STATS 790 - Statistical Learning
Project Draft

Tommy Flynn

11 April, 2023

Contents

1 Introduction

Principal component analysis (PCA) is a powerful statistical/unsupervised learning technique typically used for dimensionality reduction. It achieves this by linearly transforming the data to a basis of principal components which are the directions of maximal information. It can be given as $\mathbf{T}_{n\times p} = \mathbf{X}_{n\times p}\mathbf{P}_{p\times p}$ where $\mathbf{X}$ is the standarized data matrix, $\mathbf{T}$ is the score matrix, and $\mathbf{P}$ is the loading matrix whose columns are the principal components. The gold standard for computing PCA in R is singular value decomposition (SVD) implemented in the `prcomp` function. In this project, we will investigate an alternative method, namely, non-linear iterative partial least squares (NIPALS). It will be implemented from scratch in R and we will compare it to the gold standard based on accuracy, speed, and functionality.

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 orthonormal and $\mathbf{D}$ is diagonal. Applying SVD to the data matrix results in the relations $\mathbf{T} = \mathbf{UD}$ and $\mathbf{P} = \mathbf{V}$. R considers SVD to be the gold-standard for PCA because it is mathematically sound, computationally efficient, and numerically stable. It is preferred to eigendecomposition because it is applied directly to the data matrix without having to calculate the covariance matrix. Below is the built-in R code to perform PCA using SVD with standardization.

```
# PCA using SVD
P.svd  <- prcomp(X, center = TRUE, scale = TRUE)$rotation
```

Note: we could also calculate $\mathbf{P}$ using `svd(scale(X))$v`.

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. For the first principal component $h = 1$ we initialize a residual matrix as the standardized data matrix

$\mathbf{X}_h = \mathbf{X}$ and score vector $\vec{t}_h$ as any non-zero vector. The loading vector is calculated as

$$\vec{p}_h = \frac{\mathbf{X}_h' \vec{t}_h}{\vec{t}_h' \vec{t}_h}$$

This is regressing the columns of $\mathbf{X}_h$ onto $\vec{t}_h$ with solution $\vec{p}_h$. After we normalize $\vec{p}_h$ as

$$p_h = \frac{\vec{p}_h}{\sqrt{\vec{p}_h' \vec{p}_h}}$$

we calculate the new scores

$$t_h = \frac{\mathbf{X}_h \vec{p}_h}{\vec{p}_h' \vec{p}_h}$$

This is regressing the rows of $\mathbf{X}_h$ onto $\vec{p}_h$ with solution $\vec{t}_h$. This process iterates until the score vector converges

$$||t_{h_{l+1}} - t_{h_l}||_2 < \epsilon$$

i.e. there is a lack of progress of size less than ϵ between iteration l and $l+1$. Once we obtain $\vec{t}_h$ and $\vec{p}_h$ we can store them in column h of $\mathbf{T}$ and $\mathbf{P}$ respectively. The final step before moving to the next principal component is to remove the variability captured by the current score and loading vectors. This is known as deflating the residual matrix

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

The algorithm iterates 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. Because of these errors we quickly lose orthogonality of the scores and loadings. Andercut (2009) proposed the GS-PCA algorithm to combat this. At each iteration ($h > 1$), the scores and loadings are corrected with Gram-Schmidt orthogonalization. In practice, we have loadings correction

