Lecture 10 - Computational Linear Algebra Part 2

DSE 512

Drew Schmidt
2022-02-24

From Last Time

- Homework is out --- due Saturday
- Questions?

Revisiting Cholesky

A random draw from a multivariate normal distribution can be obtained using the Cholesky decomposition of Σ and a vector of univariate normal draws. The Cholesky decomposition of Σ produces a lower-triangular matrix A (the 'Cholesky factor') for which $AA^T = \Sigma$. If $z = (z_1, \ldots, z_d)$ are d independent standard normal random variables, then $\theta = \mu + Az$ is a random draw from the multivariate normal distribution with covariance matrix Σ.

Gelman, A., Carlin, J.B., Stern, H.S. and Rubin, D.B., 1995. Bayesian data analysis. Chapman and Hall/CRC.

Two Techniques

- Principal Components Analysis (PCA)
- Linear Regression

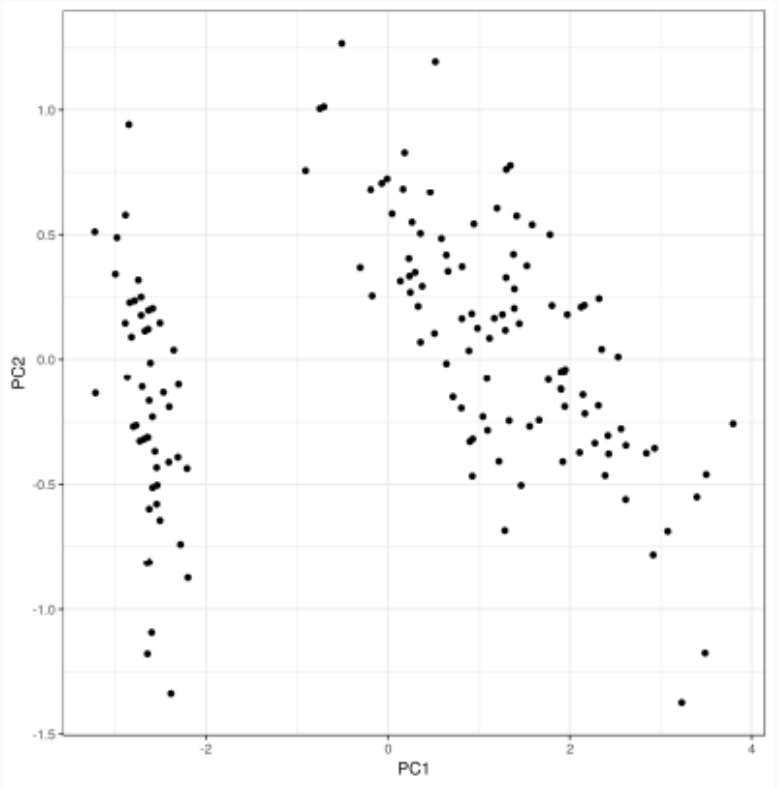
Setup

- X is an $m \times n$ (model) matrix with $m > n$
- y is an m-length (response) vector
- Example code in R
- We will ignore some efficiency tricks for the moment
  - `crossprod(X)` instead of `t(X) %*% X`
  - `backsolve()` over inversion
  - etc.
- One thing at a time...

Principal Components Analysis

PCA

	A	B	C	D
1	5.1	3.5	1.4	0.2
2	4.9	3	1.4	0.2
3	4.7	3.2	1.3	0.2
4	4.6	3.1	1.5	0.2
5	5	3.6	1.4	0.2
6	5.4	3.9	1.7	0.4
7	4.6	3.4	1.4	0.3
8	5	3.4	1.5	0.2
9	4.4	2.9	1.4	0.2
10	4.9	3.1	1.5	0.1
11	5.4	3.7	1.5	0.2
12	4.8	3.4	1.6	0.2



PCA: The Black Box

- Via covariance matrix `princomp()`
- Via SVD `prcomp()`

- Pros
  - It just works!
  - Reasonably efficient
- Cons
  - Stock methods do what they do…
  - What if we run out of memory?
  - What if we want to go parallel?

PCA: Covariance Matrix

- Step 1: compute covariance matrix Σ_X
- Step 2: compute $\Sigma_X = V \Delta V^T$
  - "Standard deviations": $\sqrt{\delta_i}$
  - Loadings: V
  - Scores: XV

```
# compare to princomp()
X_cen = scale(X, center=TRUE, scale=FALSE)
eig = eigen(crossprod(X_cen))

sdev = sqrt(eig$values)
loadings = eig$vectors
scores = X_cen %*% eig$vectors
```

- Pros
  - Very easy to implement
  - Can be very fast
  - Useful in distributed contexts
  - Memory-efficient if n is small
- Cons
  - What happens to the condition number?
  - What happens if n is very large?

