week 05: mixed feelings

goals this week

- Introduction to *single-cell RNA-seq* analysis, and how we use it to identify new "cell types" by clustering.
- Understanding the *K-means clustering* algorithm, one of a few basic clustering algorithms.
- Viewing K-means clustering as a *mixture model*, an important class of generative probability model, and viewing the K-means clustering algorithm as an example of *expectation maximization*.

clustering single cell RNA-seq data

Starting around 2009 people started doing RNA-seq experiments on single cells, and the technologies for doing these experiments have been advancing rapidly. Single cell RNA-seq is a major breakthrough because it allows us to see cell-to-cell heterogeneity in transcriptional profiles, including different cell types. So long as we're willing to use transcriptional profile as a proxy for "cell type", by using single-cell RNA-seq we can comprehensively survey all the cell types present in some tissue. This promises to be especially powerful in understanding complex biological structures composed of many cell types, like brains.

A single mammalian cell only contains about 400K mRNAs, and many are inevitably lost in the experimental procedure. Depending on the

protocol, perhaps a few million different fragments are generated per cell. There's little point in sequencing single-cell libraries to the depth of a standard RNA-seq library (30M-100M reads). Experiments might obtain 200K-5M reads per cell, and often as few as 50K [Pollen et al 2014].

These reads are distributed over 20K mammalian gene transcripts, so we only get small numbers of counts per typical gene: on the order of 1-100 mapped read counts, depending on sequencing depth. Only highly expressed genes are detected reliably, and low-level transcripts may not be sampled at all. There is an ongoing argument over whether the number of zeros is more than expected (called "dropout"). Some statistical modeling of single-cell RNA-seq data includes a model of the dropout effect [Kharchenko et al, 2014].

For these reasons single-cell RNA-seq experiments typically detect expression of fewer genes than population RNA-seq experiments do. Population RNA-seq experiments are better at deeply sampling the "complete" transcriptome of a population of cells. Single cell experiments are most useful for identifying enough higher-expressed genes to distinguish useful cell types. These used to be called marker genes, when we had few of them. Cell types are still often operationally identified by a single marker gene they express highly, such as the parvalbumin-positive (Pv+) interneurons in mammalian hippocampus.

One can pool single cell data after clustering to approximate population RNA-seq on purified cell type populations, although the higher technical difficulty of single cell RNA-seq experiments probably makes this an imperfect approximation, with higher technical noise.

Single cell RNA-seq experiments are very noisy. The amount of material is small, and has to be

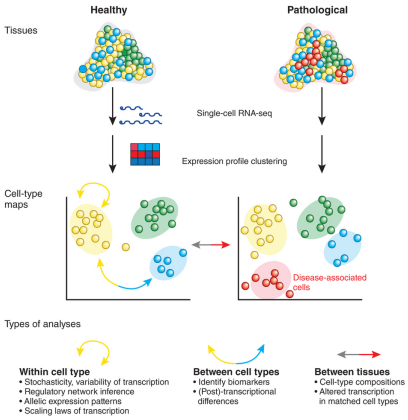


Figure 1 from R. Sandberg, Entering the era of single-cell transcriptomics in biology and medicine, Nature Methods 11:22 (2014). *('open image in new tab' to embiggen)*

amplified with noisy amplification protocols. Important experimental trickery is used, including the use of UMI barcodes (unique molecular identifiers), to detect and correct amplification artifacts. One (slightly aging) review of the experimental design, analysis, and interpretation of single-cell RNA-seq experiment is [Stegle, Teichmann, and Marioni, 2015].

The heart of the analysis, and what concerns us here, is that we want to **cluster** the transcriptional profiles of the single cells, to identify groups of cells with similar profiles.

Our particular focus this week is on understanding a classic clustering algorithm called **K-means clustering**.

k-means clustering

We have N data points X_i, and we want to assign them to clusters.

Each X_i is a point in a multidimensional space. It has coordinates X_{iz} for each dimension z. In two dimensions ($Z = 2$) we can visualize the X_i as dots on a scatter plot. It's easiest to think about K-means clustering in 2D, but important to remember that many applications of it are in high dimensions. Gene expression vectors for G different genes in each cell are points in a G-dimensional space.

A **cluster** is a group of points that are close together, for some definition of "close". The easiest to imagine is "close in space". The Euclidean distance between any two points X_i and X_j is

$$d_{ij} = \sqrt{\sum_z (X_{iz} - X_{jz})^2}$$

definition of a k-means

clustering

Suppose we know there are K clusters. (This is a drawback of K-means: we have to specify K.) Suppose we assign each point X_i to a cluster (i.e. a set) C_k, so we can refer to the indices $i \in C_k$. We can calculate the **centroid** of each cluster, μ_k: its mean location, i.e. its center of mass, simply by averaging independently in each dimension z:

$$\mu_{kz} = \frac{\sum_{i \in C_k} X_{iz}}{|C_k|}$$

where the denominator $|C_k|$ is the number of points in cluster k -- the *cardinality* of set C_k, if you haven't seen that notation before.

