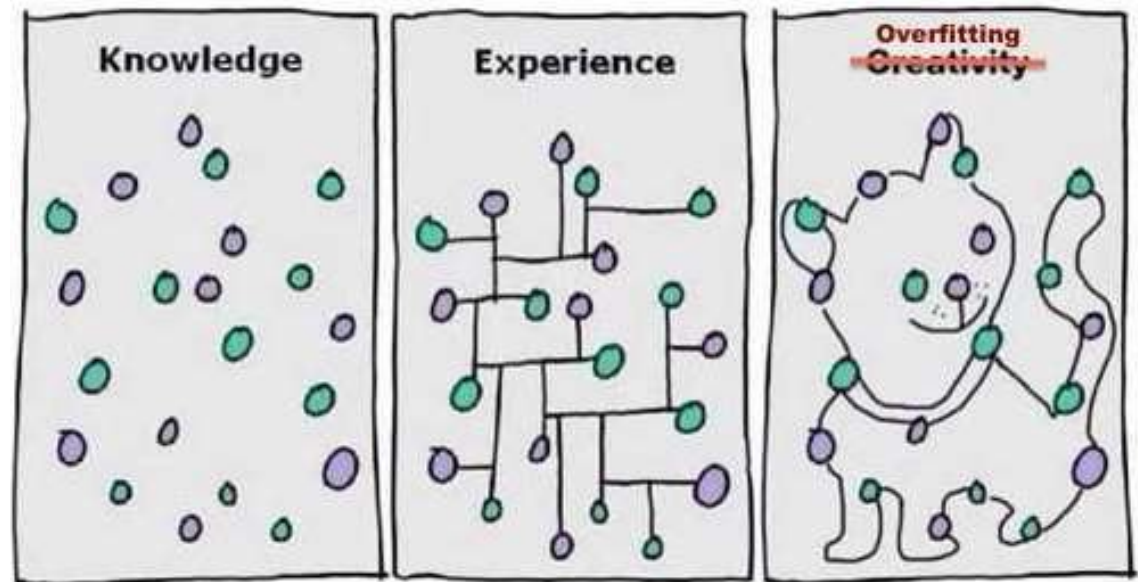


Graduate Certificate in Big Data Analytics

Recommender Systems

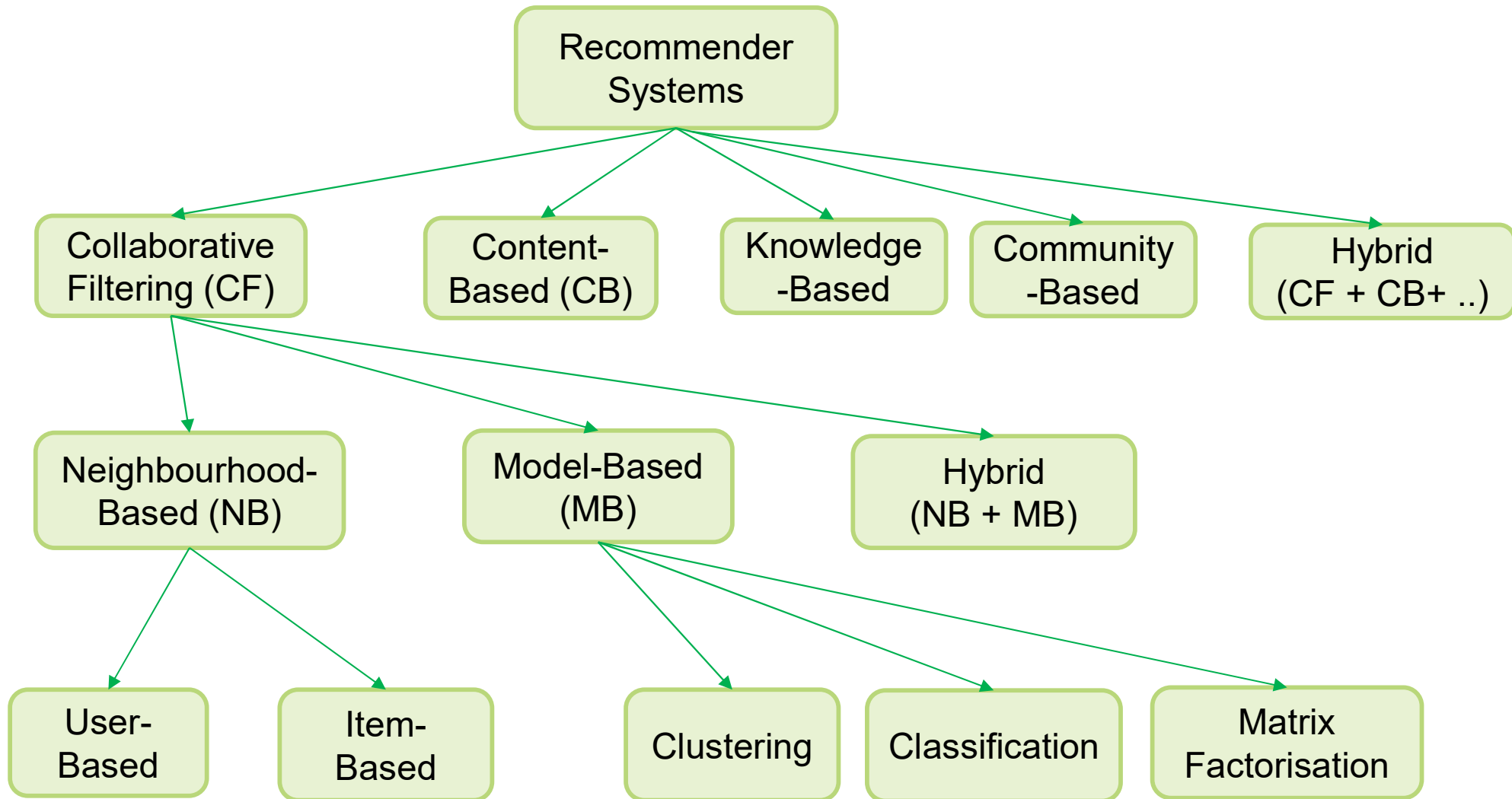
Model-Based Methods and Matrix Factorisation

