

# Recommender Systems In Context: Singular Value Decomposition

Derek G Nokes

July 7, 2019

# Introduction

- ▶ Matrix factorizations are ubiquitous in modern data science

# Introduction

- ▶ Matrix factorizations are ubiquitous in modern data science
- ▶ Our need for tools that can map a high dimensional space into a lower dimensional space of latent factors grows as the amount of data we collect increases

# Introduction

- ▶ Matrix factorizations are ubiquitous in modern data science
- ▶ Our need for tools that can map a high dimensional space into a lower dimensional space of latent factors grows as the amount of data we collect increases
- ▶ Provide a brief overview of singular value decomposition (SVD)

# Introduction

- ▶ Matrix factorizations are ubiquitous in modern data science
- ▶ Our need for tools that can map a high dimensional space into a lower dimensional space of latent factors grows as the amount of data we collect increases
- ▶ Provide a brief overview of singular value decomposition (SVD)
- ▶ Walk through simple power method implementation for SVD on a dense matrix

## Theory: Singular Value Decomposition (i.e., Spectral Decomposition)

The singular value decomposition (SVD) of a matrix  $A$  with  $m$  rows and  $n$  columns is

$$A = U\Sigma V^T = \sum_i \sigma_i u_i \circ v_i^T$$

where

- ▶  $A = m \times n$  input data matrix ( $A \in \mathbb{R}^{m \times n}$ )

## Theory: Singular Value Decomposition (i.e., Spectral Decomposition)

The singular value decomposition (SVD) of a matrix  $A$  with  $m$  rows and  $n$  columns is

$$A = U\Sigma V^T = \sum_i \sigma_i u_i \circ v_i^T$$

where

- ▶  $A = m \times n$  input data matrix ( $A \in \mathbb{R}^{m \times n}$ )
- ▶  $r = \text{rank of matrix } A$  (i.e., the number of linearly independent columns of  $A$ ).

# Theory: Singular Value Decomposition (i.e., Spectral Decomposition)

The singular value decomposition (SVD) of a matrix  $A$  with  $m$  rows and  $n$  columns is

$$A = U\Sigma V^T = \sum_i \sigma_i u_i \circ v_i^T$$

where

- ▶  $A = m \times n$  input data matrix ( $A \in \mathbb{R}^{m \times n}$ )
- ▶  $r = \text{rank of matrix } A$  (i.e., the number of linearly independent columns of  $A$ ).
- ▶  $U = m \times r$  matrix of left singular vectors



# Theory: Singular Value Decomposition (i.e., Spectral Decomposition)

The singular value decomposition (SVD) of a matrix  $A$  with  $m$  rows and  $n$  columns is

$$A = U\Sigma V^T = \sum_i \sigma_i u_i \circ v_i^T$$

where

- ▶  $A = m \times n$  input data matrix ( $A \in \mathbb{R}^{m \times n}$ )
- ▶  $r = \text{rank of matrix } A$  (i.e., the number of linearly independent columns of  $A$ ).
- ▶  $U = m \times r$  matrix of left singular vectors
- ▶  $\Sigma = r \times r$  diagonal matrix

## Theory: Singular Value Decomposition (i.e., Spectral Decomposition)

The singular value decomposition (SVD) of a matrix  $A$  with  $m$  rows and  $n$  columns is

$$A = U\Sigma V^T = \sum_i \sigma_i u_i \circ v_i^T$$

where

- ▶  $A = m \times n$  input data matrix ( $A \in \mathbb{R}^{m \times n}$ )
- ▶  $r = \text{rank of matrix } A$  (i.e., the number of linearly independent columns of  $A$ ).
- ▶  $U = m \times r$  matrix of left singular vectors
- ▶  $\Sigma = r \times r$  diagonal matrix
- ▶  $V = n \times r$  matrix of right singular vectors. The superscript  $T$  indicates the transpose of matrix  $V$ .

