Recommender Systems

Nishanth Nair

April 09, 2020

Motivation

- Problem definition
  - To predict whether (or how much) a given person will like a given item
- Broad range of applications
  - Movie ratings
  - Amazon product recommendations
  - iTunes music suggestions
  - Friend recommendations
- Collaborative Filtering
  - Bipartite graph: two types of nodes, edges join nodes of different types
  - Predictions based on preferences of similar people
  - Insight: preferences are correlated, people are people

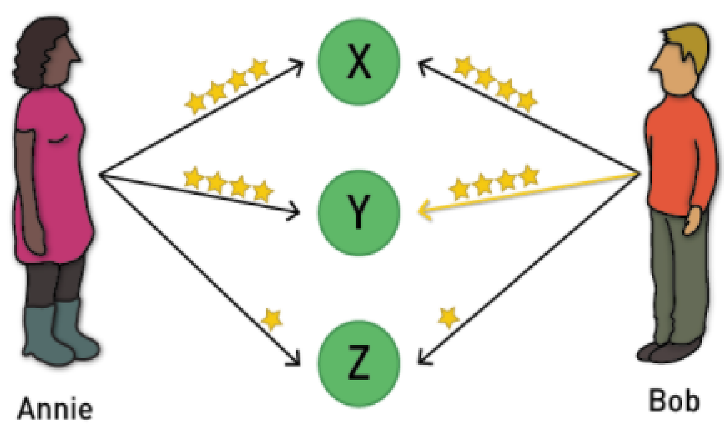


Figure 1: example

- Recommender systems can help you learn features

Types of recommendation systems

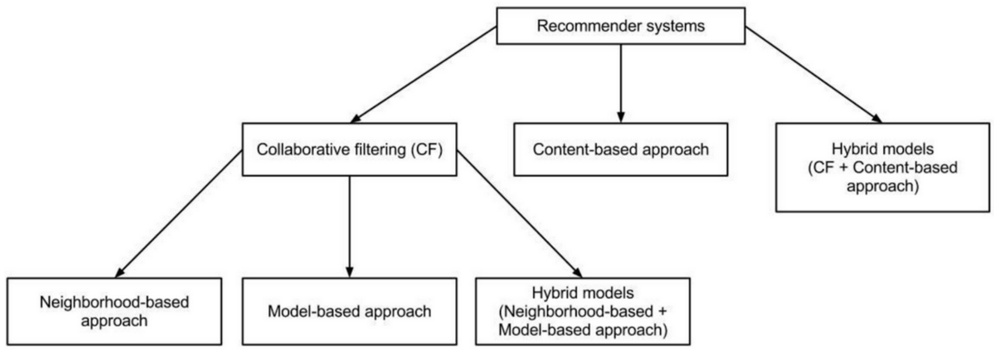


Figure 2: example

Content based recommmendations

- Look at general attributes for each item,
- try to determine whether a person likes those attributes, and
- then predict unknown ratings based on known attributes
  - Does Annie like suspense movies?
  - Does Annie like movies with Leonardo di Caprio?
  - Et cetera
- Given known features of each movie
- And given knowledge of how users rate these movies
- We can develop a model that predicts how a user will rate an unseen movie, given features of the movie
- Formally:
  - Given,
    * Features $x^m = (x_1^m, x_2^m, ..., x_k^m)$ are k features for an item m
    * Rating Y_{im} is a rating for item m from user i
  - We need to build a model such that

$$Y_{im} = f_i(x^m)$$

  - $f_i(x)$ is a model specific to each person.
  - $\hat{Y_{im}}$ would be the predicted rating for item m for user i
- Any classifier or linear regression model can be used. i.e.

$$Y_{im} = \beta_0 + \beta_{i1}x_1^m + ... + \beta_{ik}x_k^m$$

- Limitations
  - Sparse data: Many movies and features but very few available ratings
  - People's tastes are generally not that straightforward
    - Difficult and time consuming to get very content specific features.
    - Some features may be intangible, subjective, complex
  - Difficult to determine what features to use.