Dr. Barry Shepherd
Institute of Systems Science
National University of Singapore
Email: barryshepherd@nus.edu.sg



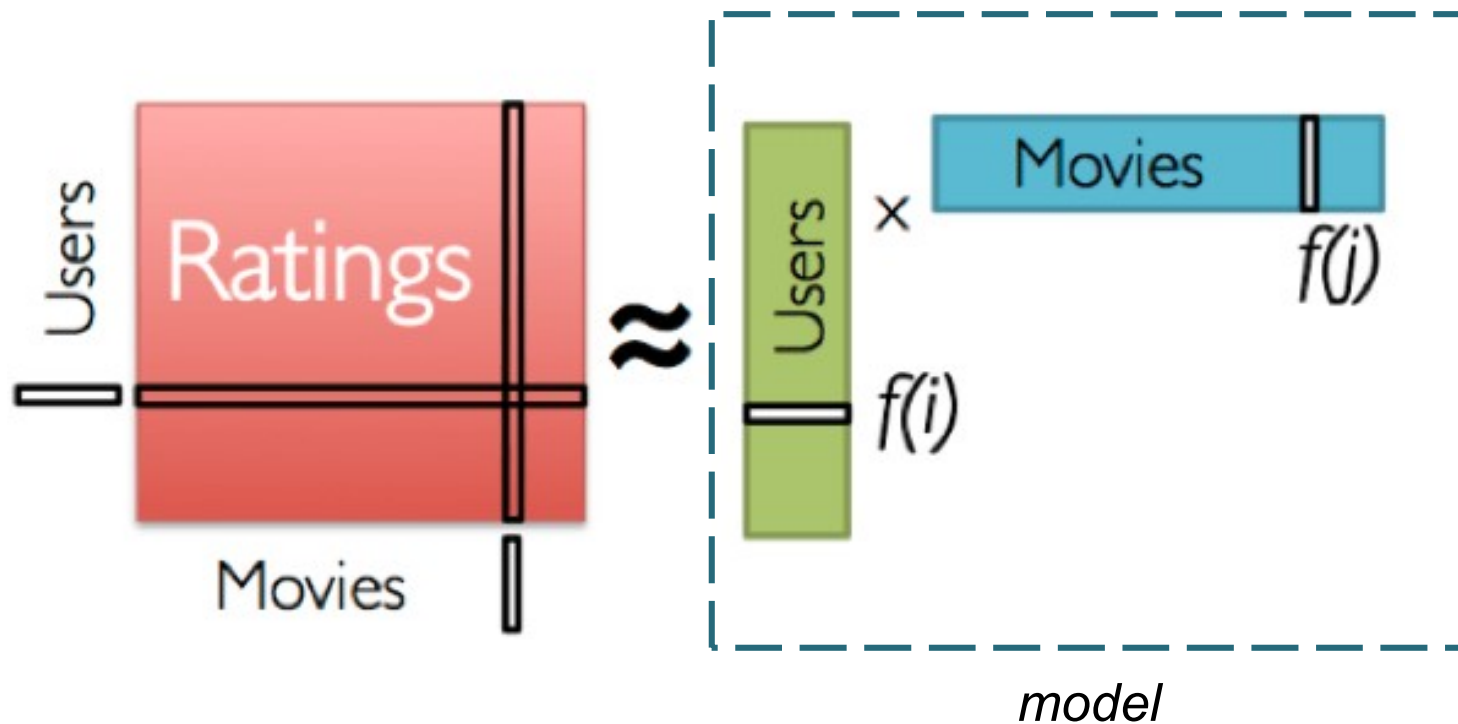
© 2023 NUS. The contents contained in this document may not be reproduced in any form or by any means, without the written permission of ISS, NUS, other than for the purpose for which it has been supplied.

