## Concept of Vectors and Matrices

Mathematically, A vector is represented by a single dimension array of numbers. Example: V = [25, 3, 46, 30, 33]. This makes no sense to a Machine Learning practitioner until it can be related to something real. Let us say the numbers represents certain data of an individual - Age, years of Work experience, Annual Salary(in 000 USD), Average Expenses per year (in 000 USD), % of Annual salary saved per year. If such data are collected for multiple individuals and stacked up one over the other, then we have an array of vectors which is called a *Matrix*.

From a Linear Algebra perspective, a Matrix can represent one of the two:

1. **Systems of Linear Equations** - Matrix operations can be used to represent a set of linear equations. This helps us solve the equations, define inverse of matrices.
2. **Linear Mapping** - Matrices can be used to define Linear Mappings. Linear mapping define transformation actions that can be performed on Vectors. Examples of Vector Transformations that fall into this bucket are: Rotation, Stretching along specific axes, change of basis

## Matrix Decomposition

Let us take an example of a Matrix that represents a set of Vectors - i.e . Data. Let us assume that the Data consists of rows - which represents users, and columns which represents products that they have purchased. So we have a user-product matrix. The matrix maps each user to each product. If a user Ui has purchased product Vj then the cell ij will contain the number of items purchased. If a user has never purchased a product, then the corresponding cell will have a 0. Please note that at this stage, we just have Sales Data arranged in a certain way (in the form of matrix). This does not give any additional information. ***So how do we break it down into chunks of comprehendable pieces? Enter Factorization!***

### Concept of Matrix Factorization

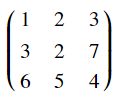
Suppose you had a number - say 28, and you want to understand it better. What do you do? You can say 28 is 7 times 4, which is 7 times 2 times 2 (7 x 2 x 2). This tells us that 7 and 2 are latent in the number 28. What happens if we apply factorization to Matrices? By extending the logic, we must be able to understand the latent factors of matrices. The process of decomposing a matrix into certain factors of interpretable matrices is known as Matrix Factorization or Matrix Decomposition

### Matrix Decomposition Methods

There are two ways to decompose matrices - ***Eigen Decomposition, Singular Value Decomposition***. Both methodologies are similar. The fundamental difference is that Eigen decomposition can only be done on square matrices. Whereas Singular Value Decomposition can be done on any matrix and is generic in a sense. Regardless, at the core of Matrix Decomposition method is identifying values that capture key information about the parent matrix.

#### Eigen Decomposition

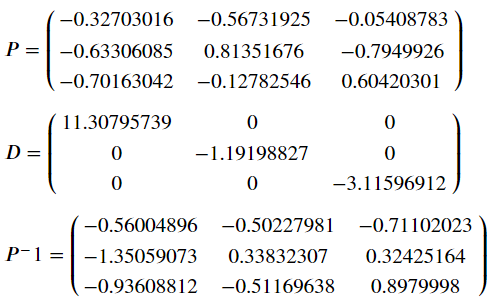
Let A =



A Square matrix can be decomposed into Eigen values and Eigen vectors as follows

* Let be Eigen Vectors of a matrix A, such that where L1, L2, L3,..Ln are Eigen values
* Then Matrix arranged in a columnar fashion, D = Diagonal matrix of , - inverse of P
* P is matrix of eigen vectors, D is eigenvalues arranged in a diagonal matrix, and is inverse of matrix of eigen vectors
* It can be Mathematically proven that (Dot products)
* Eigen values and Eigen vectors capture information about a matrix. The first eigen vector and first eigen value captures highest information about the matrix. Similarly, second eigen vector and second eigen value captures 2nd highest information so on...
* This technique is useful in dimensional reduction/compression

Post Eigen decomposition of A, we have the following



An illustration of Eigen Values and Eigen Vectors in action - Google Page Rank Algorithm.

Here is the URL to a page with a neat illustration of how Eigen values work <http://pi.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture3/lecture3.html>

#### Code

|  |
| --- |
| import numpy as np import pandas as pd  A = np.array([[1,2,3],[3,2,7],[6,5,4]])  D,P = np.linalg.eig(A)  P  array([[-0.32703016, -0.56731925, -0.05408783],  [-0.63306085, 0.81351676, -0.7949926 ],  [-0.70163042, -0.12782546, 0.60420301]])  P\_inv = np.linalg.inv(P)  P\_inv  array([[-0.56004896, -0.50227981, -0.71102023],  [-1.35059073, 0.33832307, 0.32425164],  [-0.93608812, -0.51169638, 0.89799982]])  D  array([11.30795739, -1.19198827, -3.11596912]) |

#### Singular Value Decomposition

Okay, Eigen decomposition works when the matrix is a square matrix. But in most real-life scenarios, the matrices are not square. Therefore, Eigen decomposition is not useful in many cases. So, what do we decompose non-square matrices? Singular Vector Decomposition is the answer to this problem.