Collaborative Filtering

- Collaborative filtering is commonly used for recommender systems.
- It relies only on past user behavior.
  - for example, previous transactions or product ratings without requiring the creation of explicit profiles.
- This approach is known as collaborative filtering, a term coined by the developers of Tapestry, the first recommendersystem.
- Collaborative filtering analyzes relationships between users and interdependencies among products to identify new user item associations.
- These techniques aim to fill in the missing entries of a user-item association (eg. user movie rating)

Nearest Neighbor CF

Overview

- Based on the following intiution:
  - Similar people rate same item similarly (user-based CF).
  - Same person rates similar items similarly (item-based CF).
- Below is an example to make the idea more concrete. Below is a table of Ratings given by different users for ddifferent movies(Y).

Movie	Y_1	Y_2	Y_3	Y_4	Y_5
Annie	4	5	5	-	4
Bob	4	-	5	2	4
Carol	-	4	1	-	1
Dave	3	-	-	5	3

Figure 3: example

- **User based CF**
  - Lets say we want to determone Annie's rating for movie 4 (Y_4)
  - Look for people similar to Annie. Here, Bob seems similar, use a rating similar to Bob for Annie.

Movie	Y_1	Y_2	Y_3	Y_4	Y_5
Annie	4	5	5	-	4
Bob	4	-	5	2	4
Carol	-	4	1	-	1
Dave	3	-	-	5	3

Movie	Y_1	Y_2	Y_3	Y_4	Y_5
Annie	4	5	5	2	4
Bob	4	-	5	2	4
Carol	-	4	1	-	1
Dave	3	-	-	5	3

Figure 4: User Based CF example

- **Item based CF**
  - Lets say we want to determone Carols's rating for movie 1 (Y_1)
  - Look for mmovvies with ratings similar to movvie 1. Here, Movie 5 seems similar, so use its corresponding rating for Carol.

Movie	Y_1	Y_2	Y_3	Y_4	Y_5
Annie	4	5	5	2	4
Bob	4	-	5	2	4
Carol	-	4	1	-	1
Dave	3	-	-	5	3

Movie	Y_1	Y_2	Y_3	Y_4	Y_5
Annie	4	5	5	2	4
Bob	4	-	5	2	4
Carol	1	4	1	-	1
Dave	3	-	-	5	3

Figure 5: Item Based CF example

- For the sake of understanding, we can think of the predicted rating for user i for movie m to be the weighted sum of the neighbors ratings for movie m.

$$\hat{Y}_{im} = \sum_{j \in N_i} W_{ij}.(Y_{jm})$$

- The weight, W_{ij} can be computed as the Pearson correlation coefficient (r)

$$W_{ij} = r = \frac{\sum_m (Y_{im} - \bar{Y_i})(Y_{jm} - \bar{Y_j})}{\sqrt{\sum_m (Y_{im} - \bar{Y_i})^2.(Y_{jm} - \bar{Y_j})^2}}$$

- The above simplistic approach doesn't take into account peoples baseline prefences.For example,
  - Alice could always rate a movie lower (and give a 2 instead of 4 or 1 instead of 3) and Bob might tend to always rate them higher.
  - By considering individual baselines and normalizing, we get a much better prediction.

$$\hat{Y}_{im} = \bar{Y}_i + \alpha. \sum_{j \in N_i} W_{ij}.(Y_{jm} - \bar{Y}_j)$$

- The above equation adds to i's baseline rating and it also reduces j's baseline to reduce bias.
- There many potential extensions to computing the above.
  - Removing bias from ratings
    - Users mean rating(depicted in the equation above)
    - Global mean rating
    - Items mean rating
    - Items mean rating + users deviation from items mean rating
    - etc..
  - Measuring similarity between people or items
    - Pearson correlation coefficient (described earlier)
    - Cosine similarity.
    - Inverse Euclidean (or other) distance.
    - Note: These metrics assume complete data. In practice, can only compute over set of items rated by both users.

Latent factor Model based CF

Overview

- In the content based approach, we assume we have the features and ratings and we compute the predictive weights of the model.
- Assume that we know the ratings and the predictive weights but not the features. Can we determine the features?
- Formally,
  - In the content based approach, we are given Y_{im} and x_k^m and we compute β_{ik}

$$Y_{im} = \beta_0 + \beta_{i1} x_1^m + ... + \beta_{ik} x_k^m$$

  - Now, we are given β and Y_{im} and we need to compute x_k^m

$$Y_{im} = \gamma_0 + x_1^m \hat{\beta_{i1}} + ... + x_k^m \hat{\beta_{ik}}$$

- This what model based collaborative filtering uses to approach the recommendations problem.
  - Given features, we can learn user preferences i.e β.
  - Given user preferences, we can learn features i.e x.
- In this approach, we can "guess" (randomly initialize) our preference parameters and use that to estimate features and then use estimated features to estimate parameters and so on.

$$\text{Guess } \beta \implies x \implies \beta...$$

Approaches

- Many approaches can be taken to solve this
  - Probabilistic
  - Rule based
  - Classification
  - Regression
  - Matrix factorization
  - etc..

