week 13: t-SNE

Many old friends make their farewell appearance in this final week of the course: Student's t distribution, entropy of probability distributions, optimization of objective functions using gradient information, and principal components analysis among them. My goodness, it's almost as if there was a plan all along...

again the problem of visualizing "high dimensional" data

Recall that RNA-seq data is "high dimensional" in the following sense. RNA-seq data is a matrix of n samples $\times m$ genes, with mapped read counts or some other sort of quantitation (such as TPM) for each gene in each sample. Each sample (a cell, tissue, or cell type) is associated with a list of expression levels for m genes; we can think of this as a vector in an m-dimensional space. Samples with similar gene expression patterns are close in "gene space", as measured by Euclidean distance (or some other distance measure we might choose).

We can just as readily think of each *gene* being associated with a list of expression levels in n samples, hence as a vector in an n-dimensional space, and we might want to see how genes cluster in "sample space".

We want the distances in these "spaces" to reflect biologically meaningful similarity between two samples, or two genes. We have to be careful about what the numbers in these vectors are, and what we think they mean. If we're using mapped read counts, we'd want to normalize them somehow to account for the fact that one sample

might have more total reads than another; we don't want to consider two samples to be different and distant just because we mapped more reads to one of them. If we're going to make some sort of assumption about how distances are distributed -- for example, if we're going to use some clustering technique that assumes Gaussian distributions -- we might want to put our matrix in units of log counts, not raw counts, as we've seen in using k-means clustering with RNA-seq data.

It's difficult to visualize clustering in high-dimensional spaces. There are a variety of techniques for trying to project high-dimensional data down to two or three dimensions for visualization purposes. We've already seen one of the main techniques, principal components analysis (PCA). A limitation of PCA is that it corresponds to a rigid orthogonal rotation of the original space followed by a projection to 2D - you're projecting the data onto a plane through the original space. If there are only a few well-separated clusters, dominated by variance along only a few directions, PCA can do a fine job. But in many cases, a linear method like PCA isn't good enough, and we need to do some sort of nonlinear projection into 2D.

A widely used nonlinear visualization method for high-dimensional data called *t-SNE* was introduced by Laurens van der Maaten and Geoff Hinton in 2008. t-SNE stands for **t-distributed stochastic neighbor embedding**. t-SNE produces evocative and strangely beautiful visualizations. For that reason as much as any, it has become extraordinarily popular in biological data analysis and machine learning in general. A t-SNE plot is almost *de rigueur* in a single cell RNA-seq paper these days, though the cool kids are starting to switch to alternative algorithms like UMAP. The figure below shows an example from a paper that sequenced 42,035 single cells from whole *C. elegans* animals, showing how cells cluster into tissues like neurons and muscle.

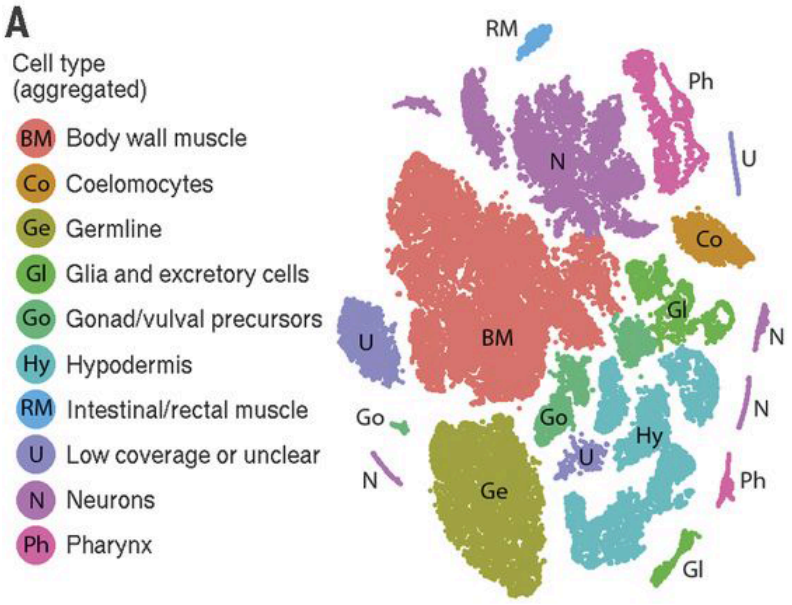

Figure 3A from [Cao et al. 2017], showing a t-SNE plot of 42,035 *Caenorhabditis elegans* single cells.