Any Matrix can be decomposed as

Where

* S - Singular matrix which is a diagonal matrix
* U - Left Singular matrix
* - Right Singular matrix

The diagonal values in Matrix S is called Singular Matrix and the Singular values arranged in the descending order from top-left

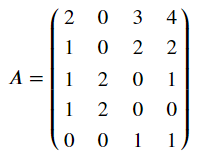
SVD is used in a number of ways in Machine learning including solving systems of linear equations, curve fitting etc. Substituting a matrix with its SVD approximation reduces calculation as well as loss. Here, we describe how this is achieved Let A be a user item matrix (4 users and 5 items) as shown below. The Values in the table describe the quantities purchased

Let us see SVD in action through an Example

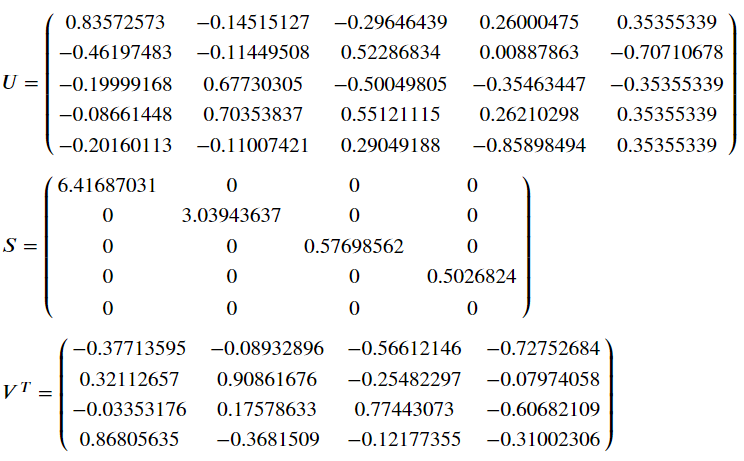
Let A be a user item matrix (4 users and 5 items) as shown below. The Values in the table describe the quantities purchased

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Cat/User | U1 | U2 | U3 | U4 |
| **Formal\_Shirt** | 2 | 0 | 3 | 4 |
| **Trouser** | 1 | 0 | 2 | 2 |
| **T\_shirt** | 1 | 2 | 0 | 1 |
| **Jeans** | 1 | 2 | 0 | 0 |
| **Blazer** | 0 | 0 | 1 | 1 |

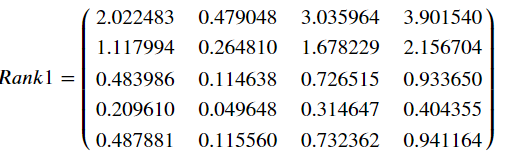
Let us get started with Decomposition



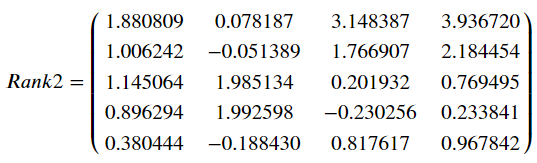
SVD Decomposition results in the following matrices



Let us calculate the first low rank matrix (Reconstructing)



If you add to Rank 1 approximation then we get Rank 2 Matrix



This is already close to the original matrix. Rank, 3 and 4 will of course add more information to it. However, from computation perspective, it is much easier to multiply 5x2, 2x2, and 2x4 matrices than trying to crunch the original 5x4 matrix

How does this reduce computation? Note that a matrix of Size mxn is replaced with a smaller matrices of mxp, pxn- where p is rank of the singular matrix. P can also be seen as the number of latent factors

In our example we saw 5x4 matrix being replaced with one 5x2 matrix and another 2x4 matrix in addition to two real numbers. In real-life situations, matrices of several thousands of rows and columns can be reduced to ones with few hundreds. Storing such matrices require less memory and information is retained as well.