Matrix Factorization (Low Dimensional Rank Factorization)

- Recommender systems rely on different types of input data, which are often placed in a matrix with one dimension representing users and the other dimension representing items of interest.
- The values in the matrix are called **latent factors**
- Matrix factorization models map both users and items to a joint latent factor space of dimensionality f, such that user item interactions are modeled as inner products in that space.
- The intiution for this is as follows:

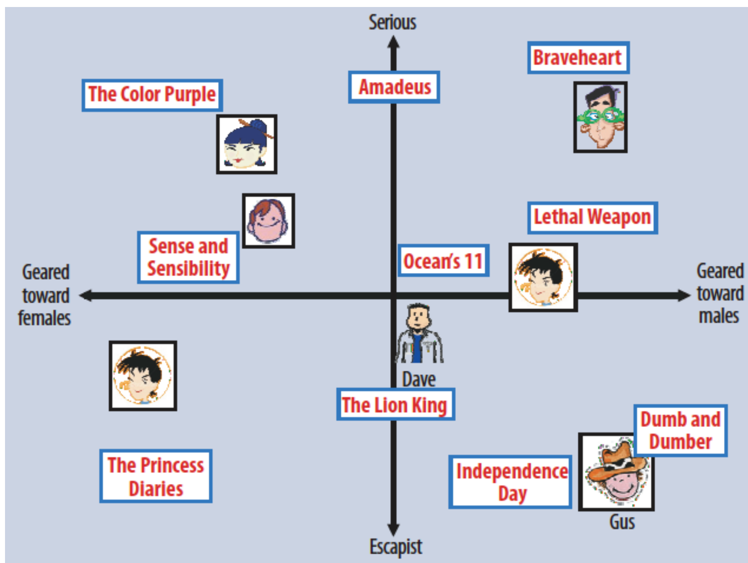


Figure 6: example

- Assume we have a vector V of movies that are characterized on a scale from serious to escapist.
  - Assume we have a vector U of users that characterizes users as male or female and their corresponding interest in movies characcterised from seriouss to escapist.
  - The resulting dor product of U and V gives us a matrix R, of the users interest or rating for a specific movie.
  - Matrix factorization reverses this. i.e, given a sparse matrix R (many entries are empty), it attempts to determine U and V

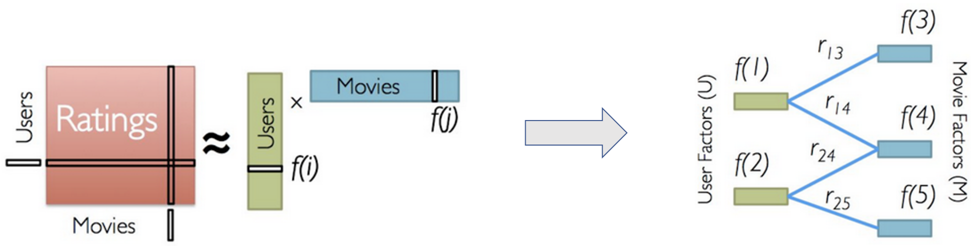


Figure 7: matrix factorization

- The primary challenge is the large number of missing values in R
- Earlier systems relied on imputation to fill in missing ratings and make the rating matrix dense.
  - This is very expensive and significantly increases amount of data.
  - Inaccurate imputation distorts the data significanlty.

Problem overview

- Given a matrix R, find lower rank matrices U and V to aprroximate R such that

$$R \approx U^T.V$$

- To learn factor vectors, we need to minize the objective (or loss) function which is the Regularized Square Error:

$$L(U,V) = (R - U^T.V)^2 + \lambda(||V||^2 + ||U||^2)$$

Solution Approaches

- Stochastic gradient descent
  - Easy to implement
  - Computation is fast
- Alternating Least Squares
  - More favorable approach
  - Allows for parallelization
  - For systems centered around implicit data.
    - data is not considered sparse, gradient descent maynot be practical.
    - ALS handles this efficiently.

