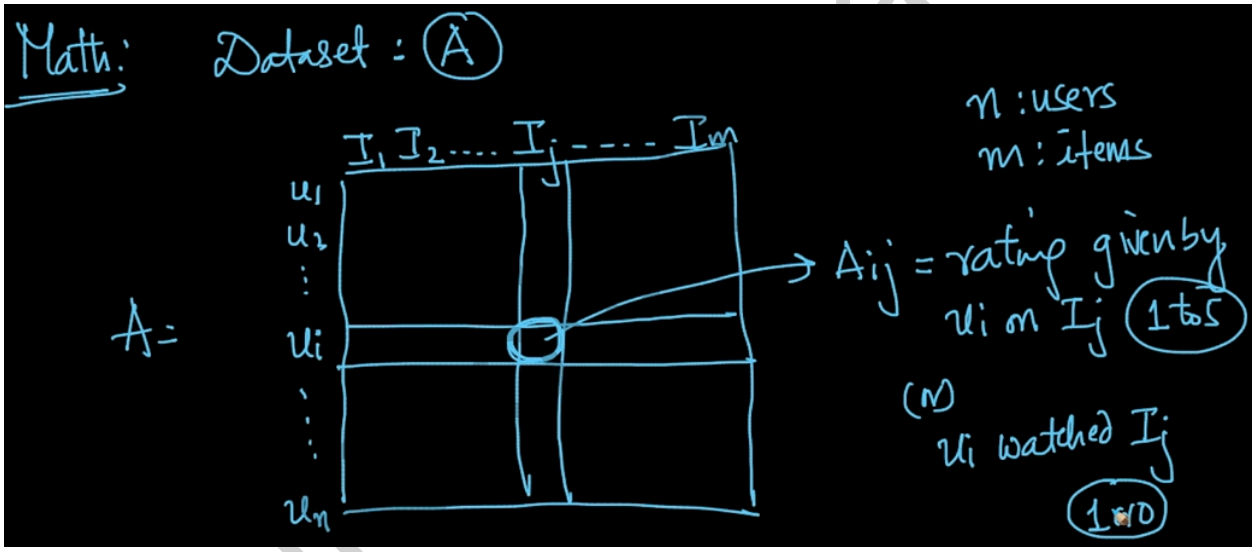
53.1 Problem Formulation: Movie Reviews

Let us look at a new type of problem called a Recommendation problem. One of the best examples of recommendation problems is recommending movies/videos on platforms like Netflix, Amazon Prime and Youtube.

Mathematical Formulation

Let us assume we were given a dataset and it is represented as a matrix 'A'. This dataset contains the data on the ratings given by the different users on different items. Let us assume we have 'n' users and 'm' items, then the matrix looks as below



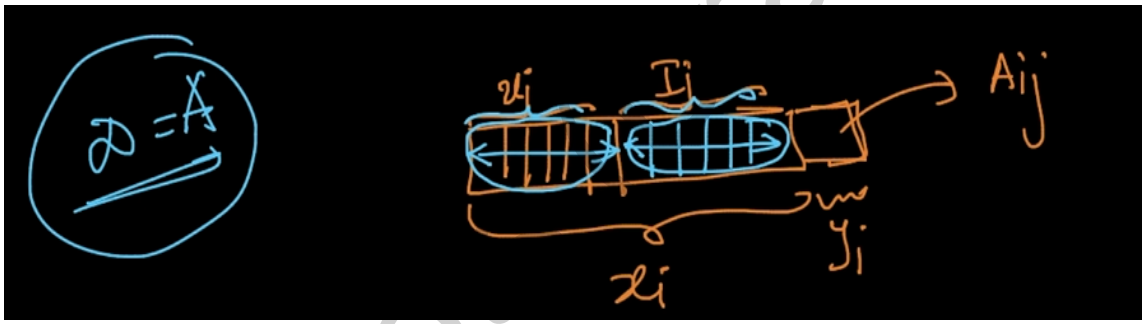
This matrix contains either binary (or) discrete numerical values. Every element in this matrix is denoted as 'A_{ij}' which denotes the rating given by the user 'i' (ie., '$\mathbf{u}_i$')on the item 'j' (ie., '$\mathbf{i}_j$').

'$\mathbf{A}_{ij}$' is binary, if the value just indicates whether the user '$\mathbf{u}_i$' has watched the item '$\mathbf{i}_j$', whereas 'Aij' is discrete numerical if the value indicates the rating given by the user '$\mathbf{u}_i$' on the item '$\mathbf{i}_j$'. (This rating could be mostly on the scale of 1 to 5, where 1 being the worst and 5 being the best). These values are given by the users, and in case, if any user hasn't given the value for any item, then we can leave it blank.

Properties of Matrix of Ratings

1) The Matrix of Ratings is very sparse. As there are a huge number of users and a huge number of items, it is not possible for every user to watch and rate every item. Hence we see this matrix sparse in most of the cases.
   Sparsity of Matrix = Number of Empty cells/Total number of cells
   The main goal of building a recommender system is if a user '$\mathbf{u_i}$' has given a rating for a few items, then the recommender system should recommend new items to the user '$\mathbf{u_i}$' depending on his/her interests.

2) Recommender System problem can also be posed as a Regression or a Classification problem.

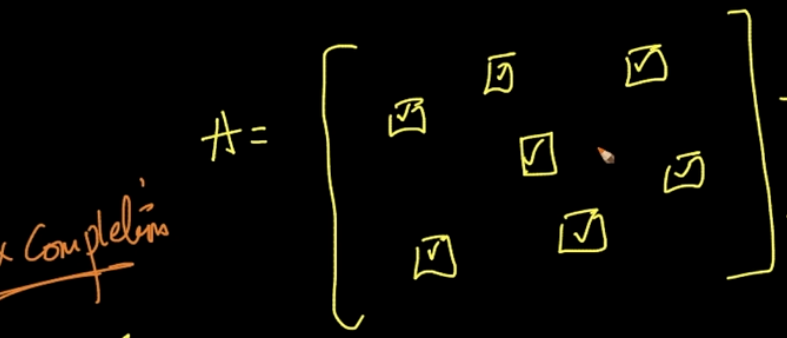


   The features for '$\mathbf{u_i}$' and '$\mathbf{i_j}$' are not given directly. We will be given only the matrix 'A' as the dataset. We have to obtain the features for '$\mathbf{u_i}$' and '$\mathbf{i_j}$' using Feature Engineering.
   Here 'D_{Train}' = All the data points with non empty cells
   'D_{Test}' = All the data point with empty cells
   We need to predict the values for all the non-empty cells. This is the way of posing a recommender system problem as a regression or a classification problem.

3) Recommender System problem can also be posed as a Matrix completion problem. Here 'A' is the given matrix, and only a few values of the matrix 'A' are given, whereas the remaining cells are empty. The problem is to fill up the empty cells with reasonable values, based on the values in the non-empty cells.

