Math for Data Science: Lab 9

Prof. Asya Magazinnik

2023-22-11

Principal Components Analysis (PCA): An Application to Image Compression

Today we will explore the dimension reduction technique of **Principal Components Analysis** (PCA), which is a way of efficiently summarizing and describing large datasets. PCA has a large variety of applications in genomics, facial recognition and computer vision, finance and economics, climate science, and social science, but today we will be using it to represent images of my cat.

Figure 1: My cat, Laszlo.

You have already learned everything you need to know to understand this powerful tool: maximization, variance, and eigendecomposition. Let's put these pieces together to see how it works.

1. PCA as a Variance-Maximization Problem

Suppose we have an n by p data matrix called $\mathbf{X}$ (n rows, p columns). For convenience, we have centered each of the columns of $\mathbf{X}$ around the column mean (i.e., subtracted off the column mean), so that all column means of $\mathbf{X}$ are now 0.

Our challenge is to convey as much information about $\mathbf{X}$ as we can in only one column of data (that is, with only one n-length vector). What vector would you choose?

Your first impulse might be to choose a random column from $\mathbf{X}$. It isn't much, but it preserves intact one column of $\mathbf{X}$ — and with only one column available to us, can we do any better?

The answer is yes. Suppose $\mathbf{X}$ contained one column of data that was pretty similar across observations (e.g., everyone's a Hertie student), and another that was quite different (e.g., people come to Hertie from different nations all over the world). Which column of data would you rather keep? Probably the one with *higher variance*, right? It lets you glean more information from the original matrix $\mathbf{X}$.

Now suppose I told you that you weren't limited to one intact column of data from $\mathbf{X}$. Instead, you're allowed to take a column $\mathbf{z}_1$ that is a *linear combination* of all the columns of $\mathbf{X}$:

$$\mathbf{z}_1 = \phi_{11}\mathbf{x}_1 + \phi_{21}\mathbf{x}_2 + ... + \phi_{p1}\mathbf{x}_p$$

Or you can write an element of $\mathbf{z}_1$, z_{i1} (where $i = 1, 2, ..., n$ corresponds to the rows of the data), as:

$$z_{i1} = \phi_{11}x_{i1} + \phi_{21}x_{i2} + ... + \phi_{p1}x_{ip}$$

Which linear combination would you choose? By the same logic as above, you'd pick the linear combination (i.e., the values of $\phi_{11}, \phi_{21}, ..., \phi_{p1}$) that maximizes the variance of your $\mathbf{z}_1$ vector. That is, you would solve the maximization problem:

$$\begin{aligned}\max_{\phi_{11},\phi_{21},...,\phi_{p1}} \text{Var}(\mathbf{z}_1) &= \max_{\phi_{11},\phi_{21},...,\phi_{p1}} \frac{1}{n}\sum_{i=1}^{n}(z_{i1} - \bar{\mathbf{z}}_1)^2 \\ &= \max_{\phi_{11},\phi_{21},...,\phi_{p1}} \frac{1}{n}\sum_{i=1}^{n} z_{i1}^2\end{aligned}$$

where the first line is the definition of variance ($\bar{\mathbf{z}}_1$ is the column mean of $\mathbf{z}_1$) and the second line follows from the fact that $\bar{\mathbf{z}}_1 = 0$ (since the means of the columns of $\mathbf{X}$ are 0).

The maximal-variance $\mathbf{z}_1$ is called the **first principal component**, and we call $\phi_1 = \begin{bmatrix}\phi_{11}\\ \phi_{21}\\ \cdots\\ \phi_{p1}\end{bmatrix}$ its **loadings**.

We will constrain the loadings so that their sum of squares is equal to one, since allowing these elements to be arbitrarily large could result in an arbitrarily large variance. This adds the following constraint to our optimization problem.

$$\sum_{j=1}^{p} \phi_{j1}^2 = 1$$

We will cover how to do constrained optimization next lecture, but you don't need it today, because it turns out the same maximization problem can be solved using eigendecomposition.

