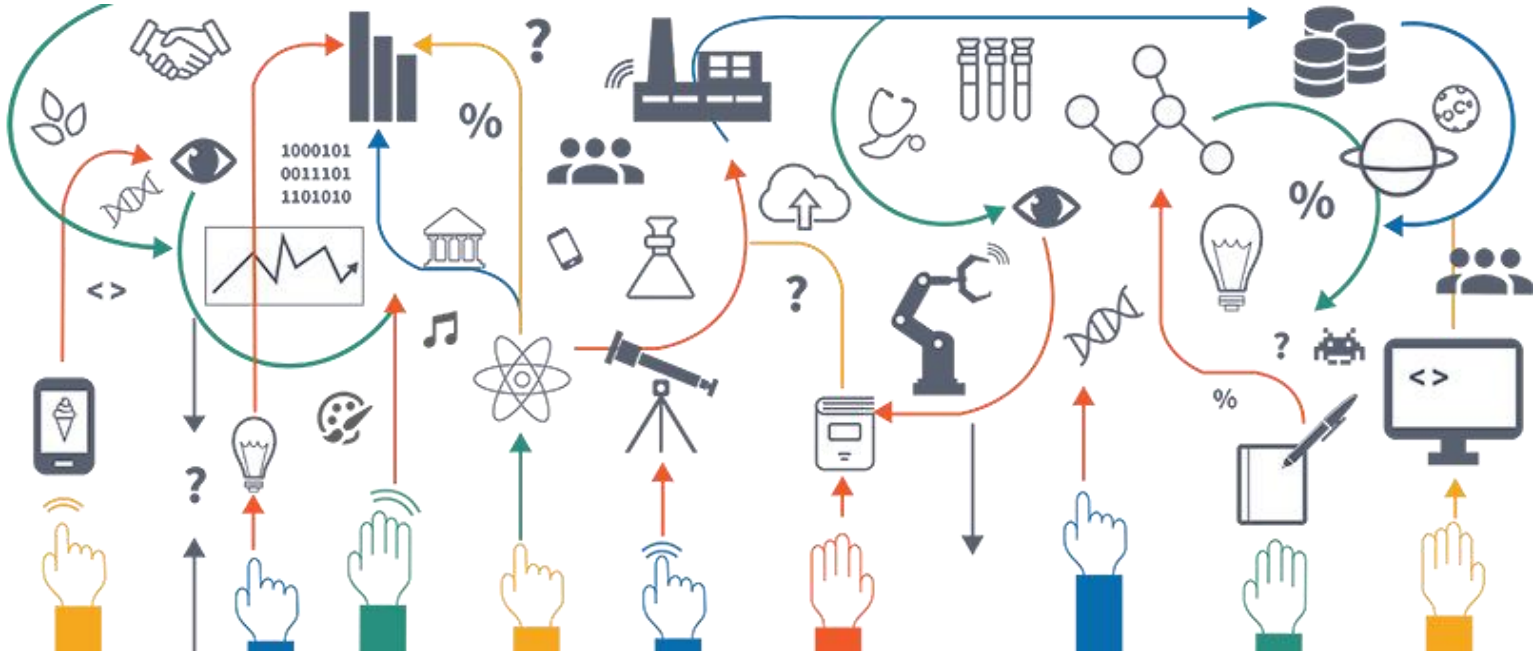
3 – Learning from Structured Data

Mathematics of Data Science



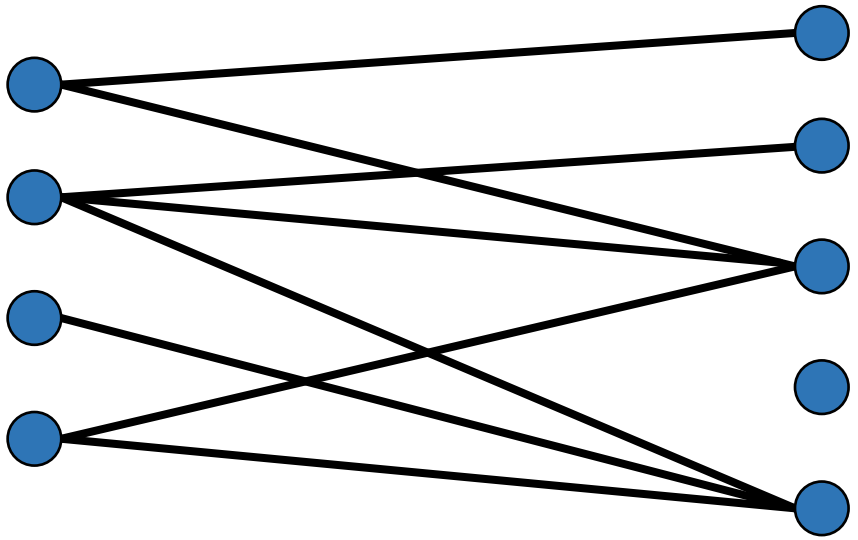
Lecturer: Dominik Dold

University of Vienna, WiSe 2025

Master's programme in Data Science

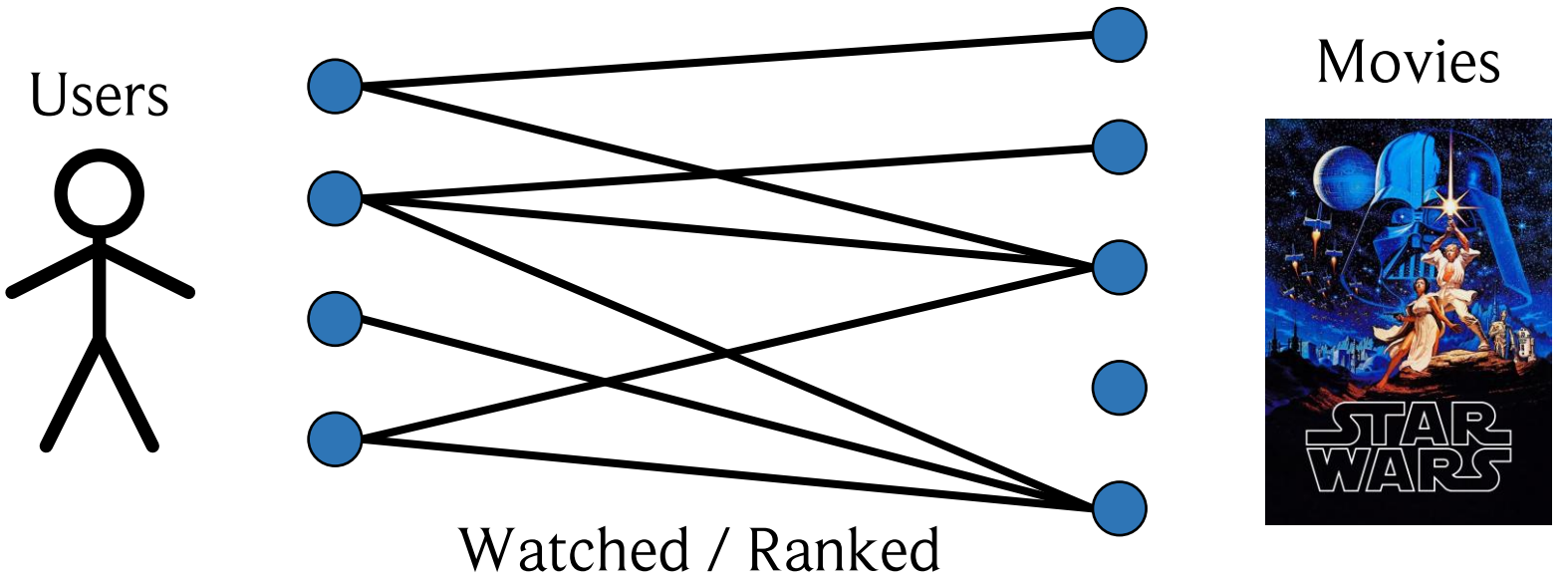
Credit: Netflix

Motivation: assume the following type of data structure