A :- RS

RS as a matrix completion problem

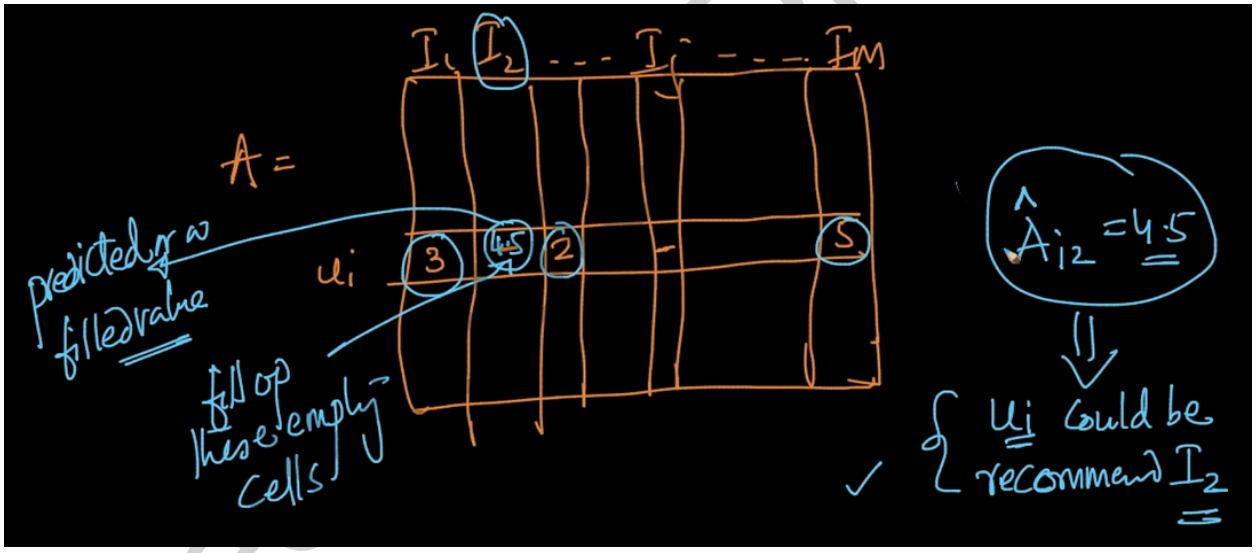
$A =$ 

→ some values of A_{ij} are given

→ many A_{ij}'s are empty

Matrix Completion

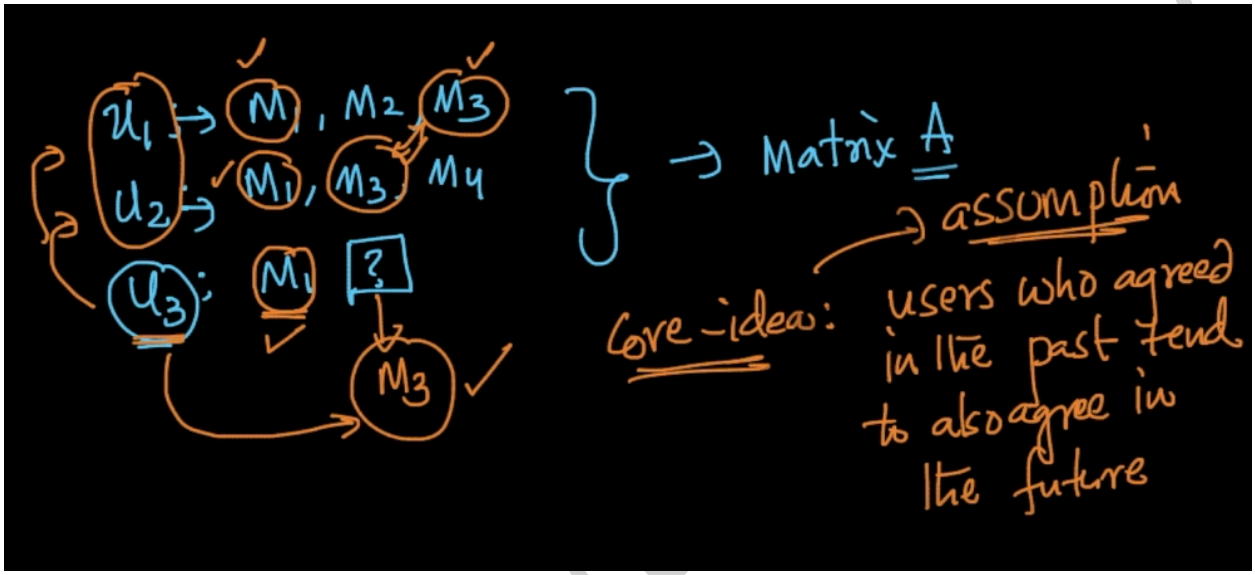
✓ Fill up the empty cells with reasonable values based on values in the non-empty cells



53.2 Content Based vs Collaborative Filtering

There are 2 types of recommendation algorithms. They are

a) Collaborative Filtering based recommendation



For example, if we have 3 users 'u_1', 'u_2' and 'u_3'. If the user 'u_1' has watched the movies 'M_1', 'M_2' and 'M_3', and the user 'u_2' has watched the movies 'M_1', 'M_3' and 'M_4'. Let us assume the user 'u_3' has watched only the movie 'M_1', and now we have to recommend more movies to the user 'u_3', on the basis of the movies that were watched by the other users, who also have watched the movies watched by 'u_3'.

So when we look at the above data, we can see that the movie 'M_1' which was watched by the user 'u_3', was also watched by the users 'u_1' and 'u_2'. So it means both the users 'u_1' and 'u_2' share the similar taste with the user 'u_3'. In such a case, it is always more appropriate to recommend the movies to 'u_3', that were watched by both 'u_1' and 'u_2', rather than those watched by only one of them. So here we could see the movie 'M_3' was watched by both 'u_1' and 'u_2'. The movie 'M_2' was watched only by 'u_1' and the movie 'M_4' was watched only by 'u_2'. So it is appropriate to recommend the movie 'M_3' first, to the user 'u_3', then 'M_2' and 'M_4'.

The core idea/assumption in collaborative filtering is the users who agreed in the past, tend to agree in the future also.

b) Content based recommendation

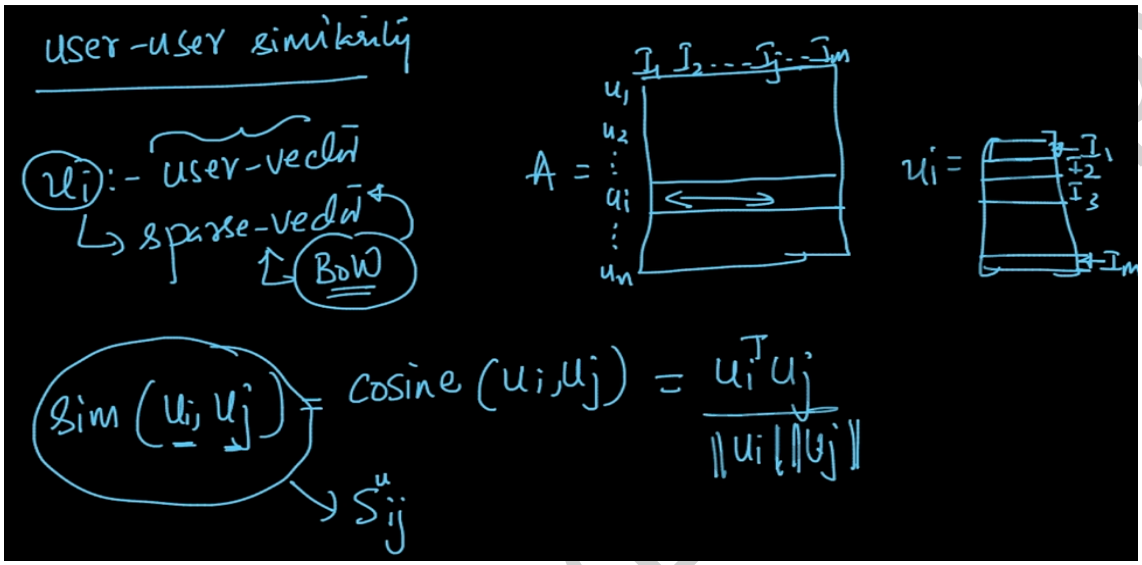
The content based recommendation system doesn't use the ratings '$\mathbf{A}_{ij}$' given by the users. This system needs a way to represent the user vector 'u_i' and the item vector 'I_i'. The point of content based filtering is that we have to know the content of both the users and the items. Usually we construct the user profile and the item profile using the content of the shared attribute space.

In this recommendation system, the engine compares the items that are already positively rated by the user with the items that he didn't rate and looks for similarities. Items similar to the positively rated ones, will be recommended to the user. The content based systems will only recommend the articles/items related to the same categories, and never recommend anything from the other categories.

53.3 Similarity based Algorithms

There are 2 types of Similarity based Recommender. They are

a) **User-User Similarity**



u_i → user vector (It is a sparse vector)

Every 'u_i' is m-dimensional. It is the rating given by each user for all the 'm' items. So the similarity between two users 'u_i' and 'u_j' is given as

S^u_{ij} = sim(u_i, u_j) = cosine-similarity(u_i, u_j) = $u_i^T u_j$/(||u_i||*||u_j||)

So we need to build a similarity matrix with all 'S_{ij}'. So 'S^u' is an nXn matrix, and 'S_{ij}' represents how similar two users 'u_i' and 'u_j' are. Larger the value of 'S^u_{ij}' , the more similar the users 'u_i' and 'u_j' are.

For example, let us assume the users 'u_1', 'u_2' and 'u_7' are the 3 users most similar to the user 'u_{10}', then we have to pick the items liked/bought by 'u_1', 'u_2' and 'u_7' and not yet bought/liked by the user 'u_{10}', and finally recommend them to the user 'u_{10}'.

