THE UNIVERSITY OF TEXAS AT AUSTIN
Department of Statistics and Data Sciences
College of Natural Sciences

Advanced Predictive Models for Complex Data

Lecture 4: PCA

Purnamrita Sarkar
Department of Statistics and Data Science
The University of Texas at Austin

https://psarkar.github.io/teaching

Principal Component Analysis

- Goal: Find the direction of the most variance.
- Say X is the data matrix
- The average is $\bar{\boldsymbol{x}} = \frac{\sum_{i=1}^{n} \boldsymbol{x}_i}{n}$
- Let $\tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i - \bar{\boldsymbol{x}}$

Principal Component Analysis

- Goal: Find the direction of the most variance.
- Say X is the data matrix
- The average is $\bar{\boldsymbol{x}} = \frac{\sum_{i=1}^{n} \boldsymbol{x}_i}{n}$
- Let $\tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i - \bar{\boldsymbol{x}}$
- The sample variance of $(\tilde{\boldsymbol{x}}_1, \ldots, \tilde{\boldsymbol{x}}_n)$ *along a direction w* is give by:

$$\frac{1}{n}\sum_{i=1}^{n}(\tilde{\boldsymbol{x}}_i^T w)^2$$

- What is the sample variance of $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ *along a direction $\boldsymbol{w}$*?

Principal Component Analysis

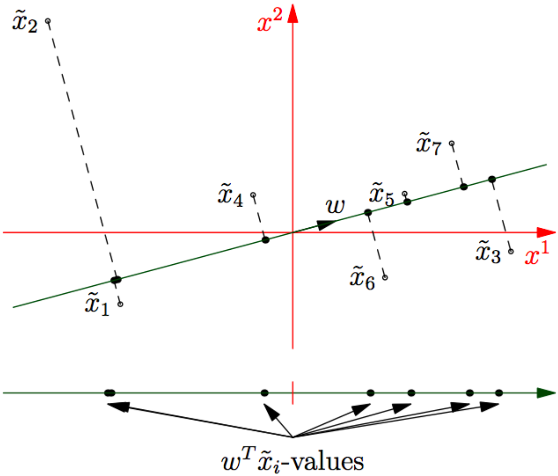


Figure 1: Picture courtesy: davidrosenberg.github.io

First component

- So the first PC direction is:

$$\boldsymbol{w}_1 = \arg\max_{\|\boldsymbol{w}\|=1} \frac{1}{n} \sum_{i=1}^{n} (\tilde{\boldsymbol{x}}_i^T \boldsymbol{w})^2$$

- And the first PC component of $\tilde{\boldsymbol{x}}_i$ is $\tilde{\boldsymbol{x}}_i^T \boldsymbol{w}_1$

First component

- So the k^{th} PC direction is:

$$\boldsymbol{w}_k = \arg \max_{\substack{\|\boldsymbol{w}\|=1 \\ \boldsymbol{w} \perp \boldsymbol{w}_1, \ldots, \boldsymbol{w}_{k-1}}} \frac{1}{n} \sum_{i=1}^{n} (\tilde{\boldsymbol{x}}_i^T \boldsymbol{w})^2$$

- And the k^{th} PC component of $\tilde{\boldsymbol{x}}_i$ is $\tilde{\boldsymbol{x}}_i^T \boldsymbol{w}_k$
- Note that $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_k$ form an orthogonal basis.

Simple algorithm

- Let W is a matrix with $\boldsymbol{w}_k$ along its columns
- $\tilde{X}W$ gives a **low dimensional** representation of $\tilde{X}$

Simple algorithm

- Let W is a matrix with $\boldsymbol{w}_k$ along its columns
- $\tilde{X}W$ gives a **low dimensional** representation of $\tilde{X}$
- We can frame the optimization problem also as

$$\boldsymbol{w}_1 = \arg\max_{\|\boldsymbol{w}\|=1} \boldsymbol{w}^T \tilde{X}^T \tilde{X} \boldsymbol{w}$$

Simple algorithm

- Let W is a matrix with $\boldsymbol{w}_k$ along its columns
- $\tilde{X}W$ gives a **low dimensional** representation of $\tilde{X}$
- We can frame the optimization problem also as

$$\boldsymbol{w}_1 = \arg\max_{\|\boldsymbol{w}\|=1} \boldsymbol{w}^T \tilde{X}^T \tilde{X} \boldsymbol{w}$$

- This is the first eigenvector of $S = \tilde{X}^T \tilde{X}$.