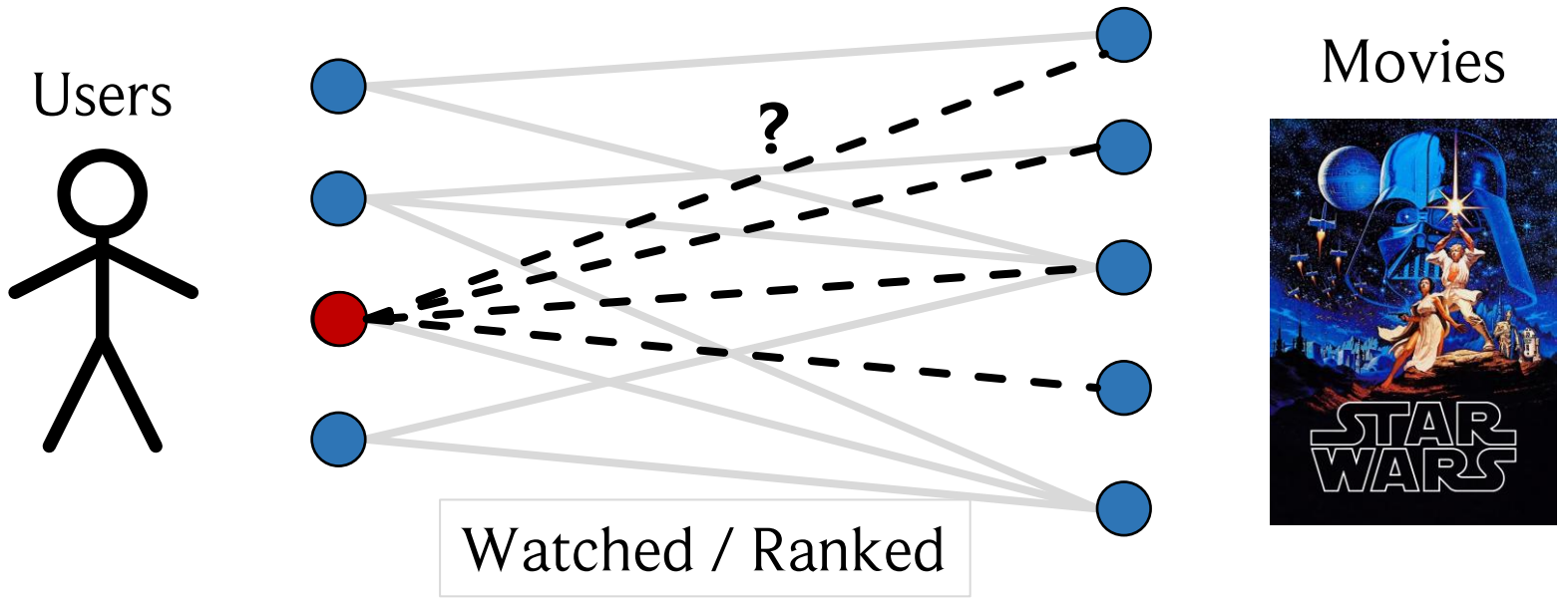


? In what kind of tasks do we encounter data that is structured like this?

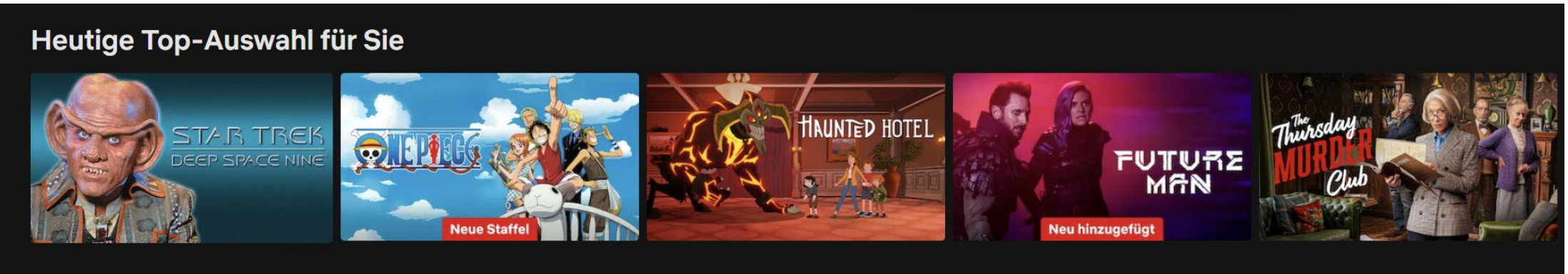
Motivation: assume the following type of data structure



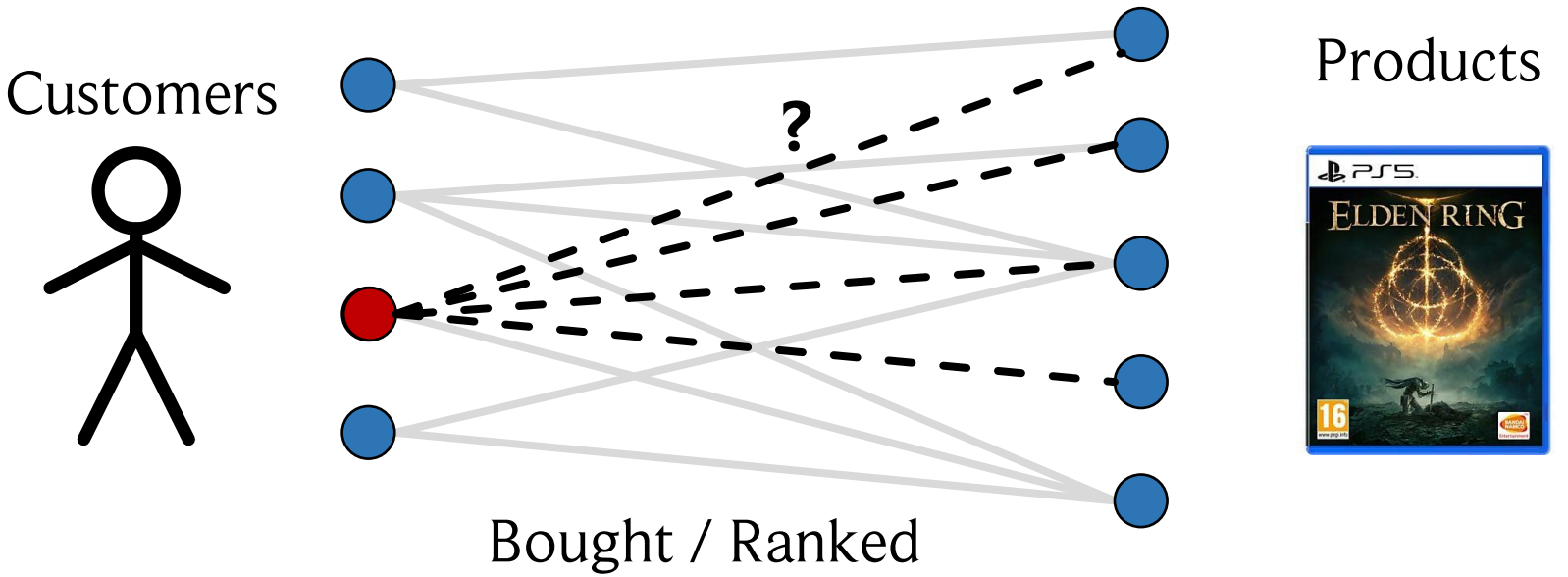
Motivation: recommending based on history