2. PCA as Eigendecomposition of the Variance-Covariance Matrix

We can *equivalently* solve the above maximization problem by applying eigendecomposition to the **variance-covariance matrix** of the centered matrix $\mathbf{X}$, which is given by:

$$\begin{aligned}\mathbf{S} &= \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^T \\ &= \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i\mathbf{x}_i^T\end{aligned}$$

Here, $\mathbf{x}_i$ is a p-length vector of the data corresponding to row i, and $\bar{\mathbf{x}}$ is a p-length vector of the *means* of the data taken over $i = 1$ to n (which all equal zero for the centered data). We perform eigendecomposition on this $p \times p$ matrix:

$$\mathbf{S}\phi_1 = \lambda_1\phi_1$$

where λ_1 is the largest eigenvalue and ϕ_1 is the eigenvector corresponding to the largest eigenvalue. This eigenvector ϕ_1 is the same as the ϕ_1 that solves the constrained maximization problem from part 1 above: it also gives the loadings on the first principal component.[1]

As before, the **first principal component** is given by:

$$\begin{aligned} \mathbf{z}_1 &= \phi_{11}\mathbf{x}_1 + \phi_{21}\mathbf{x}_2 + ... + \phi_{p1}\mathbf{x}_p \\ &= \mathbf{X}\phi_1 \end{aligned}$$

We can interpret this first principal component as a **projection** of our $n \times p$ matrix onto one dimension (i.e., into one n-length vector). This is the most efficient way to store our data matrix in one vector, where "most efficient" means we have maximized the variation captured in that first principal component.

3. Beyond the First Principal Component

So you've chosen wisely and taken from the data matrix $\mathbf{X}$ the first principal component: the data matrix multiplied by the eigenvector corresponding to the largest eigenvalue of the variance-covariance matrix of $\mathbf{X}$. Now suppose I let you have another column, $\mathbf{z}_2$. Which column will you choose next?

If you're going the maximization route, then you should choose the linear combination of the columns of $\mathbf{X}$, $\mathbf{x}_1, ..., \mathbf{x}_p$, that has maximal variance out of all the linear combinations that are *uncorrelated* with the first principal component $\mathbf{z}_1$. This ensures that your next column captures the most of the remaining variance in the data without duplicating information you already have in $\mathbf{z}_1$.

As you might have guessed, this is the same as taking the eigenvector corresponding to the second-largest eigenvalue of the variance-covariance matrix. We can keep constructing the next principal components from the next eigenvectors:

$$\begin{aligned} \mathbf{S}\phi_2 &= \lambda_2\phi_2 \rightarrow \mathbf{z}_2 = \mathbf{X}\phi_2 \\ \mathbf{S}\phi_3 &= \lambda_3\phi_3 \rightarrow \mathbf{z}_3 = \mathbf{X}\phi_3 \\ &\cdots \end{aligned}$$

4. Enough Math, Let's Meet My Cat

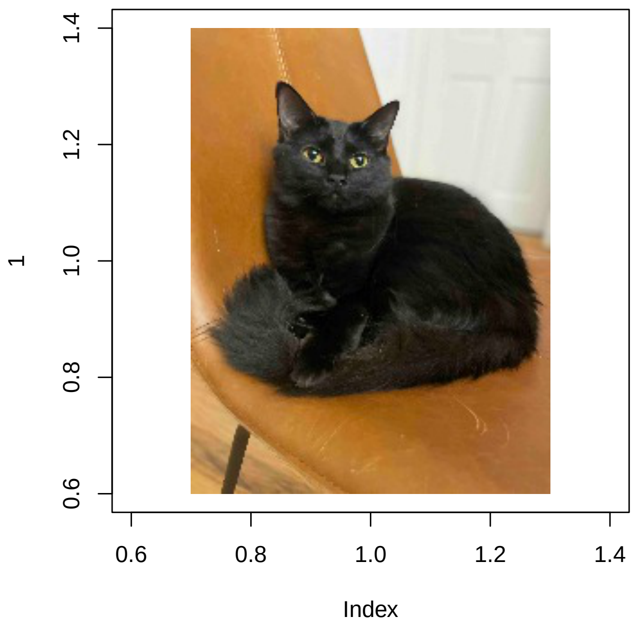
So what is our data matrix in this application? Let's do a quick tutorial on how image data is stored. We load the image of Laszlo and turn it into an **array** of three matrices:

- Each of the three matrices has 267 rows and 200 columns of data, with each data point representing a pixel of the image (our image is therefore a modest 267 by 200 pixels)
- Each of the three (267×200) matrices tells us the concentration of red, green, and blue color at that pixel
- (Any color can be represented as a combination of red, blue, and green of varying intensities.)