There is one major problem with user-user similarity. That is the user's preferences change over time. In such cases, the chances to build the best recommender systems are low. If the user preferences do not change over time, then we can build a matrix with respect to time. So an alternative to handle this problem is **item-item similarity**.

b) Item-Item Similarity

$I_i \rightarrow$ item vector

Each item is represented by an item vector. Every item vector 'I_i' is m-dimensional, and it is the rating given for each item. So the similarity between two items is given as

$S^i_{ij} = sim(I_i, I_j) = \text{cosine-similarity}(I_i, I_j) = I_i^T I_j / (||I_i|| * ||I_j||)$

The major disadvantage with the user-user based similarity is the preferences of the user change over time, but the advantage of item-item similarity is the ratings on the given item do not change significantly.

For example, if a user 'u_{10}' likes the items 'I_1', 'I_3' and 'I_7', then we have to generate 3 different sets with the items similar to 'I_1', 'I_3' and 'I_7' respectively. If any item is present in more than one of these sets, then the chances that itm to be liked by the user 'u_{10}' are more.

Rule of Thumb

When we have more number of users when compared to the items, and if the item ratings do not change much over the time after the initial period, then it is better to use item-item similarity approach over the user-user similarity approach.

53.4 Matrix Factorization: PCA, SVD

To solve the recommendation system problems, we have a technique called **Matrix Factorization**. The code idea of Matrix Factorization is to decompose a given matrix into a product of two or more matrices.

If we can decompose matrix 'A' into a product of two or more matrices, we call it **Multiplicative Matrix Decomposition** (or) **Matrix Factorization**.

Principal Component Analysis (PCA)

We are given a data matrix 'X_{nxd}', which is centered. The covariance matrix of 'X' is given as $\mathbf{S_{dxd} = X^TX/(n-1)}$

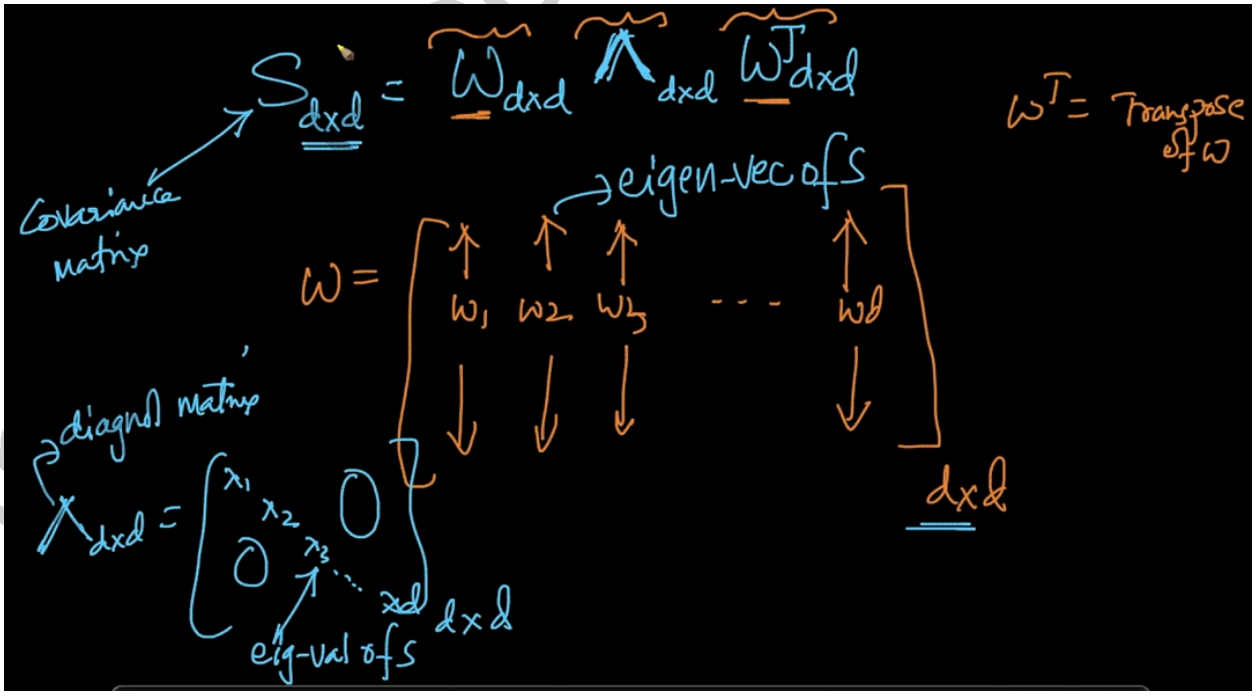
Let the eigenvalues be denoted as $\lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_d$

Let the respective eigenvectors be denoted as $w_1, w_2, w_3, \ldots, w_d$

Now the covariance matrix Sdxd can be decomposed into a product of two or more matrices as

$$\mathbf{S_{dxd} = w_{dxd}\ \Lambda_{dxd}\ w^T_{dxd}}$$

Where '$\mathbf{w_{dxd}}$' is a matrix of the eigenvectors as the columns, and '$\mathbf{\Lambda_{dxd}}$' is a diagonal matrix of the eigenvalues as shown below.



This whole decomposition ($\mathbf{S=W\Lambda W^T}$) is called **Eigen Decomposition**. PCA is the eigen decomposition of the covariance matrix 'S'.

Singular Value Decomposition (SVD)

SVD is a matrix factorization technique related to PCA. PCA is applied only on the covariance matrix (S) which is square and symmetric. SVD on the other hand can be applied on any rectangular matrix. So we can apply SVD directly on the data matrix which is of the **nxd** shape.

$\mathbf{X_{nxd} = u_{nxn}\ \Sigma_{nxd}\ V^T_{dxd}}$

$\mathbf{u_{nxn}}$ → Left singular Vectors

$\mathbf{v^T_{dxd}}$ → Right singular vectors

$\mathbf{\Sigma_{nxd}}$ → Diagonal matrix of singular values of 'X' (ie., $s_1, s_2, s_3, \ldots, s_d$)

The reaction between the singular values (S_i) and eigenvalues (λ_i) is given by $\mathbf{s_i^2/(n-1) = \lambda_i}$

$\mathbf{\lambda_i}$ → Eigen value

$\mathbf{s_i}$ → singular value

$\mathbf{u_i}$ → i^{th} column of u → i^{th} eigenvector of $(X_{nxd}\ X^T_{dxn})_{nxn}$

$\mathbf{v_i}$ → i^{th} column of v → i^{th} eigenvector of $(X^T_{dxn}X_{nxd})_{dxd}$ → S

Note

Eigenvalues and Eigenvectors are defined for square matrices only while singular values and singular vectors are defined for all rectangular matrices.

53.5 Matrix Factorization: Non Negative Matrix Factorization (NMF)

Let us assume we are given a matrix 'A_{nxm}' which has to be decomposed as a product of two matrices 'B' and 'C^T' such that 'B' is a **nxd** matrix and 'C^T' is a **dxm** matrix, such that for all i, j → B_{ij}>=0, C_{ij}>=0.

$\mathbf{A_{nxm} = B_{nxd}\ (C^T)_{dxm}}$

53.6 Matrix Factorization for Collaborative Filtering

Consider a given matrix ‘A’ which consists of the ratings given by various users for various items.

Let A_{ij} → Rating given by the user ‘u_i’ on item ‘I_j’

Moreover ‘A’ is a sparse matrix. Now ‘A’ can be decomposed as a product of two matrices ‘B’ and ‘C^T’.

$\mathbf{A_{nxm} = B_{nxd}\ C^T_{dxm}}$ such that **d>0** and **d<=min(m,n)**.

Here ‘B’, ‘C’ are column vectors.