PCA: SVD

- Step 1: mean-center x
- Step 2: compute $X = U\Sigma V^T$
  - "Standard deviations": $\frac{\sigma_i}{n}$
  - Loadings: V
  - Scores: XV

```
# compare to prcomp()
X_cen = scale(X, center=TRUE, scale=FALSE)
s = svd(X_cen)

sdev = s$d / n
loadings = s$v
scores = X_cen %*% s$v
```

- Pros
  - Numerically accurate
- Cons
  - Harder to parallelize in distributed contexts

An Observation

- Computing PCA is tantamount to computing SVD
- Many SVD algorithms
- We will revisit this when we get to parallelism

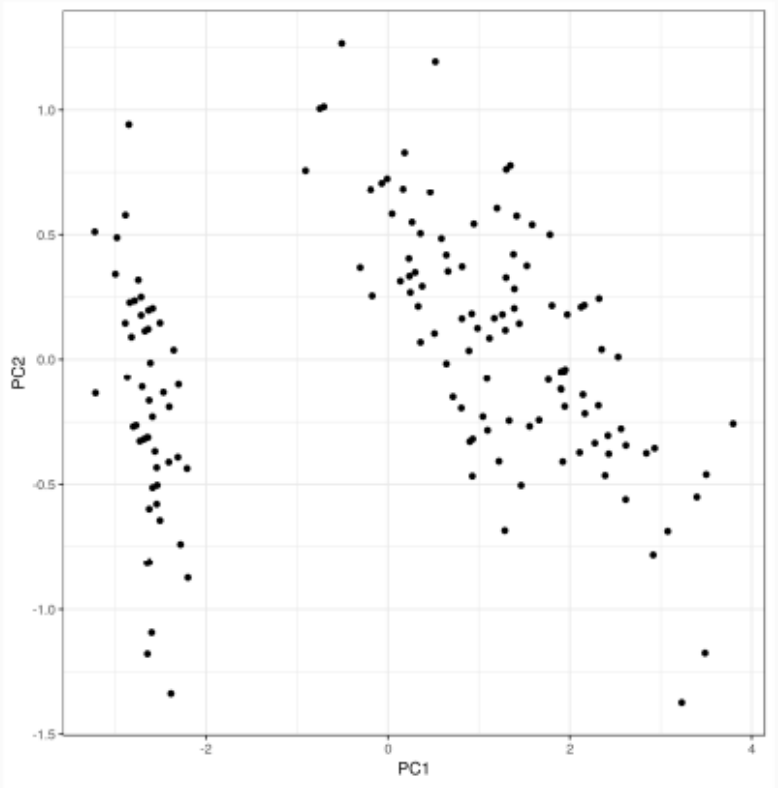
Schmidt, D., 2020, November. A Survey of Singular Value Decomposition Methods for Distributed Tall/Skinny Data. In 2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA) (pp. 27-34). IEEE.

Question

What if we just want the first few components?

PCA

	A	B	C	D
1	5.1	3.5	1.4	0.2
2	4.9	3	1.4	0.2
3	4.7	3.2	1.3	0.2
4	4.6	3.1	1.5	0.2
5	5	3.6	1.4	0.2
6	5.4	3.9	1.7	0.4
7	4.6	3.4	1.4	0.3
8	5	3.4	1.5	0.2
9	4.4	2.9	1.4	0.2
10	4.9	3.1	1.5	0.1
11	5.4	3.7	1.5	0.2
12	4.8	3.4	1.6	0.2



Truncated SVD

svd

```
function (x, nu = min(n, p), nv = min(n, p))
{
    # ...
    n <- nrow(x)
    p <- ncol(x)
    # ...
    res <- if (is.complex(x))
        .Internal(La_svd_cmplx(jobu, x, double(min(n, p)), u,
            vt))
    else .Internal(La_svd(jobu, x, double(min(n, p)), u, vt))
    res <- res[c("d", if (nu) "u", if (nv) "vt")]
    if (nu && nu < nu0)
        res$u <- res$u[, seq_len(min(n, nu)), drop = FALSE]
    if (nv && nv < nv0)
        res$vt <- res$vt[seq_len(min(p, nv)), , drop = FALSE]
    res
}
```

Randomized SVD

Prototype for Randomized SVD

Given an $m \times n$ matrix $\boldsymbol{A}$, a target number k of singular vectors, and an exponent q (say $q = 1$ or $q = 2$), this procedure computes an approximate rank-$2k$ factorization $\boldsymbol{U\Sigma V}^*$, where $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthonormal, and $\boldsymbol{\Sigma}$ is nonnegative and diagonal.