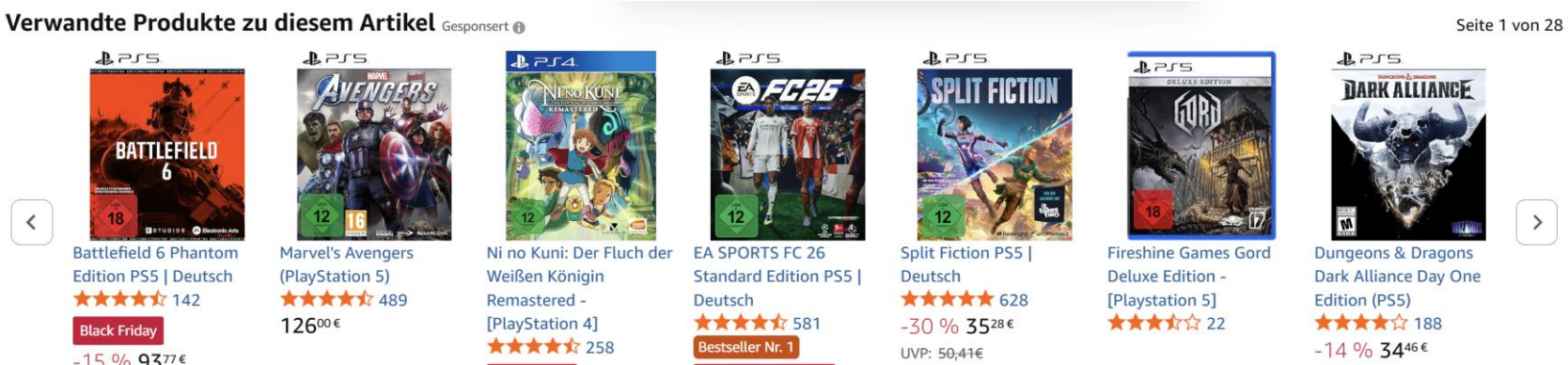
Task: recommend movies based on what red user and other users watched / liked.



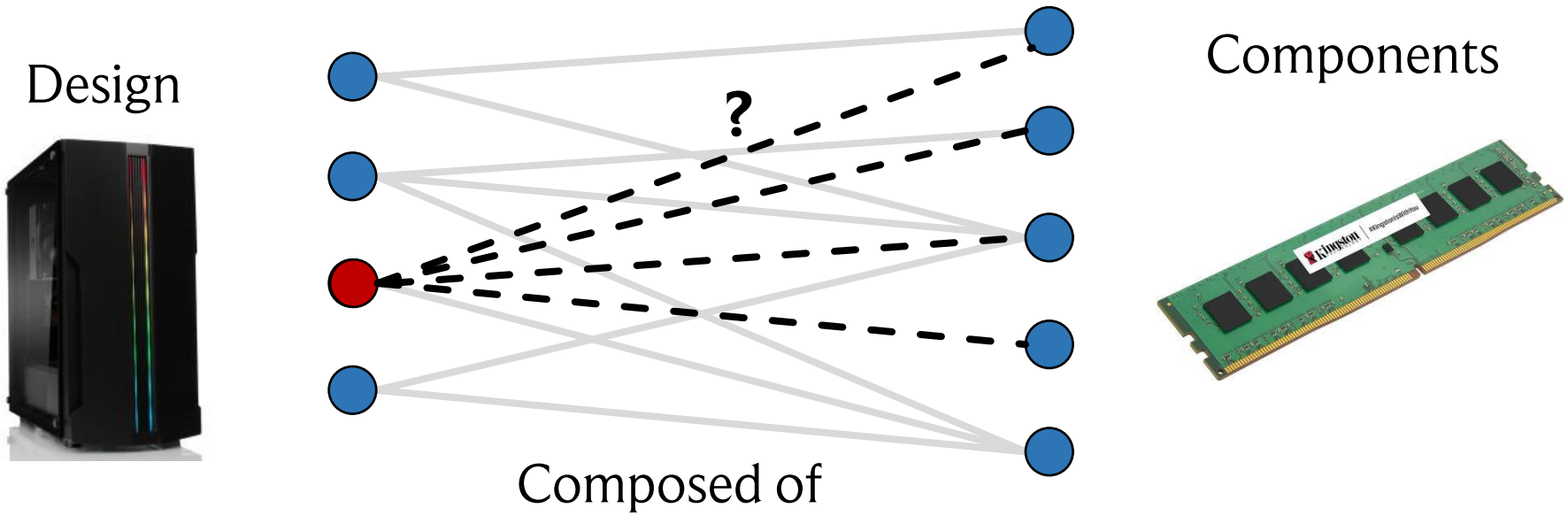
Motivation: recommending based on history



Task: recommend products based on what red customer and other customers bought.



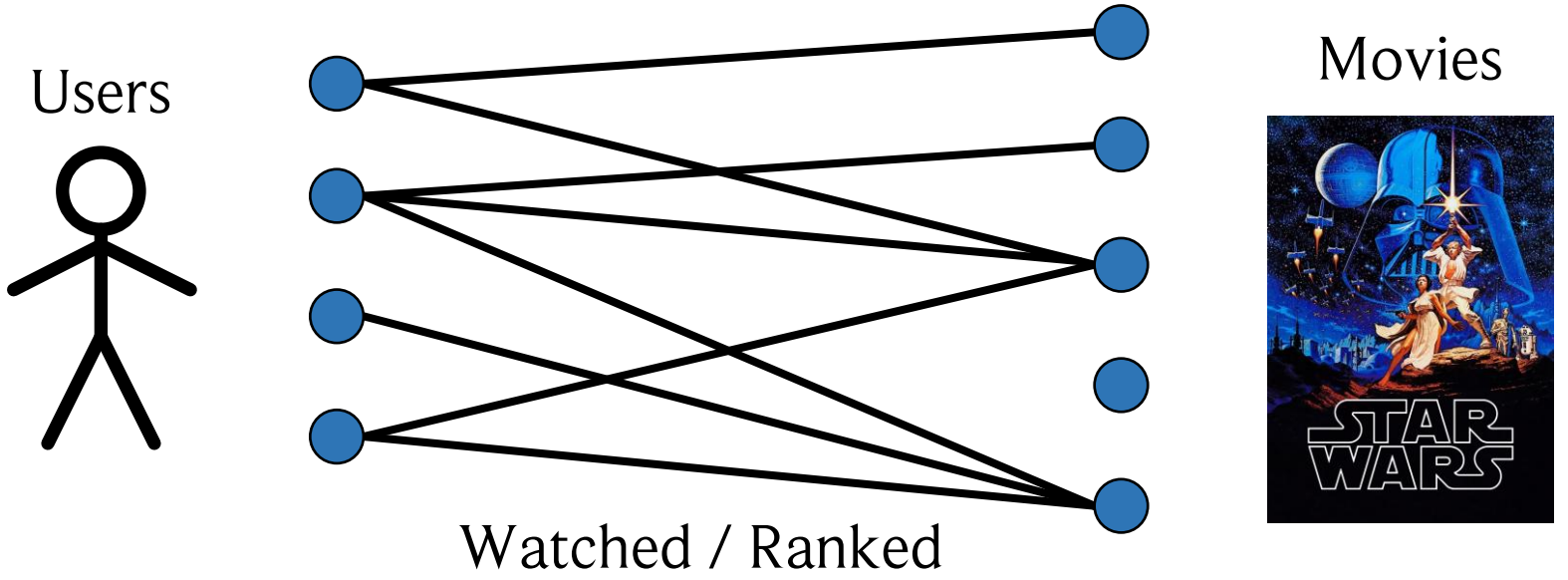
Motivation: recommending based on history



