Quick Introduction to Nonnegative Matrix Factorization

Norm Matloff
University of California at Davis

1 The Goal

Given an $u \times v$ matrix $A$ with nonnegative elements, we wish to find nonnegative, rank-$k$ matrices $W$ ($u \times k$) and $H$ ($k \times v$) such that

$$A \approx WH$$  \hspace{1cm} (1)

We typically hope that a good approximation can be achieved with

$$k \ll \text{rank}(A)$$ \hspace{1cm} (2)

The benefits range from compression ($W$ and $H$ are much smaller than $A$) to avoidance of “overfitting” in a prediction context.

2 Notation

We’ll use the following notation for a matrix $Q$

- $Q_{ij}$: element in row $i$, column $j$
- $Q_i$: row $i$
- $Q_j$: column $j$

Note the key relation

$$(WH)_{j} = \sum_{i=1}^{v} H_{ij} W_{i}$$ \hspace{1cm} (3)$$
In other words, in (1), we have that:

- Column $j$ of $A$ is approximately a linear combination of the columns of $W$, with the coefficients in that linear combination being the elements of column $j$ of $H$.

- Thus the columns $W_1, \ldots, W_k$ then (approximately) form a basis for the vector space spanned by the columns of $A$. Since the relation is only approximate, let’s use the term pseudo-basis.

- If (1) is a good approximation, then most of the information carried by $A$ is contained in $W$.

3 Applications

This notion of nonnegative matrix factorization has become widely used in a variety of applications, such as:

- **Image recognition:**
  Say we have $n$ image files, each of which has brightness data for $r$ rows and $c$ columns of pixels. We also know the class, i.e. the subject, of each image, say car, building, person and so on. From this data, we wish to predict the classes of new images. Denote the class of image $j$ by $C_j$.
  
  We form a matrix $A$ with $rc$ rows and $n$ columns, where the $i$th column, $A_{i,:}$, stores the data for the $i$th image, say in column-major order.
  
  If we can successfully find matrices $W$ and $H$ as above, we can use this for prediction, say using a 1-nearest neighbor approach, as follows: Given the $rc$-element vector $q$ of the new image, we find its representation $q_p$ in the pseudo-basis, and then find $g$ such $H_g$ best matches $q_p$. Our predicted class is then $C_g$.

- **Text classification:**
  Here $A$ consists of, say, word counts. We have a list of $d$ key words, and $m$ documents of known classes (politics, finance, sports etc.). $A_{ij}$ is the count of the number of times word $i$ appears in document $j$.
  
  Otherwise, the situation is the same as for image recognition above. We find the NMF, and then given a new document to classify, with word counts $q$, we find its coordinates $q_p$, and then predict class by matching to the columns of $H$.

\footnote{Or, at least for the nonnegative orthant in that space.}
Recommender systems:

Here most of $A$ is unknown. For instance, we have users, who have rated movies they’ve seen, with $A_{ij}$ being the rating of movie $j$ by user $i$. Our goal is to fill in estimates of the unknown entries.

It is customary to initially insert 0s for those entries, then perform NMF, producing $W$ and $H$. We then compute $WH$ as our estimate of $A$, and now have estimates for the missing entries.

4 The R Package NMF

The R package NMF is quite extensive, with many, many options. In its simplest form, though, it is quite easy to use. For a matrix $a$ and desired rank $k$, we simply run

```r
> nout <- nmf(a, k)
```

The factors are then in `nout@fit@W` and `nout@fit@H`.

Let’s illustrate it in an image context, using the following:

Here we have only one image, and we’ll use NMF to compress it, not do classification. First obtain $A$:

```r
> library(pixmap)
> mtr <- read.pnm('MtRush.pgm')
> a <- mtr@grey
```

Now, perform NMF, find the approximation to $A$, and display it:
> aout <- nmf(a,50)
> w <- aout@fit@W
> h <- aout@fit@H
> approxa <- w %% h

# brightness values must be in [0,1]
> approxa <- pmin(approxa,1)
> mtrnew <- mtr
> mtrnew@grey <- approxa
> plot(mtrnew)

Here is the result:

![Image of approximated matrix]

This is understandably blurry. The original matrix has dimension 194 × 259, and thus presumably has rank 194.\(^2\) We’ve approximated the matrix by one of rank only 50, with a 75% storage savings. Not important for one small picture, but possibly worthwhile if we have many. The approximation is not bad in that light, and may be good enough for image recognition or other applications.

Indeed, in many if not most applications of NMF, we need to worry about overfitting. This is beyond the scope of this document, but the point is that overfitting in this context amounts to using too high a value for our rank, something to be avoided.

\(^2\)This is confirmed by running the function `rankMatrix()` in the *Matrix* package.
5 Algorithms

How are the NMF solutions found? What is \texttt{nmf()} doing internally?