The centroid is just another point in the same multidimensional space. We can calculate a distance d_{ik} of any of our items X_i to a centroid μ_k, the same way we calculated the distance between two points i, j.

For any given assignment C consisting of K sets C_k, we can calculate the K centroids μ_k, and then calculate the distance d_{ik} from each point X_i to its assigned centroid μ_k. We can calculate a total *squared distance* for a clustering C:

$$\begin{aligned} D_C &= \sum_k \sum_{i \in C_k} d_{ik}^2 \\ &= \sum_k \sum_{i \in C_k} \sum_z (X_{iz} - \mu_{kz})^2 \end{aligned}$$

A **K-means clustering** of the data is defined by finding a clustering C that minimizes the total squared distance D_C, given K. We literally try to place K means (centroids) in the space such that each point X_i is close to one of them. That's the *definition* of a K-means clustering; but how do we *find* one?

Why *squared* distance? Why doesn't it just minimize the sum of the distances? We'll see.

k-means clustering algorithm

The following algorithm almost does the job — well enough for practical purposes. (I say almost, because it is a local optimizer rather than a global optimizer, as we'll discuss in a sec.) Start with some randomly chosen centroids μ_k. Then iterate:

- **Assignment step:** Assign each data point X_i to its closest centroid μ_k.
- **Update step:** Calculate new centroids μ_k.

Iterate until the assignments don't change.

It is possible for a centroid to have no points assigned to it, so you have to do something to avoid dividing by zero in the update step. One common thing to do is to leave such a centroid unchanged.

The algorithm is guaranteed to converge, albeit to a local rather than a global optimum.

The k-means algorithm is sometimes also called *Lloyd's algorithm*.

local minima problem; initialization strategies

K-means is extremely prone to spurious local optima. For instance, you can have perfectly obvious clusters yet end up with multiple centroids on one cluster and none on another.

People take two main approaches to this problem. They try to choose the initial guessed centroid locations "wisely", and they run K-means many times and take the best solution (best being the clustering that gives the lowest total distance D_C).

Various ways of initializing the centroids include:

- choose K random points in space;
- choose K random points from X and put the initial centroids there;
- assign all points in X randomly to clusters C_k, then calculate centroids of these.

soft k-means

Standard K-means is distinguished by a *hard* assignment: data point i belongs to cluster k, and no other cluster. We can also imagine doing a *soft* assignment, where we assign a probability that point i belongs to each cluster k.

Let's introduce A_i, the *assignment* of point i, so we can talk about $P(A_i = k \mid X_i, \mu)$: the probability that we assign point X_i to cluster k given the K-means centroids $\mu = \mu_1 .. \mu_K$.

What should this probability be, since we haven't described K-means in probabilistic terms? (Not yet, anyway.) The standard definition of soft k-means calculates a "responsibility" r_{ik}:

$$r_{ik} = \frac{\exp\left(-\beta d_{ik}^2\right)}{\sum_{k'} \exp\left(-\beta d_{ik'}^2\right)}$$

where $\sum_k r_{ik} = 1$ for each point i, since the denominator forces normalization. We call it a "responsibility" instead of a probability because we haven't given it any well-motivated justification (again, not yet).

People call the β parameter the *stiffness*. The higher β is, the closer a centroid k has to be to a point i to get any responsibility r_{ik} for it. In the limit of $\beta \to \infty$, the closest centroid gets $r_{ik} = 1$, and we get standard K-means hard clustering as a special case.

The centroid of each cluster k is now a *weighted* mean, with each point X_i weighted by its responsibility (remember that each point is a vector; we're going to drop the indexing on its dimensions, for more compact notation):

$$\mu_k = \frac{\sum_i r_{ik} X_i}{\sum_i r_{ik}}$$

The soft K-means algorithm starts from an initial guess of the μ_k centroids and iterates:

- **Assignment step:** Calculate responsibilities

r_{ik} for each data point X_i, given the current centroids μ_k.

- **Update step:** Calculate new centroids μ_k, given responsibilities r_{ik}.

Fine, but this "responsibility" business looks *ad hoc*. Is there a justification for what we're doing?

mixture models

A **mixture model** is a generative probability model composed of other probability distributions. A mixture model is specified by:

- Q **components**.
- each component has a **mixture coefficient** π_q, the probability of sampling each component.
- each component has its own parameters θ_q, and generates a data sample x with some probability $P(x \mid \theta_q)$.

