An Introduction to amanpg

Shixiang Chen, Justin Huang, Benjamin Jochem, Lingzhou Xue, Hui Zou

2022-10-02

Contents

- Introduction
  - Algorithm
  - Convergence
  - Pseudocode
- Installation
- Documentation
  - R Usage
  - Python Usage
  - Arguments
  - Values
- Quick Start
  - R Quick Start
  - Python Example
- References

Introduction

sparsepca and amanpg find sparse loadings in principal component analysis (PCA) via an alternating manifold proximal gradient method (A-ManPG). Seeking a sparse basis allows the leading principal components to be easier to interpret when modeling with high-dimensional data. PCA is modeled as a regularized regression problem under the elastic net constraints (linearized L1 and L2 norms) in order to induce sparsity. Due to the nonsmoothness and nonconvexity numerical difficulties, A-ManPG is implemented to guarantee convergence. The package provides a function for performing sparse PCA and a function for normalizing data.

The authors of A-ManPG are Shixiang Chen, Shiqian Ma, Lingzhou Xue, and Hui Zou. The Python and R packages are maintained by Justin Huang and Benjamin Jochem. A MATLAB implementation is maintained by Shixiang Chen.

Algorithm Description

The A-ManPG algorithm can be applied to solve the general manifold optimization problem

$$\min F(A, B) := H(A, B) + f(A) + g(B) \text{ subject to (s.t.) } A \in \mathcal{M}_1, B \in \mathcal{M}_2$$

where $H(A, B)$ is a smooth function of $A, B$ with a Lipschitz continuous gradient, $f(\cdot)$ and $g(\cdot)$ are lower semicontinuous (possibly nonsmooth) convex functions, and $\mathcal{M}_1, \mathcal{M}_2$ are two embedded submanifolds in the Euclidean space.

For sparse PCA, the following function definitions are used:

- $H(A, B) = \text{Tr}(B^TX^TXB) - 2\text{Tr}(A^TX^XB)$
- $f(A) \equiv 0$
• $g(B) = \lambda_2 \sum_{j=1}^{k} \|B_j\|_2^2 + \sum_{j=1}^{k} \lambda_{1,j} \|B_j\|_1$
• $\mathcal{M}_1 = \text{St}(p,k)$
• $\mathcal{M}_2 = \mathbb{R}^{p \times k}$

where $\mathbf{X}$ is the $n \times p$ data matrix or $n \times n$ covariance matrix, $\mathbf{A}$ is the scores and $\mathbf{B}$ is the loadings, $k$ is the rank of the matrices (in other words, how many principal components are desired), $\lambda_1$ is the L1 norm penalty and $\lambda_2$ is the L2 norm penalty. Both the L1 and L2 norm are used as elastic net regularization to impose sparseness within the loadings. Note that a different L1 norm penalty is used for every principal component, and the algorithm operates differently when the L2 norm penalty is set to a large constant ($\text{np.inf}$ or $\text{Inf}$).

The A-ManPG algorithm uses the following subproblems with an alternating updating scheme to solve sparse PCA, computed in a Gauss-Seidel manner for faster convergence.

\[ D_{\mathbf{A}}^k := \arg \min_{D_{\mathbf{A}}} \langle \nabla_{\mathbf{A}}^H (\mathbf{A}_k, \mathbf{B}_k), D_{\mathbf{A}} \rangle + f(\mathbf{A}^k + D_{\mathbf{A}}) + \frac{1}{2t_1} \|D_{\mathbf{A}}\|_F^2 \quad \text{s.t.} \quad D_{\mathbf{A}} \in T_{\mathbf{A}_k} \mathcal{M}_1 \]  

\[ D_{\mathbf{B}}^k := \arg \min_{D_{\mathbf{B}}} \langle \nabla_{\mathbf{B}}^H (\mathbf{A}_k^+, \mathbf{B}_k), D_{\mathbf{B}} \rangle + f(\mathbf{B}^k + D_{\mathbf{B}}) + \frac{1}{2t_2} \|D_{\mathbf{B}}\|_F^2 \quad \text{s.t.} \quad D_{\mathbf{B}} \in T_{\mathbf{B}_k} \mathcal{M}_2 \]  

where $\mathbf{A}_{k+1}$ is obtained via a retraction operation (in this case, polar decomposition), $t_1 \leq L_A$, and $t_2 \leq L_B$. $L_A$ and $L_B$ are the least upper bounds of the Lipschitz constants for $\nabla_{\mathbf{A}}^H (\mathbf{A}, \mathbf{B})$ and $\nabla_{\mathbf{B}}^H (\mathbf{A}, \mathbf{B})$, respectively. The subproblems are solved using an adaptive semismooth Newton method.

