Section 08: Non-negative Matrix Factorization

Notes by Mary Richardson (2021) adapted from Wendy Valencia-Montoya (2020), June Shin (2018), William Mallard (2017)

- Non-negative matrix factorization
- Example problem
- Example notebook
- Practical details
- Additional resources

Non-negative matrix factorization

Non-negative matrix factorization **(NMF)** is a matrix decomposition technique that splits a matrix into two smaller non-negative matrices whose product is approximately equal to the original matrix.

Why? The idea is that there is some rule-driven process generating our data – a generative model. We have taken a number of noisy measurements of the system's output, and we would like to discover the underlying structure of our data.

There are several **dimensionality reduction techniques** that can be used to infer hidden structure. Some commonly used method for this are k-means clustering (your favorite week!), hierarchical clustering, PCA (coming next week!), and NMF. Global clustering methods like PCA are best when groups are disjoint. NMF is helpful when we are interested in pulling out interpretable components that can be combined to recreate the whole. This week, we'll use NMF.

How? We have a matix of observations V of size $n \times m$. We are decomposing it into two matrices W ($n \times r$) and H ($r \times m$). Since $W \times H$ only approximates V, this is actually **lossy compression**. The goal is to solve for W and H such that the matrix multiplication of those two components is appromiately equal to the data V:

$$V \approx WH$$

Example problem

We want to find the underlying structure of the population(s) of Andean beetles we are studying. Are they all from one population or from several distinct populations? What is the proportion of each population?

Climatic fluctuations in the mountains can result in up or downslope shifts of entire ecosystems. A clever way of inferring the history of these shifts is by studying genetic patterns of current species that are tightly associated with a particular kind of ecosystem such a rainforest, cloud forest, or alpine tundra along an altitudinal gradient. Field biologists from a beetle genetic lab set up to investigate ecosystem shifts of the cloud forest of Andean mountains. One hypothesis is that a species of cloud forest beetle forms a single population, consistent with long-term altitudinal stability of this ecosystem. The competing model is that the beetle species consists of several populations despite there are none obvious geographic barriers. The latter hypothesis is based on their belief that the different populations formed in isolation in the mountaintops of different peaks. If the cloud forest just recently shifted downslope, it may not have been enough time to erode the genetic differentiation between populations, although hybridization may be expected.

The beetle lab has genotyped 300 loci for 120 individuals and summarized the genetic variation by counting the number of sites with derived alleles per locus. Their data is a simple whitespace-delimited table, available here.

Basically, the goal is to find the number of populations (the hidden parameter) that resulted in the observed data (genotypes). Additionally, you want to find potential hybrid individuals in case there is more than one population as a signature of ongoing genetic exchange. You have taken MCB112 and distinctly remember you can approach this problem through non-negative matrix factorization since you aim to find hidden components (populations) that group your individuals, but also admixture coefficients to identify hybrid individuals (weights of the different components).

What we know: We have data from $m = 120$ beetles. For each beetle we have the observed allele counts from $n = 300$ genomic loci. We're counting the number of derived (mutated from the ancestor) alleles.

What we want: We want to use the information we have about allele counts to infer the admixture proportion each beetle population. How many different populations do we have and what are their proportions? We know we have two terms of interest: (1) the proportion of individuals in each population and (2) the allele frequencies in each population. How can we infer these?

What we'll do: We'll use NMF which will let us decompose the data into information about the proportion of individuals in each population *and* information about the allele frequencies in each population.

NMF overview

We have an **allele counts matrix** V. We will decompose this matrix into an **admixture proportions matrix** W and an **allele frequencies matrix** H and an . We'll also have a **counts**

vector C that we'll use to make sure we get back counts:

$$V_{i\mu} \approx C_\mu \sum_a W_{ia} H_{a\mu}$$

We can rewrite this in terms of matrix multiplication of W and H:

$$V_{i\mu} \approx C_\mu (WH)$$

The **allele counts matrix** V **(**$n \times m$**)** is n individuals by m loci. Each position in the matrix $V_{i\mu}$ contains the counts of derived alleles at locus μ in individual i.