The code below loads the picture of Laszlo and plots it. Explore this object. Break it down into its component `r`, `g`, and `b` matrices. Understand how these matrices are structured.

```r
# load laszlo and plot
laszlo <- readJPEG("laszlo.jpg")
plot(1, type = "n")
rasterImage(laszlo, 0.7, 0.6, 1.3, 1.4)
```

[1]We can show this equivalence after we cover constrained optimization in the next lecture.



```r
# store rgb components in separate matrices
r <- laszlo[,,1]
g <- laszlo[,,2]
b <- laszlo[,,3]
```

5. Computing Laszlo's First Principal Component

We will now compute the first principal component of the `r` matrix, which tells us the concentration of red color in every pixel.

- Center the data using the `scale` command (use options `center = TRUE` and `scale = FALSE`).
- Check that it worked by computing the column means of **X**.
- Start by computing the variance-covariance matrix of `r` using the `cov` command. Call this object `vc`.
- Spend some time with this object. What is its dimensionality? Why does it have these dimensions?
- Reconstruct `vc[1,1]` by hand. Interpret this value.
- Do the same for `vc[1,2]` and `vc[1,100]`. Interpret these values. Intuitively, why is `vc[1,2]` a larger (more positive) value than `vc[1,100]`?
- Now use the `eigen` command to find the eigendecomposition of the variance-covariance matrix `vc`. Call this object `eigs`. Use `str` to inspect this object. What does it contain and how is it structured? Where are the loadings?
- Perform matrix multiplication to obtain the first principal component, $\mathbf{z}_1$. That is, multiply the centered data by the first eigenvalue. Look at the dimensions of the output. Do they make sense?

```r
# center the data
r.centered <- scale(r, center = TRUE, scale = FALSE)

# check that it worked

# make variance-covariance matrix
vc <- cov(r.centered)

# check dimensions

# reconstruct cells
# perform eigendecomposition on the variance-covariance matrix
eigs <- eigen(vc)

# multiply centered data by first eigenvector
pcr.1 <- r.centered %*% eigs$vectors[,1]
```

6. The Proportion of Variance Explained

The loading vector ϕ_1 defines a direction in feature space along which the data vary the most. We can compute the **proportion of variance explained** by a principal component as the variance of that principal component divided by the total variance in the data.

The variance of the first principal component is given by:

$$\frac{1}{n}\sum_{i=1}^{n} z_{i1}^2$$

And the total variance of the matrix **X** is:

$$\sum_{j=1}^{p} \mathrm{Var}(\mathbf{x}_j) = \sum_{j=1}^{p} \frac{1}{n}\sum_{i=1}^{n} x_{ij}^2$$

Compute both these quantities and use them to get the total variance explained by our first principal component.

```r
var.pcr.1 <- sum(pcr.1^2) / length(pcr.1)
var.total <- sum(apply(r.centered, 2, function(x) sum(x^2))) / nrow(r.centered)
var.pcr.1/var.total
```

```
## [1] 0.7671094
```

