bold	a covariance matrix
$\prod$	capital pi	multiply these together
θ, $\boldsymbol{\theta}$	theta	parameters in general, when the model is not specifically a regression

Everything else

Symbol	Read as	Means
$\in$	is in, belongs to	membership of a set
$\mathbb{R}^{n \times p}$	R, n by p	the real matrices with n rows and p columns
$\{0, 1\}$	the set zero, one	the two values a binary target can take
$\mathcal{N}_k(\mathbf{x})$	the neighbourhood of x	a named set, here the k nearest points
$\mathbf{A}^\top$	A transpose	rows and columns swapped
$\mathbf{A}^{-1}$	A inverse	the matrix undoing $\mathbf{A}$
$\hat{x}$	x hat	an estimate computed from data
$\bar{x}$	x bar	a sample mean

Symbol	Read as	Means
$\lvert x \rvert$	the absolute value of x	size, ignoring sign
$\lVert \mathbf{x} \rVert_2$, $\lVert \mathbf{x} \rVert_1$	the two-norm, the one-norm	root of the sum of squares; sum of absolute values
$\mathbb{E}[\cdot]$	the expectation of	the population average
$P(A \mid B)$	the probability of A given B	a conditional probability
$\sim$	is distributed as	names the distribution of a random variable
$\mathbf{1}\{\cdot\}$	the indicator of	1 if true, 0 if false
∇	nabla, grad	the vector of all partial derivatives
∂	partial	a derivative holding other variables fixed
$\arg\min$, $\arg\max$	arg min, arg max	the input that minimises or maximises
$\approx$	is approximately	
$\propto$	is proportional to	equal up to a constant factor that does not matter here
$\Rightarrow$	implies	the next line follows from the last
$\forall$, $\exists$	for all, there exists	rare in this course, common in the textbooks

If you want a proper refresher EXTRA

None of this is required, and none of it is examined. It is here for the case where a section above was unfamiliar rather than rusty.

Essence of linear algebra a short video series that shows what a matrix *does*, geometrically. About three hours in total, and the best available cure for matrices feeling like bookkeeping.

An Introduction to Statistical Learning chapter 3 is linear regression written at exactly the level this course assumes, and chapter 6 is where ridge, lasso and elastic net appear in the notation used above. Free PDF, and already the course's conceptual reference.

Linear algebra review and reference the Stanford CS229 handout. Thirty pages, dense, and the closest thing to a formal version of this page if you want the definitions stated properly.