Recommender System Approaches



Matrix Factorisation and CF

- Matrix factorisation can be applied to the ratings matrix
- Converts it into two smaller matrices which can be thought of as a model.
- Once we have done the factorisation we make rating predictions using the two factor matrices – not the original ratings matrix



Matrix Factorisation Example

- Assume the items to be recommended have a set of properties.
E.g. Movies could have properties: type, actors, directors, film studios, location, length etc. Properties are numerical strengths, e.g. values 0 (low) to 10 (high)

	Length	Disney	Universal	Action	Comedy	SciFi	T. Hanks	B. Pitt
Movie1	4	10	0	2	8	0	8	2
Movie2	8	0	10	9	1	7	0	9

- Assume users express their preferences using the same properties

	Length	Disney	Universal	Action	Comedy	SciFi	T. Hanks	B. Pitt
User1	2	7	5	4	8	5	5	9
User2	9	3	7	9	1	7	2	5

Matrix Factorisation Example

- The users rating for any movie can now be computed as a simple product of user preferences * movie properties
- E.g. User1's rating score for movie 2 is

$$= 2*8 + 7*0 + 5*10 + \dots \quad (\text{to normalise divide by the max score})$$

	Length	Disney	Universal	Action	Comedy	Sci.Fi	T. Hanks	B. Pitt
M1	4	10	0	2	8	0	8	2
M2	8	0	10	9	1	7	0	9

	Length	Disney	Universal	Action	Comedy	Sci.Fi	T. Hanks	B. Pitt
U1	2	7	5	4	8	5	5	9
U2	9	3	7	9	1	7	2	5

Matrix Factorisation Example

- BUT we don't have the movie properties or user preferences, we don't even know what the properties should be. We only have a set of ratings from the users (the ratings matrix **R**). E.g.

	Movie1	Movie2	Movie3	Movie4	Movie5	Movie6	MovieN
U1	-	3	-	-	8	0	3
U2	8	0	10	9	1	7	-
U3							6

- Solution ~ Assume the movies have K properties (e.g. say 100) and use matrix factorisation on **R** to derive the user preference matrix (U) and the movie properties matrix (M):

$$\mathbf{R} = \mathbf{U}^T \mathbf{M}$$

- We don't need to know what the K properties are, we just assume they exist, they are called **latent** (hidden) variables

Matrix Factorisation Example

$$\begin{array}{c}
 \text{User ratings} \\
 \left[\begin{array}{c} 1 \dots\dots M \\ \vdots \\ U \end{array} \right] \\
 (500K * 17K = 8,500M)
 \end{array}
 =
 \begin{array}{c}
 \text{User preferences} \\
 \left[\begin{array}{c} 1 \dots\dots K \\ \vdots \\ U \end{array} \right] \\
 (500K * 100 = 50M)
 \end{array}
 *
 \begin{array}{c}
 \text{Movies properties} \\
 \left[\begin{array}{c} 1 \dots\dots M \\ \vdots \\ K \end{array} \right] \\
 (17K * 100 = 1.7M) \\
 \text{model}
 \end{array}$$

- The resulting matrices form the model ~ usually a much smaller dataset!
- 50 to 100 latent features are often all that is required for good results

Matrix Factorisation Example

- Once factorisation is done we can predict the ratings for all users and all items

		Item			
		W	X	Y	Z
User	A		4.5	2.0	
	B	4.0		3.5	
	C		5.0		2.0
	D		3.5	4.0	1.0

Rating Matrix

=

A	1.2	0.8
B	1.4	0.9
C	1.5	1.0
D	1.2	0.8

User Matrix

X

	W	X	Y	Z
A	1.5	1.2	1.0	0.8
B	1.7	0.6	1.1	0.4

Item Matrix

$$R = U * I^T$$

Why are the known ratings not always predicted correctly?

Performing the Factorisation

- SVD (Singular Value Decomposition) is common method for matrix factorisation
- Issues ~ much of the ratings matrix is empty (very sparse matrix)
- Straight SVD doesn't work. Hard to find exact factors
- Instead the Netflix winner used incremental learning solution based on gradient descent by taking derivatives of the approximation error

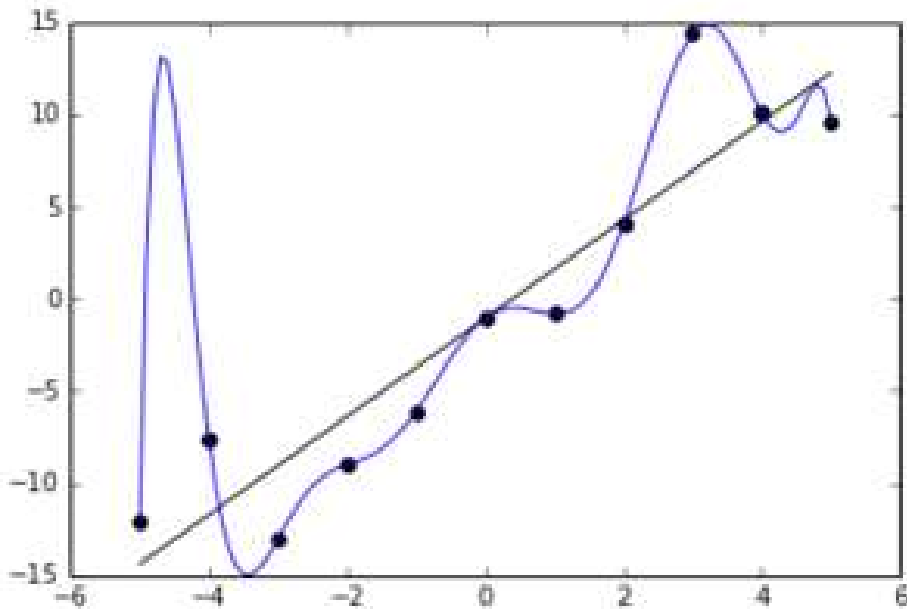
E.g. Use stochastic gradient descent to minimise the squared error between the predicted rating and the actual rating:

$$\min_{x_*, y_*} \sum_{r_{u,i} \text{ is known}} (r_{ui} - x_u^T y_i)^2 + \lambda (\|x_u\|^2 + \|y_i\|^2)$$