Task: based on previous designs, propose components for a partially completed design.

So… how can we do this?

What we want to build is a **recommender system**.
Lets look at the Netflix example:



This is a discrete structure. But we can find vector embeddings that allow us to make recommendations!

$$\boldsymbol{e}_{\text{user } i}^{T} \boldsymbol{e}_{\text{movie j}} \approx \text{score for matching user } i \text{ and movie } j$$

Content

- PageRank
- Singular Value Decomposition
- Funk Algorithm
- Non-negative Matrix Factorization
- DeepWalk
- Knowledge graphs
- Open and closed world assumption
- Tensor factorization
- Translational embeddings
- Graph Neural Networks

The secret sauce of early Google: PageRank

Larry Page (1973 -)
& Sergey Brin (1973 -)

Problem: You search webpages containing a certain "term", but you are overwhelmed by the number of results…

So, what can we do?

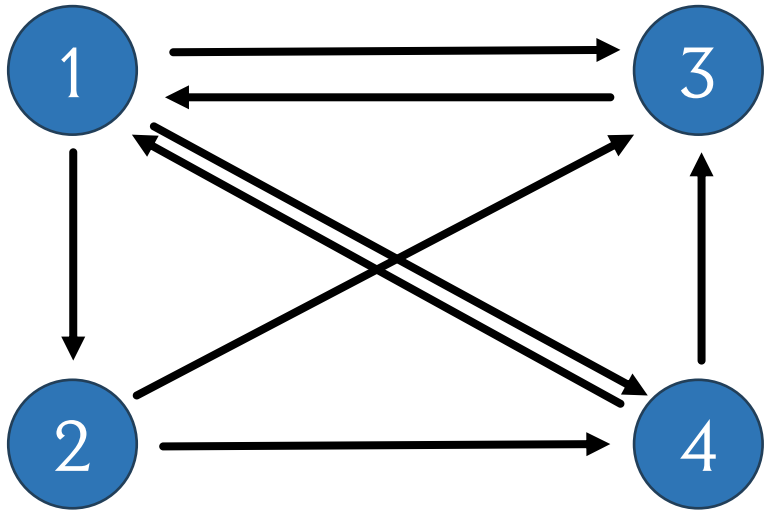
Google

Idea: sort the webpages by ***relevance***!

The importance of any web page can be judged by looking at the pages that link to it.

The internet: a graph of pages linking to each other

Larry Page (1973 -)
& Sergey Brin (1973 -)



Vertices: *Webpages*
Edges: *Links to…*

Let's define an **importance score** of each webpage: x_k

How do we get this score? Some ideas…

1. $x_k = |L_k|$, where L_k = set of pages linking to page k

   $x_1 = 2, x_2 = 1, x_3 = 3, x_4 = 2$

   Problem:
   links from important pages should count more!
2. $x_k = \sum_{i \in L_k} x_i$

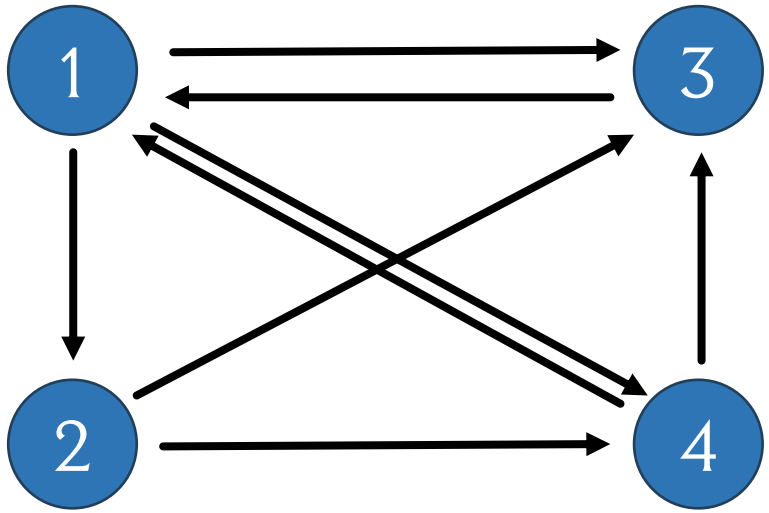
   Problem:
   Now you can spam links to pages to boost scores!
3. $x_k = \sum_{i \in L_k} x_i / n_i$, where n_i =#links of page i

   i.e., every page distributes its "votes" to all pages!

Source: The IT Crowd Sitcom

PageRank: turning ranking to an eigenvector problem

Larry Page (1973 -)
& Sergey Brin (1973 -)



Vertices: *Webpages*
Edges: *Links to…*

We go with $x_k = \sum_{i \in L_k} x_i \,/ n_i$, which can be written as a system of linear equations! For our example, we get:

$$x_1 = x_3 + \frac{x_4}{2} \quad x_2 = \frac{x_1}{3} \quad x_3 = \frac{x_1}{3} + \frac{x_2}{2} + \frac{x_4}{2} \quad x_4 = \frac{x_1}{3} + \frac{x_2}{2}$$

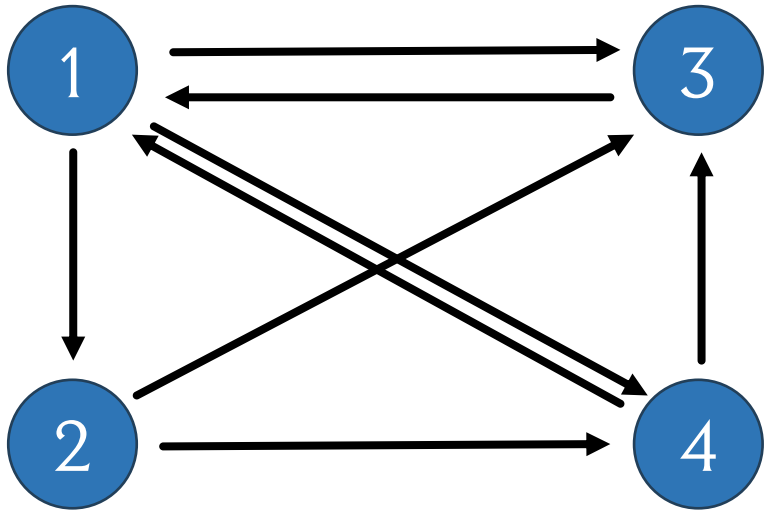
Or rather: $\boldsymbol{Ax = x}$

$$\boldsymbol{A} = \begin{pmatrix} 0 & 0 & 1 & 1/2 \\ 1/3 & 0 & 0 & 0 \\ 1/3 & 1/2 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{pmatrix} \qquad \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$$

