STATS 790 - Statistical Learning
Project - Final

Tommy Flynn

27 April, 2023

Contents

1 Introduction

Principal component analysis (PCA) is a powerful statistical learning technique for dimensionality reduction and uncorrelated feature learning. It achieves this by linearly transforming the data to a basis of principal components which are the directions of maximal information. PCA on n observations and p variables can be given as $\mathbf{T}_{n\times p} = \mathbf{X}_{n\times p}\mathbf{P}_{p\times p}$ where $\mathbf{X}$ is the standardized data matrix, $\mathbf{T}$ is the score matrix, and $\mathbf{P}$ is the loading matrix whose columns are the principal components. Typically, PCA is computed using singular value decomposition (SVD) as it is computationally efficient, numerically stable, and mathematically sound. Because SVD is an exact method, it necessarily computes all principal components even if only a subset is desired. In this project, we will investigate an alternative method, namely, non-linear iterative partial least squares (NIPALS). Unlike SVD, NIPALS is an iterative method which terminates at the desired number of principal components and which naturally accommodates missing data. It will be implemented in R and compared to R's built-in SVD-PCA algorithm based on speed, memory, and accuracy.

2 Methods

2.1 Singular Value Decomposition (SVD)

SVD is a matrix factorization of the form $\mathbf{X}_{n\times p} = \mathbf{U}_{n\times n}\mathbf{D}_{n\times p}\mathbf{V}'_{p\times p}$ where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal and $\mathbf{D}$ is diagonal. SVD of the data matrix solves PCA through the relations $\mathbf{T} = \mathbf{UD}$ and $\mathbf{P} = \mathbf{V}$. It's solid mathematically framework facilitates numerical stability along with a time and space complexity of $\mathcal{O}(np\min(n,p))$ and $\mathcal{O}(np)$ [5]. Since it is applied directly to the data matrix it also avoids computing and storing the covariance matrix, a requirement of other methods. Below is the built-in R code to perform SVD-PCA on the standardized data matrix.

```r
# PCA using SVD - returns loading matrix
pca.svd  <- prcomp(X, center = FALSE, scale = FALSE)
```

2.2 Non-linear Iterative Partial Least Squares (NIPALS)

The NIPALS algorithm can be applied to PCA to obtain one principal component at a time. Let h denote the principal component and $\mathbf{X}$ the standardized data matrix. For $h = 1$, define the residual

matrix as $\mathbf{R}_h := \mathbf{X}$, and the score vector $\vec{t}_h$ as any non-zero column vector. Then the loading vector is calculated and normalized.

$$\vec{p}_h := \frac{\mathbf{R}_h' \vec{t}_h}{\vec{t}_h' \vec{t}_h}, \quad \vec{p}_h := \frac{\vec{p}_h}{\sqrt{\vec{p}_h' \vec{p}_h}}$$

This is regressing the columns of $\mathbf{R}_h$ onto $\vec{t}_h$ with solution $\vec{p}_h$. Next the new score vector is calculated.

$$\vec{t}_{h_{\text{new}}} := \frac{\mathbf{R}_h \vec{p}_h}{\vec{p}_h' \vec{p}_h}$$

This is regressing the rows of $\mathbf{R}_h$ onto $\vec{p}_h$ with solution $\vec{t}_{h_{\text{new}}}$. This process iterates until convergence which is achieved when the difference between two successive score vectors has squared Euclidean norm smaller than some tolerance ϵ.

$$||\vec{t}_{h_{\text{new}}} - \vec{t}_h||_2^2 < \epsilon$$

After convergence, we store the vectors $\vec{t}_h$ and $\vec{p}_h$ in column h of matrices $\mathbf{T}$ and $\mathbf{P}$. The final step is to deflate the residual matrix which removes the variability captured by the current score and loading vectors.

$$\mathbf{R}_{h+1} = \mathbf{R}_h - \vec{t}_h \vec{p}_h'$$

The algorithm continues from $h = 1$ to the desired number of principal components. One of the main issues with NIPALS is the accumulation of floating-point errors at each iteration. This undermines the orthogonality of the scores and loadings. To combat this, Andercut (2008) proposed the GS-PCA algorithm [1]. At each iteration with $h > 1$, the loadings and scores are corrected with Gram-Schmidt orthogonalization using the previously calculated columns of $\mathbf{P}$ and $\mathbf{T}$.

$$\vec{p}_h := \vec{p}_h - \mathbf{P}_h \mathbf{P}_h' \vec{p}_h, \quad \vec{t}_{h_{\text{new}}} := \vec{t}_{h_{\text{new}}} - \mathbf{T}_h \mathbf{T}_h' \vec{t}_{h_{\text{new}}}$$