Actual rating
Predicted rating
Regularisation Term

Aside: Overfitting & Regularisation

- Given a high enough degree, a polynomial (or any other model) can be tuned to exactly learn the training data (over-fit). This is bad, the model will not generalise well.



We want the model to represent the training data, not to reproduce the training data!

- Regularization tries to avoid overfitting by adding a penalty term into the error function. The penalty term encourages the coefficients to be small so that the model doesn't become over-complex

$$L(x, y) = \sum_{i=1}^n (y_i - h_{\theta}(x_i))^2$$

$$\text{where } h_{\theta}x_i = \theta_0 + \theta_1x_1 + \theta_2x_2^2 + \theta_3x_3^3 + \theta_4x_4^4$$

$$L(x, y) \equiv \sum_{i=1}^n (y_i - h_{\theta}(x_i))^2 + \lambda \sum_{i=1}^n \theta_i^2$$



$$f(x_i) = h_{\theta}x = \theta_0 + \theta_1x_1 + \theta_2x_2^2 + \theta_3x_3^3 + \theta_4x_4^4$$

$$f(x_i) = h_{\theta}x = \theta_0 + \theta_1x_1 + \theta_2x_2^2$$

For Matrix Factorisation: Keeping the coefficients small means keeping the values in the factorized matrix small

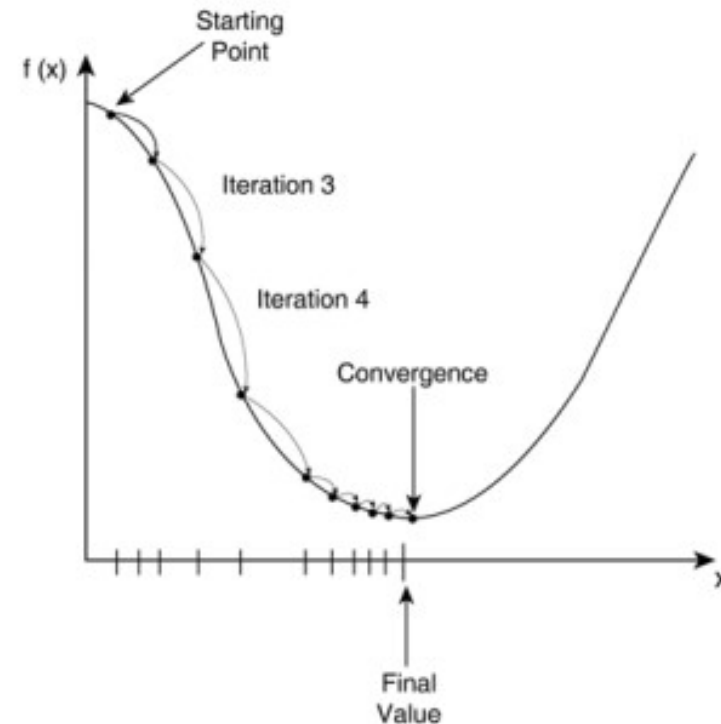
Factorisation using Gradient Descent*

Procedure:

Present the training examples in turn and update the weights after each (similar to NN learning).

Continue until convergence (error stops reducing)

Training examples are triplets:
(user, movie, rating)



- An example update rule for the user preference matrix \mathbf{U} and the item properties matrix \mathbf{M} is:

$$\mathbf{u}_{ki}^{t+1} = \mathbf{u}_{ki}^t + 2\gamma (\mathbf{r}_{ij} - \mathbf{p}_{ij}) \mathbf{m}_{kj}^t - \lambda \mathbf{u}_{ki}^t$$

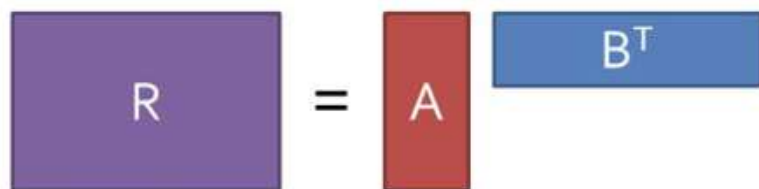
$$\mathbf{m}_{kj}^{t+1} = \mathbf{m}_{kj}^t + 2\gamma (\mathbf{r}_{ij} - \mathbf{p}_{ij}) \mathbf{u}_{ki}^t - \lambda \mathbf{m}_{kj}^t$$

\mathbf{u}_{ki} = value of kth preference for user i
 \mathbf{m}_{kj} = value of kth property for item j
 \mathbf{r}_{ij} = rating of user i on item j
 \mathbf{p}_{ij} = predicted rating of user i on item j
 γ = the learning rate
 λ = the regularisation constraint