## Singular Value Decomposition (SVD) - Visualized

$$\boxed{\mathbf{A}} = \boxed{\mathbf{U}} \times \boxed{\Sigma} \times \boxed{\mathbf{V}^T}$$

$(m \times n)$        $(m \times r)$        $(r \times r)$        $(r \times n)$

- If  $A$  has rank  $r$ , then  $U$  is  $m \times r$ ,  $\Sigma$  is an  $r \times r$  diagonal matrix with non-negative, non-increasing entries (i.e., sorted from largest to smallest),  $V$  is  $n \times r$  and  $V^T$  is  $r \times n$ .

## Singular Value Decomposition (SVD) - Visualized

$$\boxed{\mathbf{A}} = \boxed{\mathbf{U}} \times \boxed{\Sigma} \times \boxed{\mathbf{V}^T}$$

$(m \times n)$        $(m \times r)$        $(r \times r)$        $(r \times n)$

- ▶ If  $A$  has rank  $r$ , then  $U$  is  $m \times r$ ,  $\Sigma$  is an  $r \times r$  diagonal matrix with non-negative, non-increasing entries (i.e., sorted from largest to smallest),  $V$  is  $n \times r$  and  $V^T$  is  $r \times n$ .
- ▶  $U$  and  $V$  are *column orthonormal*

# Singular Value Decomposition (SVD) - Visualized

$$\boxed{\mathbf{A}} = \boxed{\mathbf{U}} \times \boxed{\Sigma} \times \boxed{\mathbf{V}^T}$$

$(m \times n) \qquad (m \times r) \qquad (r \times r) \qquad (r \times n)$

- ▶ If  $A$  has rank  $r$ , then  $U$  is  $m \times r$ ,  $\Sigma$  is an  $r \times r$  diagonal matrix with non-negative, non-increasing entries (i.e., sorted from largest to smallest),  $V$  is  $n \times r$  and  $V^T$  is  $r \times n$ .
- ▶  $U$  and  $V$  are *column orthonormal*
- ▶ By *column orthonormal*, we mean that  $U^T U = I$  and  $V^T V = I$ , where  $I$  is an identity matrix

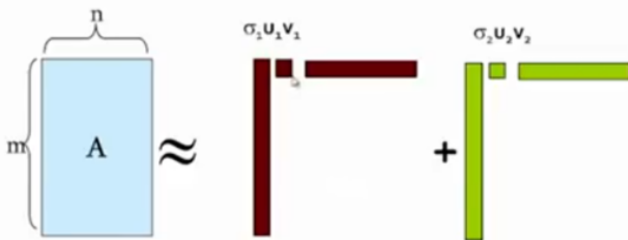
# Singular Value Decomposition (SVD) - Visualized

$$\boxed{\mathbf{A}} = \boxed{\mathbf{U}} \times \boxed{\Sigma} \times \boxed{\mathbf{V}^T}$$

$(m \times n)$        $(m \times r)$        $(r \times r)$        $(r \times n)$

- ▶ If  $A$  has rank  $r$ , then  $U$  is  $m \times r$ ,  $\Sigma$  is an  $r \times r$  diagonal matrix with non-negative, non-increasing entries (i.e., sorted from largest to smallest),  $V$  is  $n \times r$  and  $V^T$  is  $r \times n$ .
- ▶  $U$  and  $V$  are *column orthonormal*
- ▶ By *column orthonormal*, we mean that  $U^T U = I$  and  $V^T V = I$ , where  $I$  is an identity matrix
- ▶ Two vectors  $u$  and  $v$  whose dot product is  $u \cdot v = 0$  (i.e., the vectors are perpendicular) are said to be orthogonal

