week 08: regression

regression, /rəˈgreSH(ə)n/, *noun* (from *Wikipedia*):

- a return to a former or less developed state
- (statistics) a measure of the relation between the mean value of one variable and correspnding values of other variables.

regress and review: where we are

What we've been building up is a systematic way of approaching biological data analysis problems, as well as a way of trying to understand the implicit assumptions of an apparently *ad hoc* method:

- First, write down the probability "of everything":
    - Figure out how you're going to calculate the probability of your data D given a probability model and its parameters θ, $P(D \mid \theta)$. Think of this as a *generative model* for the data. You can simulate synthetic data from your model, and if it doesn't look enough like real data, you can adjust your model.
    - Figure out whether you want some parameter values of θ to be *a priori* more plausible than others: i.e. specify a prior $P(\theta)$ that generates parameter values.
    - Multiplied together, the likelihood and the prior give you the joint probability

$P(D, \theta)$: "the probability of everything".

- Then use probability algebra to get the conditional probability of what you want to infer, in terms of the data you observe, marginalized over what you don't care about.
    - Usually involving Bayes' theorem to get $P(\theta \mid D)$ in one form or another. In scientific data analysis, we're typically trying to use observed data to deduce unknown underlying models and parameters.

Much of what we've been seeing in the course can then be broken down as:

- various approaches for constructing explicit probability models for various sorts of data and analysis problems
- how to use probability algebra
- techniques for inferring hidden parameters -- ideally their posterior distribution, but if necessary, point estimates such as maximum likelihood or MAP (*maximum a posteriori*) parameters.
    - Sometimes we can analytically solve for optimal parameters;
    - sometimes we have to do something iterative and numerical, like expectation maximization (EM), or (as we'll see this week), various other kinds of function optimization;
    - iterative numerical algorithms like EM are the cousins of iterative numerical algorithms for sampling a full posterior distribution (as opposed to just finding optimal parameters), such as Markov chain Monte Carlo (MCMC) algorithms.

We're using probability as a *degree of belief*, which

makes us Bayesian, but we just call it **probabilistic inference**.

An advantage of this approach is that it forces us to put all our assumptions on the table and make them explicit. If an analysis fails, we can go back and revisit our assumptions -- not only by questioning them intuitively, but also by testing them experimentally, by comparing data synthesized from the model to the actual data.

Another advantage of this approach is that by specifying a *generative probability model* of the data, it's easy to write a Python script to generate synthetic control data sets. Indeed, the ability to generate synthetic control data and the soundness of your analysis go hand in hand: if you don't know how to synthesize synthetic control data, you haven't yet thought through your generative probability model.

You may also find, as I do, that traditional statistical analysis methods start to make sense, rather than seeming like a bag of unrelated tricks. Statistics was taught to me as lore, in pieces and recipes: linear regression, Student's t-tests, Mann/Whitney U-tests, and so on. For each recipe, you have to remember what problem it applies to, and what hidden assumptions it makes. In the course, we're seeing examples of how various traditional statistical analyses come out as some specific case of a probabilistic inference approach. I find that when I can derive a statistical approach from first principles - even if I can only remember the sketch of the derivation - I have a much easier time remembering the approach's limitations and strengths.

So, time for another example: **linear regression**.

the lowly line

The familiar equation for a straight line with 0 intercept:

$$y = \beta x$$

where β is the slope. If I give you a set of n observed data points $(x_1, y_1) . . (x_i, y_i) . . (x_n, y_n)$, how do you find the best slope β?

least squares fitting

A standard answer is **least squares fitting**. For a given β, you can calculate the expected value of y_i according to your model, $\hat{y}_i = \beta x_i$, and compare your prediction to the actual y_i. The difference is called the **residual** e_i:

$$\begin{aligned} e_i &= y_i - \hat{y} \\ &= y_i - \beta x_i \end{aligned}$$

A good fit is one with small residuals. How should we define "small"? We can define the **residual sum of squares** as:

$$\begin{aligned} \mathrm{RSS} &= \sum_i e_i^2 \\ &= \sum_i (y_i - \beta x_i)^2 \end{aligned}$$

Suppose we decide to find the slope that minimizes the RSS. Easy. To minimize the RSS, take its derivative with respect to β and set to zero:

$$\frac{\partial \mathrm{RSS}}{\partial \beta} = -2 \sum_i (y_i - \beta x_i) x_i = 0$$

We can rearrange and solve for β:

$$\hat{\beta} = \frac{\sum_i y_i x_i}{\sum_i x_i^2}$$

Great! We've fitted an "optimal" slope $\hat{\beta}$ to our data!