Alternating Least Squares (ALS)

- Given the objective function,

$$L(U,V) = (R - U^T.V)^2 + \lambda(||V||^2 + ||U||^2)$$

- Can we jointly optimize U and V?
  - No, because the loss funcction is not convex.Both U and V are unknown.
  - We know, for ridge regression, the objectivve function takes the form:

$$L = (Y - \theta^T.X)^2 + \lambda.||\theta||^2$$

  - Based on this, we can break our loss function as follows and solve as 2 sub problems:

$$L(U,V) = (R - U^T.V)^2 + \lambda(||V||^2) + \lambda(||U||^2)$$

Problem 1: Given V, solve - $L(U) = (R - U^T.V)^2 + \lambda(||U||^2)$

Problem 2: Given U, solve - $L(V) = (R^T - V^T.U)^2 + \lambda(||V||^2)$

ALS Algorithm

- STEP 0: Intialize random U_0 and V_0
- STEP 1: Minimize L(U). Fix V and determine U.

$$L(U) = (R - U^T.V)^2 + \lambda(||U||^2)$$

  - Closed form solution:

$$U = (V.V^T + \lambda I)^{-1}.(V.R^T)$$
$$U^* = U^T = R.V_0^T.(V_0^T.V_0 + \lambda I)^{-1}$$

- STEP 2:Minimize L(V). Fix U and determine V

$$L(V) = (R^T - V^T.U)^2 + \lambda(||V||^2)$$

  - Closed form solution:

$$V^* = (U.U^T + \lambda I)^{-1}.(U.R)$$
$$\implies V^* = ((U^*)^T.U^* + \lambda.I)^{-1}.((U^*)^T.R)$$

- STEP 3:Repeat 1 and 2 with updated U and V values until convergence.

Single Node ALS Algorithm

```
lambda_ = 0.1
n_factors = 100
m, n = R.shape
n_iterations = 20
U = 5 * np.random.rand(m, n_factors)
V = 5 * np.random.rand(n_factors, n)

def get_error(R, U, V, W):
    return np.sum((W * (R - np.dot(U,V)))**2)

errors = []
for ii in range(n_iterations):
    U = np.linalg.solve(np.dot(V, V.T)
                        + lambda_ * np.eye(n_factors),
                        np.dot(V, R.T)).T
    V = np.linalg.solve(np.dot(U.T, U)
                        + lambda_ * np.eye(n_factors),
                        np.dot(U.T, R))
    if ii % 100 == 0:
        print('{}th iteration is completed'.format(ii))
    errors.append(get_error(R, U, V, W))
R_hat = np.dot(U, V)
print('Error of rated movies:{}'.format(get_error(R, U, V, W)))
```

Weighted ALS Algorithm

- In regular ALS, Mean squared error is large because the optimization also includes errors for missing ratings.
- In the weighted version, include a ratings indicator, W having value 1 if user has rated and 0 if not.

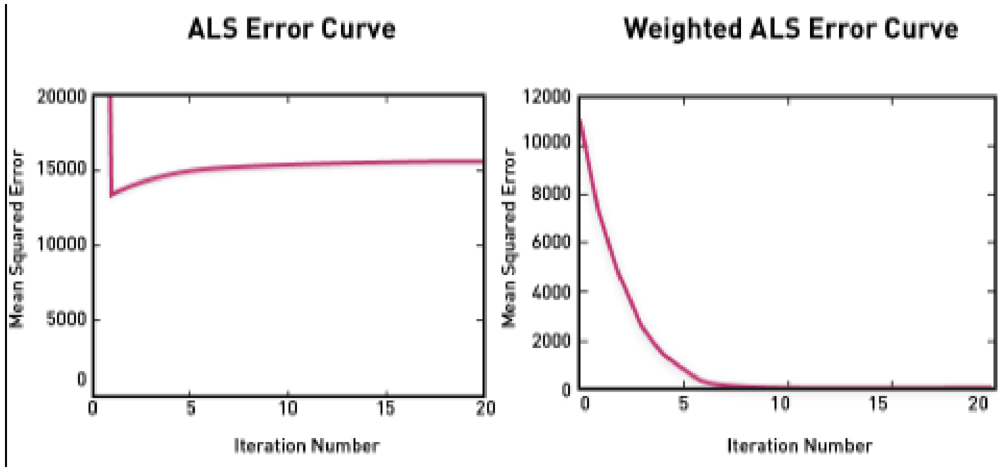


Figure 8: Regular ALS vs weighted ALS

- STEP 0: Intialize random U_0 and V_0
- STEP 1: Minimize L(U). Fix V and determine U.