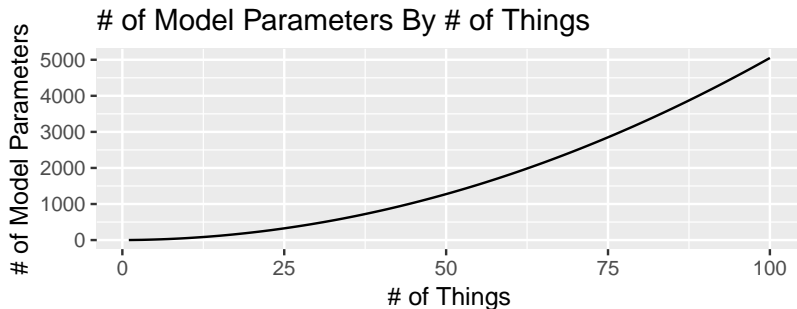
$$A \approx U \Sigma V^T = \sum_i \sigma_i \mathbf{u}_i \circ \mathbf{v}_i^T$$



$\sigma_i$  ... scalar  
 $\mathbf{u}_i$  ... vector  
 $\mathbf{v}_i$  ... vector

- This is key to the algorithm we will implement later

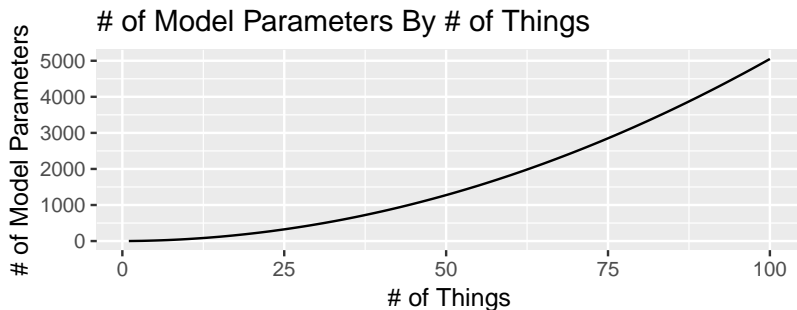
# Why Does Orthogonal Matter - Too Many Moving Parts



- ▶ Number of independent parameters to be estimated grows with square of number of things

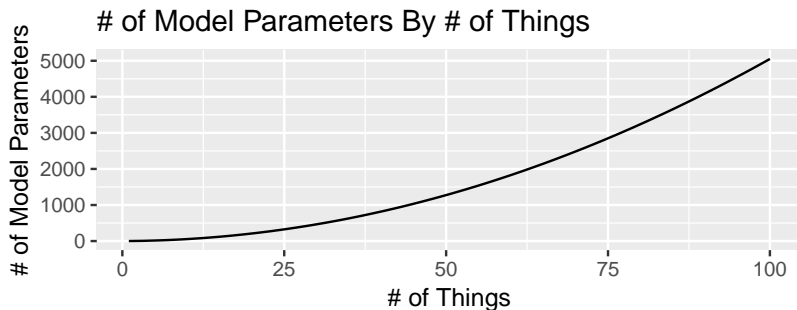


# Why Does Orthogonal Matter - Too Many Moving Parts



- ▶ Number of independent parameters to be estimated grows with square of number of things
- ▶ Number of data points available to estimate a covariance matrix grows only linearly with number of things

# Why Does Orthogonal Matter - Too Many Moving Parts



- ▶ Number of independent parameters to be estimated grows with square of number of things
- ▶ Number of data points available to estimate a covariance matrix grows only linearly with number of things
- ▶ Larger the number of things, the more data we typically need to estimate reliably

# Importance of Normalization

- Magnitudes of attribute values comprising input data matrix should be scaled into roughly same range [Skillicorn 2007]
- Want to ensure that properties are compared in a way consistent with real world
  - ▶ i.e., if measure some attributes in inches and others in miles, magnitude of numbers cannot be compared without rescaling
- May be more appropriate to normalize by keeping zero entries fixed when a matrix is sparse