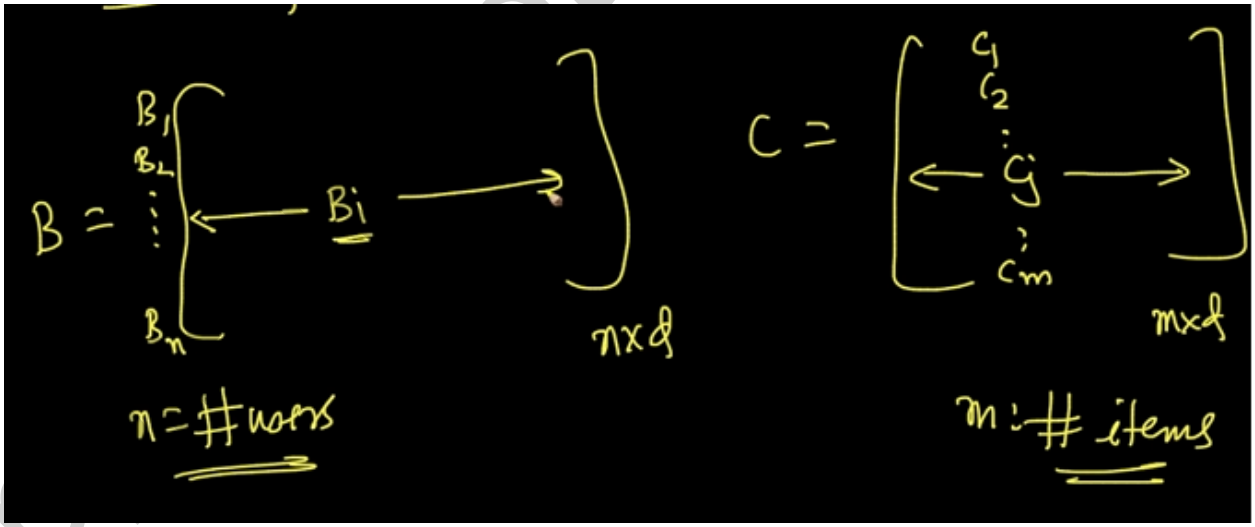
n → number of users

m → number of items

So the optimization problem is $\mathbf{argmin_{B,C}\ \Sigma_{i,j}\ (A_{ij} - B_i^T C_j)^2}$, such that ‘$A_{ij}$’ is not empty.

The goal is to find ‘B’ and ‘C’ matrices. So here we have to perform this summation only where ‘A_{ij}’ is not empty.

Using the Stochastic Gradient Descent algorithm, we now have to solve the optimization problem $\mathbf{argmin_{B,C}\ \Sigma_{Aij\ is\ not\ empty}\ (A_{ij} - B_i^T C_j)^2}$. It is like minimizing the squared loss. After solving the optimization problem, we’ll be having the values of the matrices ‘B’ and ‘C’.



After obtaining the values of the matrices ‘B’ and ‘C’, we have to complete the matrix ‘A’ by filling the missing values of ‘A_{ij}’ using the formula $\mathbf{A_{ij} = B^T_i C_j}$. Let us name this product matrix obtained as $\mathbf{A^\wedge = BC^T}$. The matrix ‘**A^**’ has no empty cells.

The key thing now is **A-A^** has to be very small. We should ensure that for all i,j, the value of '$\mathbf{A}_{ij}$' should be very close to '$\mathbf{\hat{A}}_{ij}$'.

Now '$\mathbf{\hat{A}}_{ij}$' is a complete form of 'A' using matrix factorization. The values in 'A^' are the predicted ratings, whereas the values in 'A' are the actual ratings.

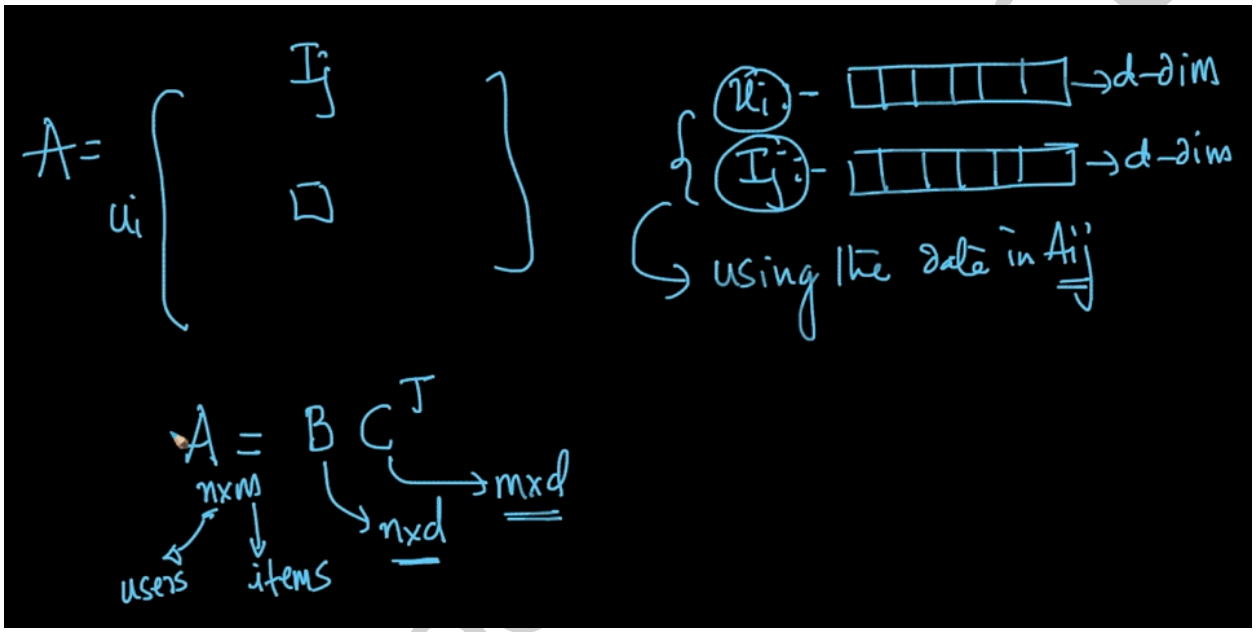
53.7 Matrix Factorization for Feature Engineering

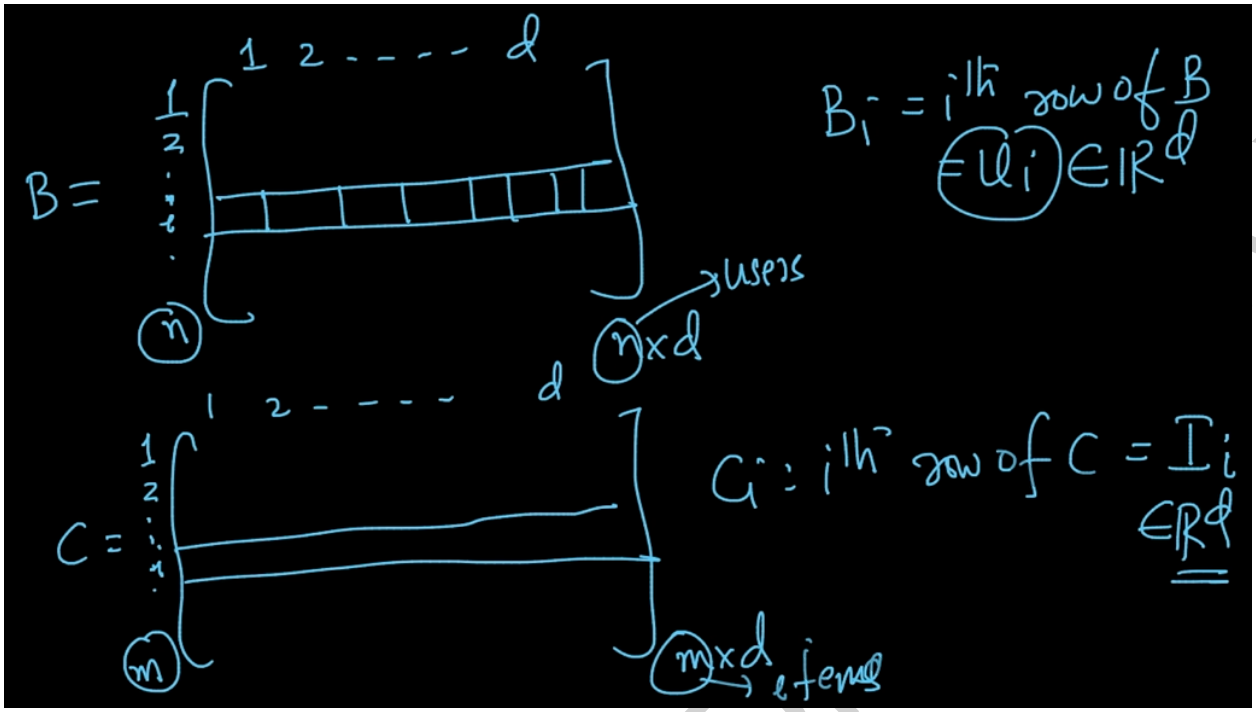
Let us assume we are given a matrix ‘A’. Let us now decompose the matrix ‘A’ into a product of two matrices.

$$\mathbf{A}_{nxm} = \mathbf{B}_{nxd}\ \mathbf{C}^{T}_{mxd}$$

Where $\mathbf{B}_{nxd}$ → Matrix of user vectors

$\mathbf{C}^{T}_{mxd}$ → Matrix of item vectors





As long as **d>0** and **d<=min(m,n)**, we can represent both 'B' and 'C' in d-dimensional form. So by leveraging the data in '$\mathbf{A_{ij}}$', we have to represent the user and the item vectors in d-dimensional form. We arrive at d-dimensional representation using the matrix factorization.