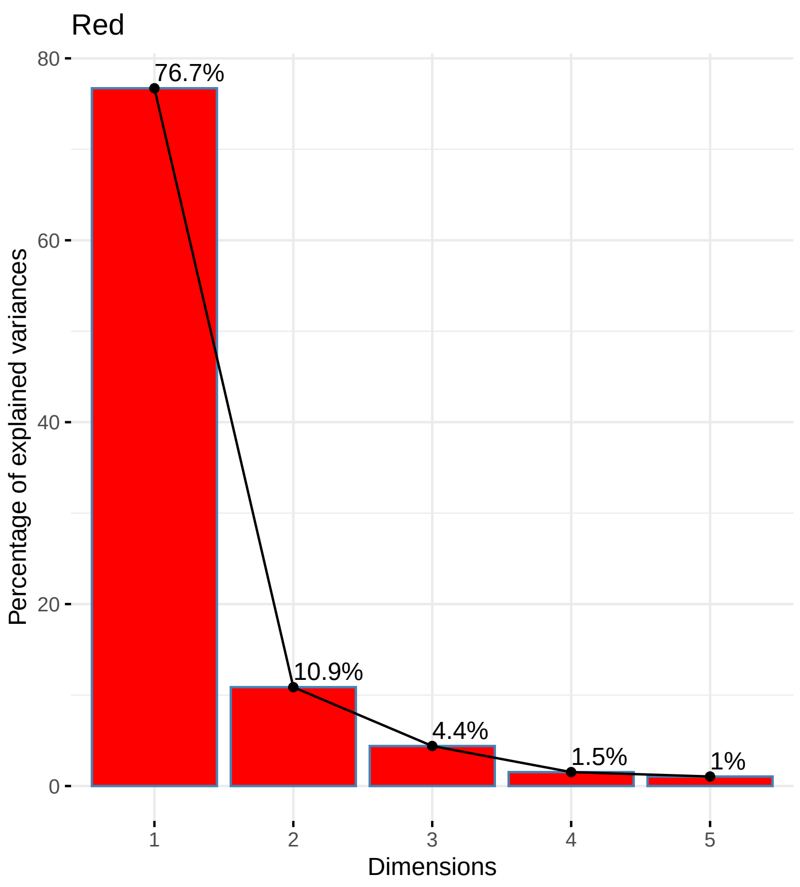
7. Using `prcomp` to Speed Up PCA

Now that we understand what's happening for the first principal component, we can use a function to speed up computation on the remaining principal components.

- Use the `prcomp` function to run PCA on the `r` matrix. Output this to an object called `r.pca`.
- Find the first eigenvector in `r.pca` and compare it to the one you made yourself.[2]
- Find the first principal component in `r.pca` and compare it to the one you made yourself.
- Use the function `fviz_eig` to graph the proportion of variance explained for the first 5 principal components. (This is called a *scree plot*.)

```r
# run pca
r.pca <- prcomp(r, center = TRUE, scale. = FALSE)

# compare eigenvectors to manual

# compare principal components to manual

# automatically compute the scree plot
fviz_eig(r.pca, main = "Red", barfill = "red", ncp = 5, addlabels = TRUE)
```

[2]Note that you can flip the sign on both an eigenvector and its associated eigenvalue and get the same result, so you can ignore differences in signs.



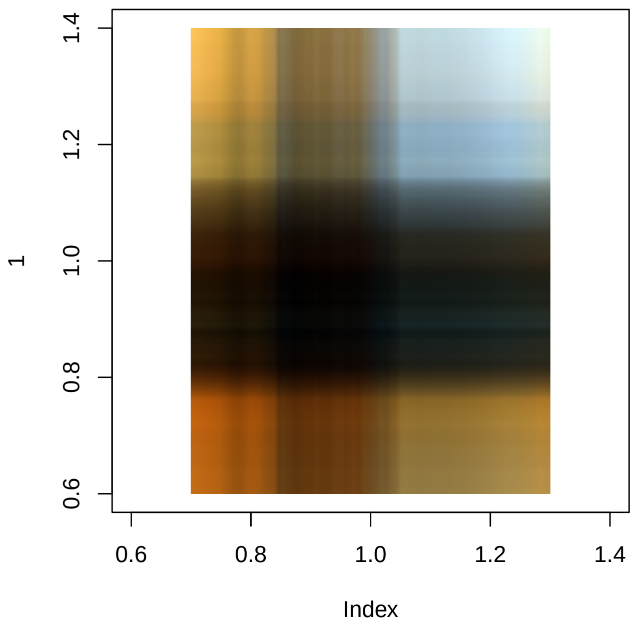
8. Using PCA for Image Compression

Let's see what this image actually looks like for different numbers of principal components. Start by running PCA on the `r`, `g`, and `b` matrices, this time without centering the data (this messes up the color).

```
# run pca
r.pca <- prcomp(r, center = FALSE, scale. = FALSE)
g.pca <- prcomp(g, center = FALSE, scale. = FALSE)
b.pca <- prcomp(b, center = FALSE, scale. = FALSE)

# put them together
rgb.pca <- list(r.pca, g.pca, b.pca)
```