# Importance of Normalization

- Magnitudes of attribute values comprising input data matrix should be scaled into roughly same range [Skillicorn 2007]
- Want to ensure that properties are compared in a way consistent with real world
  - ▶ i.e., if measure some attributes in inches and others in miles, magnitude of numbers cannot be compared without rescaling
  - ▶ Common to divide entries in each column of a dense matrix by standard deviation of that column
- May be more appropriate to normalize by keeping zero entries fixed when a matrix is sparse

# Importance of Normalization

- Magnitudes of attribute values comprising input data matrix should be scaled into roughly same range [Skillicorn 2007]
- Want to ensure that properties are compared in a way consistent with real world
  - ▶ i.e., if measure some attributes in inches and others in miles, magnitude of numbers cannot be compared without rescaling
  - ▶ Common to divide entries in each column of a dense matrix by standard deviation of that column
- May be more appropriate to normalize by keeping zero entries fixed when a matrix is sparse
  - ▶ Mean of non-zero entries is typically subtracted from non-zero entries so that they become zero-centered

# Importance of Normalization

- Magnitudes of attribute values comprising input data matrix should be scaled into roughly same range [Skillicorn 2007]
- Want to ensure that properties are compared in a way consistent with real world
  - ▶ i.e., if measure some attributes in inches and others in miles, magnitude of numbers cannot be compared without rescaling
  - ▶ Common to divide entries in each column of a dense matrix by standard deviation of that column
- May be more appropriate to normalize by keeping zero entries fixed when a matrix is sparse
  - ▶ Mean of non-zero entries is typically subtracted from non-zero entries so that they become zero-centered
  - ▶ Only non-zero entries are divided by standard deviation of the column mean

## Importance of Normalization - Continued

- Must carefully consider form of normalization because selecting correct approach can significantly improve predictive performance

# Implementation

- ▶ Modern libraries implementing SVD are based on extensive research using methods more complex than the method we are going to walk through here [GolubAndLoan 2013]



# Implementation

- ▶ Modern libraries implementing SVD are based on extensive research using methods more complex than the method we are going to walk through here [GolubAndLoan 2013]
- ▶ Use simplest available approach for illustrative purposes

# Implementation

- ▶ Modern libraries implementing SVD are based on extensive research using methods more complex than the method we are going to walk through here [GolubAndLoan 2013]
- ▶ Use simplest available approach for illustrative purposes
- ▶ There are more efficient and more numerically stable methods available [GolubAndLoan 2013]

## Sample Data

```
# create sample matrix
A = np.array([
    [4, 1, 1],
    [2, 5, 3],
    [1, 2, 1],
    [4, 5, 5],
    [3, 5, 2],
    [2, 4, 2],
    [5, 3, 1],
    [2, 2, 5],
], dtype='float64')
# determine n rows and m columns
m,n = A.shape
print("A is a "+str(m)+" x "+str(n)+" matrix")

## A is a 8, x 3 matrix
```

# Random Unit Vector

First, we define a function to create a random unit vector:

```
def randomUnitVector(n):  
    unnormalized = [random.normalvariate(0,  
        1) for _ in range(n)]  
    xNorm = sqrt(sum(x * x for x in unnormalized))  
    return [x / xNorm for x in unnormalized]
```

Create a random unit vector:

```
# build random unit vector  
x = randomUnitVector(min(m,n))  
print(np.round(x,4))
```

```
## [ 0.8037 -0.4859  0.3435]
```