The Rank P approximation can also be related to the number of latent factors. In fact, the Rank P is also equal to the number of latent factors. Let us delve a little deeper into the example we have taken.

In Matrix U above, the first cell of the first column (-0.83) indicates that **formal\_shirt** is unique and has its own factor. Similarly, the 3rd and 4th cell in the second column are close to each other and have relatively large values. These values correspond to Jeans and T-Shirts. **This indicates that Jeans and T-Shirts are part of same cluster (most likely casual clothing).** The other columns, even though giving some information about how products might be related, are not significant because the singular values are much lower.

Similarly in Matrix Users 3,4 are close to each other in their tastes (Some one that buys formals and casuals) as shown in the third and fourth cells of 1st row. User 2 is unique in his taste and hence has a value of 0.90 and no other users are close to this user as indicated in 2nd cell of Row 2 in the Matrix. The last two rows, although, giving some information about the users, are not significant because the singluar values are much lower than the first two

Matrix factorization methods are extensively used in several use cases

1. Clustering
2. Dimensionality Reduction
3. Product Recommendations

to name a few.

#### Code

|  |
| --- |
| A = np.array([[2,0,3,4],[1,0,2,2],[1,2,0,1],[1,2,0,0],[0,0,1,1]])  U,S,VT = np.linalg.svd(A)  S  array([6.41687031, 3.03943637, 0.57698562, 0.5026824 ])  U  array([[-0.83572573, -0.14515127, -0.29646439, 0.26000475, 0.35355339],  [-0.46197483, -0.11449508, 0.52286834, 0.00887863, -0.70710678],  [-0.19999168, 0.67730305, -0.50049805, -0.35463447, -0.35355339],  [-0.08661448, 0.70353837, 0.55121115, 0.26210298, 0.35355339],  [-0.20160113, -0.11007421, 0.29049188, -0.85898494, 0.35355339]])  U\_df = pd.DataFrame(U)  U\_df  0 1 2 3 4 0 -0.835726 -0.145151 -0.296464 0.260005 0.353553 1 -0.461975 -0.114495 0.522868 0.008879 -0.707107 2 -0.199992 0.677303 -0.500498 -0.354634 -0.353553 3 -0.086614 0.703538 0.551211 0.262103 0.353553 4 -0.201601 -0.110074 0.290492 -0.858985 0.353553  S  array([6.41687031, 3.03943637, 0.57698562, 0.5026824 ])  VT  array([[-0.37713595, -0.08932896, -0.56612146, -0.72752684],  [ 0.32112657, 0.90861676, -0.25482297, -0.07974058],  [-0.03353176, 0.17578633, 0.77443073, -0.60682109],  [ 0.86805635, -0.3681509 , -0.12177355, -0.31002306]])  VT\_df = pd.DataFrame(VT)  VT\_df  0 1 2 3 0 -0.377136 -0.089329 -0.566121 -0.727527 1 0.321127 0.908617 -0.254823 -0.079741 2 -0.033532 0.175786 0.774431 -0.606821 3 0.868056 -0.368151 -0.121774 -0.310023  comps = [1,2,3,4,5]  for i in range(4):  low\_rank = U[:, :comps[i]] @ np.diag(S[:comps[i]]) @ VT[:comps[i], :]  low\_rank\_df = pd.DataFrame(low\_rank).round(1)  print("\n rank %d \n" % comps[i])  print(low\_rank\_df)  rank 1    0 1 2 3 0 2.0 0.5 3.0 3.9 1 1.1 0.3 1.7 2.2 2 0.5 0.1 0.7 0.9 3 0.2 0.0 0.3 0.4 4 0.5 0.1 0.7 0.9   rank 2    0 1 2 3 0 1.9 0.1 3.1 3.9 1 1.0 -0.1 1.8 2.2 2 1.1 2.0 0.2 0.8 3 0.9 2.0 -0.2 0.2 4 0.4 -0.2 0.8 1.0   rank 3    0 1 2 3 0 1.9 0.0 3.0 4.0 1 1.0 0.0 2.0 2.0 2 1.2 1.9 -0.0 0.9 3 0.9 2.0 0.0 0.0 4 0.4 -0.2 0.9 0.9   rank 4    0 1 2 3 0 2.0 0.0 3.0 4.0 1 1.0 -0.0 2.0 2.0 2 1.0 2.0 0.0 1.0 3 1.0 2.0 -0.0 0.0 4 0.0 0.0 1.0 1.0 |