For example, if 'u_i', 'u_j' are similar in their ratings of products, then the distance between 'u_i' and 'u_j' is very small. Similarly, if 'I_i' and 'I_j' are similar on the basis of the ratings, then the distance between 'I_i' and 'I_j' is very small.

$B_i \rightarrow i^{th}$ row of 'B' = $u_i \in R^d$

$C_i \rightarrow i^{th}$ row of 'C' = $I_i \in R^d$

In the later chapters, we'll learn more in detail on how the matrix factorization can be used in creating the word vectors.

53.8 Clustering as MF

Let us take the example of the KMeans algorithm. The given dataset is denoted as D = $\{x_i\}$. The task is to compute centroids $\{C_1, C_2, C_3, \ldots, C_k\}$ and the cluster sets $\{S_1, S_2, S_3, \ldots\ldots, S_k\}$ such that $S_i \cap S_j = \emptyset \ \forall$ i,j.

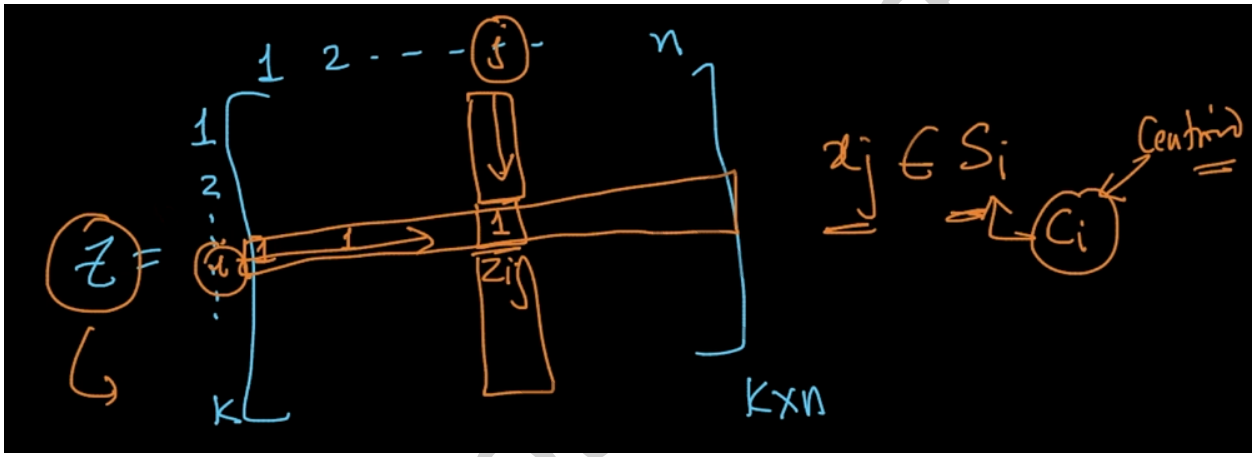
The optimization problem we have to solve is

$\mathbf{min_{Ci}\ \Sigma_{i=1}^{k}\ \Sigma_{x \in Si}\ ||x-C_i||^2}$ where $x \in R^d$

Let us define a matrix 'Z' (of shape **kxn**) such that

Zij = 1, if $x_j \in S_i$

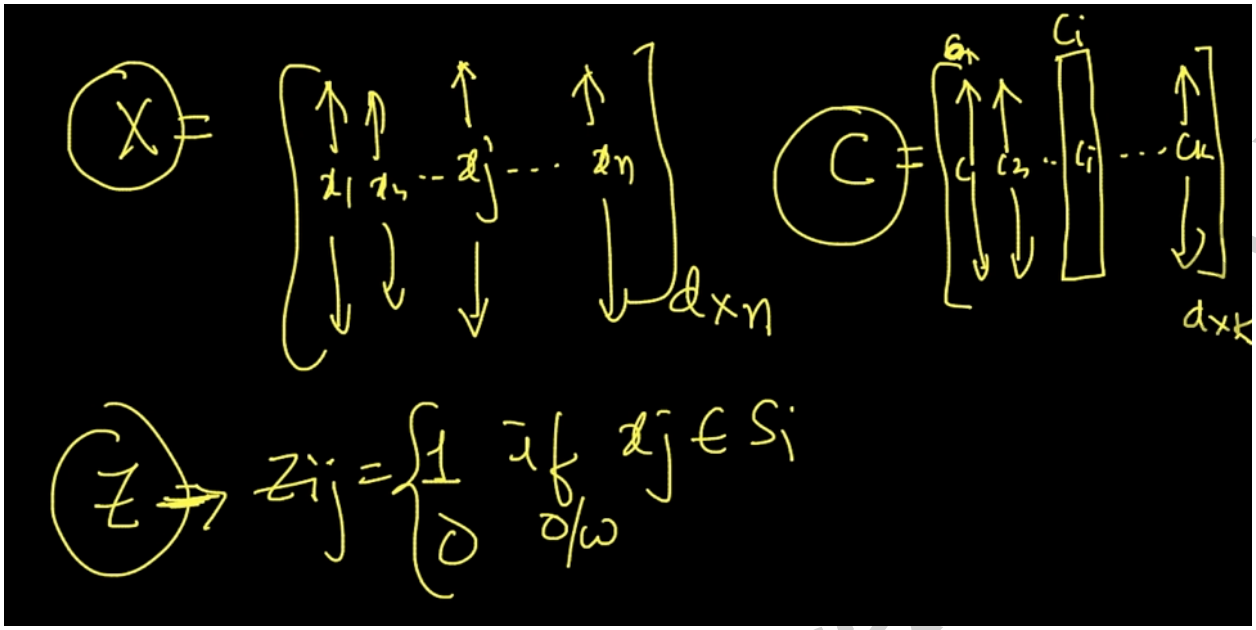
0, otherwise



Here 'Z' is a binary matrix. So if Z_{ij}=1, then it means the point 'xj' belongs to the cluster 'S_i' with the centre 'C_i'. The actual optimization problem was $\mathbf{min_{Ci}\ \Sigma_{i=1}^{k}\ \Sigma_{x \in Si}\ ||x-C_i||^2}$ where $x \in R^d$. Now this problem becomes

$\mathbf{min_{Ci,Zij}\ \Sigma_{i=1}^{k}\ \Sigma_{j=1}^{n} Z_{ij}\ ||x_j-C_i||^2}$

We have to convert the above problem into a matrix factorization problem. Let us take the matrices 'X' (of shape **dxn**)and 'C' (of shape **dxk**) where 'X' consists of all the data points, whereas 'C' consists of all the centroids. The optimization problem now becomes $\mathbf{min_{C,Z}\ ||X-CZ||^2}$



So if we want to write the KMeans clustering problem as a recommender system, then we have to decompose it as **X=CZ**, such that **Z_{ij}=1 (or) 0** (this is called **0-1 constraint**) and in every column 'Z_j' of the matrix 'Z', only one value should be 1, and the remaining all values should be 0. (ie., sum of all the elements of the column vector Z_j = 1). There is no chance for the existence of negative values in the matrix 'Z'.

But finding the accurate solution for KMeans is NP-hard. Hence we go with approximation. So

KMeans = Matrix Factorization + (0-1 constraint) + column constraint

Without these constraints, KMeans is the same as Matrix Factorization.

Note

In the case of general matrix factorization, we do not have any constraints on the values of the matrices 'C' and 'Z'.

In the case of Non negative matrix factorization, we have the constraints **C_{ij}>=0** and **Z_{ij}>=0**.

In the case of KMeans, the constraint is **Z_{ij} = 0 or 1**, and **$\Sigma_{i=1}^{k} Z_{ij} = 1$**.

53.9 Hyperparameter Tuning

Let ‘A’ be a ‘nxm’ matrix and is decomposed into a product of two matrices.