What does t-SNE do? How does it work? What does it assume?

the key idea behind t-SNE (and SNE in general)

Each data point is given an ordered preference for a relatively small number of neighbors in the original space. In the so-called "embedded space" (in the two-dimensional t-SNE plot), the points are arranged to preserve that preference and order.

A bit more specifically, each point has a specific preference in terms of a probability distribution for "visiting" neighboring points. As it visits its neighborhood, some points are visited more than others; distant points are rarely if ever visited.

The size of the "neighborhood" is defined as an effective number of neighbor points that each point cares about. This is a controllable parameter to t-SNE. Perplexingly, it's called the **perplexity**. We'll see why; it's information theory jargon, related to Shannon entropy and the precise definition of what we mean by an "effective number" of neighbors, when they have different visitation probabilities.

The particular form of the probability distribution

that t-SNE uses in the original high-dimensional space is a spherical Gaussian, where each point has its own parametric standard deviation σ_i; whereas in the embedded low-dimensional space, it uses the same distribution for every point (a Cauchy distribution, a.k.a. a Student t distribution with one degree of freedom). This has the effect that points in sparse versus dense regions in the original space tend to get more evenly distributed in the embedded space, because t-SNE cares about preserving neighbor relations, not absolute distances.

This evenness probably contributes to the visually pleasing geometry of t-SNE plots.

Go visit this excellent article on t-SNE, written by Martin Wattenberg, Fernanda Viegas, and Ian Johnson at Google. The article is written in distill.pub, a fabulous hypermodern forum for articles that use high-quality, interactive data visualization to explain machine learning techniques. It will give you a good high-level intuition for t-SNE -- and it uses MCB112-style control experiments, using t-SNE to project *known 2D data clusters* into 2D t-SNE space, and you can watch many ways in which t-SNE fails spectacularly but always beautifully.

I think the distill.pub article has one problem: it doesn't sufficiently consider the fact that t-SNE's objective function isn't convex, which means it can get stuck in local optima. The article doesn't seem to give enough consideration to the fact that some of the bizarre plots may simply be spurious local optima that would look more sensible if you re-ran t-SNE a few times and chose a better optimum. The local optimum problem only comes up pretty far down in the article, and even then, it presents results of 5 runs as if they're equivalent, without considering the value they achieved for the objective function.

the road to t-SNE lies through SNE

My explanation here will follow the concise and clear explanation in the 2008 van der Maaten and Hinton paper. They start by explaining SNE, stochastic neighbor embedding without the t, which was introduced in 2002 by Geoff Hinton and Sam Roweis.

We are given n data points $x_1 . . x_n$, each of which is a vector in an m-dimensional space.

We aren't going to need to index the dimensions; we're going to write everything in compact vector

notation. But just so you remember: when we write $||x_i - x_j||$ for the Euclidean distance between points x_i and x_j (i.e. the Euclidean length, or *norm*, of the resultant vector $x_i - x_j$), that's $\sqrt{\sum_{d=1}^{m}(x_{i,d} - x_{j,d})^2}$. The fancy name for an ordinary Euclidean distance is the L^2 (or just L2) norm.

We aim to project the original points $\mathbf{X}$ to new data points $\mathbf{y}_1 . . \mathbf{y}_n$ in two dimensions.

First we define a conditional probability $p_{j|i}$ for the probability of "visiting" neighbor j starting from point i. If we assume that we visit neighbor j as a function of its distance in proportion to a spherical Gaussian distribution, then the probability that we visit j as opposed to all other points (other than i itself) is:

$$p_{j|i} = \frac{\exp\left(-||x_i - x_j||^2 / 2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-||x_i - x_k||^2 / 2\sigma_i^2\right)}$$

The $p_{i|i}$ are set to zero.

To do this, we'll need a σ_i for each point i. We are going to set σ_i so that point i has a certain *effective number of neighbors*. We'll make that notion precise using Shannon entropy. The Shannon entropy of the $p_{j|i}$ distribution is:

$$H_i = -\sum_j p_{j|i} \log_2 p_{j|i}$$

You can think of a Shannon entropy as the number of yes/no questions it takes to specify one of N equiprobable outcomes; for example, it takes 2 bits to specify one of the 4 DNA nucleotides (2^2), and 10 bits to specify one of the 1024 possible DNA sequences of length 5 ($2^{10} = 1024$). Turning that around, if I let you ask H yes/no questions, you could distinguish one of 2^H equiprobable possibilities. For non-uniform probabilities, we can still calculate 2^H and consider it some sort of "effective" number of

possible outcomes for our probability distribution. This is sometimes called the **perplexity** of a probability distribution: $\text{Perplexity} = 2^H$.