Below is an implementation of the GS-PCA algorithm in R.

```r
nipals <- function(X, pcs = min(nrow(X), ncol(X)),
                   center = TRUE, scale = TRUE,
                   tol = 1e-6, max.iter = 500) {
  # ARGUMENTS
  # X        - data (matrix)
  # pcs      - number of principal components (int)
  # center   - center the data to mean zero (bool)
  # scale    - scale data to unit variance (bool)
  # tol      - tolerance for convergence (numeric)
  # max.iter - max iterations without convergence (int)

  # VALUES
  # P         - loadings (matrix)
  # Tm        - scores (matrix)
  # eig       - eigenvalues (vector)
  # explained - variance explained by each component (vector)

  # initialize
  n <- nrow(X)                                    # rows
  p <- ncol(X)                                    # columns
  X <- scale(X, center = center, scale = scale)   # standardize
  Tm <- matrix(0, nrow = n, ncol = pcs)           # score matrix
  P <- matrix(0, nrow = p, ncol = pcs)            # loading matrix
  R <- X                                          # residual matrix
  TSS <- sum(R^2)                                 # total sum of squares
  eig <- rep(0, pcs)                              # eigenvalues
  explained <- rep(0, pcs)                        # variance explained

  # for each component
  for (i in 1:pcs) {
    t.h <- R[, i] # initial score vector
```

```
  # iterations
  for (j in 1:max.iter) {
    # loading vector / GS correction / normalize
    p.h <- crossprod(R, t.h) / sum(t.h^2)
    if (i > 1){
      p.h <- p.h - P[, 1:(i-1)] %*% crossprod(P[, 1:(i-1)], p.h)
    }
    p.h <- p.h / sqrt(sum(p.h^2))

    # score vector / GS correction / score vector difference
    t.h2 <- R %*% p.h
    if (i > 1){
      t.h2 <- t.h2 - Tm[, 1:(i-1)] %*% crossprod(Tm[, 1:(i-1)], t.h2)
    }
    e <- t.h2 - t.h

    # convergence check
    if (sum(e^2) < tol){
      break
    } else{
      t.h <- t.h2
    }
  }

  # update and deflate
  Tm[, i] <- t.h
  P[, i] <- p.h
  eig[i] <- sum(t.h^2)
  explained <- eig[i] / TSS
  R <- R - tcrossprod(t.h, p.h)
}
```

```
  return(list(P=P, Tm=Tm, eig=eig, explained=explained))
}
```

3 Results

We will compare `prcomp` to `nipals` based on speed, memory, and accuracy. The comparisons will use square matrices of sizes in $S = \{10, 50, 100, 500, 1000, 1500, 2000\}$. Note that speed, memory, and accuracy are intimately related to tolerance, maximum iterations, and number of principal components. Scikit Learn uses a tolerance of 10^{-6} and maximum iterations of 500 for their partial least squares function so we will use the same [3]. In every case of PCA we are interested in at least the first two principal components so we will compute up to $h = 2$. These results serve as a lower bound considering the possibility of more optimized code or even the use of C++. Below is the general set-up.

```
# testing set-up
library(ggplot2)
library(microbenchmark)
library(peakRAM)
set.seed(13)

# generates (n x p) standardized data matrix
generate.data <- function(n, p){
  X <- scale(matrix(rnorm(n*p), nrow = n, ncol = p), center=TRUE, scale=TRUE)
  return(X)
}

# test sizes
sizes <- c(10, 50, 100, 500, 1000, 1500, 2000)
N <- length(sizes)
```

3.1 Speed

Figure 1 shows the runtime comparison and below is the code used to generate the benchmark. We see that SVD outperforms NIPALS until the matrix grows to a size of around 1000×1000. Keep in mind that this is only for two principal components. Moreover, SVD continues to grow linearly at the edge of the plot whereas NIPALS appears to have a slightly diminishing slope.