Source: The IT Crowd Sitcom

PageRank: turning ranking to an eigenvector problem

Larry Page (1973 -)
& Sergey Brin (1973 -)



Vertices: *Webpages*
Edges: *Links to…*

$$\boldsymbol{Ax} = \boldsymbol{x}\,, \quad \boldsymbol{A} = \begin{pmatrix} 0 & 0 & 1 & 1/2 \\ 1/3 & 0 & 0 & 0 \\ 1/3 & 1/2 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{pmatrix} \quad \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$$

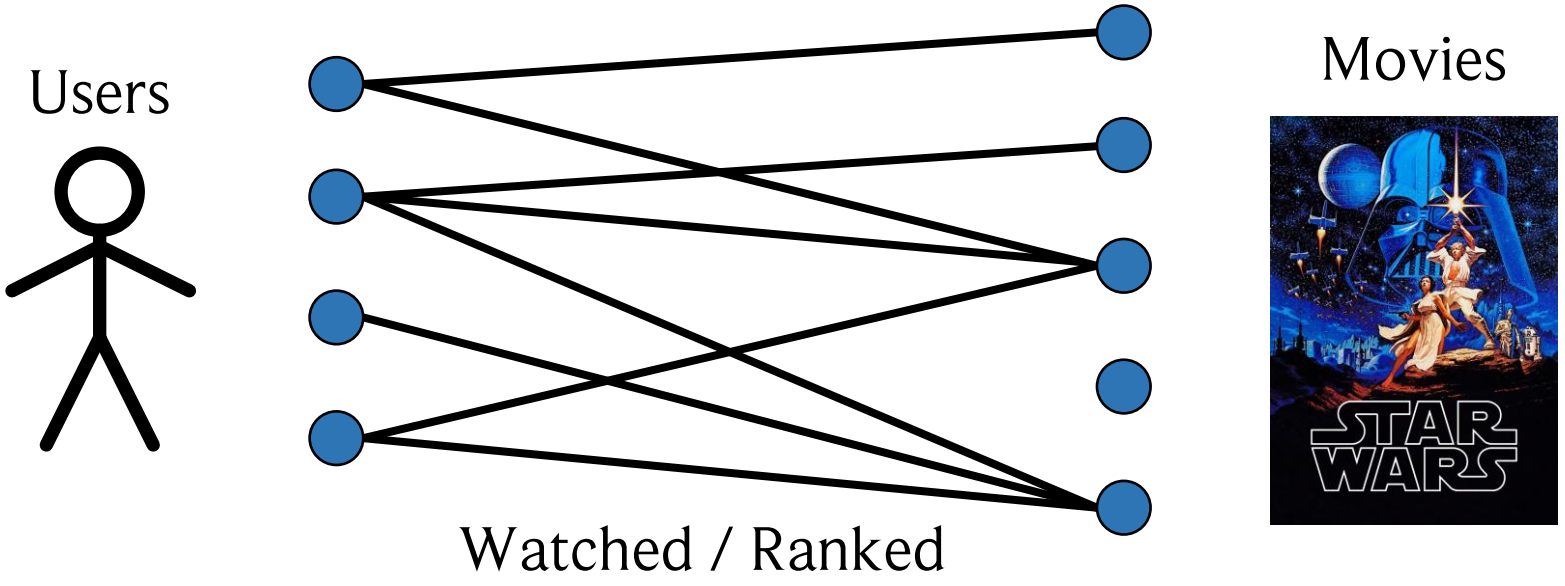
1. Thus, we seek the eigenvector of $\boldsymbol{A}$ with eigenvalue 1.
2. Note that $\boldsymbol{A}^{\mathrm{T}}$ is column stochastic (it defines a random walk on the graph!). **We know from diffusion maps:** the 1-vector is its eigenvector with eigenvalue 1, which is also the largest eigenvalue.
3. Moreover, $\boldsymbol{A}^{\mathrm{T}}$ and $\boldsymbol{A}$ have the same eigenvalues (but not necessarily eigenvectors).
4. In our case the eigenvector is: $\boldsymbol{x} = \left(\frac{12}{31}, \frac{4}{31}, \frac{9}{31}, \frac{6}{31}\right)$

Recommender systems

Using PageRank, we can **recommend** the most famous webpages!

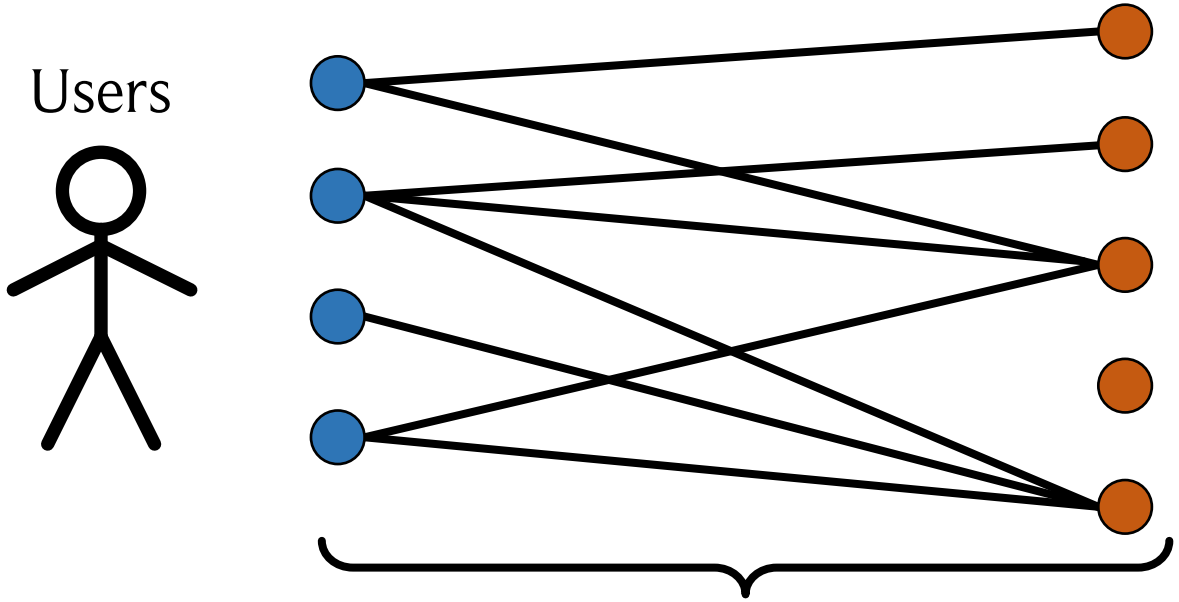
In particular, its another example of how a graph problem turns into an eigenvalue problem!

But what about cases like this?



Representing our problem as a user-movie matrix

Let's represent this in a different way!



