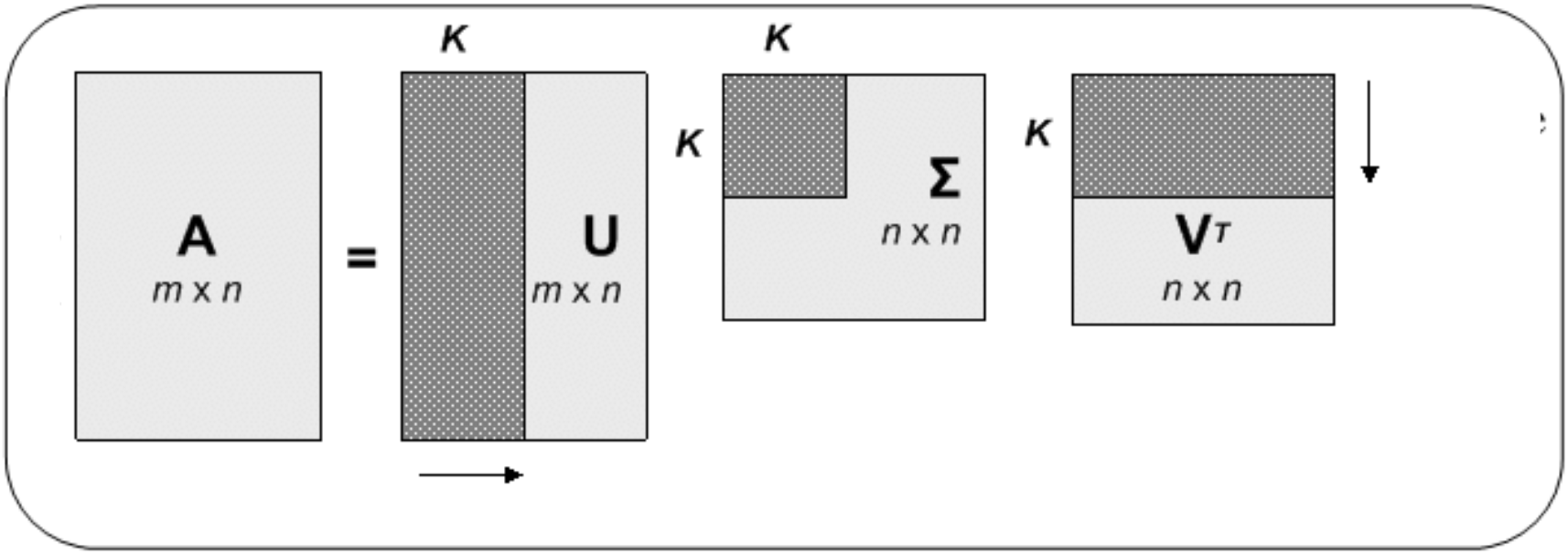
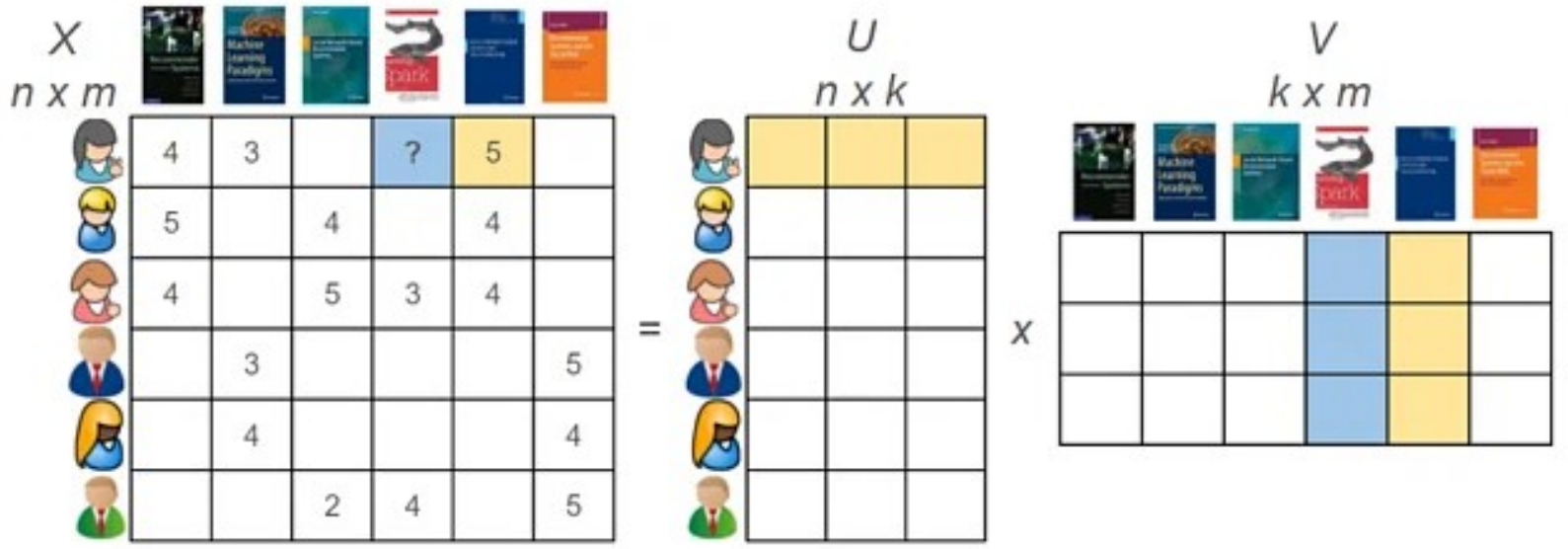
Recommender Systems