But what did we actually optimize here? What assumptions did we make? Why should we minimize the sum of *squares*? Some textbooks will

tell you we do this because some residuals are positive and some are negative, and squaring them keeps them from cancelling out. But if so, why not minimize the sum of their absolute values?

This should be reminding you of k-means a couple weeks ago, where the same question arose - why does k-means optimize the total sum of *squared* distances from centroids to data points?

a generative model of noisy data on a line

Assume that data points y_i are generated by adding a normally-distributed random error to the predicted value:

$$y_i = \beta x_i + \epsilon_i$$
$$\epsilon_i = N(0, \sigma^2)$$

which says that the *residual*, the error, is normally distributed with mean 0 and variance σ^2.

(Notationally, we will try to be careful to distinguish ϵ (a parameter of our generative model) from e (an observed residual in our data), much like we try to be careful to distinguish a Gaussian parametric mean μ from an observed data mean $\bar{x}$.)

To sample data from this model at any given x_i, we calculate βx_i and add a random Gaussian-distributed value $\epsilon_i = N(0, \sigma^2)$.

This is an important class of probabilistic model. Until now we've more seen models that are in some sense *inherently* probabilistic: balls and urns type problems, like picking an mRNA transcript randomly according to its abundance. Now here's a problem where we sort of imagine an underlying "true", ideal value βx_i, but we've made a noisy observed measurement of it to obtain y_i.

In our model of noisy data on a line, we've already made specific assumptions:

- that the residuals are Gaussian distributed.
- that the variance of the residuals ϵ_i is the

same everywhere, independent of i and x_i.

We could assume a different distribution for the errors (maybe something long-tailed that would account better for outliers?), and we could allow errors to have different distributions (perhaps the observations y_i come from different experiments with different measurement errors?) but let's go with this simple model of the data for now.

Given observed data $(x_1, y_1) \ldots (x_i, y_i) \ldots (x_n, y_n)$, we can write down the probability of the data:

$$P(\text{data}) = \prod_i \frac{1}{\sqrt{2\sigma^2\pi}} e^{-\frac{(y_i - \beta x_i)^2}{2\sigma^2}}$$

which is just a product of Gaussian PDFs for the residuals $y_i - \beta x_i$.

As usual it's almost always a good thing to work with the log likelihood:

$$\log P(\text{data}) = \sum_i \log \frac{1}{\sqrt{2\sigma^2\pi}} - \sum_i \frac{(y_i - \beta x_i)^2}{2\sigma^2}$$

What's the maximum likelihood β? You can already sneak a peek into the log likelihood equation and see what's going to happen -- look past constants that depend only on σ, and you'll see that the term that depends on β is $-\sum_i (y_i - \beta x_i)^2$: to maximize the likelihood, we'll minimize the sum of squared residuals, which means we're supposed to do a least squares fit!

We see the answer in there but let's carry on anyway - if we want to maximize the likelihood with respect to β, we take the derivative of the likelihood with respect to β, set to zero, and solve:

$$\frac{\partial \log P(\text{data})}{\partial \beta} = \frac{1}{2\sigma^2} \sum_i 2(y_i - \beta x_i) x_i = 0$$

$$0 = \frac{1}{\sigma^2} \sum_i (x_i y_i - \beta x_i^2)$$

$$\beta \sum_i x_i^2 = \sum_i x_i y_i$$

$$\hat{\beta} = \frac{\sum_i x_i y_i}{\sum_i x_i^2}$$

So, congratulations, we've re-derived least squares fitting from first principles. Instead of arbitrarily stating "hey, let's minimize the sum of squared residuals", we started from writing down a generative probability model of our data, specified our assumptions, and inferred a maximum likelihood point estimate of our desired unknown parameter β, given observed data. When you do least squares, you're implicitly assuming the same model.