$$L(U) = (R - U^T.VW)^2 + \lambda(||U||^2)$$

`- Closed form solution:`

$$U^* = R.WV_0^T.(V_0^T.WV_0 + \lambda I)^{-1}$$

- STEP 2:Minimize L(V). Fix U and determine V

$$L(V) = (R^T - V^T.WU)^2 + \lambda(||V||^2)$$

  - Closed form solution:

$$\implies V^* = ((WU^*)^T.U^* + \lambda.I)^{-1}.((WU^*)^T.R)$$

- STEP 3:Repeat 1 and 2 with updated U and V values until convergence.

Single Node Weighted ALS Algorithm

```
weighted_errors = []
for ii in range(n_iterations):
  print('{}th iteration begins'.format(ii))

  for u, Wu in enumerate(W):
    U[u] = np.linalg.solve(np.dot(V, np.dot(np.diag(Wu), V.T))
                           + lambda_ * np.eye(n_factors),
                      np.dot(V, np.dot(np.diag(Wu), R[u].T))).T

  for i, Wi in enumerate(W.T):
    V[:,i] = np.linalg.solve(np.dot(U.T, np.dot(np.diag(Wi), U))
                             + lambda_ * np.eye(n_factors),
                      np.dot(U.T, np.dot(np.diag(Wi), R[:, i])))

  weighted_errors.append(get_error(R, U, V, W))
  print('{}th iteration is completed'.format(ii))

weighted R hat = np.dot(X,Y)
```

Distributed ALS Algorithm

Distributed closed form solution

- The single node closed form solution described earlier is inefficient.
- We can further break down the problem to achieve parallelization in a single node.
- In the computation of U, each row of U is independent of the other and thus can be computed in parallel.
  - 1 row of U depends on the corresponding row of R and the entire V.
- Similarly.
  - 1 row of V depends on the corresponding column of R (or row of R^T) and the entire U.
- So now that we can parallelize, how do we distribute this.

STEP 1: Initialize with Random values.

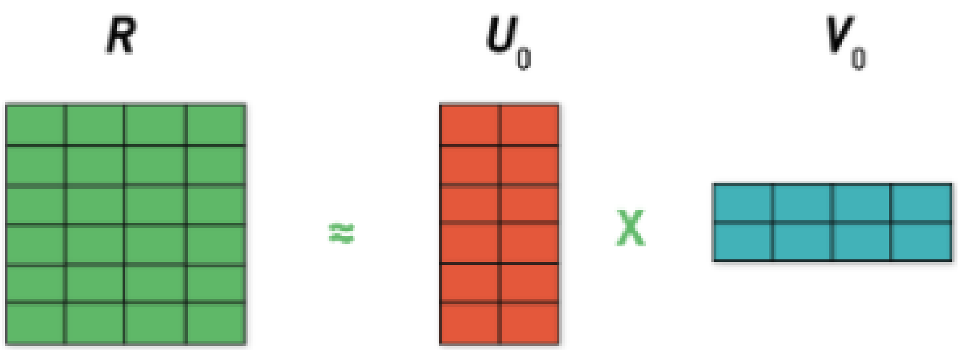


Figure 9: Distributed: init

STEP 2: Fix V and compute U

```
- Partition R by  rowID
- Broadcast V to all worker nodes
- Compute each row of U in the mappers using the closed form solution
```