**Convergence** Let $\epsilon$ represent a tolerance level to detect convergence. An $\epsilon$-stationary point is defined as a point $(\mathbf{A}, \mathbf{B})$ with corresponding $D_{\mathbf{A}}$ and $D_{\mathbf{B}}$ that satisfy the following:

\[ \|D_{\mathbf{A}}/t_1\|_F^2 + \|D_{\mathbf{B}}/t_2\|_F^2 \leq \epsilon^2 \]  

The algorithm reaches an $\epsilon$-stationary point in at most

\[ \frac{2(F(\mathbf{A}_0, \mathbf{B}_0) - F^\star)}{((\gamma \bar{\alpha}_1 t_1 + \gamma \bar{\alpha}_2 t_2)\epsilon^2)} \]

iterations, where:

• $(\mathbf{A}_0, \mathbf{B}_0)$ are the initial values
• $F^\star$ is the lower bound of $F$, from the general manifold optimization problem
• $\bar{\alpha}_1$ and $\bar{\alpha}_2$ are positive constants

**Pseudocode** The following describes the algorithm used for solving the general manifold optimization problem using A-ManPG. For solving sparse PCA, the algorithm is implemented with the aforementioned definitions.

**Pseudocode**

Input initial point $(\mathbf{A}_0, \mathbf{B}_0)$ and necessary parameters for the required problem

for $i=0,1,...$ do
    Solve the first subproblem for $D_{\mathbf{A}}$
    Set $\alpha = 1$

    while $F(\text{Retr}(\alpha \cdot D_{\mathbf{A}}), \mathbf{B}) > F(\mathbf{A}, \mathbf{B}) - \alpha \cdot \text{norm}(D_{\mathbf{A}}) \cdot 2 / (2 \cdot t_1)$ do
        $\alpha = \gamma \cdot \alpha$
    end while

    Set $\mathbf{A} = \text{Retr}(\alpha \cdot D_{\mathbf{A}})$
Solve the second subproblem for Db
Set alpha = 1

while F(A, Retr(alpha * Db)) > F(A, B) - alpha * norm(Db)^2 / (2 * t2) do
    alpha = gamma * alpha
end while

Set B = Retr(alpha * Db)
end for

Return A as the scores and B as the sparse loadings

Installation
To install the R package, install \texttt{amanpg} directly from CRAN.
\begin{verbatim}
install.packages("amanpg")
\end{verbatim}

To install the Python package, use \texttt{pip} to obtain \texttt{sparsepca} from PyPI.
\begin{verbatim}
pip3 install sparsepca
\end{verbatim}

Documentation

\begin{verbatim}
spca.amanpg(z, lambda1, lambda2,
            f_palm = 1e5, x0 = NULL, y0 = NULL, k = 0, type = 0,
            gamma = 0.5, maxiter = 1e4, tol = 1e-5,
            normalize = TRUE, verbose = FALSE)
\end{verbatim}

R Usage

\begin{verbatim}
spca(z, lambda1, lambda2,
    x0=None, y0=None, k=0, gamma=0.5, type=0,
    maxiter=1e4, tol=1e-5, f_palm=1e5,
    normalize=True, verbose=False):
\end{verbatim}