The larger σ_i is, the more spread out the Gaussian, and our point i can visit more neighbors. As σ_i decreases toward zero, the Gaussian contracts and only the one closest neighbor is ever visited, relative to the others. For each point i, we dial σ_i in to get 2^{H_i} to our chosen perplexity value. (We'll talk about the numerical details of that below; it becomes an example of a general class of problems called *one dimensional root-finding* problems).

It's worth noting that the perplexity is a *global* parameter to SNE: all the points i want to visit the same number of neighbors, even if some i are in dense clusters and some are outliers in the original space. This is responsible for some of the counterintuitive behaviors described in the distill.pub t-SNE visualizations.

In the embedded 2D space, we define the neighbor relations with conditional probabilities $q_{j|i}$ that are also calculated as Gaussians, but now with a *fixed* $\sigma = \frac{1}{\sqrt{2}}$ for all i, so that:

$$q_{j|i} = \frac{\exp\left(-\|y_i - y_j\|^2\right)}{\sum_{k \neq i} \exp\left(-\|y_i - y_j\|^2\right)}$$

with $q_{i|i}$ defined to zero.

Now we try to find positions $y_1 . . y_n$ that make all $q_{j|i} \sim p_{j|i}$. One typical measure for the difference between two probability distributions P and Q is the so-called Kullback-Leibler divergence, $\text{KL}(P\|Q) = \sum_i P_i \log \frac{P_i}{Q_i}$. The sum of the n KL divergences for all points i and their neighbor distributions is:

$$f(\mathbf{y}) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$$

I write that this is a function $f(\mathbf{y})$, because what we're going to do in SNE is to find points $\mathbf{y}$ that minimize this objective function, given the original points $\mathbf{x}$. We can use standard numerical function minimization, starting from a random guess at the $\mathbf{y}$. The objective function is differentiable, so we also have gradient information to help this minimization. (But I won't bother with showing the gradient for SNE, since we're just using it as a stepping stone on the way to t-SNE, coming up next.)

The SNE objective function isn't convex; it has local optima. We at least need to do multiple optimization runs from multiple starting points. The local optima problem turns out to be very serious. The optimization surface is very rugged. Points get stuck, unable to move past each other in the iterations of the optimization. SNE implementations use special tricks to help, as discussed in the van der Maaten paper.

The van der Maaten paper describe some of the problems of SNE, one of which is the *crowding problem*. To alleviate these problems (which I won't go further into), the 2008 paper introduced t-SNE.

on to t-SNE

t-SNE makes a small change to the calculation of P (the probabilities in the original high-dimensional space), and a big change in the calculation of Q in the embedded space.

First t-SNE converts the asymmetric SNE $p_{j|i}$'s to symmetric p_{ij} "joint probabilities" that sum to one over all i, j by:

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$$

It seems to me that this is essentially arbitrary -- the symmetric version is a little easier to work with because it has a simpler gradient. There are

other ways to symmetricize the P, but this one has an advantage that each point i is treated with uniform probability $p(i) = \frac{1}{n}$, so that we're considering two ways of obtaining the joint p_{ij} either by $p_{j|i}p(i)$ or $p_{i|j}p(j)$, and since these aren't the same number (because what we're doing is arbitrary!), we're just averaging them. This is not the heart of what makes t-SNE tick, but it's part of the definition of the method.

t-SNE also assumes a symmetric q_{ij} distribution, but here is where the big change is. t-SNE assumes a *heavy-tailed* distribution. It will assign higher probabilities to points that are further away in the 2D embedded space than they were in the original m-dimensional space. This alleviates the crowding problem: points that were clustered in the original roomy hypervolume need to be laid out in a much more cramped area in a mere two dimensions, and something has to give, so we want to let them spread out some. Specifically, t-SNE assumes a Student t distribution with one degree of freedom, also known as the Cauchy distribution, and calculates:

$$q_{ij} = \frac{\left(1 + \|y_i - y_j\|^2\right)^{-1}}{\sum_{k \neq l} \left(1 + \|y_k - y_l\|^2\right)^{-1}}$$

Again the q_{ii} are defined as zero. Notice that the denominator is a sum over *all* pairs $k \neq l$, not just a sum over the possible neighbors $k \neq i$ for one point i: this is what makes the q_{ij} a symmetric "joint probability" that sums to one over all i, j.