The probability of sampling (generating) an observed data point x goes through two steps:

- sample a component q according to its mixture coefficient π_q
- sample data point x from the component q's probability distribution $P(x \mid \theta_q)$.

This is a simple example of a *hierarchical model*. More complicated hierarchical models have more steps. It's also a simple example of a *Bayesian network* composed of two random variables, with an random variable for the observed data dependent on a hidden random variable for the component choice.

The probability that an observed data point x is generated by a mixture model is the marginal sum over the unknown component q:

$$P(x \mid \theta) = \sum_q \pi_q P(x \mid \theta_q)$$

If we had to infer which (hidden) component q generated observed data point x, we can calculate the posterior probability:

$$P(q \mid x) = \frac{\pi_q P(x \mid \theta_q)}{\sum_{q'} \pi_{q'} P(x \mid \theta_{q'})}$$

The mean (average) over many data points X_i assigned to a mixture is, for each component q:

$$\mu_q = \frac{\sum_i P(q \mid X_i)\, X_i}{\sum_i P(q \mid X_i)}$$

You can see this is the update step of soft K-means, with a probability $P(q \mid X_i)$ in place of the ad hoc responsibility r_{ik}. But there's a missing piece: we haven't said anything yet about what the component probability distributions $P(x \mid \theta_q)$ can be.

That's because the components $P(x \mid \theta_q)$ can be anything you like. It's most common to make them all the same elemental type of distribution. So we can have *mixture exponential* distributions with parameters λ_q; we can have *Poisson mixture* distributions also with parameters λ_q; we can have *mixture multinomial* distributions with parameters p_q; and so on.

For example, one common mixture model is a *mixture Gaussian* distribution, with parameters μ_q and σ_q for each component. This might arise in trying to fit data that had multiple modes (peaks), with different peak widths, and different proportions of data in each peak. A mixture Gaussian in a *single* dimension would give:

$$P(x, q \mid \theta) = \pi_q \left(\frac{1}{\sqrt{2\pi\sigma_q^2}} \exp\left(\frac{-(x-\mu_q)^2}{2\sigma_q^2} \right) \right)$$

We could use a two-component mixture Gaussian to specify a bimodal distribution for a real-valued random variable X, for example.

soft K-means as a mixture probability model

Let's stare at the mixture Gaussian model for a while, and compare it to soft K-means. Specifically, consider what the posterior probability $P(q \mid X_i, \theta)$ looks like for a multidimensional mixture Gaussian, compared to the responsibility r_{ik} calculation.

First notice that the $(x - \mu_q)^2$ term in the one-dimensional Gaussian looks sort of like it's parallelling the d_{iq}^2 term in the multidimensional k-means responsibility. Turns out there's an especially simple multidimensional generalization of the Gaussian, a **spherical Gaussian**, where we replace a one-dimensional distance $(x - \mu)$ from the mean with the multidimensional distance d_{iq}. A spherical Gaussian is still controlled by a single variance parameter σ^2; its probability density falls off symmetrically in all directions from its mean. (More general multidimensional Gaussian distributions allow different variances in each dimension, and correlations between dimensions.)

So how about the σ_q for each different spherical Gaussian component, where are they in the responsibility? A mixture Gaussian requires that we specify a variance σ_q^2 for each component q, and no such component-dependent term appeared in soft K-means. So in additional to assuming spherical Gaussian components, let's go a step further, and assume that there is a single constant $\sigma = \sigma_q$ for all spherical Gaussian components q.

Let's also go yet another step further, and assume that all components are equiprobable: $\pi_q = \frac{1}{Q}$.

Now you should be able to prove to yourself that after making these assumptions, the mixture Gaussian posterior probability equation reduces to:

$$P(q \mid X_i, \theta) = \frac{\exp\left(\frac{-d_{iq}^2}{2\sigma^2}\right)}{\sum_{q'} \exp\left(\frac{-d_{iq'}^2}{2\sigma^2}\right)}$$

As if you haven't already recognized the equation for the responsibilities r_{ik} in soft K-means, let's go ahead and define a stiffness $\beta = \frac{1}{2\sigma^2}$, and:

$$P(q \mid X_i, \theta) = \frac{\exp\left(-\beta d_{iq}^2\right)}{\sum_{q'} \exp\left(-\beta d_{iq'}^2\right)}$$

implicit assumptions of K-means

Having derived the responsibility update equation for soft K-means from first principles, we've laid bare its implicit assumptions:

- the data are drawn from a mixture Gaussian distribution;
- composed of spherical Gaussians;
- each component has an identical spherical Gaussian variance σ -- each component is the same "width"; and
- each component has an equal expected number of data points.