To be careful, I suppose we also want to be sure that this solution (i.e., this critical point) we've found is a maximum, not a minimum or an inflection point, which means we want to show that the second derivative is negative there. Work it through and you'll see that the second derivative is

$$\frac{\partial^2 \log P(\text{data})}{\partial \beta^2} = -\frac{1}{\sigma^2} \sum_i x_i^2,$$

which is negative everywhere in β.

derivation of weighted least-squares regression

Suppose we didn't want to assume that all the variances are equal. For example, what if we knew a specific variance σ_i for each data point (x_i, y_i)? Then our log likelihood is:

$$\log P(\text{data}) = \sum_i \log \frac{1}{\sqrt{2\sigma_i^2 \pi}} - \sum_i \frac{(y_i - \beta x_i)^2}{2\sigma_i^2}$$

and when we differentiate it with respect to β and set to zero, the σ_i terms stay in the equation:

$$\frac{\partial \log P(\text{data})}{\partial \beta} = \sum_i \frac{(y_i - \beta x_i) x_i}{\sigma_i^2} = 0$$

$$0 = \sum_i \frac{y_i x_i - \beta x_i^2}{\sigma_i^2}$$

$$\beta \sum_i \frac{x_i^2}{\sigma_i^2} = \sum_i \frac{y_i x_i}{\sigma_i^2}$$

$$\beta = \frac{\sum_i \frac{y_i x_i}{\sigma_i^2}}{\sum_i \frac{x_i^2}{\sigma_i^2}}$$

If you look up **weighted least squares regression** you might see something like, *minimize the sum of the weighted squared residuals* $\sum_i w_i (y_i - \beta x_i)^2$ for some arbitrary weights w_i on each data point. Then you will probably see a recipe to use weights $w_i = \frac{1}{\sigma_i^2}$. Now you see why: once we assume a σ_i for each data point, the part of our log likelihood that depends on β is the term $\frac{(y_i - \beta x_i)^2}{2\sigma_i^2}$.

the zero-dimensional special case

Suppose that $x_i = 1$ for all i. Now $y_i = \beta + N(0, \sigma^2)$. In other words, $y_i = N(\beta, \sigma^2)$: just a Gaussian distribution.

For equal σ, the maximum likelihood estimate (MLE) for β is (plugging 1 in for x_i):

$$\hat{\beta} = \frac{\sum_i y_i}{n}$$

The familiar arithmetic mean is the MLE for the location parameter of a Gaussian, under the assumption that all samples were drawn with equal variance.

If the samples were drawn with unequal variances σ_i^2, we get:

$$\hat{\beta} = \frac{\sum_i \frac{y_i}{\sigma_i^2}}{\sum_i \frac{1}{\sigma_i^2}}$$

which is a familiar formula for a weighted arithmetic mean, with weights $w_i = \frac{1}{\sigma^2}$.

multiple regression

Suppose we extend our 0-intercept line $y = \beta x$ to one that has a nonzero intercept, $y = \beta_0 + \beta_1 x$. Then suppose we extend that to the case where we have multiple predictor variables $x_1 . . x_m$, so we have $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 . . . \beta_m x_m$. It'll be useful to imagine that we have a $x_0 = 1$ term in there too:

$$y = \beta_0 x_0 + \beta_1 x_1 + \beta_2 x_2 . . . \beta_m x_p$$

because then we can write this in compact matrix notation:

$$y = \mathbf{X}^T \beta$$

with the first element of $\mathbf{X}$ as 1.

The β are the **coefficients**. The $\mathbf{X}$ are the **predictors**.

To be a linear regression problem, y must be a linear function of the parameters β, or at least a linear function of some temporary parameters that we can derive our desired parameters from.

for example: harmonic regression

In this week's homework, we'll be seeing an example of harmonic regression, where the observed y are varying sinusoidally with time t, given a phase parameter ϕ and an amplitude A, as in:

$$y_t = A \sin(2\pi\omega(t + \phi))$$

This doesn't look like a linear regression problem, because this function is nonlinear in our parameter ϕ. But it turns out we can transform it into a linear regression problem, by using a trigonometric identity:

$$\sin(a + b) = \sin a \ \cos b + \sin b \ \cos a$$

We can use this on y_t to get:

$$y_t = \beta_1 \sin(2\pi\omega t) + \beta_2 \cos(2\pi\omega t)$$

where

$$\begin{aligned} \beta_1 &= A \cos(2\pi\omega\phi) \\ \beta_2 &= A \sin(2\pi\omega\phi) \end{aligned}$$