# One-Dimensional SVD

```
def svdPowerMethod_1d(A, epsilon=1e-10):  
    m, n = A.shape  
    x = randomUnitVector(min(m,n))  
    lastV = None  
    V = x  
    if m > n:  
        B = np.dot(A.T, A)  
    else:  
        B = np.dot(A, A.T)  
    while True:  
        lastV = V  
        V = np.dot(B, lastV)  
        V = V / norm(V)  
        if abs(np.dot(V, lastV)) > 1 - epsilon:  
            return V
```

```
def svdPowerMethodRow(A, k=None, epsilon=1e-10):  
    A = np.array(A, dtype=float)  
    m, n = A.shape  
    svdEstimate = []  
    if k is None:  
        k = min(m, n)  
    for i in range(k):  
        matrixFor1D = A.copy()  
        for singularValue, u, v in svdEstimate[:i]:  
            matrixFor1D -= singularValue * np.outer(u, v)  
        v = svdPowerMethod_1d(matrixFor1D, epsilon=epsilon)  
        u_unnormalized = np.dot(A, v)  
        sigma = norm(u_unnormalized)  
        u = u_unnormalized / sigma  
        svdEstimate.append((sigma, u, v))  
    output = [np.array(x) for x in zip(*svdEstimate)]  
    singularValues, us, vs = output  
    return singularValues, us.T, vs
```

```
def svdPowerMethodColumn(A, k=None, epsilon=1e-10):
    A = np.array(A, dtype=float)
    m, n = A.shape
    svdEstimate = []
    if k is None:
        k = min(m, n)
    for i in range(k):
        matrixFor1D = A.copy()
        for singularValue, u, v in svdEstimate[:i]:
            matrixFor1D -= singularValue * np.outer(u, v)
        u = svdPowerMethod_1d(matrixFor1D, epsilon=epsilon)
        v_unnormalized = np.dot(A.T, u)
        sigma = norm(v_unnormalized)
        v = v_unnormalized / sigma
        svdEstimate.append((sigma, u, v))
    output = [np.array(x) for x in zip(*svdEstimate)]
    singularValues, us, vs = output
    return singularValues, us.T, vs
```

## SVD of a Matrix, $A$ , Using the Power Method

```
def svdPowerMethod(A, k=None, epsilon=1e-10):  
    A = np.array(A, dtype=float)  
    m, n = A.shape  
    if m > n:  
        s, uT, vs=svdPowerMethodRow(A,  
            k, epsilon=1e-10)  
    else:  
        s, uT, vs=svdPowerMethodColumn(A,  
            k, epsilon=1e-10)  
    return s, uT, vs
```



## Run `svdPowerMethod()`

```
# compute SVD  
singularValues,U,V = svdPowerMethod(A)
```

The 8 by 3 matrix of left singular vectors  $U$ :

```
print(U)  
  
## [[ 0.22155208 -0.52086662 -0.39333697]  
##   [ 0.39458523  0.23924155  0.35445363]  
##   [ 0.15830232  0.03055162  0.15299691]  
##   [ 0.53347447  0.19168664 -0.19949776]  
##   [ 0.39692635 -0.08648347  0.41053093]  
##   [ 0.31660463  0.06110323  0.30599382]  
##   [ 0.34630265 -0.64128879 -0.07381356]  
##   [ 0.32840218  0.45969551 -0.62355828]]
```

The 3 by 3 diagonal singular value matrix  $\Sigma$ :

```
print(np.diag(singularValues))
```

```
## [[ 15.09626916  0.          0.          ]  
##   [  0.          4.30056855  0.          ]  
##   [  0.          0.          3.40701739]]
```

The 3 by 3 matrix of right singular vectors  $V$ :

```
print(V)
```

```
## [[ 0.54184843  0.67070986  0.50650623]  
##   [-0.75152966  0.11682294  0.6492731 ]  
##   [-0.37630233  0.73246207 -0.56735868]]
```

We reconstitute the matrix  $A$ :

```
# reconstitute matrix A  
Sigma = np.diag(singularValues)  
# reconstitute matrix A  
AA=np.dot(U, np.dot(Sigma, V))
```

```
print(AA)
```

```
## [[ 4.  1.  1.]  
##  [ 2.  5.  3.]  
##  [ 1.  2.  1.]  
##  [ 4.  5.  5.]  
##  [ 3.  5.  2.]  
##  [ 2.  4.  2.]  
##  [ 5.  3.  1.]  
##  [ 2.  2.  5.]]
```

We can see that the original and reconstituted matrices are the same to 10 decimal places:

```
print(np.round(A - AA, decimals=nDigits))
```