Eigenvector and eigenvalues

- Any square symmetrix matrix S has real eigenvalues
- The i^{th} eigenvalue,vector pair satisfy $S\boldsymbol{w}_i = \lambda_i \boldsymbol{w}_i$
- The eigenvectors are orthogonal to each other, and normalized to have length 1.

Eigenvector and eigenvalues

- Any square symmetrix matrix S has real eigenvalues
- The i^{th} eigenvalue,vector pair satisfy $S\boldsymbol{w}_i = \lambda_i \boldsymbol{w}_i$
- The eigenvectors are orthogonal to each other, and normalized to have length 1.
- In matrix terms, we can write:

$$S = U \Sigma U^T, \text{ where}$$

  - columns of U are the orgonal eigenvectors, and
  - Σ is a diagonal matrix with eigenvalues on the diagonal
- The larger the magnitude of the eigenvalue, more important the eigenvector

Back to PCA: Simple algorithm

- Let W is a matrix with $\boldsymbol{w}_k$ along its columns
- $\tilde{X}W$ gives a **low dimensional** representation of $\tilde{X}$

Back to PCA: Simple algorithm

- Let W is a matrix with $\boldsymbol{w}_k$ along its columns
- $\tilde{X}W$ gives a **low dimensional** representation of $\tilde{X}$
- We can frame the optimization problem also as

$$\boldsymbol{w}_1 = \arg\max_{\|\boldsymbol{w}\|=1} \boldsymbol{w}^T \tilde{X}^T \tilde{X} \boldsymbol{w}$$

Back to PCA: Simple algorithm

- Let W is a matrix with $\boldsymbol{w}_k$ along its columns
- $\tilde{X}W$ gives a **low dimensional** representation of $\tilde{X}$
- We can frame the optimization problem also as

$$\boldsymbol{w}_1 = \arg\max_{\|\boldsymbol{w}\|=1} \boldsymbol{w}^T \tilde{X}^T \tilde{X} \boldsymbol{w}$$

- This is the first eigenvector of $S = \tilde{X}^T \tilde{X}$

Back to PCA: Simple algorithm

- Let W is a matrix with $\boldsymbol{w}_k$ along its columns
- $\tilde{X}W$ gives a **low dimensional** representation of $\tilde{X}$
- We can frame the optimization problem also as

$$\boldsymbol{w}_1 = \arg\max_{\|\boldsymbol{w}\|=1} \boldsymbol{w}^T \tilde{X}^T \tilde{X} \boldsymbol{w}$$

- This is the first eigenvector of $S = \tilde{X}^T \tilde{X}$
- What is S?

Back to PCA: Simple algorithm

- Let W is a matrix with $\boldsymbol{w}_k$ along its columns
- $\tilde{X}W$ gives a **low dimensional** representation of $\tilde{X}$
- We can frame the optimization problem also as

$$\boldsymbol{w}_1 = \arg\max_{\|\boldsymbol{w}\|=1} \boldsymbol{w}^T \tilde{X}^T \tilde{X} \boldsymbol{w}$$

- This is the first eigenvector of $S = \tilde{X}^T \tilde{X}$
- What is S?
- Its the scalar multiple of the sample covariance matrix

$$\hat{\Sigma} = \frac{1}{n} \sum_i \tilde{\boldsymbol{x}}_i \tilde{\boldsymbol{x}}_i^T = \frac{S}{n}$$

Back to PCA: Simple algorithm

- Let W is a matrix with $\boldsymbol{w}_k$ along its columns
- $\tilde{X}W$ gives a **low dimensional** representation of $\tilde{X}$
- We can frame the optimization problem also as

$$\boldsymbol{w}_1 = \arg\max_{\|\boldsymbol{w}\|=1} \boldsymbol{w}^T\tilde{X}^T\tilde{X}\boldsymbol{w}$$

- This is the first eigenvector of $S = \tilde{X}^T\tilde{X}$
- What is S?
- Its the scalar multiple of the sample covariance matrix

$$\hat{\Sigma} = \frac{1}{n}\sum_i \tilde{\boldsymbol{x}}_i\tilde{\boldsymbol{x}}_i^T = \frac{S}{n}$$

- So, all you have to do is to calculate eigenvectors of the covariance matrix.

Back to PCA: Simple algorithm

- So, all you have to do is to calculate eigenvectors of the covariance matrix.
- But, do I even need to do that?

Back to PCA: Simple algorithm

- So, all you have to do is to calculate eigenvectors of the covariance matrix.
- But, do I even need to do that?
- The right singular vectors of $\tilde{X}$ is just fine.