Movies

User 1

$$\mathbb{R}_{+}^{n\times m} \ni \boldsymbol{S} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix} \approx \boldsymbol{U} \cdot \boldsymbol{M}$$

Movie 1

$$\boldsymbol{U} \in \mathbb{R}^{n\times d}$$

$$\boldsymbol{M} \in \mathbb{R}^{d\times m}$$

$$d < n, m$$

Finding vector embeddings for users and movies

What are U and M?

They are called **embeddings / latent representations** of the users and movies!
It's **low-dimensional vectors** that contain **collaborative information** from all users/movies.

Original representation

Rows and columns of $\boldsymbol{S}$

$$\boldsymbol{u}_2 = (1 \quad 1 \quad 0 \quad 1)$$

Movies watched by user 2
No information about other users!

$$\boldsymbol{m}_0 = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

Users that watched movie 0
No information about other movies!

Embeddings

Entries contain global information!
For users, it could be genres they like
For movies, how much they align with that genre

$$\boldsymbol{U}_2 = (0.5 \quad 0.2) \qquad M_0 = \begin{pmatrix} 0.8 \\ 0.2 \end{pmatrix}$$

Often, the elements are not cleanly interpretable though!

Question!

How can we find such embeddings?!

First idea: spectral decomposition!

Reminder: If a matrix $\boldsymbol{S}$ is diagonalizable, we can represent it in the following way

$$\boldsymbol{S} = \boldsymbol{V} \cdot \boldsymbol{D} \cdot \boldsymbol{V}^{-1}$$

User embeddings — Feature importance (diagonal matrix) — Movie embeddings

However, we don't want to perfectly reproduce the original matrix!

Truncate! I.e., only keep the d highest eigenvalues: $\boldsymbol{S} \approx \boldsymbol{U} \cdot \boldsymbol{\Delta} \cdot \boldsymbol{M}$

($\boldsymbol{U}$: all rows, only first d columns of $\boldsymbol{V}$, $\boldsymbol{M}$: first d rows, all columns of $\boldsymbol{V}^{-1}$, $\boldsymbol{\Delta}$: first d rows and columns of $\boldsymbol{D}$)

Scoring a new users-movie combination $i-j$? $s_{ij} = \boldsymbol{U}_i \cdot \boldsymbol{\Delta} \cdot \boldsymbol{M}_j^{\mathrm{T}}$

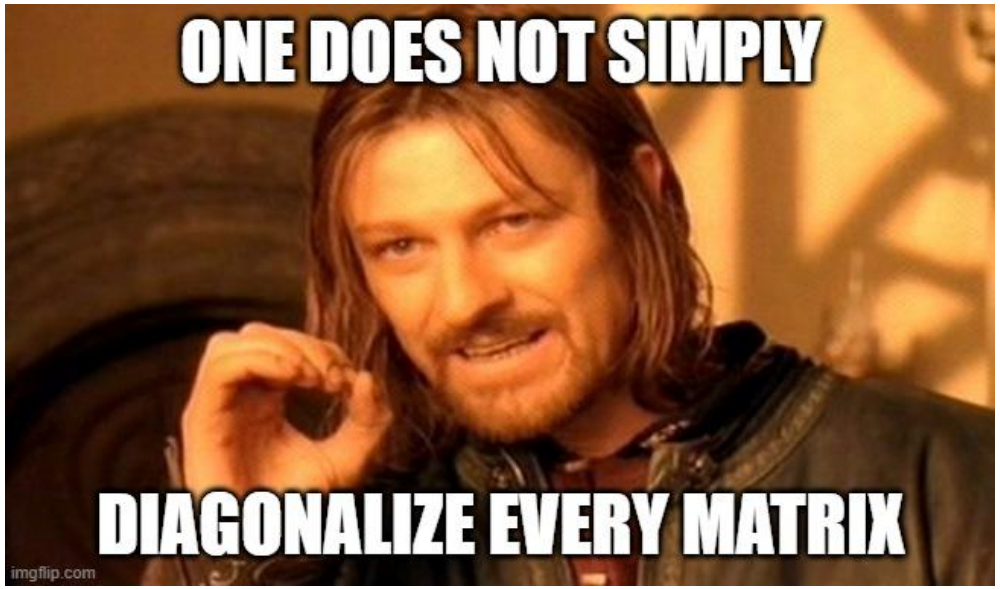
Isn’t this just dimensionality reduction?

In principle, yes!

In the previous example, we tried to find low-dimensional vectors representing users and movies such that the distance between vectors is

- **low** if a user will like the movie,
- **high** otherwise,

with the distance given by the inner product $\boldsymbol{U}_i \boldsymbol{\Delta} \boldsymbol{M}_j^{\mathrm{T}}$.



Singular Value Decomposition (SVD) to the rescue!

A decomposition that always works is SVD!

SVD: Let $\boldsymbol{S} \in \mathbb{R}^{n \times m}$. Then there exists a $\boldsymbol{U} \in \mathbb{R}^{n \times n}$, $\boldsymbol{V} \in \mathbb{R}^{m \times m}$, and $\boldsymbol{\Sigma} \in \mathbb{R}^{n \times m}$, such that

$$\boldsymbol{S} = \boldsymbol{U} \cdot \boldsymbol{\Sigma} \cdot \boldsymbol{V}^{\mathrm{T}}$$

Where $\boldsymbol{\Sigma}$ has σ_i, the square-roots of the eigenvalues of $\boldsymbol{S}^{\mathrm{T}}\boldsymbol{S}$, on its diagonal and is 0 otherwise, and $\boldsymbol{U}^{\mathrm{T}}\boldsymbol{U} = \boldsymbol{I}, \boldsymbol{V}^{\mathrm{T}}\boldsymbol{V} = \boldsymbol{I}$. σ_i are called singular values of $\boldsymbol{S}$.

Proof: see handwritten notes

An important property!

Let $\boldsymbol{A} \in \mathbb{R}^{n \times m}$ with SVD $\boldsymbol{A} = \boldsymbol{U}\,\boldsymbol{\Sigma}\,\boldsymbol{V}^{\mathrm{T}}$. Moreover, let $\boldsymbol{A_s}$ be the matrix reconstructed from the truncated SVD, i.e., using only the s largest singular values, $\boldsymbol{A_s} = \sum_{k=1}^{s} \sigma_k\, \boldsymbol{u}_k \boldsymbol{v}_k^{\mathrm{T}}$.

Then, among **all** matrices with rank s, $\boldsymbol{A_s}$ is the best possible approximation of $\boldsymbol{A}$ with respect to the Frobenius and Operator norm.