The **counts vector** C **(**$1 \times m$**)** is m loci long. Each position in the vector C_μ contains the total derived allele counts for locus μ. The columns of V sum to make C:

$$\sum_i V_{i\mu} = C_\mu$$

The **admixture proportions matrix** W **(**$n \times r$**)** is n individuals by r populations. Each position in the matrix W_{ia} indicates whether individual i is in population a. The columns of W sum to one:

$$\sum_i W_{ia} = 1$$

The **allele frequencies matrix** H **(**$r \times m$**)** is r populations by m loci. Each position in the matrix $H_{a\mu}$ contains the frequency of derived alleles at locus μ in population a. The columns of H sum to one:

$$\sum_a H_{a\mu} = 1$$

Generative model

As usual, we want to generate synthetic data so we can validate NMF gets the right answer. To do this, we need to generate known W and H

matrices, and use them to generate an **expected counts matrix** λ equal to the expected counts $< V_{i\mu} >$. Finally, we'll add Poisson noise to λ to generate an **observed counts matrix** V (because real biological data is noisy!)

$$\lambda_{i\mu} =< V_{i\mu} >= C_{\mu} \sum_{a} W_{ia} H_{a\mu}$$

Let's convert this to matrix space because it will make the impementation much easier.

$$\lambda_{i\mu} =< V_{i\mu} >= C_{\mu}(WH)$$

In steps:

1. Initialize W. For W, each row represents a individual, and each column represents a population. Each row should have a non-zero entry in only one column, unless the individual is a hybrid between two populations. We'll assign each individual to a population, then choose hybrid individuals. Each column should sum to one because we are creating a matrix of weights, so we can perform column normalization. See python tips.

2. Initialize H. For H, each row represents a population, and each column represents a locus. This can be random, but each column should sum to one. We'll generate a ones matrix, and apply `np.random.dirichlet()` along each column.

3. Calculate λ. For λ, we'll find the dot product (matrix multiplication) of W and H using either `np.dot()` or `@`.

4. Calculate V. For V, we'll finally add Poisson noise to λ with `np.random.poisson()`.

We'll feed this V into our NMF function, and try to recover the true W and H.

Objective function

Our objective function for NMF is log likelihood. Since we are assuming Poisson noise, we can calculate the likelihood of our data V by calculating the probability that each observed count $V_{i\mu}$ was generated by a Poisson distribution around our expected count $\lambda_{i\mu}$

$$P(V_{i\mu}|W,H) = \frac{\lambda_{i\mu}^{V_{i\mu}} e^{-\lambda_{i\mu}}}{V_{i\mu}!}$$

To calculate the probability of *all* of the counts in V rather than just the count at $V_{i\mu}$, we need to find the joint probability over all positions in the matrix (over all i and all μ). As usual, we'll do this by taking the product (assuming independence).

$$P(V|W,H) = \prod_{\mu} \prod_{i} P(V_{i\mu}|W,H)$$

But wait – we're multiplying lots of probabilities together now and we run the risk of underflow! Time to convert to log space using our familiar log rules.

$$logP(V|W,H) = \sum_{\mu} \sum_{i} logP(V_{i\mu}|W,H)$$

Let's plug our equation for $P(V_{i\mu}|W,H)$ back in and calculate the log.

$$logP(V|W,H) = \sum_{\mu} \sum_{i} log\frac{\lambda_{i\mu}^{V_{i\mu}} e^{-\lambda_{i\mu}}}{V_{i\mu}!}$$

$$logP(V|W,H) = \sum_{\mu} \sum_{i} V_{i\mu}\lambda_{i\mu} - \lambda_{i\mu} - log(V_{i\mu}!)$$