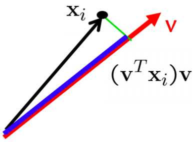
Back to PCA: Simple algorithm

- So, all you have to do is to calculate eigenvectors of the covariance matrix.
- But, do I even need to do that?
- The right singular vectors of $\tilde{X}$ is just fine.
- How many PC's? (more of a dissertaiton question)

Singular value decomposition

- The columns of U are orthogonal eigenvectos of AA^T
- The columns of V are orthogonal eigenvectos of A^TA
- A^TA and AA^T have the same eigenvalues, which are all positive, and squares of Σ_{ii}.

Second interpretation



- Minimum reconstruction error:

$$(\boldsymbol{x}_i - (\boldsymbol{x}_i^T\boldsymbol{w})\boldsymbol{w})^T(\boldsymbol{x}_i - (\boldsymbol{x}_i^T\boldsymbol{w})\boldsymbol{w}) = \boldsymbol{x}_i^T\boldsymbol{x}_i - (\boldsymbol{x}_i^T\boldsymbol{w})^2$$

- So, the first PC direction gives the direction projecting on which has the **minimum reconstruction error**.

Lets do some coding

- We will make a covariance matrix and generate independent multivariate gaussian random variables

```
d=1000
Sigma=np.zeros([d,d])
Sigma[0:200,0:200]=.3;
np.fill_diagonal(Sigma,1)
```

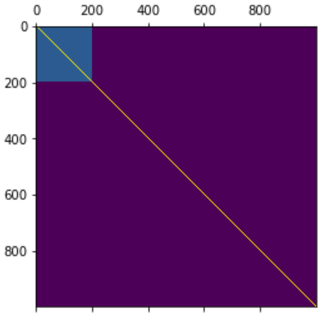
Lets do some coding

- We will make a covariance matrix and generate independent multivariate gaussian random variables

```
d=1000
Sigma=np.zeros([d,d])
Sigma[0:200,0:200]=.3;
np.fill_diagonal(Sigma,1)
```

```
plt.matshow(Sigma)
```

```
<matplotlib.image.AxesImage at 0x188c373d0>
```



Lets do some coding

- Now lets compute eigenvectors of the covariance matrix.

```
X=np.random.multivariate_normal(np.zeros(d),Sigma,5000)
```

```
S=np.cov(np.transpose(X))
u,s,vt=svd(S)
```

Lets do some coding

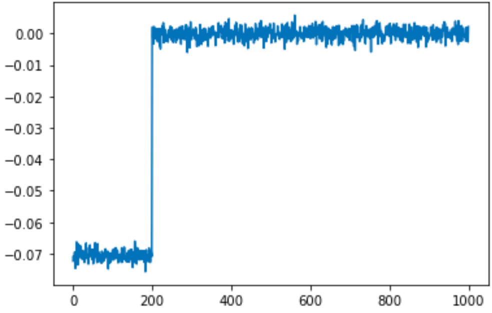
- Now lets compute eigenvectors of the covariance matrix.

```
X=np.random.multivariate_normal(np.zeros(d),Sigma,5000)
```

```
S=np.cov(np.transpose(X))
u,s,vt=svd(S)
```

```
plot(u[:,0])
```

```
[<matplotlib.lines.Line2D at 0x167e5ae50>]
```



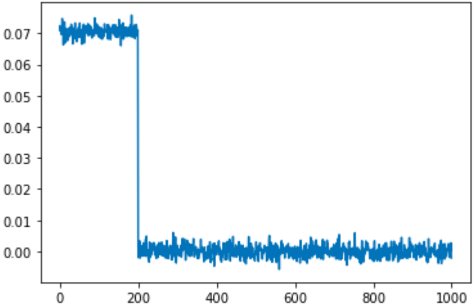
Lets do some coding

- Now lets do SVD on the data matrix X.

```
u,s,vt=svd(X)
```

```
plot(vt[0,:])
```

```
[<matplotlib.lines.Line2D at 0x1680810a0>]
```



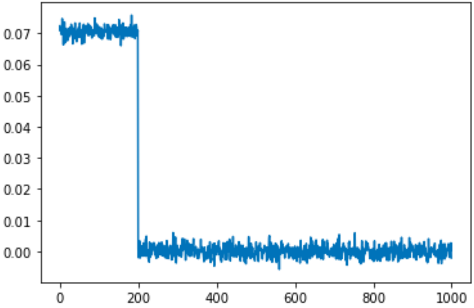
Lets do some coding

- Now lets do SVD on the data matrix X.

```
u,s,vt=svd(X)
```

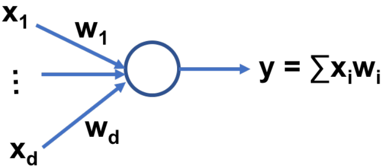
```
plot(vt[0,:])
```

```
[<matplotlib.lines.Line2D at 0x1680810a0>]
```