So we can use multiple linear regression to solve for coefficients β_1, β_2 given predictors $\sin(2\pi\omega t), \cos(2\pi\omega t)$ and the observed y_t, then obtain our A, ϕ fit from the β_1, β_2.

maximum likelihood optimization in SciPy

Suppose we want to find the "best" parameter values θ for our model, given some observed data D. People often call this "fitting" a model to the data. Thinking about it in the context of fitting the slope of a line is a good place to start.

maximum likelihood parameters, definition

If our model is a probability model, then we can probably write down the likelihood $P(\text{data} \mid \theta)$.

We'll typically immediately go from there to the **log likelihood** $\log P(\text{data} \mid \theta)$, both because it's easier to deal with mathematically (products become sums, and it's easier to differentiate functions that are sums), and because it's more stable numerically (products of probabilities quickly underflow your machine). They're monotonic: maximizing the log likelihood maximizes the likelihood.

Then a natural definition of "best" θ are the θ that maximize this likelihood. We call this **maximum likelihood estimation (MLE)**, and we call these the **maximum likelihood (ML)** parameters.

Notationally, we use a hat, $\hat{\theta}$, to indicate an optimized point estimate of some sort. We might specifically indicate an ML point estimate as $\hat{\theta}^{\text{ML}}$.

If we've got prior information that some values of θ are more likely than others, we might instead aim to optimize the product of the likelihood and the prior, which (by Bayes' theorem) is proportional to the posterior probability of θ:

$$P(\text{data} \mid \theta)P(\theta) \propto P(\theta \mid \text{data})$$

If this is what we're trying to maximize, instead of just the likelihood, we say we're looking for **maximum a posteriori** parameters, $\hat{\theta}^{\text{MAP}}$.

Notationally, we use argmax_θ to denote "find the values of θ that maximize whatever comes next". So we write:

$$\hat{\theta}^{\text{ML}} = \text{argmax}_\theta \log P(\text{data} \mid \theta)$$
$$\hat{\theta}^{\text{MAP}} = \text{argmax}_\theta \log P(\text{data} \mid \theta)P(\theta)$$

optimizing functions

Now let's forget about whether we're talking specifically about ML or MAP estimation. Suppose we have a generic function, $f(\theta)$, in terms of one or more parameters θ. It may happen to be a log

likelihood for us, but what we're now about to talk about applies to any $f(\theta)$.

The very first thing to get is that in numerical optimization, it just happens that it's conventional to talk in terms of **minimization** of functions, and optimizers (including `scipy.optimize.minimize`) work as minimizers. Of course, maximization and minimization of functions are trivially related to each other. Maximizing $f(x)$ is the same as minimizing $-f(x)$. ML optimization approaches will therefore typically implement a function that calculates the **negative log likelihood (NLL)**, so that function can be handed off to a numerical function minimizer.

OK, so we want to find the values θ that minimize our NLL function $f(\theta)$.

The intuition is that your function is a landscape: θ are your coordinates, and $f(\theta)$ is your elevation at those coordinates. Imagine a topo map, especially if you've ever been hiking in mountains. You're trying to find your way downhill to the lowest point.

In outline, a minimization algorithm typically works much like you would, if you were dropped into the wilderness somewhere:

- Start at random initial coordinates θ_0:
- Then iterate:
- look around and find a downhill direction
- move downhill some distance

This iterative cycle is reminiscent of expectation maximization.

generic interfaces to general function minimizers

This looks pretty general, and it makes function minimization a generic problem. Minimizers only need to be able to calculate $f(\theta)$ as they propose steps to new coordinates θ; they don't necessarily

need to know any of the details of what's in your function, or what it means.

So typically, a numerical minimization function has a function interface that asks you to provide a pointer to your **objective function** that returns a single scalar $f(\theta)$, and an initial guess θ_0. Your function $f(\theta)$ probably needs access to other information besides just the current point θ -- the observed data D for example -- so you typically pass a packet of information to the minimizer that the minimizer will just blindly pass on to your $f(\theta)$ every time it needs to call it.