Singular Value Decomposition (SVD)

SVD



Truncated SVD in Rec Sys



How to make recommendation?

- Estimate U and V matrices
- Multiply them to estimate null values of the rating matrix
- For each user, sort the estimated values descendingly
- Recommend the items with top m largest estimated values

Challenges of Estimating U and V

- Missing values
- Huge matrix, memory issues
- Slow and costly matrix decomposition algorithms
- Can’t do batch learning; all data is needed for decomposition
- It’s an unsupervised problem
- No control over complexity (e.g. via regularization type tuning)
- Can’t take into account other interactions (such as implicit interactions)

An iterative algorithm

- Perform decomposition via loss-minimization

$$\min_{p,q} \sum_{i,j \in I_R} (r_{i,j} - p_i^T q_j)^2$$

where

- r_ij are ratings that user i gave to item j
- I_R is the set of tuples of row and column indecies from matrix R that is not null
- p's are rows of user matrix (U)
- q's are columns of item matrix (V)

Note how the issues in the previous page can be addresses.

```
n = np.nan
R = pd.DataFrame( [[5, n, 1, n , n, 2],
                   [1, n, n, 3 , n, n],
                   [1, 2, n, 3 , 2, n]],
                  columns = ['i1', 'i2', 'i3', 'i4', 'i5', 'i6'],
                  index=['u1', 'u2', 'u3'])
R
```

I_R = {(1, 1), (1, 3), (1, 6), (2, 1), (2, 4), .. (3, 5)}

Iterpretations

- p and q are user and item embeddings for each user and item on the k-dimentional space
- Inner product of p and q:
  - Larger values of inner product mean they’re closer to each other (corresponding to larger value of rating)
  - Smaller values mean they’re far apart (corresponding to smaller value of rating)

Regularization

$$\min_{p,q} \sum_{i,j \in I_R} (r_{i,j} - p_i^T q_j)^2 + \lambda(\|p\|^2 + \|q\|^2)$$

- Can control the complexity via controlling the size of p and q and the regularization hyperparameter
- The noise in p or q will not be increases/ propagated by keeping the size of the vectors small
- Extreme cases:
  - Lambda tends to infinity -> Nothing will be learned (low variance, large bias)
  - Lambda is zero -> Might overfit (not generalize well; large variance, low bias)

Training

1) Define the right matrices, and right dimensions, and define the model: outcome = p^tq
    Note the number of parameters in this model (=(n+m)*k)
2) Define the loss function (with the right regularization)
3) Define an optimizer such as gradient descent
4) Convert your huge data to batches of data with size like 32, 128, or 512, ...
5) Define the number of epochs
6) Put the model in the training mode (vs. Evaluation mode)
7) Put the matrices in GPU for faster matrix operations
8) Train the model:
    1) In each epoch, go through all iterations (batches of data) and repeat the following
    2) Equate
9) Visualize the loss through steps (iterations or epochs)

```python
from torch import nn

class SVD(nn.Module):
    def __init__(self, num_factors, num_users, num_items, device, **kwargs):
        super(SVD, self).__init__(**kwargs)
        self.device = device
        # plain MF params
        self.P = nn.Embedding(num_users, num_factors).to(self.device)
        self.Q = nn.Embedding(num_items, num_factors).to(self.device)

    def forward(self, user_id, item_id):

        P_u = self.P(user_id)
        Q_i = self.Q(item_id)

        outputs = (P_u * Q_i).sum(axis=1)

        return outputs.flatten()
```

Model, optimization, and hyperparameters

```
num_factors = 5
batch_size = 128

l2_reg = 1e-5
lr = 0.001
num_epochs = 100
```

```
model = SVD(num_factors, num_users, num_items, device)

loss_fn = nn.MSELoss(reduction='mean')

optimizer = torch.optim.Adam((param for param in model.parameters()
                              if param.requires_grad),
                             weight_decay=l2_reg,
                             lr=lr)
```