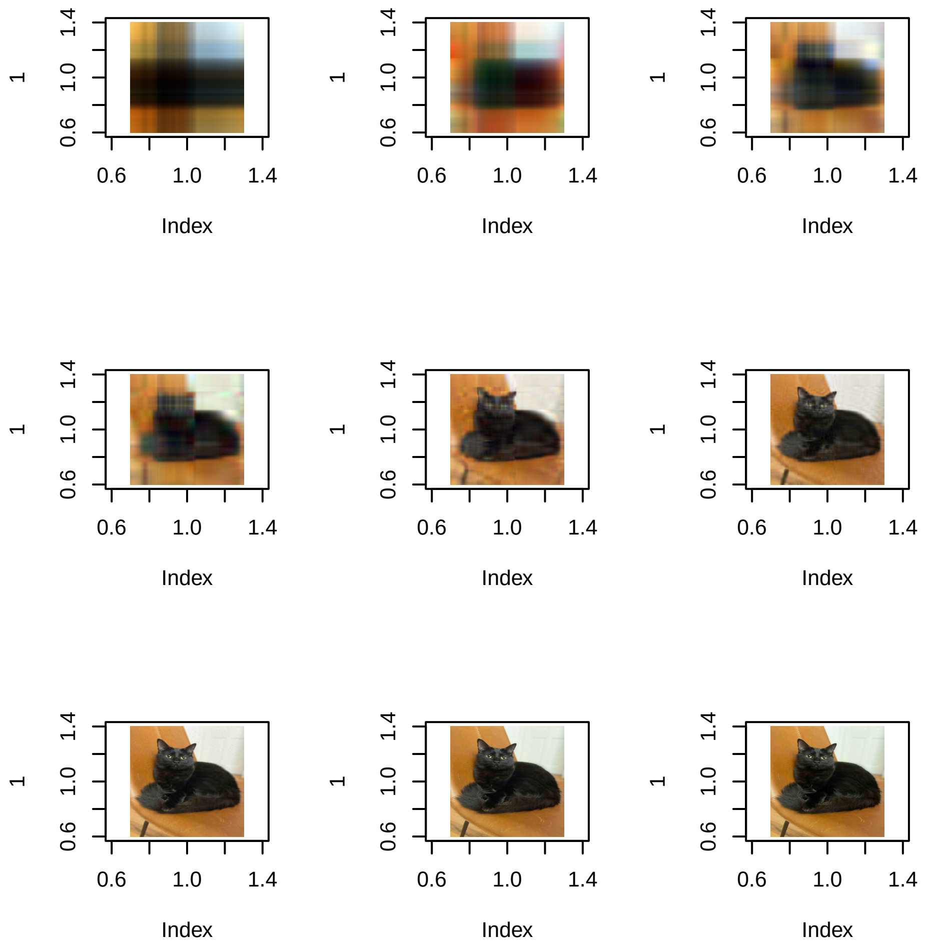
To see what image is created by the first principal component, project $\mathbf{z}_1$ back into p-dimensional space by multiplying it by ϕ_1^T. Plot this image.

```
expanded.1 <- sapply(rgb.pca, function(j) {
  # recompose the compressed image
  new.RGB <- j$x[,1] %*% t(j$rotation[,1])
  # rescale to between 0 and 1
  new.RGB <- (new.RGB - min(new.RGB)) / (max(new.RGB) - min(new.RGB))
}, simplify = "array")

plot(1, type = "n")
rasterImage(expanded.1, 0.7, 0.6, 1.3, 1.4)
```



Although you don't see a cat, this is already pretty impressive: you get the general color palette of the foreground and background of the picture. Repeat this by looping over some values of PCA components and plot the results. How many principal components does it take to recognize my dear cat?

```r
vec <- c(1, 2, 3, 4, 5, 10, 20, 50, 100, 200)
for(i in vec) {
  photo.pca <- sapply(rgb.pca, function(j) {
    # recompose the compressed image
    new.RGB <- j$x[,1:i] %*% t(j$rotation[,1:i])
    # rescale to between 0 and 1
    new.RGB <- (new.RGB - min(new.RGB)) / (max(new.RGB) - min(new.RGB))
  }, simplify = "array")
  assign(paste("photo_", round(i, 0), sep = ""), photo.pca)
}

par(mfrow=c(3,3))
plot(1, type = "n")
rasterImage(photo_1, 0.7, 0.6, 1.3, 1.4)
plot(1, type = "n")
rasterImage(photo_2, 0.7, 0.6, 1.3, 1.4)
plot(1, type = "n")
rasterImage(photo_3, 0.7, 0.6, 1.3, 1.4)
plot(1, type = "n")
rasterImage(photo_5, 0.7, 0.6, 1.3, 1.4)
plot(1, type = "n")
rasterImage(photo_10, 0.7, 0.6, 1.3, 1.4)
plot(1, type = "n")
rasterImage(photo_20, 0.7, 0.6, 1.3, 1.4)
plot(1, type = "n")
rasterImage(photo_50, 0.7, 0.6, 1.3, 1.4)
plot(1, type = "n")
rasterImage(photo_100, 0.7, 0.6, 1.3, 1.4)
plot(1, type = "n")
rasterImage(photo_200, 0.7, 0.6, 1.3, 1.4)
```



Time permitting, try PCA on a black and white image. Create a new matrix that is the sum of the `r`, `g`, and `b` matrices, then divide every cell by the maximum value. Repeat PCA with this object.