[illegible]

- ▶ Have shown results from full-rank decomposition

- ▶ Have shown results from full-rank decomposition
- ▶ In many applications of SVD - including those involving recommender systems - our objective is to map a (sometimes sparse) high-dimensional space to a dense low-dimensional space

- ▶ Have shown results from full-rank decomposition
- ▶ In many applications of SVD - including those involving recommender systems - our objective is to map a (sometimes sparse) high-dimensional space to a dense low-dimensional space
- ▶ For example, if we have a large user-by-item explicit ratings matrix, we may want to create a more compact representation of that ratings space using linearly independent latent factors

- ▶ Have shown results from full-rank decomposition
- ▶ In many applications of SVD - including those involving recommender systems - our objective is to map a (sometimes sparse) high-dimensional space to a dense low-dimensional space
- ▶ For example, if we have a large user-by-item explicit ratings matrix, we may want to create a more compact representation of that ratings space using linearly independent latent factors
- ▶ We can think of the latent factors as an abstraction that allows us to represent each user and item by a linear combination of other users and items



- ▶ Have shown results from full-rank decomposition
- ▶ In many applications of SVD - including those involving recommender systems - our objective is to map a (sometimes sparse) high-dimensional space to a dense low-dimensional space
- ▶ For example, if we have a large user-by-item explicit ratings matrix, we may want to create a more compact representation of that ratings space using linearly independent latent factors
- ▶ We can think of the latent factors as an abstraction that allows us to represent each user and item by a linear combination of other users and items
- ▶ Classical SVD is one approach to solve for such latent factors, but unfortunately it can only be applied to a dense matrix

- ▶ Have shown results from full-rank decomposition
- ▶ In many applications of SVD - including those involving recommender systems - our objective is to map a (sometimes sparse) high-dimensional space to a dense low-dimensional space
- ▶ For example, if we have a large user-by-item explicit ratings matrix, we may want to create a more compact representation of that ratings space using linearly independent latent factors
- ▶ We can think of the latent factors as an abstraction that allows us to represent each user and item by a linear combination of other users and items
- ▶ Classical SVD is one approach to solve for such latent factors, but unfortunately it can only be applied to a dense matrix
- ▶ Using independent latent factors often simplifies the interpretation of large datasets because it drastically reduces the number of interacting variables.

## References

Gene H. Golub and Charles F. Van Loan. Matrix Computations. The John Hopkins University Press., 4th edition, 2013.

Jeremy Kun. Source Code for SVD, January 2018. URL <https://github.com/j2kun/svd>.

David Skillicorn. Understanding Complex Datasets: Data Mining with Matrix Decompositions. Chapman and Hall/CRC, 2007.

## Appendix A - Power Method

- Power method for SVD of a matrix  $A \in \mathbb{R}^{m \times n}$  works as follows:
  - Start by computing the first singular value  $\sigma_1$  and the left and right singular vectors  $u_1$  and  $v_1$  of  $A$ , for which  $\min_{i>j} \log(\sigma_i/\sigma_j) \geq \lambda$

## Appendix A - Power Method

- Power method for SVD of a matrix  $A \in \mathbb{R}^{m \times n}$  works as follows:
  - Start by computing the first singular value  $\sigma_1$  and the left and right singular vectors  $u_1$  and  $v_1$  of  $A$ , for which  $\min_{i>j} \log(\sigma_i/\sigma_j) \geq \lambda$
- 1. Generate  $x_0$  such that  $x_0(i) \sim N(0,1)$

## Appendix A - Power Method

- Power method for SVD of a matrix  $A \in \mathbb{R}^{m \times n}$  works as follows:
  - ▶ Start by computing the first singular value  $\sigma_1$  and the left and right singular vectors  $u_1$  and  $v_1$  of  $A$ , for which  $\min_{i>j} \log(\sigma_i/\sigma_j) \geq \lambda$
- 1. Generate  $x_0$  such that  $x_0(i) \sim N(0,1)$
- 2. for  $i$  in  $[1, \dots, k]$  where  $k = \min(m, n)$ :