*Method by Simon Funk (Netflix competitor)

Alternating Least Squares Algorithm

Model R as product of user and movie feature matrices A and B of size $U \times K$ and $M \times K$



$$R = A B^T$$

We wish to find values for \mathbf{a}_i and \mathbf{b}_j that minimises the total squared error:

$$\text{cost} = \sum_{ij} (r_{ij} - \mathbf{a}_i \cdot \mathbf{b}_j)^2$$

If we fix \mathbf{B} and optimize for \mathbf{A} alone, the problem is reduced to the problem of linear regression:

$$\mathbf{y} = \boldsymbol{\beta} \mathbf{X} \quad \} \quad y = \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \dots$$

where:

\mathbf{A} corresponds to the regression coefficients $\boldsymbol{\beta}$

\mathbf{B} corresponds to the training data (the input variables \mathbf{X})

\mathbf{R} corresponds to the output variables (\mathbf{y})

Alternating Least Squares (ALS)

- » Start with random \mathbf{A} & \mathbf{B}
- » Optimize user vectors (\mathbf{A}) based on movies
- » Optimize movie vectors (\mathbf{B}) based on users
- » Repeat until converged

<https://datasciencemadesimpler.wordpress.com/tag/alternating-least-squares/>

Alternating Least Squares

$$R = A * B^T$$

	i1	i2	i3	i4	i5	i6	i7	i8		f1	f2	f3	f4		i1	i2	i3	i4	i5	i6	i7	i8
u1		4			5				u1	a1	a2	a3	a4	f1	b1	b2	b3	b4	b5	b6	b7	b8
u2	1				2		5		u2	a5	a6	a7	a8	f2	b9	b10	b11	b12	b13	b14	b15	b16
u3		3	4	3		5			u3	a9	a10	a11	a12	f3	b17	b18	b19	b20	b21	b22	b23	b24
u4	2	2			5	5		3	u4	a13	a14	a15	a16	f4	b25	b26	b27	b28	b29	b30	b31	b32
u5			1			4		2	u5	a17	a18	a19	a20									
u6	3			5	4		3		u6	a21	a22	a23	a24									

$$E.g. 4 = (a1*b2 + a2*b10 + a3*b18 + a4*b26)$$

Step1: Assign random weights to A and B
(weights and ratings should be normalised to range 0 to 1)

Step2 : fix A and use **linear regression*** to estimate weights in B
will require building 8 regression models (one per item)

$$\longrightarrow y = B_1*x_1 + B_2*x_2 + B_3*x_3 + \dots$$

Step3: fix B and use **linear regression*** to estimate weights in A
will require building 6 regression models (one per user)

$$\longrightarrow y = A_1*x_1 + A_2*x_2 + A_3*x_3 + \dots$$

Repeat from step2 until convergence (or time limit etc)

Alternating Least Squares - Conceptual

$$R = A * B^T$$

	i1	i2	i3	i4	i5	i6	i7	i8
u1		4			5			
u2	1				2		5	
u3		3	4	3		5		
u4	2	2			5	5		3
u5			1			4		2
u6	3			5	4		3	

	f1	f2	f3	f4
u1	a1	a2	a3	a4
u2	a5	a6	a7	a8
u3	a9	a10	a11	a12
u4	a13	a14	a15	a16
u5	a17	a18	a19	a20
u6	a21	a22	a23	a24

	i1	i2	i3	i4	i5	i6	i7	i8
f1	b1	b2	b3	b4	b5	b6	b7	b8
f2	b9	b10	b11	b12	b13	b14	b15	b16
f3	b17	b18	b19	b20	b21	b22	b23	b24
f4	b25	b26	b27	b28	b29	b30	b31	b32

E.g. training data to learn the i2 weights in B:

x1	x2	x3	x4	Output (Y)
a1	a2	a3	a4	4
a9	a10	a11	a12	3
a13	a14	a15	a16	2



$$Y = MX + C$$



f1	f2	f3	f4
b2	b10	b18	b26

... training data to learn the i7 weights in B:

x1	x2	x3	x4	Output (Y)
a5	a6	a7	a8	5
a21	a22	a23	a24	3



f1	f2	f3	f4
b7	b15	b23	b31

Alternating Least Squares - Conceptual

$$R = A * B^T$$

	i1	i2	i3	i4	i5	i6	i7	i8
u1		4			5			
u2	1				2		5	
u3		3	4	3		5		
u4	2	2			5	5		3
u5			1			4		2
u6	3			5	4		3	

	f1	f2	f3	f4
u1	a1	a2	a3	a4
u2	a5	a6	a7	a8
u3	a9	a10	a11	a12
u4	a13	a14	a15	a16
u5	a17	a18	a19	a20
u6	a21	a22	a23	a24

	i1	i2	i3	i4	i5	i6	i7	i8
f1	b1	b2	b3	b4	b5	b6	b7	b8
f2	b9	b10	b11	b12	b13	b14	b15	b16
f3	b17	b18	b19	b20	b21	b22	b23	b24
f4	b25	b26	b27	b28	b29	b30	b31	b32