It should not be surprising that K-means may tend to fail when any of these assumptions is violated.

Standard descriptions of K-means may tell you it can fail when the individual data clusters are asymmetrically distributed, have different variances, or have different sizes in terms of number of assigned data points — but they will typically not tell you *why*.

fitting mixture models with expectation maximization

Soft K-means, as a probability model, is a simplified example of an important class of optimization algorithm called **expectation maximization (EM)**.

Expectation maximization is a common way of fitting mixture models, and indeed other models with hidden variables. You recall I already

mentioned that kallisto and other RNA-seq quantitation methods use EM to infer which transcript a read maps to, when the read is compatible with multiple reads. That's essentially a clustering problem too.

For example, here's how you'd use expectation maximization to fit a mixture Gaussian model to data. To simplify things, let's assume that there is a single known, fixed variance parameter σ, as in a soft K-means stiffness. But let's relax one of the other K-means assumptions, and allow different mixture coefficients π_q. Then, starting from an initial guess at the means μ_q:

- **Expectation step:** Calculate posterior probabilities $P(q \mid X_i, \mu, \sigma, \pi)$ for each data point X_i, given the current means (centroids) μ_q, fixed σ, and mixture coefficients π_q.
- **Maximization step:** Calculate new centroids μ_k and mixture coefficients π_q given the posterior probabilities $P(q \mid X_i, \mu, \sigma, \pi)$.

We can follow the total log likelihood $\log P(X \mid \theta)$ as we iterate, and watch it increase and asymptote. When we start from different starting conditions, a better solution has a better total log likelihood.

When we calculate the updated mixture coefficients π_q, that's just the expected fraction of data points assigned to component q:

$$\hat{\pi}_q = \frac{\sum_i P(q \mid \cdot)}{N}$$

If we had to fit more than just a mean parameter μ_q for each component, we would have some way of doing maximum likelihood parameter estimation given posterior-weighted observations.

For example, if we were also trying to estimate σ_q terms, we could calculate the expected variance for each component q, not just the expected mean.

the negative binomial distribution

Single-cell RNA-seq data consist of a small number of counts y_g for each gene (or transcript) g. Many analysis approaches model these observed counts directly, rather than trying to infer quantitation (in TPM).

If a single-cell RNA-seq experiment generated observed counts from a mean expression level like pulling balls out of urns, then the differences we see across replicates of the same condition would just due to finite count sampling error. We would expect the observed counts y_{gi} to be Poisson-distributed around a mean μ_g.

The Poisson distribution only has one parameter. The variance σ^2 of the Poisson is equal to its mean μ. This is a strong assumption about σ^2.

In real biological data the counts y_{gi} in replicates -- biological replicates, or even just technical replicates -- have more variance than the Poisson predicts: they are **overdispersed**. This shouldn't surprise you. If the samples are outbred organisms with different genotypes, for example, we get expression variation due to genotype variation. We also get variation from a host of other environmental and experimental factors that are changing when we do biological replicates.

Merely as a practical matter, we want a distribution form that allows us to control the variance separately from the mean, so we can model this overdispersion. Empirically, two different distributions have been seen to fit RNA-seq count data reasonably well: the **negative binomial** distribution and the **lognormal** distribution. The homework this week is based on the negative binomial distribution.

definition of the negative binomial

The negative binomial distribution appears in

various notations and guises. We're going to sneak up on the more perplexing-looking version that's most convenient for RNA-seq analysis, starting with a version that's most easy on the eyes and most straightforward for understanding how the NB arises.

negative binomial with Bernoulli successes/failures

Suppose we have some sort of random discrete event (like flipping heads with a coin or rolling a one on a die) where we can talk about a probability p of *success* and $1 - p$ of *failure*. (Each event is called a *Bernoulli trial*, in the jargon.) What's the probability of obtaining k failures before reaching exactly n successes in a sequence of Bernoulli trials (where n is pre-specified)?