$$U^* = RV^T(VV^T + \lambda.I)^{-1}$$

\- Combine all rows of U in the reducer

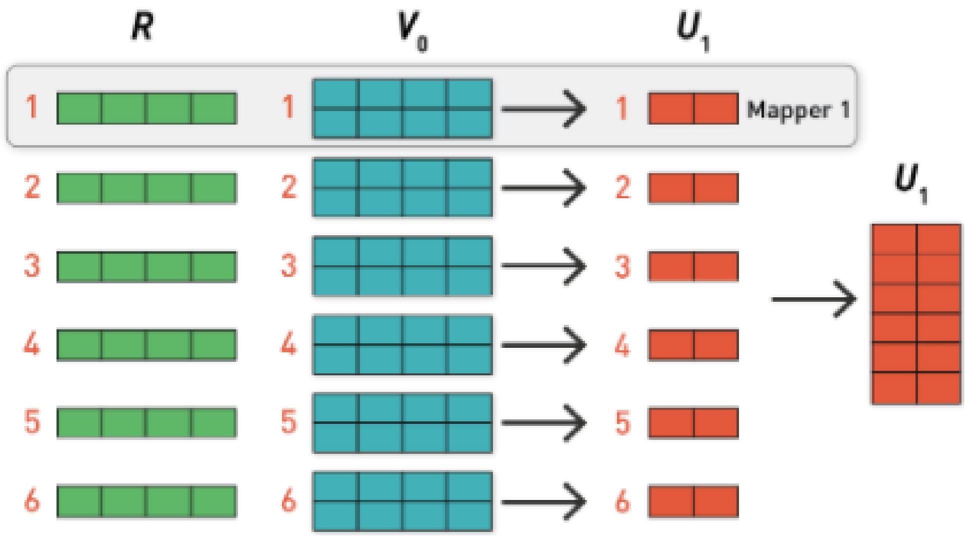


Figure 10: Distributed: compute U

STEP 3: Fix U and compute V

```
- Partition R by  columnID
- Broadcast $U^*$ to all worker nodes
- Compute each row of V in the mappers using the closed form solution
```

$$V^* = R^T U (U^T U + \lambda . I)^{-1}$$

- Combine all columns of V in the reducer

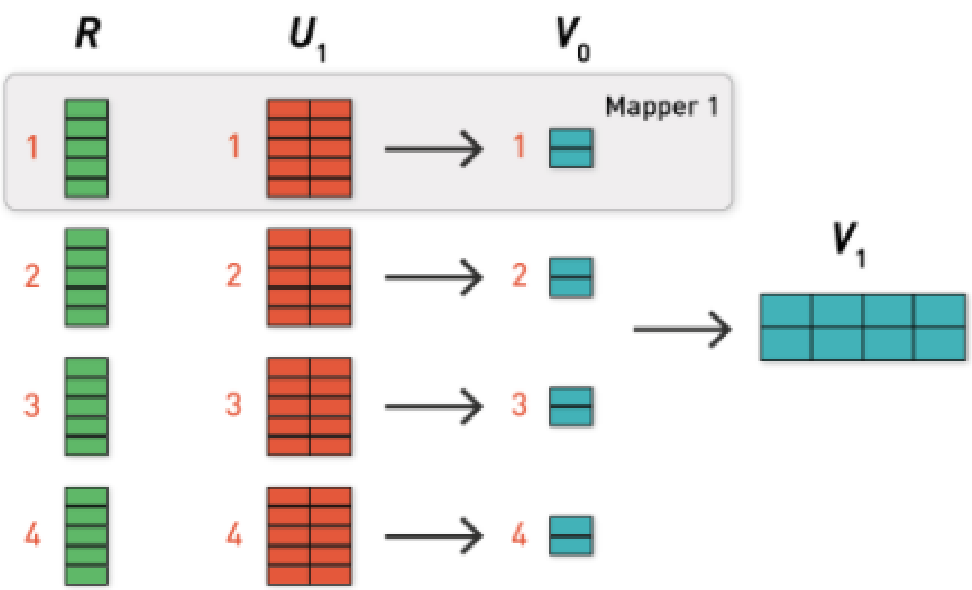


Figure 11: Distributed: compute V

STEP 4: Iterate steps 1 and 2 until convergence.