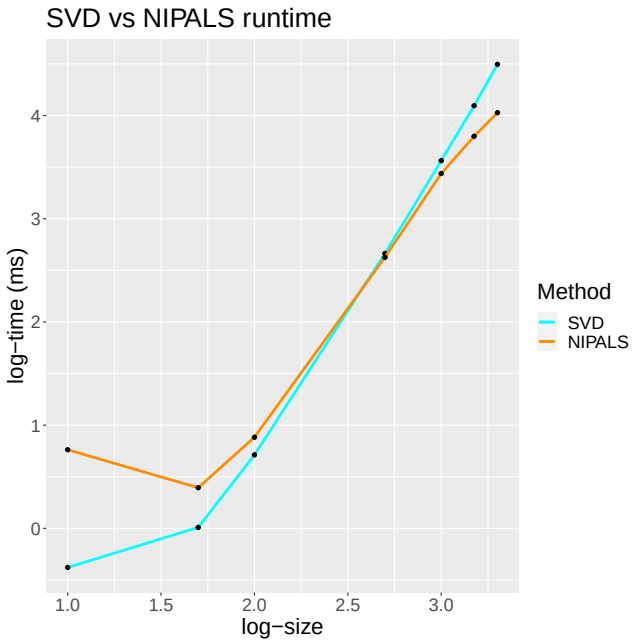

Figure 1: Runtime log-log plot.

```
# time benchmark
time.list <- vector('list', N)
for (i in 1:N) {
    X <- generate.data(sizes[i], sizes[i])
    times <- microbenchmark(
      prcomp(X, center=FALSE, scale=FALSE),
      nipals(X, 2, center=FALSE, scale=FALSE),
      times = 10,
      unit ='ms')
    # store
    time.list[[i]] <- summary(times)$mean
}
```

```r
# time data
time.r <- log10(sapply(time.list, '[', 1))
time.nipals <- log10(sapply(time.list, '[', 2))
logx <- log10(sizes)

# time plot
p <- ggplot() +
  geom_line(aes(x=logx, y=time.r, color='SVD'), size = 1.15) +
  geom_point(aes(x=logx, y=time.r), color='black') +
  geom_line(aes(x=logx, y=time.nipals, color='NIPALS'), size = 1.15) +
  geom_point(aes(x=logx, y=time.nipals), color='black') +
  scale_color_manual(values = c('SVD'='cyan', 'NIPALS'='darkorange')) +
  theme(text = element_text(size=20)) +
  labs(x = 'log-size',
       y = 'log-time (ms)',
       title = 'SVD vs NIPALS runtime',
       color = 'Method')
```

3.2 Accuracy and memory

Given that NIPALS begins to outperform SVD in terms of speed at a size of 1000×1000, we will assess accuracy and memory at this point for 2 and 10 principal components. To assess the accuracy we will use the spectral norm of the loading matrix difference. Note that the signs can be different so we must apply the absolute value function to each element of the matrices.

$$||\mathbf{D}||_2 = \sqrt{\lambda_{\max}(\mathbf{D}'\mathbf{D})} = \sigma_{\max}(\mathbf{D}), \quad \mathbf{D} = \text{abs}(\mathbf{P}_{\text{SVD}}) - \text{abs}(\mathbf{P}_{\text{NIPALS}})$$

Below is the R code to calculate the accuracy and the peak RAM values during the computations. For two principal components, the average norm value and peak RAM values were 0.000286, 9016167 MiB, and 9462102 MiB for SVD and NIPALS respectively. For 10 principal components, the values were 0.3348609, 9016167 MiB, and 9421940 MiB so some accuracy was lost.

```r
set.seed(13)


# accuracy and memory
pcs <- 10
peak.r <- 0
peak.nipals <- 0
accuracy <- 0
for (i in 1:10) {
  X <- generate.data(10^3, 10^3)
  mem.r <- peakRAM(P.r <- prcomp(X, scale = FALSE, center = FALSE))
  mem.nipals <- peakRAM(P.nipals <- nipals(X, pcs = pcs,
                                           scale = FALSE, center = FALSE))
  if (mem.r$Peak_RAM_Used_MiB > peak.r) {
    peak.r <- mem.r$Peak_RAM_Used_MiB
  }
  if (mem.nipals$Peak_RAM_Used_MiB > peak.nipals) {
    peak.nipals <- mem.nipals$Peak_RAM_Used_MiB
  accuracy <- accuracy + norm(abs(P.r$rotation[, 1:pcs]) - abs(P.nipals$P), "2")
  }
}
accuracy <- accuracy/10
```

4 Conclusions

Overall, NIPALS is an accurate PCA method for all sizes of data consistently having a low spectral norm with SVD. For small to medium sized datasets, NIPALS is much slower and more memory intensive than SVD. It is only when we reach sizes of 10^3 that we start to see the benefits of an iterative method. If we extrapolate Figure 1, we can imagine for extremely large p that NIPALS is very useful. It is also an option when missing data is prevalent. Therefore, in most cases SVD is advised until the data becomes too large or we need to handle missing data.

5 References

[1] Andrecut, M. 2008. *Parallel GPU implementation of iterative PCA algorithms.* arxiv. https://arxiv.org/pdf/0811.1081.pdf

[2] Dunn, Kevin. 2023. *Process Improvement Using Data.* https://learnche.org/pid/latent-variable-modelling/principal-component-analysis/algorithms-to-calculate-build-pca-models

[3] Scikit Learn. 2022. *PLSRegression.* https://scikit-learn.org/stable/modules/generated/sklearn.cross_decomposition.PLSRegression.html

[4] Stanstrup, Jan. 2021. *nipals.R.* GitHub. https://github.com/kwstat/nipals/blob/main/R/nipals.R

[5] V, Vasudevan., & Ramakrishna, M. 2019. *A Hierarchical Singular Value Decomposition Algorithm for Low Rank Matrices.* arXiv. https://arxiv.org/pdf/1710.02812.pdf#:~:text=Computing%20the%20SVD%20of%20an,expensive%20for%20large%20data%20sets.

[6] Wright, Kevin. 2017. *The NIPALS algorithm.* https://cran.r-project.org/web/packages/nipals/vignettes/nipals_algorithm.html