Needless to say, the methods are all iterative, but here we will focus on the Alternating Least Squares algorithm. By the way, this is not the default for \texttt{nmf()}; to select it, set \texttt{method = 'snmf/r'}.

We also need a criterion for goodness of approximation. Here we will take that to be the Frobenius norm, which is just the Euclidean norm:

\[
\|Q\| = \sqrt{\sum_{i,j} Q_{ij}^2}
\]  

(4)

So our criterion for error of approximation will be

\[
\|A - WH\|
\]  

(5)

This measure is specified in \texttt{nmf()} by setting \texttt{objective = 'euclidean'}.

So, how does it work? It’s actually quite simple. Suppose just for a moment that we know the exact value of $W$, with $H$ unknown. Then for each $j$ we could minimize

\[
\|A_j - WH_j\|
\]  

(6)

We are seeking to find $H_j$ that minimizes (6), with $A_j$ and $W$ known. But since the Frobenius norm is just a sum of squares, that minimization is just a least-squares problem, i.e. linear regression. The solution is well-known to be

\[
H_j = (W'W)^{-1}W'A_j
\]  

(7)

R’s \texttt{lm()} function does this for us

\[
> h[, j] \leftarrow \text{lm}(a[, j] \sim w - 1)
\]

for each $j$. (The -1 specifies that we do not want a constant term in the model.) On the other hand, suppose we know $H$ but not $W$. We could take transposes,

\[
A' = H'W'
\]  

(8)

and then just interchange the roles of $W$ and $H$ above. Here a call to \texttt{lm()} gives us a row of $W$, and we do this for all rows.
Putting all this together, we first choose initial guess for $W$ and $H$; \texttt{nmf()} gives us various choices as to how to do this. Then we alternate: Compute the new guess for $W$ assuming $H$ is correct, then choose the new guess for $H$ based on that new $W$, and so on.

During the above process, we may generate some negative values. If so, we simply truncate to 0.

6 Predicting New Cases

Once we have settled on $W$ and $H$, what can we do with them? In the recommender system application mentioned earlier, we simply multiply them. But recall the description given above for the image and text processing examples:

Given the $rc$-element vector $q$ of the new image, we find its representation $q_p$ in the pseudo-basis, and then find $g$ such $H_g$ best matches $q_p$. Our predicted class is then $C_g$.

How do we find $q_p$? The answer is that again we can use the least-squares above.

\begin{verbatim}
> qp <- lm(q ~ w)
\end{verbatim}

7 Convergence and Uniqueness Issues

There are no panaceas for applications considered here. Every solution has potential problems.

With NMF, an issue may be uniqueness — there might not be a unique pair $(W,H)$ that minimizes (5).\footnote{See Donoho and Stodden, When Does Non-Negative Matrix Factorization Give a Correct Decomposition into Parts?, https://web.stanford.edu/~vcs/papers/NMFCDP.pdf.} In turn, this may result in convergence problems. The NMF documentation recommends running \texttt{nmf()} multiple times; it will use a different random seed each time.

The Alternating Least Squares method used here is considered by some to have better convergence properties, since the solution at each iteration is unique.

8 How Do We Choose the Rank?

This is not an easy question. One approach is to first use SVD, and see how many eigenvalues are substantial. If all but 5 of the eigenvalues are tiny, for instance, then we could take our rank to be 5.
9 Why Nonnegative?

In the applications we’ve mentioned here, we always have $A \geq 0$. However, that doesn’t necessarily mean that we need $W$ and $H$ to be nonnegative, since we could always truncate.

Indeed, we could consider using Singular Value Decomposition instead. This too is beyond the scope of this tutorial, but the point is that SVD basically involves finding eigenvectors, say of $A^TA$, and taking the ones corresponding to large eigenvalues as our pseudo-basis.

There are a couple of reasons NMF may be preferable. First, truncation may be difficult if we have a lot of negative values. But the second reason is rather philosophical, as follows:

In the image recognition case, there is hope that the vectors $W_j$ will be sparse, i.e. mostly 0s. Then we might have, say, the nonzero elements of $W_1$ correspond to eyes, $W_2$ correspond to nose and so on with other parts of the face. We are then “summing” to form a complete face. This may enable effective parts-based recognition.

Sparsity (and uniqueness) might be achieved by using regularization methods, in which we minimize something like

$$||A - WH|| + \lambda(||W||_1 + ||H||_1)$$

(9)

where the subscript 1 (“L1 norm”) means the norm involves sums of absolute values rather than sums of squares. This guards against one of the factors becoming “too large,” and it turns out that this also can lead to sparse solutions. We can try this under $\text{nmf()}$ by setting $\text{method = 'pe-nmf'}$. 