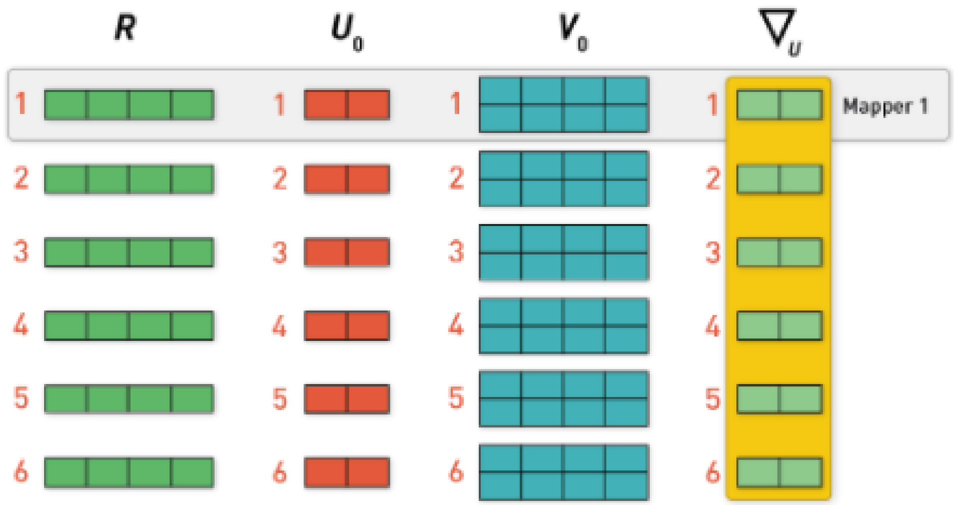
Implementation in spark

- Matrix R needs to be used in each iteration. It should be cached.
- R needs to be partitioned in two ways: by row and by column.
- Updated U and V need to be broadcast iteratively to each worker.
- Partition U or V in the same way as R.
- Operate on blocks to lower communication.
- For detailed implementation in spark, refer:

https://s3.amazonaws.com/static.datascience.berkeley.edu/DATASCI+W261+Machine+Learning+at+Scale/Week+14/14.7/ALS+in+Spark-MateiPaper.pdf

Distributed Gradient descent solution

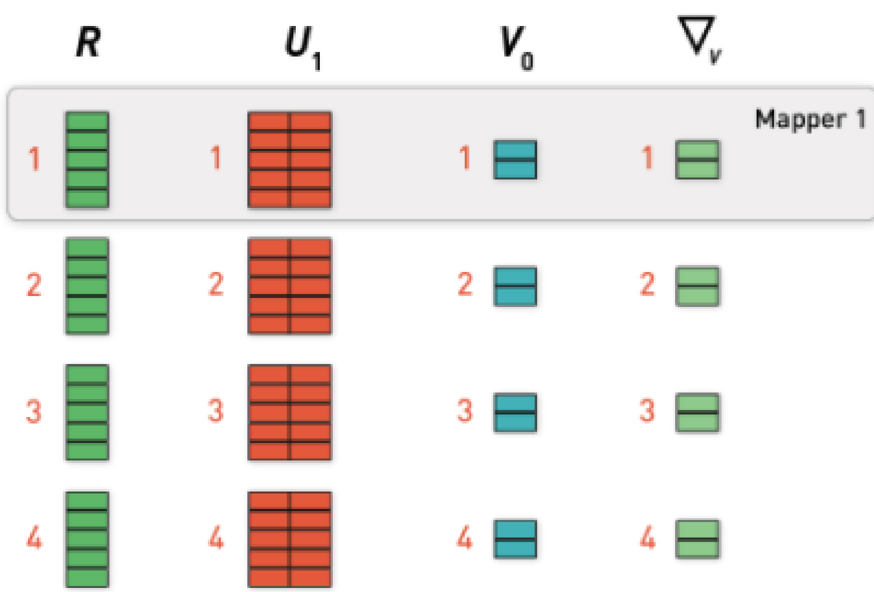
- If U and V are too big, the closed form solution won't work.
- Executors will crash.
- With gradient descent approach, we can parallelize computation to a cell level.
- In essence, we compute the gradient for each cell in the mappers and then combine the final values in the reducers.
- The price we pay is time to execute will be vvery large and a lot of network traffic would be generated.



Calculate gradient for each **partition**:

$$\nabla_U^i = 2 \cdot \left(R^i - U_0^i \cdot V_0^i\right) \cdot V_0^{i^T} + 2 \cdot \lambda \cdot U_0^i$$

Figure 12: Distributed: compute U



Calculate gradient of V for each partition:

$$\nabla_V^i = 2 \cdot \left(R^i - U_1^i \cdot V_0^i\right) \cdot U_1^{i^T} + 2 \cdot \lambda \cdot V_0^i$$

Figure 13: Distributed: compute V