Online PCA - Oja's algorithm

- Erikki Oja wrote a seminar paper in 1982 about a simple neural network model.
- He was inspired by the Hebbian principle (1949, "The organization of behavior", Donald Hebb) which claims that the synaptic energy increases from presynaptic cells stimulating post-synaptic cells.



Online PCA - Oja's algorithm

- For each data-point, you do:

$$w_{t+1} \leftarrow w_t + \eta_t (x_t^T w_t) x_t$$

$$w_{t+1} \leftarrow w_{t+1} / \|w_{t+1}\|$$

Online PCA - Oja's algorithm

- For each data-point, you do:

$$w_{t+1} \leftarrow w_t + \eta_t (x_t^T w_t) x_t$$

$$w_{t+1} \leftarrow w_{t+1} / \|w_{t+1}\|$$

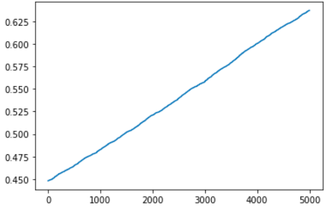
- Note that here, you do not need to construct the covariance matrix explicitly, which is extremely useful, when d is much larger than n, i.e. in high dimensional settings.
- Step size η_t can be set as $c \log n/n$ or $\eta_t \propto 1/t$
- Sharp error bounds show that the final solution converges to the principal component and the error has weak dependence on dimensionality d

Lets do some coding

- Now lets do Oja's algorithm for X.
- Set $\eta = 0.001 \log n / n$

```
plot(dot_prod)
```

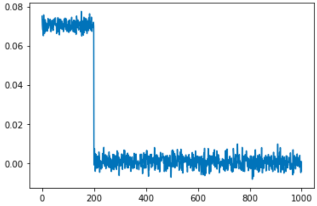
```
[<matplotlib.lines.Line2D at 0x165092100>]
```



Dot product with truth

```
plot(z)
```

```
[<matplotlib.lines.Line2D at 0x1679e1100>]
```



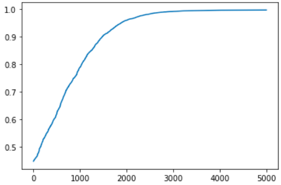
final vector

Lets do some coding

- Now lets do Oja's algorithm for X.
- Set $\eta = 0.01 \log n/n$

```
plot(dot_prod)
```

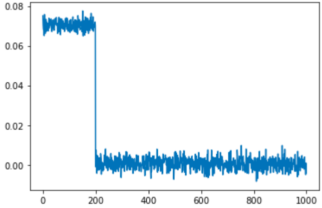
```
[<matplotlib.lines.Line2D at 0x168107130>]
```



Dot product with truth

```
plot(z)
```

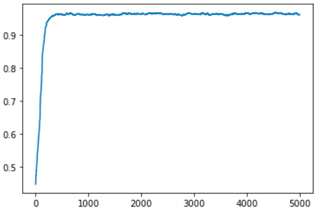
```
[<matplotlib.lines.Line2D at 0x1679e1100>]
```



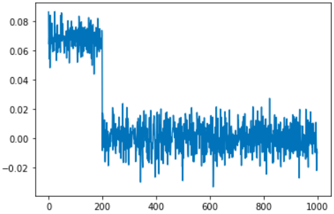
final vector

Lets do some coding

- Now lets do Oja's algorithm for X.
- Set $\eta = 0.1 \log n/n$



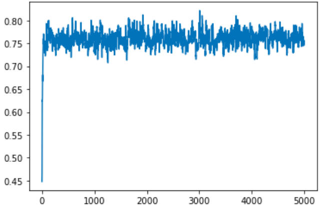
Dot product with truth



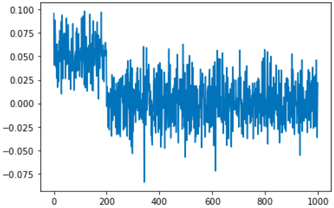
final vector

Lets do some coding

- Now lets do Oja's algorithm for X.
- Set $\eta = 1 \log n/n$



Dot product with truth



final vector