But log(V_{i\mu}!) is a constant, so we can drop this term, since it won't affect our comparisons (it will never change, so our difference in log likelihood will always cancel this term out!)

$$logP(V|W,H) = \sum_{\mu} \sum_{i} V_{i\mu}\lambda_{i\mu} - \lambda_{i\mu}$$

In matrix space, getting the term inside the sum is simple. We just want to multiply two matrices that are the same size, then subtract another matrix that is the same size. This is simply position-wise multiplication and subtraction.

$$Vlog\lambda - \lambda$$

In steps:

1. Calculate the matrix inside the sum $Vlog\lambda - \lambda$
2. Sum over the whole calculated matrix (over all i and all μ) using `np.sum()`.

Optimization

We're ready to optimize! We're using **gradient ascent** here, which is hopefully a familiar idea after having done gradient descent in previous weeks. Rather than using `scipy.optimize.minimize()`, we're going to do this one ourselves. To perform gradient ascent, we need to be able to find the direction of greatest increase in log likelihood at each point. To get this map, we will compute the **gradient** (∇) of the log likelihood.

$$\nabla logP(V|W,H) = \nabla \sum_{\mu} \sum_{i} V_{i\mu}\lambda_{i\mu} - \lambda_{i\mu}$$

For this gradient, we'll have two terms: the change in W_{ia} and the change in $H_{a\mu}$.

$$\nabla = \frac{\partial}{\partial W_{ia}}\hat{\imath} + \frac{\partial}{\partial H_{a\mu}}\hat{\jmath}$$

Let's solve for each one using differentiation rules.

$$\frac{\partial}{\partial W_{ia}} = \sum_{\mu} (\frac{V_{i\mu}}{\lambda_{i\mu}} C_{\mu} H_{a\mu}) - \sum_{\mu} C_{\mu} H_{a\mu}$$

$$\frac{\partial}{\partial H_{a\mu}} = \sum_{i} (\frac{V_{i\mu}}{\lambda_{i\mu}} C_{\mu} H_{a\mu}) - \sum_{i} C_{\mu} H_{a\mu}$$

Now we're ready to get our updated W'_{ia} and $H'_{a\mu}$, which is the current value plus a step size Δ. We'll choose a convenient step size Δ for each direction.

$$\Delta_{ia} = \frac{W_{ia}}{\sum_{\mu} C_{\mu} H_{a\mu}}$$

$$\Delta_{a\mu} = \frac{H_{a\mu}}{\sum_{\mu} C_{\mu} W_{ai}}$$

Let's get our updated W'_{ia} and $H'_{a\mu}$.

$$W'_{ia} = W_{ia} + \Delta_{ia} \frac{\partial}{\partial W_{ia}} = W_{ia} \frac{\sum_{\mu} (\frac{V_{i\mu}}{\lambda_{i\mu}} C_{\mu} H_{a\mu})}{\sum_{\mu} C_{\mu} H_{a\mu}}$$

$$H'_{a\mu} = H_{a\mu} + \Delta_{a\mu} \frac{\partial}{\partial H_{a\mu}} = H_{a\mu} \frac{\sum_{i} (\frac{V_{i\mu}}{\lambda_{i\mu}} C_{\mu} W_{ia})}{\sum_{\mu} C_{\mu} W_{ai}}$$

We can simplify our update equations. We can ignore the denominator in W'_{ia} and then renormalize W''_{ia}. We can also cancel out C_{μ} in $H'_{a\mu}$ and use the fact that $\sum_i W_{ia} = 1$ to remove the denominator in $H'_{a\mu}$. The renormalization of W''_{ia} ensures that $\sum_i W_{ia} = 1$ and consequently $\sum_a H_{a\mu} = 1$

$$W'_{ia} = W_{ia} \sum_{\mu} (\frac{V_{i\mu}}{\lambda_{i\mu}} C_{\mu} H_{a\mu})$$

$$W''_{ia} = \frac{W'_{ia}}{\sum_j W'_{ja}}$$

$$H'_{a\mu} = H_{a\mu} \sum_{i} (\frac{V_{i\mu}}{\lambda_{i\mu}} W_{ia})$$

These NMF update equations contain three different subscripts, so the easiest solution is to use three nested for loops for the first W' update step, and another three for the H' update step.