In particular, the approximation error is:

$$\|\boldsymbol{A} - \boldsymbol{A}_s\|_{\mathrm{F}} = \left(\sum_{k=s+1}^{r} \sigma_k^2 \right)^{1/2}, \quad \|\boldsymbol{A} - \boldsymbol{A}_s\|_2 = \sigma_{s+1}, \qquad \text{where } r = \operatorname{rank}(\boldsymbol{A}).$$

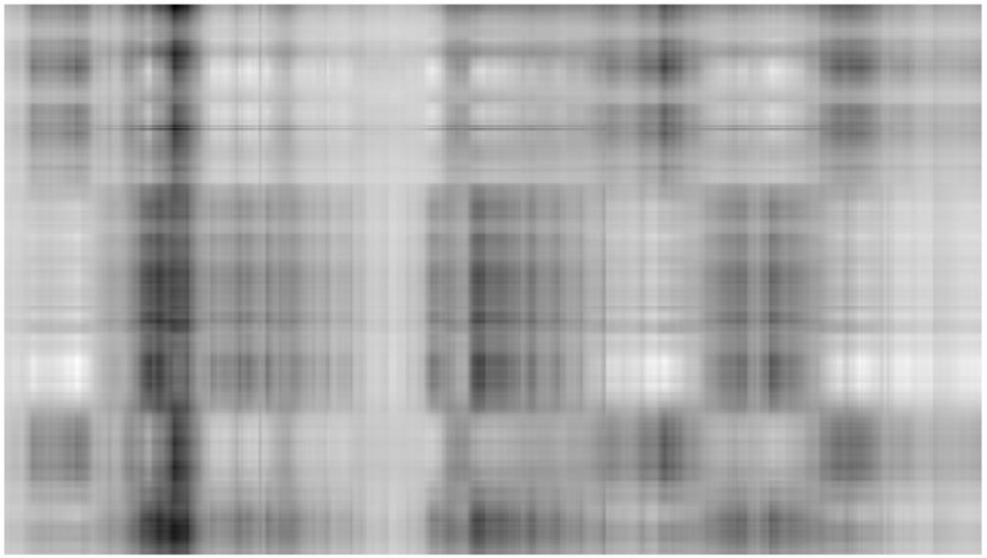
Details: see handwritten notes

Reminder: Frobenius norm $\|\boldsymbol{A}\|_{\mathrm{F}} = \sum_{i,j} A_{ij}$ Operator norm $\|\boldsymbol{A}\|_2 = \sup_{||x||=1} ||Ax||$

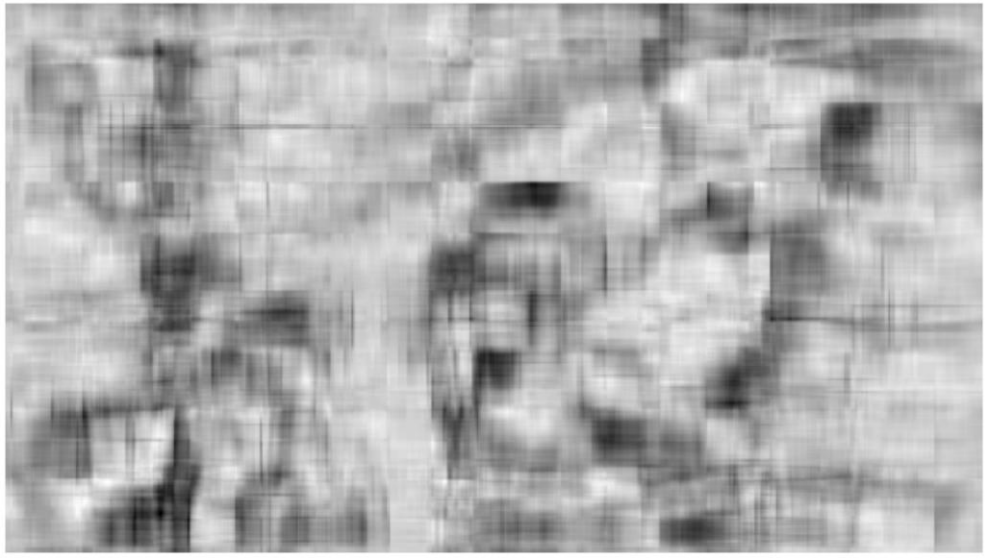
Example using images

$\boldsymbol{A}$
(421 × 748)

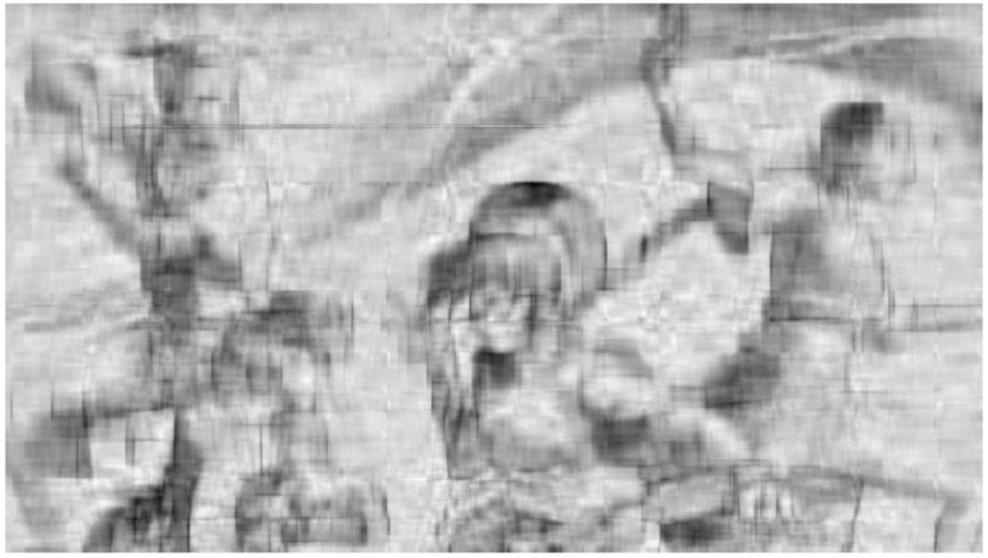
$\boldsymbol{A}_1$



$\boldsymbol{A}_{10}$



$\boldsymbol{A}_{20}$



Example using images

$\boldsymbol{A}$
(421 × 748)

This is **lossy** compression!

Useful for:

- Memory reduction (compression)
- Noise reduction / removal
- **Filling in missing values!**

$\boldsymbol{A}_{50}$

$\boldsymbol{A}_{100}$

“Recommending” missing pixel values

Here, some pixels have missing values which we filled with 0.

$s = 200$

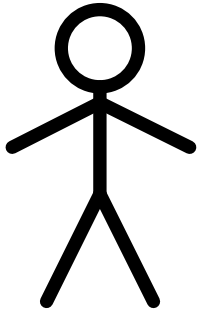
$s = 40$

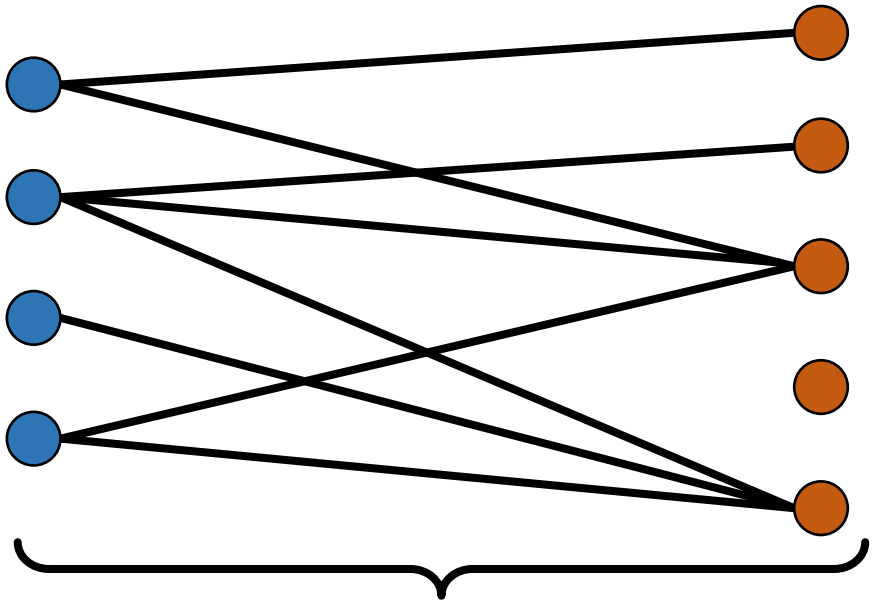
We recover those values through a truncated SVD! I.e., do the SVD and then replace the missing values with the corresponding ones from $\boldsymbol{A}_s$!