E.g. training data to learn the u2 weights in A:

x1	x2	x3	x4	Output (Y)
b1	b9	b17	b25	1
b5	b13	b21	b29	2
b7	b15	b23	b31	5



$$Y = MX + C$$



f1	f2	f3	f4
a5	a6	a7	a8

... training data to learn the u5 weights in A:

x1	x2	x3	x4	Output (Y)
				1
				4
				2



f1	f2	f3	f4
a17	a18	a19	a20

ALS: Handling New Users / Items

- No cold-start: Cannot predict for a user with no existing ratings (or a new item)
 - Ask the user for sample ratings, else wait until they have done some page-views, transactions etc
- Must re-factorise (which can take time)...

$$R = A * B^T$$

	i1	i2	i3	i4	i5	i6	i7	i8
u1		4			5			
u2	1				2		5	
u3		3	4	3		5		
u4	2	2			5	5		3
u5			1			4		2
u6	3			5	4		3	
u7		2	4			1		

	f1	f2	f3	f4
u1	a1	a2	a3	a4
u2	a5	a6	a7	a8
u3	a9	a10	a11	a12
u4	a13	a14	a15	a16
u5	a17	a18	a19	a20
u6	a21	a22	a23	a24
u7	?	?	?	?

	i1	i2	i3	i4	i5	i6	i7	i8
f1	b1	b2	b3	b4	b5	b6	b7	b8
f2	b9	b10	b11	b12	b13	b14	b15	b16
f3	b17	b18	b19	b20	b21	b22	b23	b24
f4	b25	b26	b27	b28	b29	b30	b31	b32

- Or factorise just for the new user (reuse the existing item factors matrix)

Matrix Factorisation at scale, e.g. Facebook system

<https://code.fb.com/core-data/recommending-items-to-more-than-a-billion-people/>

Other Popular Factorisation Methods

- Non-Negative Matrix Factorisation
 - https://en.wikipedia.org/wiki/Non-negative_matrix_factorization
- Logistic Matrix Factorisation
- SVD++
 - Incorporates Explicit and Implicit ratings
- Netflix and Parallel ALS
 - https://www.researchgate.net/publication/220788980_Large-Scale_Parallel_Collaborative_Filtering_for_the_Netflix_Prize