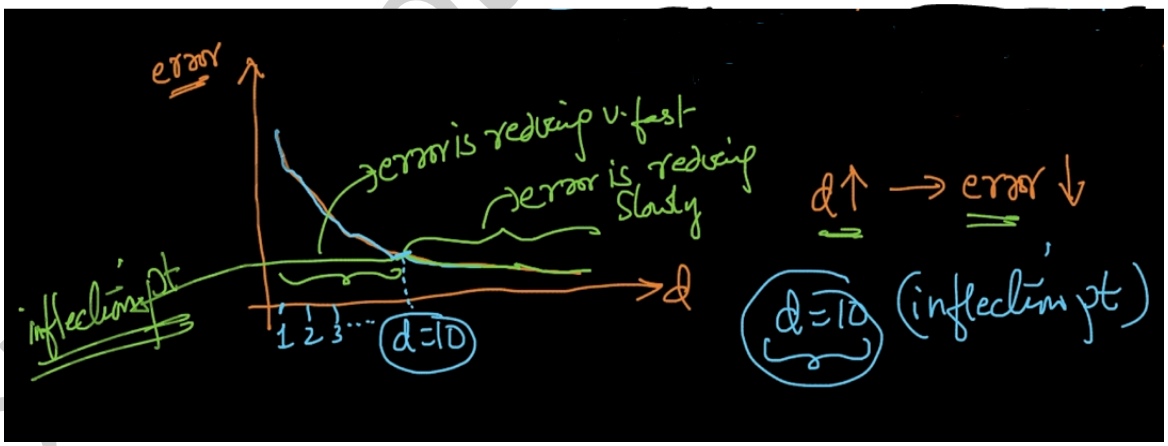
$\mathbf{A_{nxm} = B_{nxd}\ C^{T}_{mxd}}$ (where d>0, and d<=min(m, n))

Here ‘d’ is the hyperparameter.

How to select the value of ‘d’?

1) It is problem specific. Here it depends on the task we are solving. For example, if we are working on the problem of predicting the ratings given by a user for an item, then if we do not need larger dimensional (or) smaller dimensional representation for both the user and the item vectors, then the domain expert could select some moderate value like 100. It could be specified by the domain expert.

2) The optimization problem we are solving for matrix factorization is $\mathbf{min_{B,C}\ \Sigma\ (A_{ij}-B_i^TC_j)^2}$. We can find the ‘d’ value by minimizing the loss. Whichever value of ‘d’ gives the least amount of loss, that would be the optimal value.



Plot the values of ‘d’ on the ‘X’ axis and the error on the ‘Y’ axis. As the ‘d’ value increases, the error falls. Upto the value of d<10, the error is reducing fast, and after crossing the point d=10, the error is reducing slowly. So here d=10 is the inflection point, because we could see a huge change in the behaviour of the loss with the increasing number of ‘d’ values.

3) The other way to find out the ‘d’ value is by plotting the ‘d’ values on the ‘X’ axis and the rate of change of the loss/error (ie., slope) on the ‘Y’ axis, and picking the inflection point.

53.10 Matrix Factorization for Recommender Systems: Netflix Prize Solution

Let $\mathbf{r_{ui}}$ → Rating given by the user 'u' on the item 'i'.

$\mathbf{q_i}$ → Item Vector

$\mathbf{p_u}$ → User Vector

So the optimization problem is

$$\mathbf{min}_{q,p} \Sigma_{u,i} (r_{ui} - q_i^T p_u)^2 + \lambda(||q_i||^2 + ||p_u||^2)$$

In the above optimization problem,

$\mathbf{(r_{ui} - q_i^T p_u)^2}$ → Squared Loss

$\mathbf{(||q_i||^2 + ||p_u||^2)}$ → L_2 regularization to avoid overfitting

Solution 1

Using SGD, we have to compute $\partial L/\partial p_u$, $\partial L/\partial q_i$. But it takes more time, as there will be a huge number of users and items.

Solution 2 (Alternative Least Squares (ALS))

Step (a)

_____Assuming 'p_u' as constant and running the gradient descent to find the best 'q_i'.

Step(b)

Assuming 'q_i' as constant and running the gradient descent to find the best 'p_u'.

This way, we get the optimal values for both 'p_u' and 'q_i' by alternating. Both steps (a) and (b) are executed alternatively, till it converges to minima.

ALS is faster than SGD. Also ALS is a well suited algorithm when working with the least squares.

Let us assume 3 scalars. They are

b_u → Bias term for each user

b_i → Bias term for each item

μ → Mean term

The optimization problem now becomes

$$\mathbf{min}_{q,p,b}\ \Sigma_{u,i}(r_{ui}-\mu-b_u-b_i-p_u^T q_i)^2 + \lambda(||q_i||^2 + ||p_u||^2 + b_u^2 + b_i^2)$$

Where **μ** → Average ratings of all the users across all the items.
(ie., average of all the non empty cell ratings)
Here '**μ**' is a constant , whereas '$\mathbf{b_u}$' and '$\mathbf{b_i}$' are variables.

The main advantage of using Matrix Factorization in recommender systems is incorporation of problem specific changes is easy (like here we have incorporated the scalars '**μ**', '$\mathbf{b_u}$' and '$\mathbf{b_i}$'). This advantage is not directly available in item-item and user-user similarities.

Note

Refer to the research paper in the link given below
https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf

53.11 Cold Start Problem

In case if we have a new user 'u^l' coming into the ecosystem, then we do not have any ratings from this user, as he is new to the ecosystem. So this new user vector will have empty cells. Similarly if a new item 'i^l' is added to the inventory, then we do not have any ratings for that item. Here this new item vector will have empty cells. Such a problem is called a **cold-start** problem.

Case 1 (When we have a new user added in the ecosystem)

In this case, we recommend top items based on the metadata. For example, if the new user is from San Francisco, and is using an ipad with safari browser, then those items that are used/purchased by the people in the same area, with the same devices are recommended to this new user. This will be a content based recommendation system.

Case 2 (When we have a new item added to the inventory)

In this case, it checks to which category this item belongs to (like electronics, clothing, sports, provisions, etc), cost of the product, description of the item and then recommends this new item to those users who bought similar items from the same categories. This also will be a content based recommendation system.

These 2 cases come under a cold-start problem.

If a new user creates an account (or) if a new item is added to the inventory, for such users/items, the system doesn't have enough information to make a decision.

Cold-Start problem is that problem, where the system is not able to recommend the items to the users. For every recommender system, it is required to build a user profile by considering the user's activities and behaviours with the system. On the basis of the user's previous history, the system makes the decisions and recommends the items consequently.

If a new user has not rated some items and not yet viewed/visited some items, then it would be difficult for the system to build a model on that basis. The Cold-Start problem arises in 3 different situations.

a) For new users
b) For new items
c) For new community

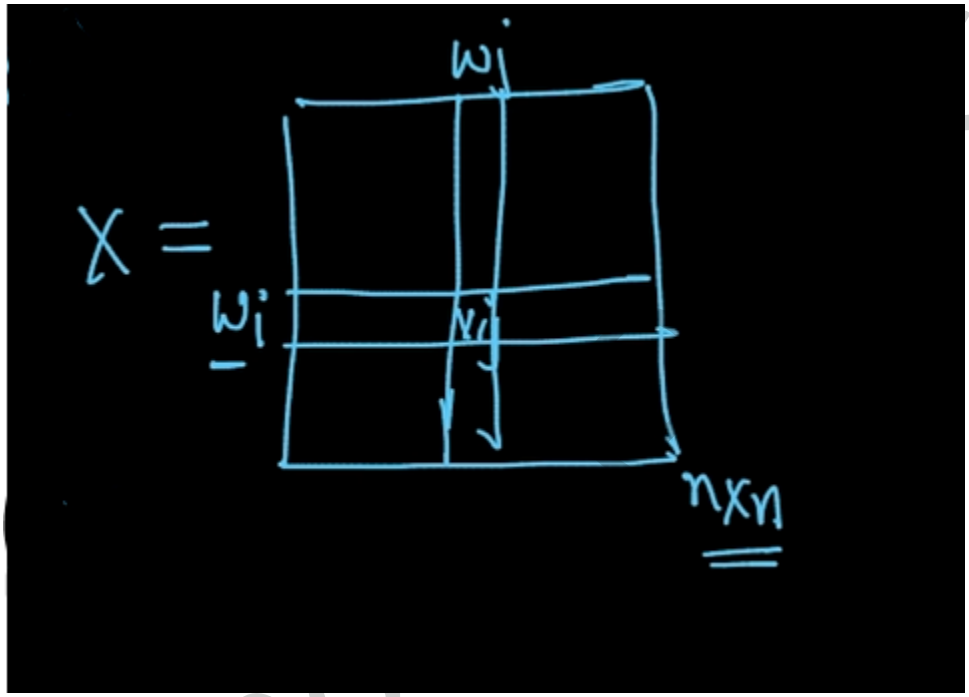
So till we get sufficient data to recommend the items, we use content based recommendations. Once after getting enough data, we can go for collaborative recommendations.

53.12 Word Vectors as MF

So far we have obtained Word Vectors from models like Word2Vec, GLoVe, PLSA, LDA, etc. Here we will use matrix factorization techniques like SVD to compute the word vectors.