Python Usage

Arguments

<table>
<thead>
<tr>
<th>Name</th>
<th>Python Type</th>
<th>R Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>numpy.ndarray</td>
<td>matrix</td>
<td>Either the data matrix or sample covariance matrix</td>
</tr>
<tr>
<td>lambda1</td>
<td>float list</td>
<td>numeric vector</td>
<td>List of parameters of length n for L1-norm penalty</td>
</tr>
<tr>
<td>lambda2</td>
<td>float or numpy.inf</td>
<td>numeric or Inf</td>
<td>L2-norm penalty term</td>
</tr>
<tr>
<td>Name</td>
<td>Python Type</td>
<td>R Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>x0</td>
<td>numpy.ndarray</td>
<td>matrix</td>
<td>Initial x-values for the gradient method, default value is the first n right singular vectors</td>
</tr>
<tr>
<td>y0</td>
<td>numpy.ndarray</td>
<td>matrix</td>
<td>Initial y-values for the gradient method, default value is the first n right singular vectors</td>
</tr>
<tr>
<td>k</td>
<td>int</td>
<td>int</td>
<td>Number of principal components desired, default is 0 (returns min(n-1, p) principal components)</td>
</tr>
<tr>
<td>gamma</td>
<td>float</td>
<td>numeric</td>
<td>Parameter to control how quickly the step size changes in each iteration, default is 0.5</td>
</tr>
<tr>
<td>type</td>
<td>int</td>
<td>int</td>
<td>If 0, b is expected to be a data matrix, and otherwise b is expected to be a covariance matrix; default is 0</td>
</tr>
<tr>
<td>maxiter</td>
<td>int</td>
<td>int</td>
<td>Maximum number of iterations allowed in the gradient method, default is 1e4</td>
</tr>
<tr>
<td>tol</td>
<td>float</td>
<td>numeric</td>
<td>Tolerance value required to indicate convergence (calculated as difference between iteration f-values), default is 1e-5</td>
</tr>
<tr>
<td>f_palm</td>
<td>float</td>
<td>numeric</td>
<td>Upper bound for the F-value to reach convergence, default is 1e5</td>
</tr>
<tr>
<td>normalize</td>
<td>bool</td>
<td>logical</td>
<td>Center and normalize rows to Euclidean length 1 if True, default is True</td>
</tr>
<tr>
<td>verbose</td>
<td>bool</td>
<td>logical</td>
<td>Function prints progress between iterations if True, default is False</td>
</tr>
</tbody>
</table>
Values  Python returns a dictionary with the following key-value pairs, while R returns a list with the following elements:

<table>
<thead>
<tr>
<th>Key</th>
<th>Python Value Type</th>
<th>R Value Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadings</td>
<td>numpy.ndarray</td>
<td>matrix</td>
<td>Loadings of the sparse principal components</td>
</tr>
<tr>
<td>f_manpg</td>
<td>float</td>
<td>numeric</td>
<td>Final F-value</td>
</tr>
<tr>
<td>x</td>
<td>numpy.ndarray</td>
<td>matirx</td>
<td>Corresponding ndarray in subproblem to the loadings</td>
</tr>
<tr>
<td>iter</td>
<td>int</td>
<td>numeric</td>
<td>Total number of iterations executed</td>
</tr>
<tr>
<td>sparsity</td>
<td>float</td>
<td>numeric</td>
<td>Number of sparse loadings (loadings == 0) divided by number of all loadings</td>
</tr>
<tr>
<td>time</td>
<td>float</td>
<td>numeric</td>
<td>Execution time in seconds</td>
</tr>
</tbody>
</table>

Quick Start
Consider the two examples below for running sparse PCA on randomly-generated data: one using a finite $\lambda_2$, and the other using a large constant $\lambda_2$.

R Example  As with other libraries, begin by loading `amanpg` in R.

```r
library(amanpg)
```

Before proceeding, it is helpful to determine a few parameters. Let the rank of the sparse loadings matrix be $k = 4$ (returning four principal components), the input data matrix be $n \times p$ where $n = 1000$ and $p = 500$, $\lambda_1$ be a $4 \times 1$ “matrix” where $\lambda_{i,1} = 0.1$, and $\lambda_2 = 1$.

```r
# parameter initialization
k <- 4
n <- 1000
p <- 500
lambda1 <- matrix(data=0.1, nrow=k, ncol=1)
lambda2 <- 1
```