$$p_h = p_h - \mathbf{P}_h \mathbf{P}_h' \vec{p}_h$$

and scores correction

$$t_h = t_h - \mathbf{T}_h \mathbf{T}_h' \vec{t}_h$$

where $\mathbf{P}_h$ and $\mathbf{T}_h$ are the loading and score matrices of the first h components. Below is an implementation of the GS-PCA algorithm in R.

```r
pca.nipals <- function(X, pcs = ncol(X), center = TRUE, scale = TRUE,
                       tol = 1e-9, max.iter = 500) {

  #' Performs PCA using Non-linear Iterative Partial Least Squares (NIPALS)
  #' with Gram-Schmidt orthogonalization.
  #'
  #' @param X Data matrix.
  #' @param pcs Number of principal components to compute.
  #' @param center Center the data to zero mean.
  #' @param scale Scale the data to unit variance.
  #' @param tol Tolerance for lack of progress.
  #' @param mat.iter Max iterations without progress.
  #' @return Loading matrix P.

  # initialize
  X <- scale(X, center = center, scale = scale)           # standardize
  Tm <- matrix(0, nrow = nrow(X), ncol = pcs)             # score matrix
  P <- matrix(0, nrow = ncol(X), ncol = pcs)              # loading matrix
  X.h <- X                                                # residual matrix
  t.h <- X.h[, 1]                                         # initial score

  # for each component
  for (i in 1:pcs) {
    for (j in 1:max.iter) {
      p.h <- (t(X.h) %*% t.h) / (t(t.h) %*% t.h)[1]       # loading
      p.h <- p.h - P[, 1:i] %*% t(P[, 1:i]) %*% p.h       # GS correction
      p.h <- p.h / sqrt(t(p.h) %*% p.h)[1]                # unit vector

      t.h2 <- X.h %*% p.h / (t(p.h) %*% p.h)[1]           # score
      t.h2 <- t.h2 - Tm[, 1:i] %*% t(Tm[, 1:i]) %*% t.h2  # GS correction
```

```
      # continue until lack of progress
      if (sum((t.h2 - t.h)^2) < tol) {
        break
      }
      else {
        t.h <- t.h2
      }
    }

    # update and deflate
    Tm[, i] <- t.h
    P[, i] <- p.h
    X.h <- X.h - t.h %*% t(p.h)
  }
  return(P)
}
```

TO DO: code is accurate but needs to be optimized to run faster - perhaps better memory usage as well

3 Results

We will compare the `prcomp` function to `pca.nipals` based on accuracy, speed, and functionality. All comparisons will be performed using matrices of size $n = 10^1, \ldots, 10^5 \times p = 10$. Below is the R code to generate the test data.

```
# seed
set.seed(13)

# generates (n by p) data matrix
generate.data <- function(n, p){
  X <- matrix(rnorm(n*p), nrow = n, ncol = p)
  return(X)
```

```
}


# test sizes
sizes <- c(10, 10^2, 10^3, 10^4, 10^5)
p <- 10
N <- length(sizes)
```

3.1 Accuracy

To assess the accuracy of the method, we will use the spectral norm

$$||\mathbf{A}||_2 = \sqrt{\lambda_{\max}(\mathbf{A}'\mathbf{A})} = \sigma_{\max}(\mathbf{A})$$

on the loading matrix difference

$$|\mathbf{P}_{\text{prcomp}}| - |\mathbf{P}_{\text{pca.nipals}}|$$

Note that the signs of each matrix element could be different so we must use apply the absolute value. Below is the R code to calculate the accuracy on the test data and the results.

```
norms <- rep(0, N)
for (i in 1:N) {
  X <- generate.data(sizes[i], p = p)
  X <- scale(X, center = TRUE, scale = TRUE)
  A <- prcomp(X, center = FALSE, scale = FALSE)$rotation
  B <- pca.nipals(X, center = FALSE, scale = FALSE)
  norms[i] <- norm(abs(A) - abs(B), "2")
}
```

TO DO: insert accuracy plot and interpret results once code is optimized

3.2 Speed

To assess the speed of the method, we take the average runtime of `prcomp` and compare it to the average runtime of NIPALS with 2, 6, and 10 principal components for each test matrix.

```
time.list <- list()
for (i in 1:N) {
  X <- generate.data(sizes[i], p = p)
  X <- scale(X, center = TRUE, scale = TRUE)
  times <- microbenchmark(
    prcomp(X, center = FALSE, scale = FALSE),
    pca.nipals(X, 2, center = FALSE, scale = FALSE),
    pca.nipals(X, 6, center = FALSE, scale = FALSE),
    pca.nipals(X, 10, center = FALSE, scale = FALSE),
    times= 50,
    unit='ms')
  time.list[[i]] <- summary(times)$mean
}
```

TO DO: insert speed plot and interpret results once code is optimized

3.3 Functionality

The main benefits of using SVD are it is mathematically sound, computationally efficient, simple, and numerically stable. However, you must calculate all principal components even if only a few are desired. This can be a burden in terms of computational cost and memory usage. On the other hand, NIPALS can calculate the desired number of principal components making it more efficient for large matrices when only a few components are needed and it can naturally handle missing data. Therefore, in most cases SVD is advised. If the data becomes too large or there is missing data, NIPALS can be useful.

4 Conclusions

TO DO: write conclusion once results are in

5 References

[1] Andrecut, M. 2009. *Parallel GPU implementation of iterative PCA algorithms.* https://doi-org.libaccess.lib.mcmaster.ca/10.1089/cmb.2008.0221

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

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