We can also implement these update steps with matrix algebra, which runs much faster. Look carefully at the dimensions of your matrices when switching to matrix space.

In steps:

1. Calculate updated W''. Using the old W and H, we'll calculate the updated W' then renormalize according to the W'' equation.

2. Calculate updated H'. Using the old W and H, we'll calculate the updated H'.

Putting it all together

Finally, we're ready to put it all together to perform the optimization. The NMF function should take two arguments: the observed data V and the number of populations we think we have r. Our goal is to return the underlying **admixture proportions matrix** W and **allele frequencies matrix** H so we can see what's really going on in the beetles!

Every iteration of our optimization, we'll update W and H by calculating our new W''_{ia} and $H'_{a\mu}$. After each update step, we want to calculate the log likelihood that V was generated by our current guess at W and H. We'll continually run these update steps until the log likelihood converges.

In steps:

1. Initialize W and H to random guesses. We will make sure the columns of W sum to one and the columns of H sum to one. For this, `np.dirichlet()` is very useful.

2. Calculate λ. First we will calculate C by summing the columns of V. Then we can use the formula for λ.

3. Calculate the log likelihood. For this we will use the formula for log likelihood we derived.

4. Update W and H. For this we will use the update equations for W'_{ia}, W''_{ia}, and $H'_{a\mu}$ we derived.

5. Iterate until convergence. Repeat steps 2 through 4 until the log likelihood converges!

This should recover W and H for a given number of populations r, but we're not *sure* how many different beetle populations we have. We might have two or we might have 5, so we need to run NMF for each differen r value and compare th resulting log likelihoods. The best log likelihood solution will tell us how many populations we likely have.

In the end, the rows of W will tell us which population each beetle belongs to, and how many hybrid mice we have. (Hint: we might need to row-normalize W to see this and threshold, since the values won't be binary.) And the columns of H will tell us which alleles are common in each population.

Example notebook

Check out the example jupyter notebook for the problem here: [download] [view].

Practical details

Log rules

Basic rules

When multiplying probabilities together, remember that in log space multiplication becomes log addition.

$$log(ab) = log(a) + log(b)$$

When calculating the log of an equation with division, such as the formula for Poisson probability, remember that in log space division becomes log subtraction.

$$log(\frac{a}{b}) = log(a) - log(b)$$

When calculating the log of an equation with an exponent, such as the formula for Poisson probability, remember that in log space exponentiation becomes multiplication.

$$log(a * b) = blog(a)$$

Differentiation

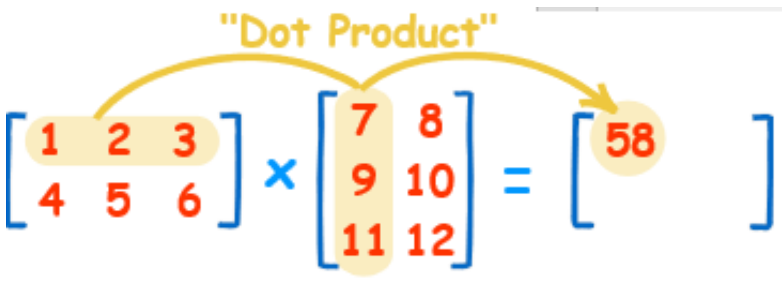
And when calculating the derivative of a log function $f(x) = logg(x)$, remember the differentiation rules.

$$f`(x) = \frac{g'(x)}{g(x)}$$

Matrix operations

Matrix multiplication

There are two ways to perform matrix multiplication (or calculate the dot product) in python: using the `@` operator or `np.dot()`.