With these definitions for t-SNE's P and Q, the KL objective function becomes:

$$f(\mathbf{y}) = \mathrm{KL}(P\|Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

We'll seek points $\mathbf{y}$ to minimize this KL divergence, using numerical optimization starting from a randomly chosen point. The gradient is:

$$\frac{\partial f}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + \|y_i - y_j\|^2)^{-1}$$

(When you go to implement this, notice that p_{ij} and q_{ij} are scalars, and $(1 + \|y_i - y_j\|^2)^{-1}$ is a scalar that you calculate from the distance between the y_i and y_j vectors... but $(y_i - y_j)$ is a *vector*, with two dimensions. The gradient is an $n \times 2$ matrix.)

t-SNE in a nutshell

Given:

- data points $X = x_1 . . x_n$, the rows of an $n \times m$ matrix.
- the chosen perplexity.

Calculation:

- Calculate $p_{j|i}$, fitting σ_i to achieve the chosen perplexity.
- Calculate $p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$.
- Sample random starting points (2D vectors) $y_1 . . y_n$ from $\mathcal{N}(0, 10^{-4} I)$
- Use numerical optimization to minimize the KL distance $f(\mathbf{y})$, using gradient information $\frac{\partial f}{\partial y_i}$.

Result:

- data points $Y = y_1 . . y_n$ in the embedded 2D t-SNE space.

The only part of that that we didn't discuss already is the sampling of the initial y_i points, which shouldn't be super important; the t-SNE paper recommends choosing them from a small 2D Gaussian centered at zero. That's what $\mathcal{N}(0, 10^{-4} I)$ means: $\mathcal{N}(\mu, \sigma^2)$ is customary notation for a Gaussian random variable, I is the 2D identity matrix, and 10^{-4} is a small variance.

three practicalities for the pset

1. one dimensional root-finding using `scipy.optimize.bisect()`

To calculate the t-SNE p_{ij} stochastic neighbor distribution, you need the SNE $p_{j|i}$ conditional distributions (n of them, for n data points). To calculate the $p_{j|i}$, you need a σ_i for each data point, which is setting the width of the Gaussian distribution that point i uses to define its stochastic neighbors.

For each data point i, for a given σ_i, we calculate the Shannon entropy H_i of the $p_{j|i}$ distribution; 2^{H_i} is the perplexity of the distribution, the effective number of neighbors. We'll dial σ_i in to get the perplexity we want.

When σ_i is smaller, the stochastic neighbor distribution is tighter, reaching out to fewer neighbors; the perplexity is low. For larger σ_i, more neighbors, larger perplexity. Perplexity is a monotonic increasing function of σ_i, and $\sigma_i > 0$. It's a simple function and there's got to be a solution.

If we cast this problem in terms of an objective function $f(\sigma_i)$ that's the difference between our current perplexity and the target perplexity, then we're looking for the value of σ_i that achieves $f(\sigma_i) = 0$. Finding a scalar parameter x such that $f(x) = 0$ is a general problem called a *root-finding problem*, which arises in many sorts of situations. It tends to get a chapter of its own in books on scientific numerical methods (such as *Numerical Recipes in C*, my go-to source).

The first step in any rootfinding problem is for you to identify a *bracketing interval*: points a and b where $f(a)$ and $f(b)$ have different signs, so there

must be a point $f(x) = 0$ in between them. A root finder is then going to narrow down that interval until it finds the root.

In our problem, with $f(\sigma_i)$ a monotonically increasing function, for $a < b$ we need to identify points such that $f(a) < 0$ and $f(b) > 0$. In a function as well behaved as what we've got here, and since we know $\sigma_i > 0$, we can lay down a test point anywhere and keep moving it toward (or away) from zero until we get the sign we need. For example, for the a side, we can do something like:

```
a = 1.0
while f(a) >= 0: a /= 2
```

By taking multiplicative steps, we guarantee that $a > 0$.

There are a variety of root finders in SciPy, including fancy methods that try to accumulate information about your $f(x)$ as you try points x, to make sophisticated predictions about where $f(x) = 0$. A simple and robust method is *bisection*, implemented in scipy.optimize.bisect().

To call `scipy.optimize.bisect()`, we provide it with the name of the function that implements $f(x)$; with the bracketing endpoints a, b; and, if necessary (and it will be necessary for t-SNE), with a tuple of data that `.bisect()` will pass as arguments to your objective function. For example, the following pseudocode:

```
def my_perplexity_diff(sigma, Di):
   # calculate the perplexity for distance matrix
   # return that minus the desired perplexity: we'

# find your a,b bracketing interval

sigma[i] = scipy.optimize.bisect(my_perplexity_dif
```

and you'd do that for each i.

