John Myles White

"Who refuses to do arithmetic is doomed to talk nonsense."

Image Compression with the SVD in R

By John Myles White on 12.17.2009

Update (2/26/2011): Thanks to Dan, I discovered that the first use of `i` in the code below was unclear. This is now fixed.

Over the next few posts, I'm going to be reviewing the use of R to implement the most commonly used matrix techniques for image manipulation. The code will be surprisingly simple to understand, because the real magic behind these techniques lies in the mathematics that R provides an abstract interface to. To start, I'm going to review the singular value decomposition of a matrix, also known as the SVD.

If you'd like to understand the magic behind the scenes here, you can read about the mathematical definition of the SVD on Wikipedia or you can watch Gilbert Strang's lectures. If your command of linear algebra isn't great, I suspect that the mathematical details of the SVD will be totally opaque to you.

Fortunately, the details of the underlying mathematics aren't at all necessary for appreciating the SVD's usefulness for manipulating images: it is arguably the best way to compress matrices into smaller representations that contain the essential information from the original matrices. The easiest way to understand this is to see it in action, so I'm going to show how the SVD allows for any degree of compression of an image represented as a real-valued matrix.

Obviously, the first thing we have to do is to represent our example image as a matrix with real-valued entries. To do this, we'll need to use a simple technique for loading an image into R that I found on the Quantitative Ecology blog. The first thing to do is use ImageMagick to transform a TIFF file into a PPM file:

```
#!/bin/bash

convert face.tiff image.ppm
```

Then we can use the "pixmap" package from CRAN to read the PPM image into R as an image object containing three matrices: one for the red components, one for the green components, and one for the blue components. The exact code I used was this:

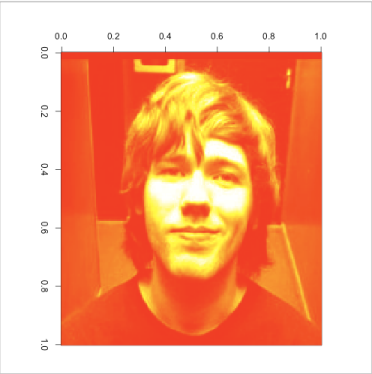
```
library('pixmap')

image <- read.pnm('image.ppm')

red.matrix <- matrix(image@red, nrow = image@size[1], ncol = image@size[2])
green.matrix <- matrix(image@green, nrow = image@size[1], ncol = image@size[2])
blue.matrix <- matrix(image@blue, nrow = image@size[1], ncol = image@size[2])
```

Before we go any further, we should look at the image we'll be playing with. Rather than work with the composite of the three color matrices, I found it easier to work with only a single color. For this specific image, I found that the green matrix looked best, so I'll use only that. At the risk of seeming self-indulgent, I'm using my own Gravatar icon as an example image because faces are particularly easy to recognize under the strange color scheme I'll be using, which I chose because it really simplifies the visualization code.

To visualize the matrix as an image, I discovered two very helpful graphical functions, `image` and `heat.colors`. `image` depicts the value of a matrix using a color scheme you specify; `heat.colors` produces a set of colors that could be used in a heat map with a granularity you specify as the first argument. Using these functions, we can see our raw input matrix:

```
image(green.matrix, col = heat.colors(255))
```

The images generated by `image` are at an unfortunate angle for viewing, so I edited them post hoc with a single 90° clockwise rotation in Preview to give this sort of image:



If you want to play with the original PPM image file, you can get it here.