For this example, the data matrix $z$ is randomly generated from the normal distribution. Although it should be centered to mean 0 and normalized to Euclidean length 1, the function will automatically preprocess the input matrix when `normalize=TRUE`.

```r
# data matrix generation
set.seed(10)
z <- matrix(rnorm(n * p), n, p)
```

```r
# only show a subset of the data matrix for brevity
knitr::kable(as.data.frame(z)[1:10,1:4])
```
Alternatively, the data can be normalized beforehand and the parameter is set to **FALSE** in the function call. However, this example won’t do so, but the output of normalize is displayed below.

```r
# see the effects of normalize()
knitr::kable(as.data.frame(normalize(z))[[1:10,]]
```

<table>
<thead>
<tr>
<th></th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0187462</td>
<td>1.0500137</td>
<td>-0.3078650</td>
<td>0.4605151</td>
<td></td>
</tr>
<tr>
<td>-0.1842525</td>
<td>0.2860926</td>
<td>0.7580856</td>
<td>0.2350253</td>
<td></td>
</tr>
<tr>
<td>-1.3713305</td>
<td>0.2405648</td>
<td>-0.5738634</td>
<td>0.6432573</td>
<td></td>
</tr>
<tr>
<td>-0.5991677</td>
<td>0.8327052</td>
<td>-0.9387445</td>
<td>0.9131981</td>
<td></td>
</tr>
<tr>
<td>0.2945451</td>
<td>-0.2229832</td>
<td>-0.0276993</td>
<td>0.9882860</td>
<td></td>
</tr>
<tr>
<td>0.3897943</td>
<td>0.2883442</td>
<td>-1.0662487</td>
<td>0.1127413</td>
<td></td>
</tr>
<tr>
<td>-1.2080762</td>
<td>-0.3403921</td>
<td>-1.3503703</td>
<td>-1.4900499</td>
<td></td>
</tr>
<tr>
<td>-0.3636760</td>
<td>1.0613346</td>
<td>0.0754557</td>
<td>-0.4432356</td>
<td></td>
</tr>
<tr>
<td>-1.6266727</td>
<td>-1.2090489</td>
<td>-0.9022730</td>
<td>1.3623441</td>
<td></td>
</tr>
<tr>
<td>-0.2564784</td>
<td>1.0524069</td>
<td>3.6667710</td>
<td>1.0452357</td>
<td></td>
</tr>
</tbody>
</table>

Now the function is called, passing through matrix `a`, `lambda1`, `lambda2`, and our desired rank `k`. The output is stored as a list in `fin_sprout`. Note that if a different initial point is desired, `x0` and `y0` should be modified, but the default value as the first `k` right singular vectors is sufficient for this example.

If further printout is desired, set `verbose=TRUE` for progress updates (time, difference for convergence, `F` value) per iteration.

```r
# function call
fin_sprout <- spca.amanpg(z, lambda1, lambda2, k=4)
print(paste(fin_sprout$iter, " iterations,", fin_sprout$sparsity, " sparsity," , fin_sprout$time))
```

```r
## [1] "280 iterations, 0.491 sparsity, 1.50452303886414"
```

The loadings can be viewed from `fin_sprout$loadings`. Note that many entries are set to zero as a result of the induced sparsity.

```r
# View loadings. Only first 10 rows for brevity
knitr::kable(as.data.frame(fin_sprout$loadings)[[1:10,]])
```

<table>
<thead>
<tr>
<th></th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0880974</td>
<td>0.0000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
<td></td>
</tr>
<tr>
<td>0.0000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
<td></td>
</tr>
<tr>
<td>0.0285272</td>
<td>0.0324180</td>
<td>0.0000000</td>
<td>0.0784247</td>
<td></td>
</tr>
</tbody>
</table>
The resulting scree plot (Figure 1) looks like this.

```r
pr.var <- (apply(fin_sprout$x, 2, sd))^2
pve <- pr.var / sum(pr.var)

par(mfrow=c(1,2))
plot(pve,
    xlab="Sparse PC",
    ylab="Proportion of Variance Explained",
    ylim=c(0,1),
    type="b")
plot(cumsum(pve),
    xlab="Sparse PC",
    ylab="Cumulative Proportion of Variance Explained",
    ylim=c(0,1),
    type="b")
```

The resulting biplot (Figure 2), with the zero loadings filtered out, can be obtained using the following:

```r
y_sub = apply(fin_sprout$loadings, 1, function(row) all(row != 0))
loadings = fin_sprout$loadings[y_sub, ]

par(mfrow=c(1,1))
bibplot(fin_sprout$x, loadings, xlab="PC 1", ylab="PC 2")
```