Model-Based CF: Libraries

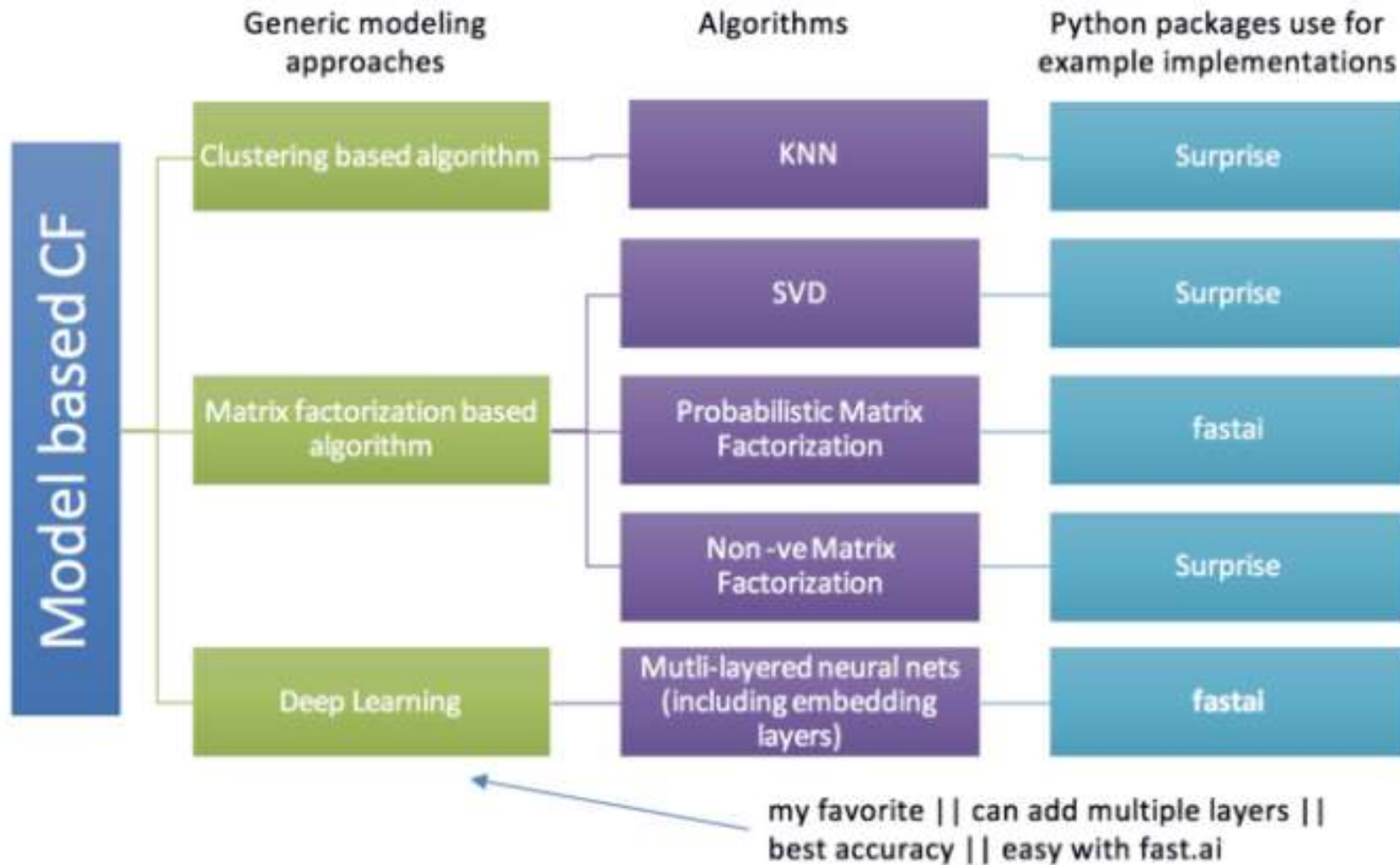


Figure 3. Types of model based collaborative filtering approaches

Libraries: Surprise (Python)

The logo for the Surprise library, featuring the word "surprise" in a white, lowercase, sans-serif font. The letter "i" is replaced by a white exclamation mark. The logo is set against a teal rectangular background.

A Python scikit for
recommender systems.

Overview

Surprise is a Python `scikit` building and analyzing recommender systems that deal with explicit rating data.

Surprise was designed with the following purposes in mind:

- Give users perfect control over their experiments. To this end, a strong emphasis is laid on `documentation`, which we have tried to make as clear and precise as possible by pointing out every detail of the algorithms.
- Alleviate the pain of `Dataset handling`. Users can use both *built-in* datasets (`Movielens`, `Jester`), and their own *custom* datasets.
- Provide various ready-to-use `prediction algorithms` such as `baseline algorithms`, `neighborhood methods`, matrix factorization-based (`SVD`, `PMF`, `SVD++`, `NMF`), and `many others`. Also, various `similarity measures` (cosine, MSD, pearson...) are built-in.
- Make it easy to implement `new algorithm ideas`.
- Provide tools to `evaluate`, `analyse` and `compare` the algorithms performance. Cross-validation procedures can be run very easily using powerful CV iterators (inspired by `scikit-learn` excellent tools), as well as `exhaustive search over a set of parameters`.

The name *SurPRISE* (roughly :)) stands for Simple Python Recommendation System Engine.

Please note that surprise does not support implicit ratings or content-based information.

Libraries: Implicit (Python)

Implicit

Fast Python Collaborative Filtering for Implicit Datasets

This project provides fast Python implementations of several different popular recommendation algorithms for implicit feedback datasets:

- Alternating Least Squares as described in the papers [Collaborative Filtering for Implicit Feedback Datasets](#) and in [Applications of the Conjugate Gradient Method for Implicit Feedback Collaborative Filtering](#). *
- [Bayesian Personalized Ranking](#)
- Item-Item Nearest Neighbour models, using Cosine, TFIDF or BM25 as a distance metric

All models have multi-threaded training routines, using Cython and OpenMP to fit the models in parallel among all available CPU cores. In addition, the ALS and BPR models both have custom CUDA kernels - enabling fitting on compatible GPU's. This library also supports using approximate nearest neighbours libraries such as [Annoy](#), [NMSLIB](#) and [Faiss](#) for [speeding up making recommendations](#).

Other Python Libraries

Top Open Source Recommender Systems In Python For Your ML Project

- LensKit. About: LensKit is an open-source toolkit for building, researching, and learning about recommender systems. ...
- Crab. ...
- Surprise. ...
- REXY. ...
- TensorRec. ...
- LightFM. ...
- Case Recommender. ...
- Spotlight.

4 Nov 2020

[https://analyticsindiamag.com › top-open-source-recomm...](https://analyticsindiamag.com/top-open-source-recomm...)

Top Open Source Recommender Systems In Python For Your ...

<https://analyticsindiamag.com/top-open-source-recommender-systems-in-python-for-your-ml-project/>

[Best Python Libraries for Recommendation Systems \(analyticsindiamag.com\)](https://analyticsindiamag.com/top-open-source-recommender-systems-in-python-for-your-ml-project/)

Numpy and Scipy: SVD

Numpy and Scipy both contain an SVD algorithm:

Factorizes the matrix a into three matrices: U , V_h , S

where $a == U @ S @ V_h$

and S is a matrix of zeros with the singular values (real, non-negative) on the diagonal

Regular SVD fails due to the nulls in the ratings matrix

But we can try replacing nulls with zeros, or use other way to estimate the nulls

```
In [41]: from scipy.linalg import svd
In [42]: ratmatrix.shape
Out[42]: (943, 1682)
In [43]: u, s, vh = svd(ratmatrix, full_matrices=True)
Traceback (most recent call last):
ValueError: array must not contain infs or NaNs
```

Using the Movielens 100K dataset (no train/test split)

```
In [51]: rowmeans = np.nanmean(ratmatrix, axis=1, keepdims=1)
...: invalid = np.isnan(ratmatrix)
...: ratmatrixFull = np.where(invalid, rowmeans, ratmatrix)
In [52]: u, s, vh = svd(ratmatrixFull, full_matrices=False)
In [53]: preds = (u.dot(np.diag(s)).dot(vt))
...: errs = np.abs(preds-ratmatrix)
...: np.nanmean(errs)
Out[53]: 1.930975157549142e-14
```