Clearly, we've got a recognizable face here. Feeling good about that, we can start to play with the SVD of this matrix. The R function to generate the SVD of a matrix is simply `svd`:

```
1  green.matrix.svd <- svd(green.matrix)
```

`svd` returns a list with three entries: `d`, `u` and `v`. `d` is a vector of the singular values ordered by decreasing size; `u` and `v` are matrices that are combined with a diagonal matrix form of `d` to give the original input matrix. To simplify things, I'll set up variables to contain the value of these elements of the SVD output list:

```
1  d <- green.matrix.svd$d
2  u <- green.matrix.svd$u
3  v <- green.matrix.svd$v
```

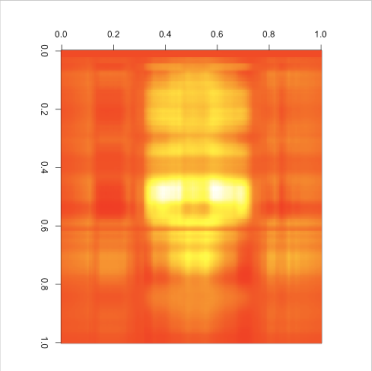
We can check that the SVD works by multiplying the relevant matrices:

```
1  green.matrix.reconstruction <- u %*% diag(d) %*% t(v)
2
3  mean(green.matrix - green.matrix.reconstruction)
4  # [1] -5.184204e-16
```

If this is the SVD by itself, it's not clear that it's useful computationally: we've just rewritten one matrix as the product of three matrices and accumulated a small bit of error along the way. The real utility of the SVD lies in the singular values: they represent, in decreasing order, the most important information about the original matrix.

To see this, you can shrink the input matrices and produce a compressed form of the matrix. For instance, my original matrix is a 212 by 201 image. Using the SVD, we can represent it as the product of a 212 by 2 matrix, a 2 by 2 matrix, and a 2 by 201 matrix. This massively reduces the amount of information we're storing, but you can already see the outline of our input image in the results:

```
1  i <- 2
2  green.matrix.compressed <- u[,1:i] %*% diag(d[1:i]) %*% t(v[,1:i])
3
4  image(green.matrix.compressed, col = heat.colors(255))
```

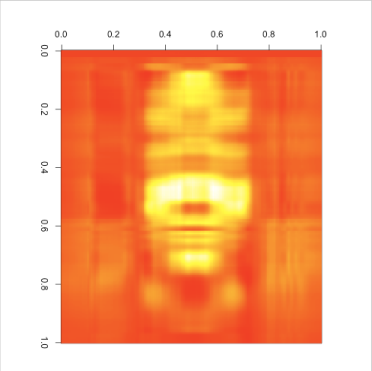


To get a feel for how this works, we can iterate over the number of singular values we include in our compressed form:

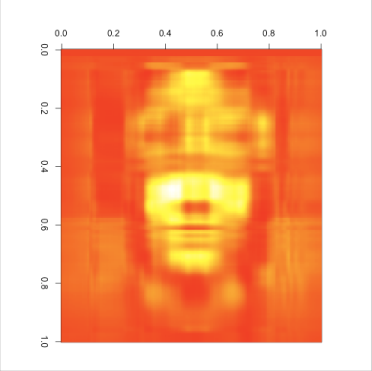
```
for (i in c(3, 4, 5, 10, 20, 30))
{
        green.matrix.compressed <- u[,1:i] %*% diag(d[1:i]) %*% t(v[,1:i])
        png(paste('images/Green Image SVD Compressed ', i, '.png', sep = ''))
        image(green.matrix.compressed, col = heat.colors(255))
        dev.off()
}
```

This produces the following images:

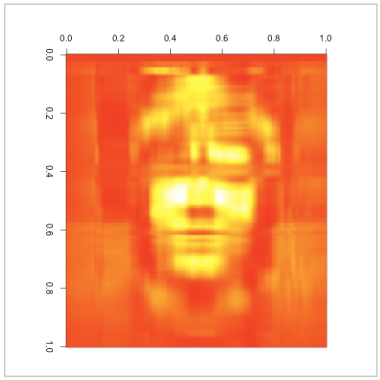
3 Singular Values



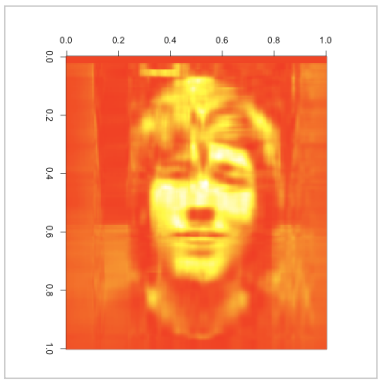
4 Singular Values



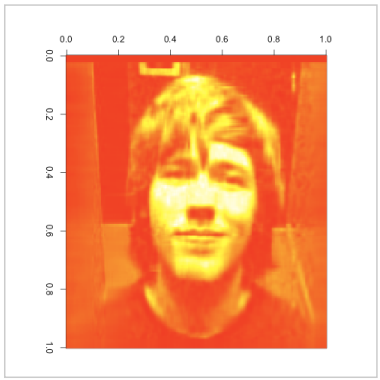
5 Singular Values



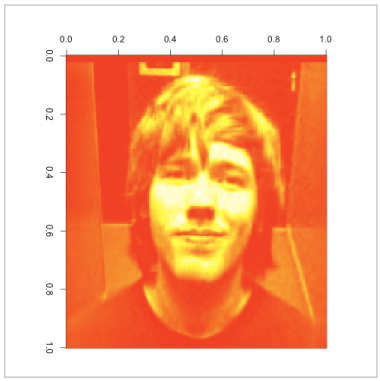
10 Singular Values



20 Singular Values



30 Singular Values



As you can see, the images get better quite quickly. Indeed, this last image looks great, despite the large reduction in the number of entries we're keeping from the original matrix. I'd say that's a pretty compelling case for the value of the SVD. It would be easy to combine the compressed matrices for red, green and blue to get a compressed version of our input image using its original color scheme. I'll leave that as an exercise for anyone interested in playing with this technique on their own.

Of course, the SVD has tons of other uses, but this simple hack for image compression struck me as pretty interesting, as well as being remarkably simple to implement in R.

Posted in Statistics | 4 Responses

4 responses to "Image Compression with the SVD in R"

Michael Sumner 12.20.2009 at 7:20 pm | *Permalink*

Hi, thanks this is great. I modified your example to use rgdal to read the file, which gives you more options for formats.

```
library(rgdal)

## this could read the TIFF directly
imdata <- readGDAL("image.ppm")

## as.matrix converts from sp's data.frame form to matrix
red.matrix <- as.matrix(imdata[1])
green.matrix <- as.matrix(imdata[2])
blue.matrix <- as.matrix(imdata[3])

## we need to flip vertically (could transpose with t() if necessary)
green.matrix <- green.matrix[,ncol(green.matrix):1]
green.matrix.svd <- svd(green.matrix)

d <- green.matrix.svd$d
u <- green.matrix.svd$u
v <- green.matrix.svd$v

op <- par(mfrow = c(3,2))
for (i in c(3, 4, 5, 10, 20, 30))
{
green.matrix.compressed <- u[,1:i] %*% diag(d[1:i]) %*% t(v[,1:i])

image(green.matrix.compressed, col = heat.colors(255), main = i)

}
```

Dan Brickley 2.26.2011 at 5:05 pm | *Permalink*

What's the 'i' in green.matrix.compressed <- u[,1:i] %*% diag(d[1:i]) %*% t(v[,1:i]) ?

I get "Error: object 'i' not found"...

Dan Brickley 2.26.2011 at 5:13 pm | *Permalink*

Ah got it, the line with the 'i' only really makes sense in the loop (although any of the given values can be substituted...

Great writeup btw, really nice to have this running locally rather than just read about it in papers :)

John Myles White 2.26.2011 at 9:26 pm | *Permalink*

Yeah, that's the right interpretation of 'i'. Sorry for the confusion. I've just edited the example code to keep future visitors from getting confused.

« Previous Next »