Training

```
for epoch in tqdm(range(num_epochs)):
    tr_rmse = 0

    model.train()
    for u, i, r in train_loader:

        u, i, r = u.to(device), i.to(device), r.to(device)

        optimizer.zero_grad()

        output = model(u, i)

        l = loss_fn(output, r)
        l.backward()
        optimizer.step()

        with torch.no_grad():
            tr_rmse += np.sqrt(loss_fn(output, r).cpu().numpy())
```

SVD in NLP

- We have a huge corpus of |S| sentences or reviews
- Need to vectorize sentences
- Create a sentence-word matrix:
  - Define vocabulary V, with size |V|
  - For each sentece, we create a vector of size |V|, all zero except certain indices get the frequency of words in that sentence
  - Now we have a huge matrix with |S| rows and |V| columns
  - Rows are sentences and columns are words and cell values are freqeuncy of words in sentences
- Now apply SVD to get a vectorization for each sentence with in a lower dimensional space, say k
- The cell values can be zero-one or TF-IDF values

gensim

- Gensim is python library for NLP
- LSI, NFM, LDA, efficient LDA, word2vec, doc2vec
- https://radimrehurek.com/gensim/models/lsimodel.html

Attentive Asymmetric SVD

- Reduce number of parameters
- Define a person based their attention to items – like a steriotype

$$\min_{p,q} \sum_{i,j \in I_R} (r_{i,j} - p_i^T q_j)^2 + \lambda(\|p\|^2 + \|q\|^2)$$

where I_R is the set of tuples of indices (k, l) where user k has given item l feedback/review/etc.

Matrix factorization approaches that avoid explicitly parametizing users have also been proposed. The size of the item set is typically much smaller than the size of the user set so this drastically reduces the amount of parameters. An example of one such approach is the Asymmetric-SVD which represents users as combination of item latent factors q_j weighted by coefficients a_k^l:

$$p_u = \sum_{u,k \in I_R} a_k . q_k = \sum_{k=1}^{N} a_k . q_k . I_{uk}$$

where . is the element wise product. Given this model, the loss function becomes

$$\min_{p,q} \sum_{i,j \in I_R} (r_{i,j} - [\sum_{i,k \in I_R} a_k . q_k]^T q_j)^2 + \lambda(\|a\|^2 + \|q\|^2)$$

SVD ++

- Remove constant bias across users, items and user-item interactions
- Address the limitation of SVD that can’t take implicit interactions into account

SVD++ estimates the rank of an item i given by user u as:

- Model definition

$$\hat{\mathbf{R}}_{ui} = \mu + b_u + b_i + \mathbf{q}_i^T \left(\mathbf{p}_u + |N(u)|^{-1/2} \sum_{j \in N(u)} \mathbf{y}_j \right)$$

In which:

- $\hat{\mathbf{R}}_{ui} \in \mathbb{R}$: predicted rating user u gives to item i
- $\mu \in \mathbb{R}$: global bias
- $b_u \in \mathbb{R}$: user bias
- $b_i \in \mathbb{R}$: item bias
- $\mathbf{q}_i \in \mathbb{R}^{1 \text{x} k}$: i^{th} row of $\mathbf{Q}$
- $\mathbf{p}_u \in \mathbb{R}^{1 \text{x} k}$: u^{th} row of $\mathbf{P}$
- $N(u) = \{i : | \mathbf{R}_{ui} \text{ is known}\}$: all items for which user u provided a rating
- $\mathbf{y}_j \in \mathbb{R}^{1 \text{x} k}$: another item latent vector for implicit feedback

- where,
  - $k << m, n$: latent factor size
  - $\hat{\mathbf{R}} \in \mathbb{R}^{m \text{x} n}$: predicted rating matrix
  - $\mathbf{P} \in \mathbb{R}^{m \text{x} k}$: user latent matrix
  - $\mathbf{Q} \in \mathbb{R}^{k \text{x} n}$: item latent matrix

- This estimation is found by minimizing the following Objective function:

$$\underset{\mathbf{P},\mathbf{Q},\mathbf{y}_*,b}{\operatorname{argmin}} \sum_{(u,i) \in \mathcal{K}} \|\mathbf{R}_{ui} - \hat{\mathbf{R}}_{ui}\|^2 + \lambda(\|\mathbf{P}\|_F^2 + \|\mathbf{Q}\|_F^2 + b_u^2 + b_i^2 + \sum_{j \in N(u)} \|\mathbf{y}_j\|^2)$$

  Where:
  - λ: regularization rate (hyper-param)
  - $\mathcal{K} = \{(u, i) \mid \mathbf{R}_{ui} \text{ is known}\}$: The (u, i) pairs for which $\mathbf{R}_{ui}$ is known and stored in the set