For example, a minimal call to `scipy.optimize.minimize()` might look something like:

```
import scipy.optimize as optimize
import numpy as np

def nll(theta, D):
    # implement the log likelihood calculation in
    LL = ...
    return -LL

theta0 = np.random.xxx(n)   # guess a random start

result = optimize.minimize(nll, theta0, (D))   # (

# <result> contains the answer, plus other informa
```

global versus local optima

There is only one **global minimum**, and that's what you're after. But there might be **local minima**: points θ at which you're in a dip, and every way you look is uphill, so you have to climb out of the dip to make more progress downhill.

An important concept is whether your function is **convex** or not. If your function is convex, it has only one minimum. You can see the bottom (the global optimum) from anywhere you stand; any step downhill takes you toward the bottom, so you can't miss it. Formally a function is convex when $f(\theta)$ is always $\geq$ its tangents; or equivalently, $f(\theta)$ in the interval $[\theta_1, \theta_2]$ is always

$\leq$ the chord between the two points at $f(\theta_1)$ and $f(\theta_2)$; or equivalently, if the second derivative is always nonnegative.

Many functions are not convex, and many minimizers only walk downhill (i.e. are *local optimizers*). If we can only walk downhill, we can get stuck in a dip. It may even be an embarrassingly shallow dip, or you could be standing in a hole. Local optimizers can be appallingly stupid.

If you aren't sure whether your function is convex, and you aren't willing or able to prove it one way or the other, a standard thing to do is to do several runs of the optimizer, starting from different starting points (a luxury you wouldn't have if you were lost in the wilderness), and to take the best solution you find. The more you find the same best solution from independent start points, the more confident you get that that's the global optimum. If every start point gets you to a different local optimum, the more depressed you get that you're caught in a horrible landscape with no clear path to the goal.

Advanced optimizers might allow "jumps", testing whether a leap of some distance in θ might improve things (even if it means jumping over some uphill stuff), or might tolerate moving uphill for a bit (**simulated annealing** algorithms are one important class of algorithms that explore occasional uphill moves while trying to work downhill overall).

But we're still focused on more basic local minimizers for the moment. One key idea is how the minimizer knows where a downhill direction is.

Using partial first derivative (gradient) information

"Downhill" means the slope. A slope means the

rate of change of $f(\theta)$ with respect to an (infinitesimal) change in θ; and that means a first derivative. If we take the partial derivative of $f(\theta)$ at our current point θ with respect to one parameter θ_i, this tells us whether a move in the $+\theta_i$ direction is uphill or downhill, and how steeply; if it's uphill we will want to move in the $-\theta_i$ direction. If we take the partial derivative with respect to each θ_i in turn, we get the **gradient**:

$$\nabla f(\theta) = \frac{\partial f}{\partial \theta_1} a_1 + \frac{\partial f}{\partial \theta_2} a_2 + \ldots \frac{\partial f}{\partial \theta_n} a_n$$

where the a_i just indicate the axes (the orthogonal unit vectors).

If we evaluate the gradient at our point θ, we get a vector of n values, which tell us how steeply uphill or downhill we will go, if we moved an infinitesimal step in a direction (vector) a. If the gradient is all zeros, we're at a local optimum.

Since we're minimizing, we want to make our steps in the direction of the negative gradient, $-\nabla f(\theta)$.

If we can calculate the gradient mathematically, it's a big help to the minimizer. We need to provide a function that returns the gradient vector, given the same arguments as your objective function (the current point θ). SciPy rather haughtily calls this function the Jacobian, although as far as I understand it, the Jacobian is the term used when we have a vector-valued $f(\theta)$, not a scalar-valued $f(\theta)$, which is what we're working in (the NLL is a scalar value) and what `scipy.optimize.minimize()` works in. When you can calculate a gradient, the setup in SciPy looks something like:

```
import scipy.optimize as optimize
import numpy as np

def nll(theta, D):
    # implement the log likelihood calculation in
    LL = ...
    return -LL
```

```
def gradient_nll(theta, D)
    for i, theta_i in enumerate(theta):
        gradient[i] = ... # calculate df/dtheta_i
    return gradient

theta0 = np.random.xxx(n)   # guess a random start

result = optimize.minimize(nll, theta0, (D), jac=g

# <result> contains the answer, plus other informa
```

If you *don't* provide a function that calculates the gradient, then SciPy can calculate it numerically by the obvious method: take a tiny step in each direction i and see what happens. An informative thing to do, when you start using SciPy's `optimize.minimize()` function: have your $f(\theta)$ function print out where it currently is, and what the current negative log likelihood is. Now you'll get to see every time `scipy.optimize.minimize()` calls your function, and what it's trying to do. If you didn't provide a gradient function, you'll see it gingerly explore a tiny step on each axis (each of the individual parameters i in your θ vector), and compare the new $f(\theta)$ to where it was. This delicate stepping you're watching is SciPy finding the gradient by numerical brute force.