1) Co-occurrence Matrix (In Text Data)

Let us denote the co-occurrence matrix by 'X'.



X_{ij} = number of times the word 'w_j' occurs in the context of 'w_i'. Let us assume we have a review 'r_{10}', and it consists of the words

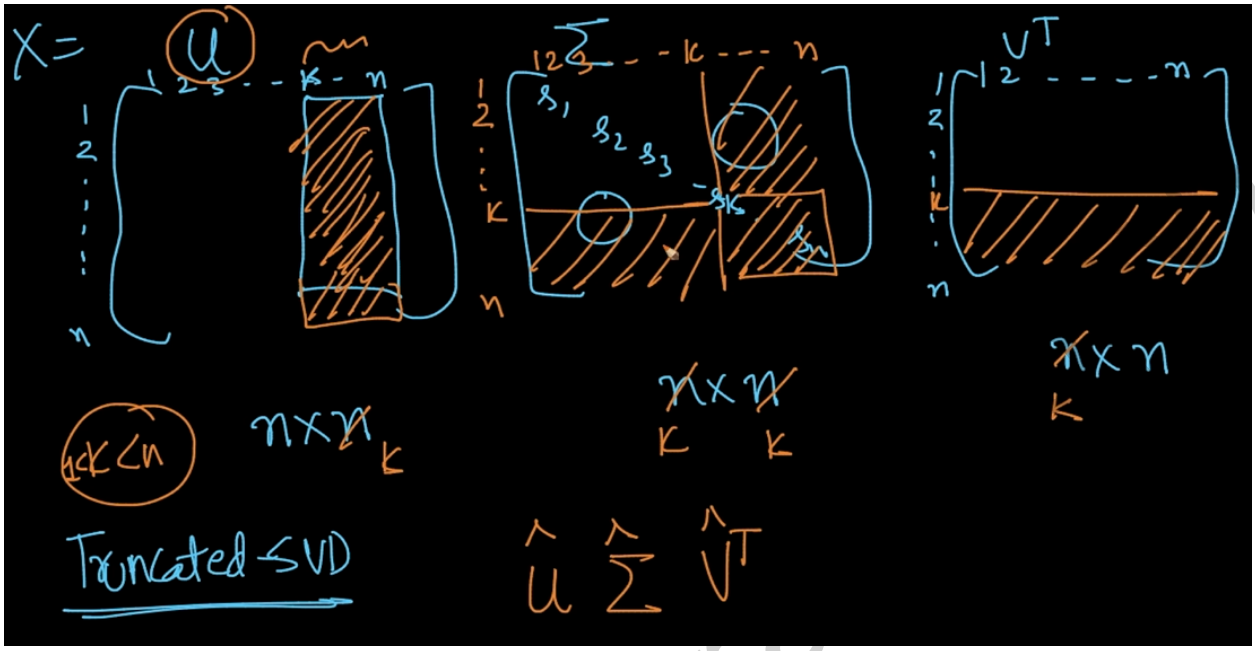
$r_{10} = w_6\ w_j\ w_1\ w_2\ w_i\ w_1\ w_3\ w_4$

We can say that the word 'w_j' occurs in the context of 'w_i', if 'w_i' and 'w_j' are within a neighborhood.

Let us assume the neighborhood distance of 10, then if 'wi' and 'wj' are separated by less than or equal to 10 words, then we can say that both these words are in the neighborhood of each other.

2) Decompose 'X' as $U\Sigma V^T$

$X_{nxn} = U_{nxn}\ \Sigma_{nxn}\ V^T_{nxn}$



Here now we are considering only the first 'K' columns from the matrix 'U', and are discarding the remaining 'n-K' columns. Let us denote it as 'U^' and its shape is **nxk**. Here the matrix 'U^' denotes the top left 'K' singular vectors.

We are considering only the first 'K' rows and first 'K' columns from the matrix 'Σ' and discard the remaining 'n-K' rows and columns. Let us denote it as 'Σ^' and its shape is **kxk**. Here the matrix 'Σ^' denotes the top 'K' singular values in diagonal order.

Similarly we are considering only the first 'K' rows from the matrix 'V^T' and are discarding the remaining 'n-K' rows. Let us denote it as 'V^{T}^' and its shape is **kxn**. Here the matrix 'V^{T}^' denotes the top right 'K' singular vectors.

In PCA, we take the top 'K' eigenvalues and the top 'K' eigenvectors. In TruncatedSVD, we take the top 'K' singular values, top 'K' left singular vectors, and the top 'K' right singular vectors.

So in TruncatedSVD(K) → 'K' is a hyperparameter

$X_{nxn} \rightarrow \hat{U}_{nxk}\ \hat{\Sigma}_{kxk}\ \hat{V^T}_{kxn}$

3) Take the rows of matrix $U^{\wedge}$ as u_1, u_2, u_3, …, u_n. Each row 'u_i' is the vector of the word 'w_i'.

The problem with the combined usage of Cooccurrence matrix + TruncatedSVD is that, if the number of words is more in co-occurrence matrix is large, then applying TruncatedSVD becomes highly expensive.

One hack to handle this kind of problem is that instead of using all the words, it is recommended to use the top 'm' words. The top 'm' words are picked on the basis of the IDF scores in TF-IDF.

53.13 Eigen-Faces

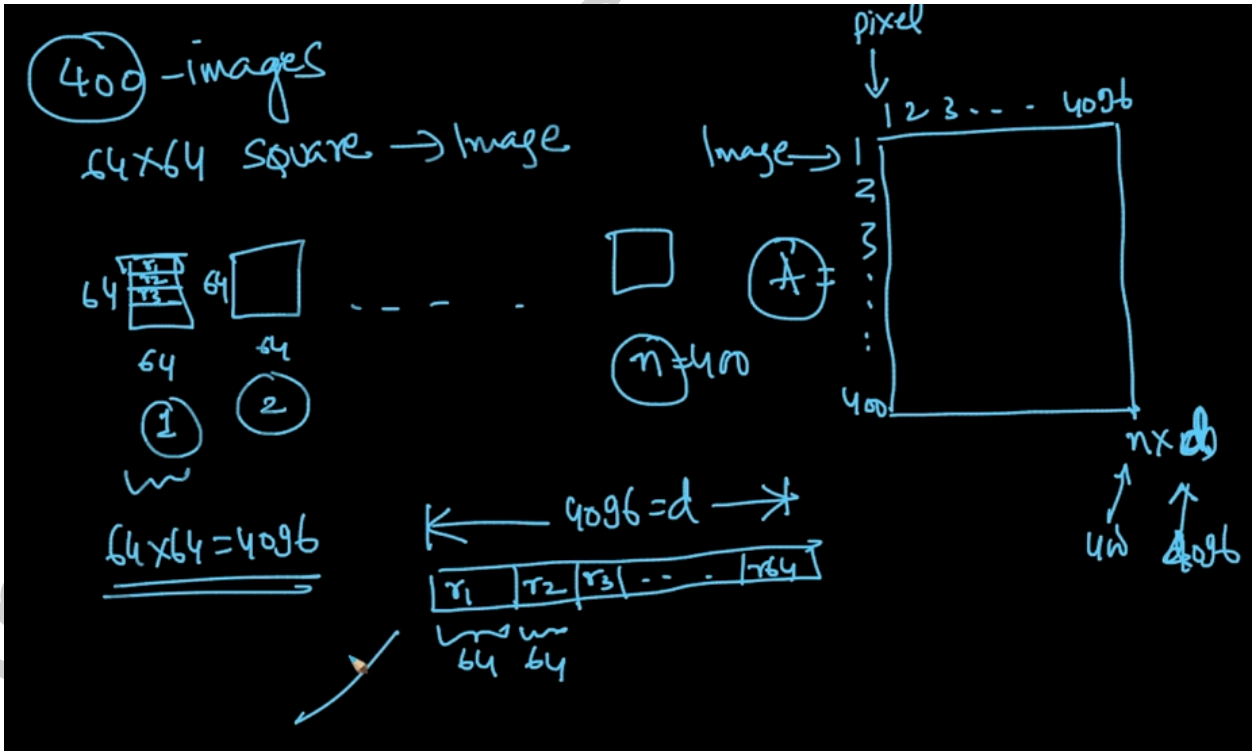
So far, for Word Vector (using co-occurrence matrix), we had used matrix factorization (using SVD). Similarly for the image dtaa, we have a technique called **EigenFaces** (using PCA). This is a very old technique, and today we use the **Convolutional Neural Networks (CNN)**.