Singular Value Decomposition

- SVD factorises a matrix into 3 components
- The left and right matrices (U and V) are analogous to those in ALS
- The middle matrix (Σ) is a diagonal matrix containing R singular values (R is the number of latent features) which indicate the relative importance of the latent features (similar to PCA)

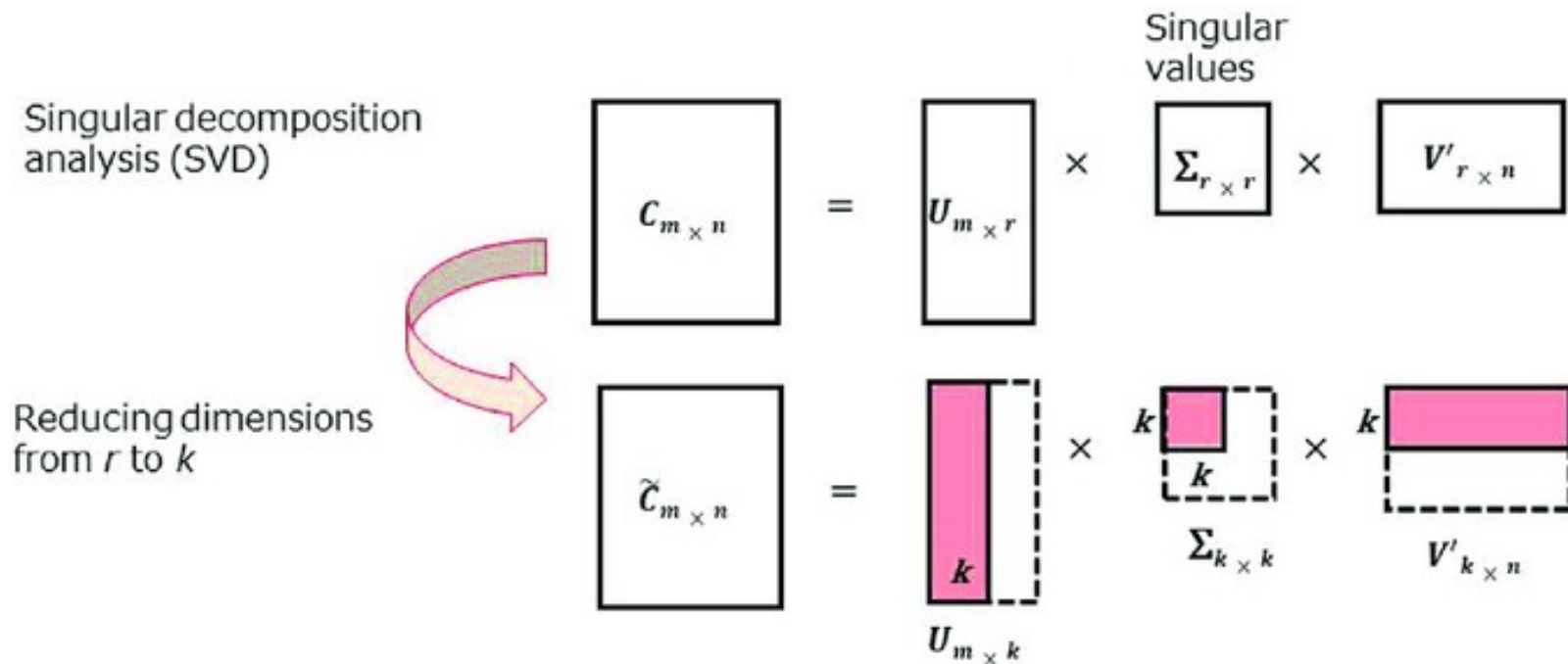
$$\begin{array}{c}
 \boxed{\begin{matrix} A \\ n \times d \end{matrix}} = \boxed{\begin{matrix} \hat{U} \\ n \times r \end{matrix}} \boxed{\begin{matrix} \hat{\Sigma} \\ r \times r \end{matrix}} \boxed{\begin{matrix} \hat{V}^T \\ r \times d \end{matrix}} \\
 \begin{matrix} U & \Sigma & V^T \\ n \times d & n \times d & d \times d \end{matrix}
 \end{array}$$

$$\Sigma = \begin{bmatrix} d_{11} & 0 & 0 & \dots & 0 \\ 0 & d_{22} & 0 & \dots & 0 \\ 0 & 0 & d_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & d_{nn} \end{bmatrix}$$

Σ is a diagonal matrix, the R values lie along the diagonal and everything else is zero. The values are sorted in descending order

Singular Value Decomposition

- Since the singular values are sorted and indicate the relative importance of the latent features, we can do data reduction by ignoring the smallest ones (those in the bottom right).



Pros & Cons of Matrix Factorisation

Pros

- Generally more accurate than User-based and Item-based CF

Cons

- They provide no explanation on the reasons behind a recommendation
- Factorisation must be redone to apply to new users / new items