The dedicated single-character infix operator `@` was added to Python3 to eliminate ambiguity about what `*` means in the context of two matrices.

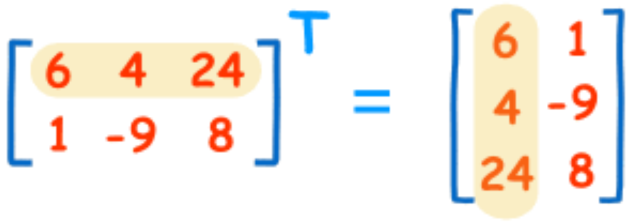
The `@` operator performs on two matrices of compatible dimensions (with the same inner dimension). eg, if `A` is an N×R matrix and `B` is an R×M matrix, then `A @ B` will give an N×M matrix.

The `*` operator performs on two matrices of the same dimensions. eg, if `A` is an N×M matrix and `B` is an N×M matrix, then `A * B` will give an N×M matrix.

```
A = np.random((N, R)) # dimensions are NxR
B = np.random((R, M)) # dimensions are RxM
A @ B # dimensions are N x M
```

Transpose

There are two ways to get the transpose of a matrix in python: `.T` and `np.transpose()`.



The transpose operation converts all columns to rows and all rows to columns. eg, if `A` is an N×R matrix, then `A.T` will give an R×N matrix.

```
A = np.zeros((N,R)) # dimensions are NxR
A.T # dimensions are RxN
```

Python tips

General tips

```
import numpy as np
import matplotlib.pyplot as plt
```

The method `np.random.dirichlet()` may be useful when initializing W and H. It generates a list of n random numbers that sum to 1. Alpha is the typical Dirichlet parameter vector; lower values of alpha generate a sparser vector. This might come in handy when generating your W matrix for your homework because a single gene modeule is likely to have a sparse relative expression vector.

```
N, M, R = 100, 50, 5 # N genes, M experiments, R modu
np.random.dirichlet(np.ones(N), size=R).T
```

The method `np.random.poisson()` will add Poisson noise to a matrix, which will come in handy for generating V.

```
N, M = 10, 5
L = 1e6 * np.random.random((N, M))
np.random.poisson(L)
```

To create an N×M matrix of zeros, use `np.zeros()`. There is also `np.ones()`, and we are already familiar with `np.random.random()`.

```
N, M = 10, 5

np.zeros((N, M))
np.ones((N, M))
np.random.random((N, M))
```

Let's say you have a function that only operates on 1D arrays -- eg, `np.random.dirichlet()`. And let's say you're trying to apply this function to each column of a matrix of ones. You can achieve this with `np.apply_along_axis()`. It takes three arguments: the function to apply, the axis to apply it along, and the matrix to apply it to. As usual, axis=0 is columns, axis=1 is rows.

```
N, R = 10, 5
W = np.ones((N, R))

np.apply_along_axis(np.random.dirichlet, 0, W)
```

You can sum all of a matrix's values with the `.sum()` method. If you want to sum the values in all columns or rows, you can use the `.sum()` method's `axis` argument.

```
N, M = 5, 3
X = np.ones((N, M))

X.sum()        # --> 30.0
X.sum(axis=0)  # --> array([ 5.,   5.,   5.])
X.sum(axis=1)  # --> array([ 3.,   3.,   3.,   3.,   3.])
```

To normalize the columns of a matrix, we can just divide the matrix by the sum of the columns.

```
N, M = 5, 3
```

```
X = np.random.random((N, M))

X /= X.sum(axis=0)
```

It's actually a little surprising that this works, given that we're asking to divide a 5×3 matrix by a 3-element vector. NumPy somehow figures out that you actually want it to perform element-wise division between the 1×3 matrix and each row of the 5×3 matrix. But it cannot figure out what you want it to do for a 5×3 matrix and a 5-element vector... at least not without a little help. To help NumPy figure out what you want, we need to use numpy broadcasting.