I slipped `D[i]` in there without defining it. Your original data are a $n \times m$ matrix X. The $p_{j|i}$ are calculated in terms of Euclidean distances

between row vectors x_i and x_j. As you dial in the σ_i, you're going to calculate those pairwise distances over and over again. You can save compute time by precalculating an $n \times n$ matrix D_{ij} of all pairwise distances between i, j points. Moreover, then for a given i, a row D_i is sufficient to calculate $p_{j|i}$. You'll see that my code for the pset is written to take advantage of a precalculated D_{ij} distance matrix, and the above pseudocode reflects that too.

2. providing gradient information to `scipy.optimize.minimize()`

We've already used the SciPy numerical optimizer in the week on regression, and you might want to look back at that. But we didn't provide the minimizer with gradient information. We didn't have to then; we were optimizing a pretty simple, low-dimensional log likelihood objective function. With t-SNE, we're in a more complex situation, and we want to use gradient information.

For t-SNE, you're going to create an objective function that takes a current set of points Y (a $n \times 2$ array), and returns the KL divergence between the stochastic neighbor distribution P in the original space (which you'll have calculated already; you only need to do that once) and Q in the embedded space (which depends on your current point Y). So your objective function $F(Y)$ in pseudocode is going to look something like:

```
def KL_dist(Y, P):
    Q  = calculate Q from Y
    KL = calculate KL(P||Q)
    return KL
```

The gradient $\nabla F(Y)$ is a $n \times 2$ matrix: how much $F(Y)$ changes, for a small change in one of the two dimensional coordinates of one point y_i. van der Maaten (2008) show the equation for the gradient on p.2584 of their paper. It depends on

Y and P too, and on calculating Q given the current Y. You could write a separate function to calculate the gradient, but then you'd have to calculate Q twice, wastefully.

It doesn't seem to be super well documented, but you can optionally have your objective function return the gradient too. This can save some calculation, compared to having a separate gradient-calculating function, because the calculation of $F(Y)$ and $F'(Y)$ often share computationally intensive calculations -- like here, where we're really rather just calculate Q once, for a given Y. So in pseudocode our objective function will look something like:

```
def KL_dist(Y, P):
    Q  = calculate Q from Y
    KL = calculate KL(P||Q)

    gradient = np.array(Y.shape)
    # calculate each gradient term...

    return KL, gradient
```

Now a detail. The SciPy optimizer is very general - it doesn't want to know about the dimensionality of your problem, it just wants to deal in terms of one big vector of current Y, and SciPy also wants the gradient to be one big vector. We use `np.flatten()` to convert a matrix to a single vector for SciPy, and we use `np.reshape()` to convert a vector back to a matrix. P is fine as an array -- SciPy passes the extra constant arguments in exactly the form you provided them.

In pseudocode, then:

```
def KL_dist(Y, P):
    n  = P.shape[0]   # or length(Y) / 2
    Y  = np.reshape(Y, (n,2))

    Q  = calculate Q from Y
    KL = calculate KL(P||Q)

    gradient = np.array(Y.shape)
    calculate each gradient term

    return KL, gradient.flatten()
```

The hard part is writing the objective function and getting its input and output in the form that SciPy needs, with the input Y and the output gradient as flattened vectors of $2n$ elements. To call the minimizer then looks like:

```
Y = np.random.normal(0., 1e-4, (n,2))       # This
result = scipy.optimize.minimize(KL_dist, Y.flatte
```

Recall that args=(P) is the interface for handing SciPy a bundle of constant additional data (as a tuple) that it needs to pass to the objective function (as additional arguments); here, the stochastic neighbor distribution in the original space that we're trying to match.

`jac=True` is the poorly-documented way of telling SciPy that our objective function is going to return both value and the gradient. "jac" means Jacobian matrix. The other way to do this is to pass the name of a function that calculates the gradient, i.e. `jac=my_gradient_func`.

When the minimizer returns, it passes a tuple of information back. The parts we care about are:

- `result.x`: the optimized solution -- our embedded Y coordinates, as a nx2-long vector that we'll reshape.
- `result.success`: True if the minimizer succeeded. We ought to check this.
- `result.fun`: the objective function value at the solution.

You'll take `result.x` and reshape it to an $n \times 2$ matrix, and and that's your t-SNE plot in two dimensions.

3. using `sklearn.manifold.TSNE()`

After all this, you will either be pleased or infuriated to know that using the canned t-SNE implementation in scikit-learn is super easy:

```
from sklearn.manifold import TSNE

X = your (n,m) data array

Y = TSNE(perplexity=10).fit_transform(X)

# Y is now (n,2); plot it.
```

Why didn't I tell you that in the first place? One of the lessons of MCB112 is we don't want to be wielding a powerful canned data analysis routine without respectful understanding.