Now consider an alternative situation where we set $\lambda_2$ to a large constant $\text{Inf}$. The algorithm changes by directly retracting $B$ without using a while loop to determine an appropriate retraction step size and only iterating $A$.

```r
# infinite lambda2
inf_sprout <- spca.amanpg(z, lambda1, lambda2=Inf, k=4)
print(paste(inf_sprout$iter, " iterations, ", inf_sprout$sparsity, " sparsity, ", inf_sprout$time))
```

```
## [1] "344 iterations, 0.253 sparsity, 1.56649899482727"
```

```r
# extract loadings. Only first 10 rows for brevity
knitr::kable(as.data.frame(inf_sprout$loadings)[1:10,])
```

<table>
<thead>
<tr>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0612782</td>
<td>0.0070632</td>
<td>0.0148628</td>
<td>0.0000000</td>
</tr>
<tr>
<td>0.0000000</td>
<td>0.0080497</td>
<td>0.0000000</td>
<td>0.0000000</td>
</tr>
<tr>
<td>0.0000000</td>
<td>0.0670739</td>
<td>-0.0796881</td>
<td>0.0616913</td>
</tr>
<tr>
<td>0.0000000</td>
<td>0.0050157</td>
<td>0.1115940</td>
<td></td>
</tr>
<tr>
<td>0.0674123</td>
<td>-0.0194809</td>
<td>0.0490157</td>
<td>-0.115940</td>
</tr>
<tr>
<td>0.0000000</td>
<td>0.194809</td>
<td>0.0490157</td>
<td>-0.125086</td>
</tr>
<tr>
<td>0.0000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
<td>-0.0207933</td>
</tr>
<tr>
<td>0.0000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
</tr>
<tr>
<td>0.0000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
</tr>
</tbody>
</table>
We obtain the scree plot (Figure 3) and the biplot (Figure 4) using the same method.

```r
pr.var <- (apply(inf_sprout$x, 2, sd)^2)
pve <- pr.var / sum(pr.var)

par(mfrow=c(1,2))
plot(pve,
    xlab="Sparse PC",
    ylab="Proportion of Variance Explained",
    ylim=c(0,1),
    type="b")
plot(cumsum(pve),
    xlab="Sparse PC",
    ylab="Cumulative Proportion of Variance Explained",
    ylim=c(0,1),
    type="b")
```

For data with high dimensionality, the biplot is still much harder to read with a lower sparsity value.

```r
y_sub = apply(inf_sprout$loadings, 1,
    function (row) all(row != 0))
loadings = inf_sprout$loadings[y_sub,]

par(mfrow=c(1,1))
biplo(t(inf_sprout$x, loadings, xlab="PC 1", ylab="PC 2")
```

**Python Example**  Note that the Python package depends on numpy.

The following example accomplishes the same situation (down to the same randomly-generated data) but in Python.

```python
import numpy as np
from sparsepca import spca

k = 4  # rank
p = 500  # dimensions
n = 1000  # sample size
lambda1 = 0.1 * np.ones((k, 1))
lambda2 = 1

np.random.seed(10)
z = np.random.normal(0, 1, size=(n, p))  # generate random normal 1000x500 matrix

fin_sprout = spca(z, lambda1, lambda2, k=k)
print(f"Finite: {fin_sprout['iter']} iterations with final value
    {fin_sprout['f_mnpg']}, sparsity {fin_sprout['sparsity']},
    timediff {fin_sprout['time']}.")

fin_sprout['loadings']
```
Figure 1: Scree plots for the finite lambda case.
Figure 2: Biplot for the finite lambda case.
Figure 3: Scree plot for lambda=inf case.
Figure 4: Biplot for lambda=inf case. Observe that with lower sparsity in the loadings and high-dimensional data, the biplot becomes less readable.
inf_sprout = spca(z, lambda1, np.inf, k=k)
print(f"Infinite: {inf_sprout['iter']} iterations with final value
  {inf_sprout['f_manpg']}, sparsity {inf_sprout['sparsity']},
  timediff {inf_sprout['time']}.")

inf_sprout['loadings']

References


13