Other useful functions to be familiar with:

```
np.multiply()
np.divide()
np.power()
```

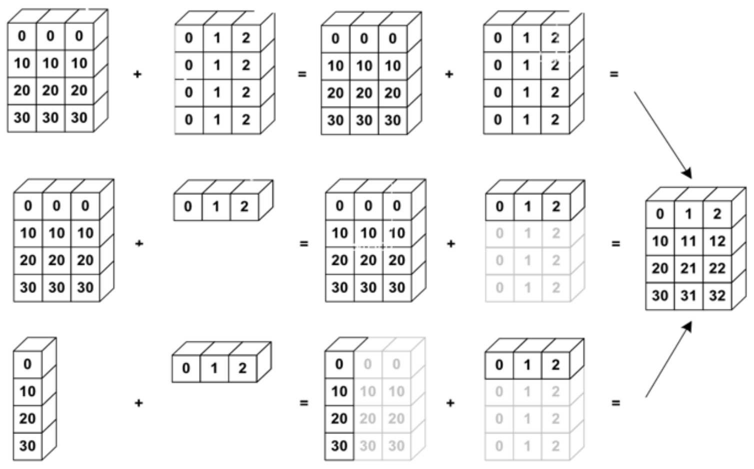
Numpy broadcasting

If you have a matrix `A` with dimensions N×M, and you want to add an M-element array `b` to all of `A`'s rows, you just need to add another axis to `b`, keeping the two same-sized axes aligned. In this example, `A.shape` is (N, M) and `b.shape` is (M,).

To add an axis to `b`, NumPy uses the notation: `B = b[np.newaxis,:]` Since we put the new axis first, `B.shape` will be (1, M). If we had instead done `b[:,np.newaxis]`, then `B.shape` would be (M, 1). But we are trying to make `B` compatible with `A`, so we made `B`'s second dimension agree with `A`'s second dimension (ie, both are of size M).

`np.newaxis` is synonymous with `None`, so you could also do `b[None,:]`.

Now that `A` and `B` are both 2D matrices, you can just say: `A + B`, and Numpy will replicate the first (and only) row of `B` N times before performing standard element-wise addition on two N×M matrices. This is best described visually:



And here it is in python:

```
A = np.array([0, 10, 20, 30])
B = np.array([0, 1, 2])

print('A:', A, 'shape:', A.shape)
print('B:', B, 'shape:', B.shape)
```

```
A: [ 0 10 20 30] shape: (4,)
B: [0 1 2] shape: (3,)
```

```
# we want to make A.shape to (4, 1)
new_A = A[:, np.newaxis]
print("new A:\n", new_A, 'shape:', new_A.shape)
```

```
new A:
 [[ 0]
 [10]
 [20]
 [30]] shape: (4, 1)
```

```
# now we can add A and B
print("A+B:\n", new_A+B)
```

```
A+B:
 [[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

Note: Broadcasting is not strictly necessary for this homework. Anything you can do with numpy broadcasting, you can also do by brute force with a `for` loop; it will just run much slower (~10-100x for this homework).

Additional resources

1. Lee DD, Seung HS. (1999) **Learning the parts of objects by non-negative matrix factorization**. *Nature* 401(6755): 788–791. doi:10.1038/44565 This is the original NMF paper described by Sean in lecture this week.

2. Frichot E, Mathieu F, Trouillon T, Bouchard G, François O. (2014) **Fast and efficient estimation of individual ancestry coefficients**. *Genetics* 196: 973–983. doi:10.1534/genetics.113.160572 We went over an example use of NMF to infer population structure, which is described in more depth here.

3. Devarajan, K. (2008) **Nonnegative matrix factorization: an analytical and interpretive tool in computational biology**. *PLoS Comput Biol* 4(7): e1000029. doi:10.1371/journal.pcbi.1000029 NMF can be applied to lots of other biological problems. This is a review of some of those applications.