Let us assume we have 400 images, and each image is of the shape 64x64 squares. Then for each image, the number of pixels = 64*64 = 4096. As we get the information for every image in a format of 4096 pixels, we have to flatten this pixel matrix into an array of 4096 dimensions. Now this pixel data is represented as a 4096 dimensional vector for every image.

As each row has 64 pixels, and there are 64 rows, then these rows should be concatenated, and we get a 4096 dimensional vector.

Step 1

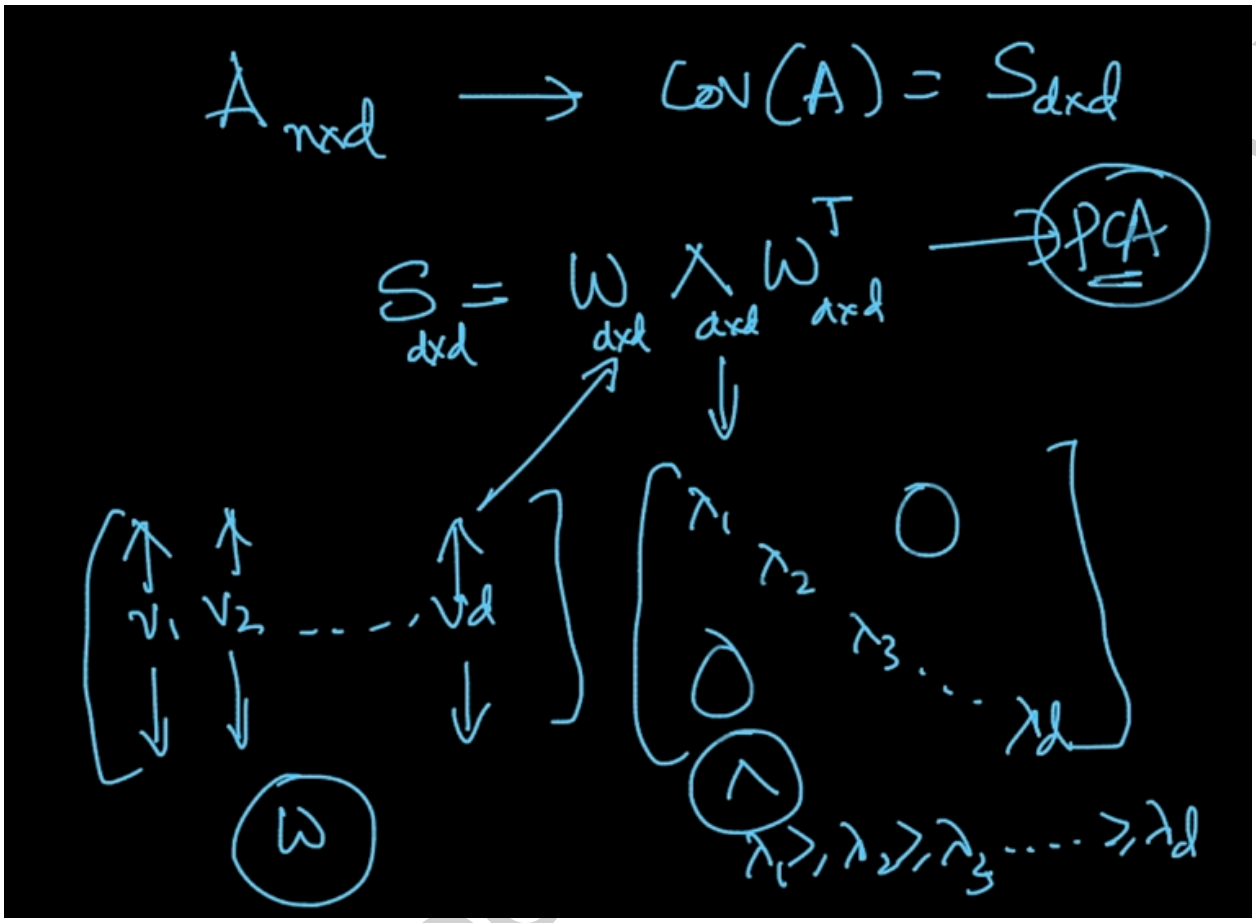
So now we have to construct a data matrix for all these 400 images. Let us denote it as $\mathbf{A}_{nxd}$. (here n = 400, d = 4096)



For the data matrix '$\mathbf{A}_{nxd}$', we have to compute the covariance matrix '$\mathbf{S}_{dxd}$'.

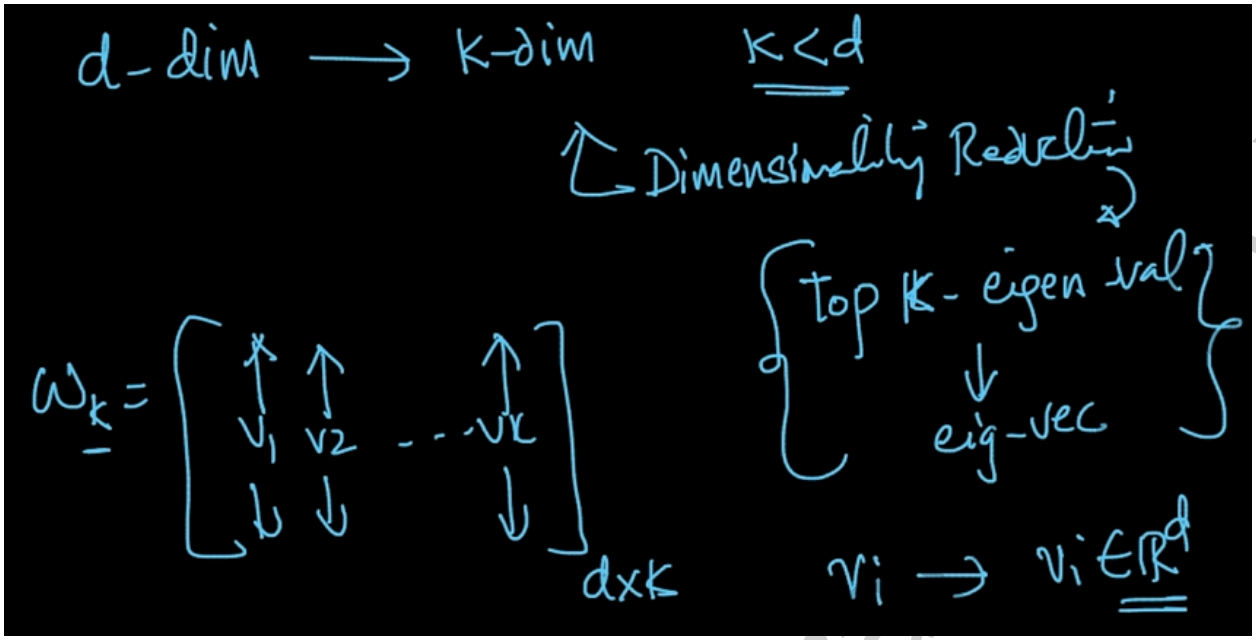
Step 2

$S_{dxd} = W_{dxd} \Lambda_{dxd} W^{T}_{dxd}$



Step 3

Let us now reduce the dimensionality from ‘d’ to ‘k’ to pick the top ‘k’ eigenvalues and eigenvectors. (here k<d).



Step 4

Now in order to get our images in k-dimensional space, we have to multiply '$\mathbf{A}_{nxd}$' with '$\mathbf{W}^{k}_{dxk}$' Let us denote the product as '$\mathbf{\tilde{A}}_{nxk}$'

$$\mathbf{A}_{nxd}\,\mathbf{W}^{k}_{dxk} = \mathbf{\tilde{A}}_{nxk}$$

This '$\mathbf{\tilde{A}}_{nxk}$' denotes the k-dimensional representation of the images.

Note

As the video lecture 53.14 is based on the code sample discussion, we are not providing any notes for it. We suggest you to go through the video lecture and the reference link for it is given below. For any queries, you can post them on the comments wall (or) you can mail us at mentors.diploma@appliedroots.com

Reference Links for Video Lectures 53.13 and 53.14

http://scikit-learn.org/stable/auto_examples/decomposition/plot_faces_decomposition.html#sphx-glr-auto-examples-decomposition-plot-faces-decomposition-py

https://github.com/sumitjha4321/Notebooks/blob/master/Notebooks/EigenFaces.ipynb

http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.NMF.html

http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html

http://scikit-learn.org/stable/auto_examples/decomposition/plot_faces_decomposition.html#sphx-glr-auto-examples-decomposition-plot-faces-decomposition-py