If you implemented it yourself, you'd find that the numerical method is prone to numerical instabilities. If we step too far, we might hit a wall and not get the correct local slope; if we step too close, the difference we measure might be so small that it underflows our machine. Implementations (presumably including SciPy's) have to be careful.

bounds and constraints on parameter values

Generic numerical optimizers, including `scipy.optimize.minimize()`, assume that all your θ_i can take on any real value $-\infty . . \infty$. But we're often going to have bounds and/or constraints on our parameters. For example, if we were fitting a Gaussian distribution to our data (parameters μ, σ), the standard deviation parameter σ is

nonnegative. If we were fitting a multinomial distribution (e.g. the unknown parameters of a loaded die), our parameters θ are probabilities, subject to $0 \leq \theta_i \leq 1$ and $\sum_i \theta_i = 1$.

SciPy's `optimize.minimize()` allows you pass lower and upper **bounds** for each θ_i as a list of n tuples, using `None` wherever there's no bound, so you can do something like:

```
bnd = [ (None,None), (0.0,None) ]
result = optimize.minimize(gaussian_nll, theta0, (D
```

when your parameters θ are a Gaussian μ, σ, for example.

If your parameters are probabilities, you might think of trying

```
bnd = [ (0.0, 1.0) for i in range(n) ]
result = optimize.minimize(multinomial_nll, theta0,
```

but that won't be enough; you're still missing the $\sum_i \theta_i = 1$ constraint.

If we were solving for an optimum analytically, rather than numerically, this is the point where I'd tell you about (or remind you of, depending on your background) how we solve constrained optimization problems in calculus using Lagrange multipliers. We use Lagrange multipliers a lot in optimizing probability parameters while maintaining a constraint of $\sum_i p_i = 1$.

One common trick in the arsenal is to use a change of variables to transform a bounded/constrained variable to an unconstrained real value and vice versa. For example, if I want $\sigma > 0$, I can define a real-valued variable θ such that:

$$\sigma = e^{\theta}$$

and now an optimizer can optimize θ to its heart's content on the full range $-\infty . . \infty$, while σ will be strictly positive.

For probabilities, we can use a more complex change of variables that enforces the sum-to-one constraint:

$$p_i = \frac{e^{\theta_i}}{\sum_j e^{\theta_j}}$$

When we differentiate to get a gradient function to help the minimizer out, we have to do our partial derivatives of our function in terms of the unconstrained θ. This will tend to create a strangely satisfying little calculus exercise, just hard enough to be a challenge, but easy enough for even a former wet lab biologist to solve.

working examples

Here is a working example of using `scipy.optimize.minimize()` to find maximum likelihood parameters for a toy problem (data drawn from a Gaussian distribution), including using a bounds argument to make sure the σ parameter is positive, compared to an alternative solution that uses a change of variables trick $\sigma = e^{\theta}$ to allow the optimizer to optimize real-valued θ that transforms to a positive σ.

After you've seen that one (with its extensive comments), here is another working example that optimizes the probability parameters of a multinomial distribution, using the change of variables trick. (In its simplest possible form. This implementation will blow up with numerical overflows if you tried it on large numbers of observed counts.)

further reading (if you're so inclined)

- "Linear Regression", chapter 3 of James, Witten, Hastie, and Tibshirani's *An Introduction to Statistical Learning*; a PDF of the book is available freely online.

- "Mathematical optimization: finding minima of functions" in the on-line SciPy Lecture Notes series

- "Minimization or Maximization of Functions", chapter 10 of *Numerical Recipes in C*, by Press, Teukolsky, Vetterling, and Flannery. This is an all-around terrific book, if you want to get deeper into how numerical algorithms work, and how to code them.