## Appendix A - Power Method

- Power method for SVD of a matrix  $A \in \mathbb{R}^{m \times n}$  works as follows:
  - Start by computing the first singular value  $\sigma_1$  and the left and right singular vectors  $u_1$  and  $v_1$  of  $A$ , for which  $\min_{i>j} \log(\sigma_i/\sigma_j) \geq \lambda$
- 1. Generate  $x_0$  such that  $x_0(i) \sim N(0,1)$
- 2. for  $i$  in  $[1, \dots, k]$  where  $k = \min(m, n)$ :
- 3.  $x_i \leftarrow A^T \leftarrow Ax_{i-1}$

## Appendix A - Power Method

- Power method for SVD of a matrix  $A \in \mathbb{R}^{m \times n}$  works as follows:
  - Start by computing the first singular value  $\sigma_1$  and the left and right singular vectors  $u_1$  and  $v_1$  of  $A$ , for which  $\min_{i>j} \log(\sigma_i/\sigma_j) \geq \lambda$
- 1. Generate  $x_0$  such that  $x_0(i) \sim N(0,1)$
- 2. for  $i$  in  $[1, \dots, k]$  where  $k = \min(m, n)$ :
- 3.  $x_i \leftarrow A^T \leftarrow Ax_{i-1}$
- 4.  $v_1 \leftarrow x_i / \|x_i\|$



## Appendix A - Power Method

- Power method for SVD of a matrix  $A \in \mathbb{R}^{m \times n}$  works as follows:
  - Start by computing the first singular value  $\sigma_1$  and the left and right singular vectors  $u_1$  and  $v_1$  of  $A$ , for which  $\min_{i>j} \log(\sigma_i/\sigma_j) \geq \lambda$
- 1. Generate  $x_0$  such that  $x_0(i) \sim N(0,1)$
- 2. for  $i$  in  $[1, \dots, k]$  where  $k = \min(m, n)$ :
- 3.  $x_i \leftarrow A^T \leftarrow Ax_{i-1}$
- 4.  $v_1 \leftarrow x_i / \|x_i\|$
- 5.  $\sigma_1 \leftarrow \|Av_1\|$

## Appendix A - Power Method

- Power method for SVD of a matrix  $A \in \mathbb{R}^{m \times n}$  works as follows:
  - Start by computing the first singular value  $\sigma_1$  and the left and right singular vectors  $u_1$  and  $v_1$  of  $A$ , for which  $\min_{i>j} \log(\sigma_i/\sigma_j) \geq \lambda$
- 1. Generate  $x_0$  such that  $x_0(i) \sim N(0,1)$
- 2. for  $i$  in  $[1, \dots, k]$  where  $k = \min(m, n)$ :
- 3.  $x_i \leftarrow A^T \leftarrow Ax_{i-1}$
- 4.  $v_1 \leftarrow x_i / \|x_i\|$
- 5.  $\sigma_1 \leftarrow \|Av_1\|$
- 6.  $u_1 \leftarrow Av_1 / \sigma_1$

## Appendix A - Power Method

- Power method for SVD of a matrix  $A \in \mathbb{R}^{m \times n}$  works as follows:
  - Start by computing the first singular value  $\sigma_1$  and the left and right singular vectors  $u_1$  and  $v_1$  of  $A$ , for which  $\min_{i>j} \log(\sigma_i/\sigma_j) \geq \lambda$
- 1. Generate  $x_0$  such that  $x_0(i) \sim N(0,1)$
- 2. for  $i$  in  $[1, \dots, k]$  where  $k = \min(m, n)$ :
- 3.  $x_i \leftarrow A^T \leftarrow Ax_{i-1}$
- 4.  $v_1 \leftarrow x_i / \|x_i\|$
- 5.  $\sigma_1 \leftarrow \|Av_1\|$
- 6.  $u_1 \leftarrow Av_1 / \sigma_1$
- 7. return  $(\sigma_1, u_1, v_1)$