Stage A:

1. Generate an $n \times 2k$ Gaussian test matrix $\boldsymbol{\Omega}$.
2. Form $\boldsymbol{Y} = (\boldsymbol{AA}^*)^q \boldsymbol{A\Omega}$ by multiplying alternately with $\boldsymbol{A}$ and $\boldsymbol{A}^*$.
3. Construct a matrix $\boldsymbol{Q}$ whose columns form an orthonormal basis for the range of $\boldsymbol{Y}$.

Stage B:

4. Form $\boldsymbol{B} = \boldsymbol{Q}^* \boldsymbol{A}$.
5. Compute an SVD of the small matrix: $\boldsymbol{B} = \widetilde{\boldsymbol{U}} \boldsymbol{\Sigma V}^*$.
6. Set $\boldsymbol{U} = \boldsymbol{Q}\widetilde{\boldsymbol{U}}$.

Note: The computation of $\boldsymbol{Y}$ in Step 2 is vulnerable to round-off errors. When high accuracy is required, we must incorporate an orthonormalization step between each application of $\boldsymbol{A}$ and $\boldsymbol{A}^*$; see Algorithm 4.4.

Halko, N., Martinsson, P.G. and Tropp, J.A., 2009. Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions.

Randomized SVD

Algorithm 4: Randomized SVD

Data: 1-d distributed real matrix A with $m > n$, integers $k < n$ and q

Result: Σ, optionally U and V

Generate $\Omega_{n \times 2k}$ random uniform or standard normal;

Let $Y_{m \times 2k} = A\Omega$;

Compute $Q_Y = \texttt{qr\_Q}(Y)$ (CAQR);

for $i \leftarrow 0$ **to** q **do**

  $Z_{n \times 2k} = A^T Q_Y$;

  $Q_Z = \texttt{qr\_Q}(Z)$ (local);

  $Y = AQ_Z$;

  $Q_Y = \texttt{qr\_Q}(Y)$ (CAQR);

end

Let $B_{2k \times n} = Q_Y^T A$;

Local SVD of B gives approximation for Σ, and optionally V^T. Recover $U = Q_Y U_B$ if desired.

Randomized SVD

```
rsvd = function(x, k=1, q=3){
  ### Stage A
  n = ncol(x)
  Omega = matrix(runif(n*2L*k),
    nrow=n, ncol=2L*k)

  Y = x %*% Omega
  Q = qr.Q(qr(Y))
  for (i in 1:q){
    Y = crossprod(x, Q)
    Q = qr.Q(qr(Y))
    Y = x %*% Q
    Q = qr.Q(qr(Y))
  }
  ### Stage B
  B = crossprod(Q, x)
  svd_B = La.svd(x=B, nu=0, nv=0)
  d = svd_B$d[1:k]
  d
}
```

- Pros
  - Fast
  - Memory efficient
  - "Good enough" for plotting
- Cons
  - Numerical accuracy
  - How do we recover U and V? (Hint: inspect B)

rsvd Benchmark

```
m = 1e6
n = 500
X = matrix(runif(m*n), nrow=m, ncol=n)

system.time(svd(X, nu=0, nv=0))
```

```
##     user  system elapsed
##  164.000  22.959  32.814
```

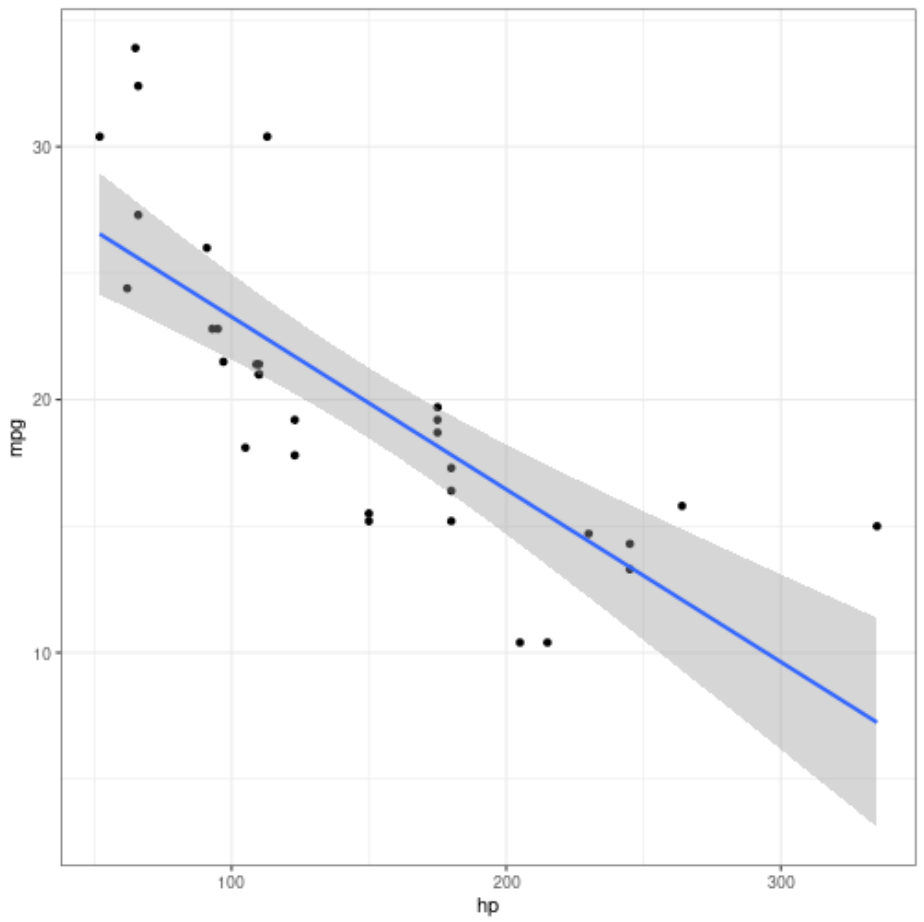
```
system.time(rsvd(X, k=2))
```

```
##    user  system elapsed
##  18.599   7.051   4.712
```