We know the chain ends in a success: we stop when we reach the n'th success. The remaining $n - 1$ successes and k failures can occur in any order (any permutation). So:

$$\begin{aligned} P(k \mid n, p) &= p \cdot \binom{k + n - 1}{n - 1} p^{n-1} (1 - p)^k \\ &= \binom{k + n - 1}{n - 1} p^n (1 - p)^k \end{aligned}$$

That's one way that the NB's probability mass function is written. In Python, there's `scipy.stats.pmf(k,n,p)` call and a `scipy.stats.logpmf(k,n,p)` functions that calculate the PMF or log PMF.

negative binomial with mean and dispersion

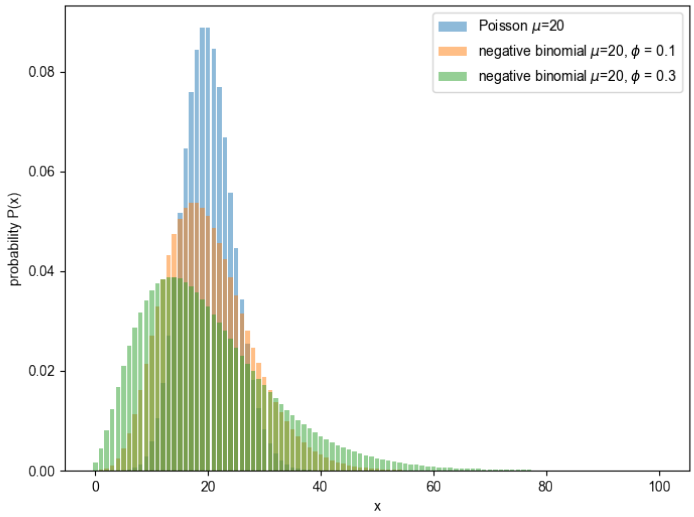
A most relevant form of the NB for us [Robinson & Smyth, 2008] is:

$$P(y \mid \mu, \phi) = \frac{\Gamma\left(y + \phi^{-1}\right)}{\Gamma(\phi^{-1})\,\Gamma(y + 1)} \left(\frac{1}{1 + \mu\phi}\right)^{\phi^{-1}} \left(\frac{\mu}{\phi^{-1} + \mu}\right)^{y}$$

Though this may look like gibberish, it's strongly related to the negative binomial distribution we wrote above. First remember (or know) that gamma functions $\Gamma(x)$ are the continuous extension of the familiar factorial, where $\Gamma(n) = (n-1)!$ for integers n, so that mess of gammas is just a binomial coefficient in continuous form. The terms inside parentheses can be written as $(1-p)$ and p, if we say $p = \frac{1}{1+\mu\phi}$, thus $(1-p) = \frac{\mu}{\phi^{-1}+\mu}$, so again familiar terms of our previous negative binomial distribution appear.

In this form, the mean of the NB is μ, and its variance is $\mu + \phi\mu^2$.

The parameter ϕ is called the **dispersion**. As $\phi \to 0$, the NB asymptotes to the Poisson.



Examples of the negative binomial distribution, compared to the Poisson. *('open image in new tab' to embiggen.)*

Once we assume the NB, then for each gene g, we need two parameters to calculate a likelihood of its counts in a sample: its mean number of counts μ_g, and its dispersion ϕ_g.

negative binomial distribution in SciPy

I mentioned that `scipy.stats` defines its negative binomial in terms of the first notation, not the

second. This creates a fiddly detail that can bite you in this week's homework. You have to be careful that you're plugging the right numbers into `scipy.stats.nbinom` functions.

You should be able to prove to yourself that when you go to calculate a probability for y observed counts (what `scipy` thinks of as the probability of k failures) you need to convert parameters for passing to `scipy.stats.nbinom` as:

$$n = \frac{1}{\phi}$$

$$p = \frac{1}{1 + \mu\phi}$$

and now when you call `scipy.stats.nbinom.logpmf(y,n,p)`, it will give you $\log P(y \mid \mu, \phi)$.

fitting the negative binomial distribution to data

Suppose I give you a bunch of samples x_i that have been sampled from a negative binomial distribution with unknown parameters μ, ϕ. How do I estimate optimal parameter values?

Maximum likelihood fitting of an NB turns out to be a bit fiddly. For almost all practical uses, an alternative parameter estimation strategy suffices, and it's a strategy worth knowing about in general: **method of moments** (MOM) estimation.

The idea is that since I know the relationship between the NB parameters and the mean and variance (in the limit of having lots of observed data):

$$\text{sample mean } \bar{x} \simeq \mu$$

$$\text{sample variance } s^2 \simeq \mu + \phi\mu^2$$

so why not calculate the mean and variance of the data, then just solve for the parameters:

$$\mu^{\text{MOM}} = \bar{x}$$

$$\phi^{\text{MOM}} = \frac{s^2 - \bar{x}}{\bar{x}^2}$$

We can often get away with MOM estimation. In the homework, you can. And in fact, you don't even need to estimate ϕ, because it's given to you. You only need to estimate the μ parameter of the NB -- and that's just the sample mean of the count data, thankfully.