## Appendix A - Power Method

- Power method for SVD of a matrix  $A \in \mathbb{R}^{m \times n}$  works as follows:
  - Start by computing the first singular value  $\sigma_1$  and the left and right singular vectors  $u_1$  and  $v_1$  of  $A$ , for which  $\min_{i>j} \log(\sigma_i/\sigma_j) \geq \lambda$
- 1. Generate  $x_0$  such that  $x_0(i) \sim N(0,1)$
- 2. for  $i$  in  $[1, \dots, k]$  where  $k = \min(m, n)$ :
- 3.  $x_i \leftarrow A^T \leftarrow Ax_{i-1}$
- 4.  $v_1 \leftarrow x_i / \|x_i\|$
- 5.  $\sigma_1 \leftarrow \|Av_1\|$
- 6.  $u_1 \leftarrow Av_1 / \sigma_1$
- 7. return  $(\sigma_1, u_1, v_1)$
- Once we have computed  $(\sigma_1, u_1, v_1)$ , we can repeat this process for  $A - \sigma_1 u_1 v_1^T = \sum_{i=2}^n \sigma_i u_i v_i^T$

## Appendix A - Power Method

- Power method for SVD of a matrix  $A \in \mathbb{R}^{m \times n}$  works as follows:
  - Start by computing the first singular value  $\sigma_1$  and the left and right singular vectors  $u_1$  and  $v_1$  of  $A$ , for which  $\min_{i>j} \log(\sigma_i/\sigma_j) \geq \lambda$
- 1. Generate  $x_0$  such that  $x_0(i) \sim N(0,1)$
- 2. for  $i$  in  $[1, \dots, k]$  where  $k = \min(m, n)$ :
- 3.  $x_i \leftarrow A^T \leftarrow Ax_{i-1}$
- 4.  $v_1 \leftarrow x_i / \|x_i\|$
- 5.  $\sigma_1 \leftarrow \|Av_1\|$
- 6.  $u_1 \leftarrow Av_1 / \sigma_1$
- 7. return  $(\sigma_1, u_1, v_1)$
- Once we have computed  $(\sigma_1, u_1, v_1)$ , we can repeat this process for  $A - \sigma_1 u_1 v_1^T = \sum_{i=2}^n \sigma_i u_i v_i^T$

## Appendix B - Matrix Approximation

We perform the matrix decomposition for the largest  $k$  singular values, then reconstitute the matrix using only the associated  $k$  latent user and item factors:

```
# compute SVD  
kSingularValues,kU,kV = svdPowerMethod(A,k=2)
```

Using the 8 by 2 matrix of left singular vectors  $U$ , the 2 by 2 diagonal singular value matrix  $\Sigma$ , and the 2 by 3 matrix of right singular vectors  $V$ , we reconstitute the approximate matrix  $\hat{A}$  using  $k$  factors (in this case, just two user factors and two item factors):

```
# reconstitute matrix A  
kSigma = np.diag(kSingularValues)  
# reconstitute matrix A  
kAA=np.dot(kU, np.dot(kSigma, kV))
```

We can compare the original matrix  $A$  and the two-factor approximation to that matrix  $\hat{A}$ :

```
print(np.round(AA-kAA,2))
```

```
## [[ 0.5  -0.98  0.76]
##  [-0.45  0.88 -0.69]
##  [-0.2   0.38 -0.3 ]
##  [ 0.26 -0.5   0.39]
##  [-0.53  1.02 -0.79]
##  [-0.39  0.76 -0.59]
##  [ 0.09 -0.18  0.14]
##  [ 0.8  -1.56  1.21]]
```

Despite the use of an extremely small dataset, the two-factor approximation looks reasonable.