Linear Regression

Linear Regression

```
library(ggplot2)
g = ggplot(data=mtcars, aes(hp, mpg)) +
  theme_bw() +
  geom_point() +
  geom_smooth(method="lm")
```



Regression: The Black Box

- Uses a rank-revealing QR
- Modified LINPACK (!) code
- Pivoting destroys statistical information…

```
lm.fit(X, y)$coef
```

- Pros
  - It just works!
  - It gets the statistics right when rank degenerate (THIS IS VERY HARD)
- Cons
  - Not very efficient
  - Same as before: memory, parallelism, etc

Regression: The Normal Equations

$$
\begin{aligned}
y = X\beta &\iff X^T y = X^T X\beta \\
&\iff \left(X^T X\right)^{-1} X^T y = \beta
\end{aligned}
$$

```
solve(t(X) %*% X) %*% t(X) %*% y
```

- Pros
  - Easy to understand
  - Great for teaching
- Cons
  - Doesn't handle rank degeneracy
  - Other accuracy problems

Regression: QR Factorization

$$
\begin{aligned}
y = X\beta &\iff y = QR\beta \\
&\iff R^{-1}Q^T y = \beta
\end{aligned}
$$

```
qr.solve(qr(X), y)
```

- Pros
  - Mostly like what `lm.fit()` is doing!
  - Numerically good
- Cons
  - LAPACK QR solver has pivoting issues (ANOVA)

Regression: SVD

$$y = X\beta \iff \beta = V\Sigma^{-1}U^T y$$

```
s = La.svd(X)
t(s$vt) %*% diag(1/s$d) %*% t(s$u) %*% y
```

- Pros
  - Accurate
  - Can accommodate rank degeneracy
- Cons
  - Not the fastest

Regression: Optimization Problem

$$\min_{\beta \in \mathbb{R}^n} \frac{1}{2m} \sum_{i=1}^{m} ((X\beta)_i - y_i)^2$$

```
cost_gaussian = function(beta, x, y){
  m = nrow(x)
  (1/(2*m))*sum((x%*%beta - y)^2)
}

reg.fit = function(x, y, maxiter=100){
  control = list(maxit=maxiter)
  beta = numeric(ncol(x))
  optim(par=beta, fn=cost_gaussian,
    x=x, y=y, method="CG", control=contro
  )
}
```

```
reg.fit(X, y)$par
```

- Pros
  - Easy to implement
  - Can easily re-purpose for other GLM's
  - Parallelizes easily!
- Cons
  - Performance depends *a lot* on optimization method
  - How even would we handle rank degeneracy?

Ungraded Homework

- Do some basic benchmarking of these methods. How do they compare to the stock R methods?
- Can we improve the performance of these (yes)?
- Implement your own conjugate gradient https://en.wikipedia.org/wiki/Conjugate_gradient_method#Example_code_in_MA
- Implement these in Python using NumPy
- What relationship will (un-)optimized BLAS have to what we have seen today?
- Let X be the 7×7 Hilbert Matrix (see https://en.wikipedia.org/wiki/Hilbert_matrix). You can generate this in R via `Matrix::Hilbert(7)`. Test your linear model fitters on this data (response can be random uniform). Does anything interesting happen?

Wrapup

- Next time:
  - GPGPU: The Easy Parts
  - Return to ISAAC
- Resources
  - McCullagh, P. and Nelder, J.A., 1989. Generalized Linear Models, no. 37 in Monograph on Statistics and Applied Probability.
  - Duda, R.O., Hart, P.E. and Stork, D.G., 1973. Pattern classification (pp. 526-528). Wiley, New York.

Questions?