Truncating is key, else we recover the original 0s we added!

Images are matrices, as is our user-movie score matrix!

Users





Movies

User embeddings

Movie embeddings

truncate!

User 1

Movie 1

$$\mathbb{R}_+^{n\times m} \ni \boldsymbol{S} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix} \approx \boldsymbol{U}_s\sqrt{\boldsymbol{\Sigma}_s}\sqrt{\boldsymbol{\Sigma}_s}\,\boldsymbol{V}_s^{\mathrm{T}} = \widetilde{\boldsymbol{U}}_s\,\widetilde{\boldsymbol{V}}_s^{\mathrm{T}}$$

Scoring new pair i, j: $\quad s_{ij} = \widetilde{\boldsymbol{U}}_{s,i}\widetilde{\boldsymbol{V}}_{s,j}^{\mathrm{T}}$

What if… all our matrix entries are positive…?

Daniel D. Lee & Sebastian Seung

In that case, a factorization algorithm exists with a multiplicative update rule that is guaranteed to converge!

Non-negative Matrix Factorization (NMF):
Given a non-negative matrix $\boldsymbol{V}$, find non-negative matrices $\boldsymbol{W}$ and $\boldsymbol{H}$ such that

$$\boldsymbol{V} \approx \boldsymbol{W}\boldsymbol{H}$$

This can be achieved by minimizing $\|\boldsymbol{V} - \boldsymbol{W}\boldsymbol{H}\|^2 = \sum_{ij}\left(V_{ij} - (\boldsymbol{W}\boldsymbol{H})_{ij}\right)^2$

This term is nonincreasing under the following updates:

$$H_{ij} \leftarrow H_{ij} \frac{\left(\boldsymbol{W}^{\mathrm{T}}\boldsymbol{V}\right)_{ij}}{\left(\boldsymbol{W}^{\mathrm{T}}\boldsymbol{W}\boldsymbol{H}\right)_{ij}} \qquad W_{ij} \leftarrow W_{ij} \frac{\left(\boldsymbol{V}\boldsymbol{H}^{\mathrm{T}}\right)_{ij}}{\left(\boldsymbol{W}\boldsymbol{H}\boldsymbol{H}^{\mathrm{T}}\right)_{ij}}$$

and $\boldsymbol{V} = \boldsymbol{W}\boldsymbol{H}$ is a fixed point of both update rules.

Proof: see handwritten notes

Second version of NMF

Daniel D. Lee & Sebastian Seung

Note: there is a second optimization goal and set of update rules!

Minimizing the divergence $D(\boldsymbol{V}||\boldsymbol{WH}) = \sum_{ij}\left(V_{ij}\log\left(\frac{V_{ij}}{(\boldsymbol{WH})_{ij}}\right) - V_{ij} + (\boldsymbol{WH})_{ij}\right)$

In case
$\sum_{ij} V_{ij} = \sum_{ij}(\boldsymbol{WH})_{ij} = 1$, i.e., the matrices form proper probability distributions,
this is the Kullback-Leibler divergence!

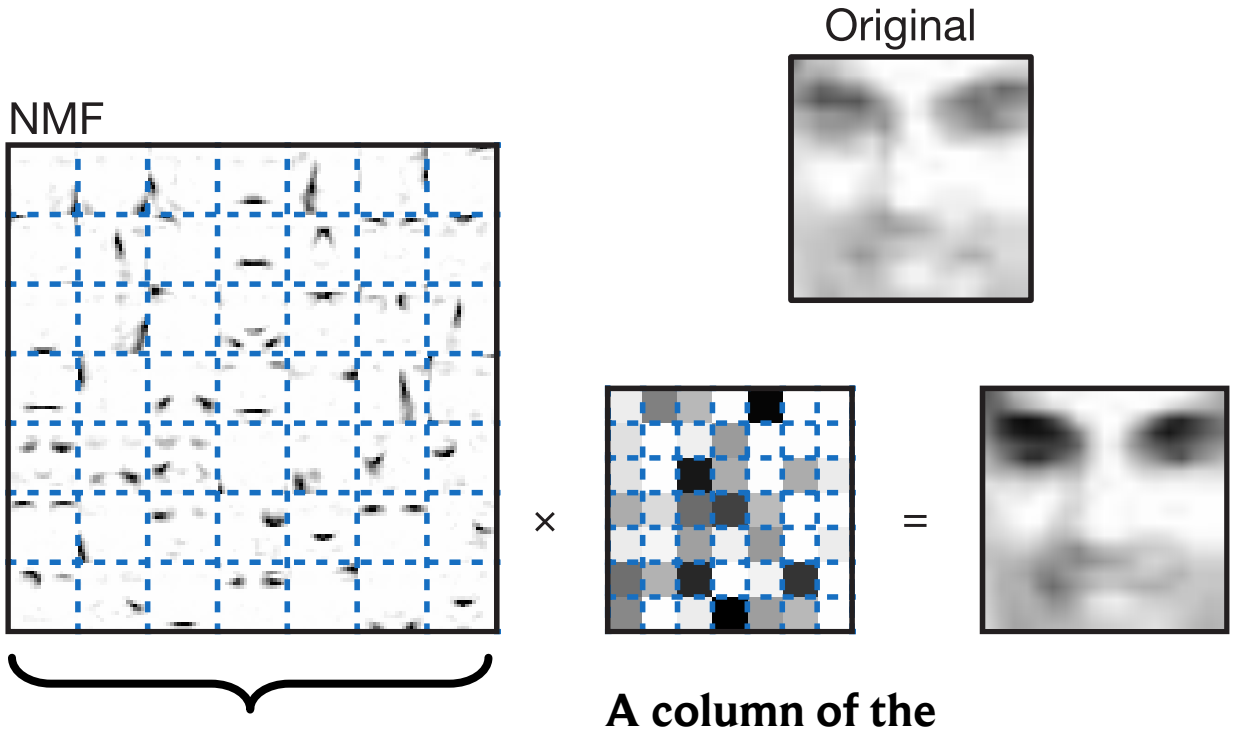
The divergence is nonincreasing under the following updates:

$$H_{ij} \leftarrow H_{ij}\frac{\sum_k W_{ki}V_{kj}/(\boldsymbol{WH})_{kj}}{\sum_k W_{ki}} \qquad W_{ij} \leftarrow W_{ij}\frac{\sum_k H_{jk}V_{ik}/(\boldsymbol{WH})_{ik}}{\sum_k H_{jk}}$$

and $\boldsymbol{V} = \boldsymbol{WH}$ is again a fixed point of both update rules.

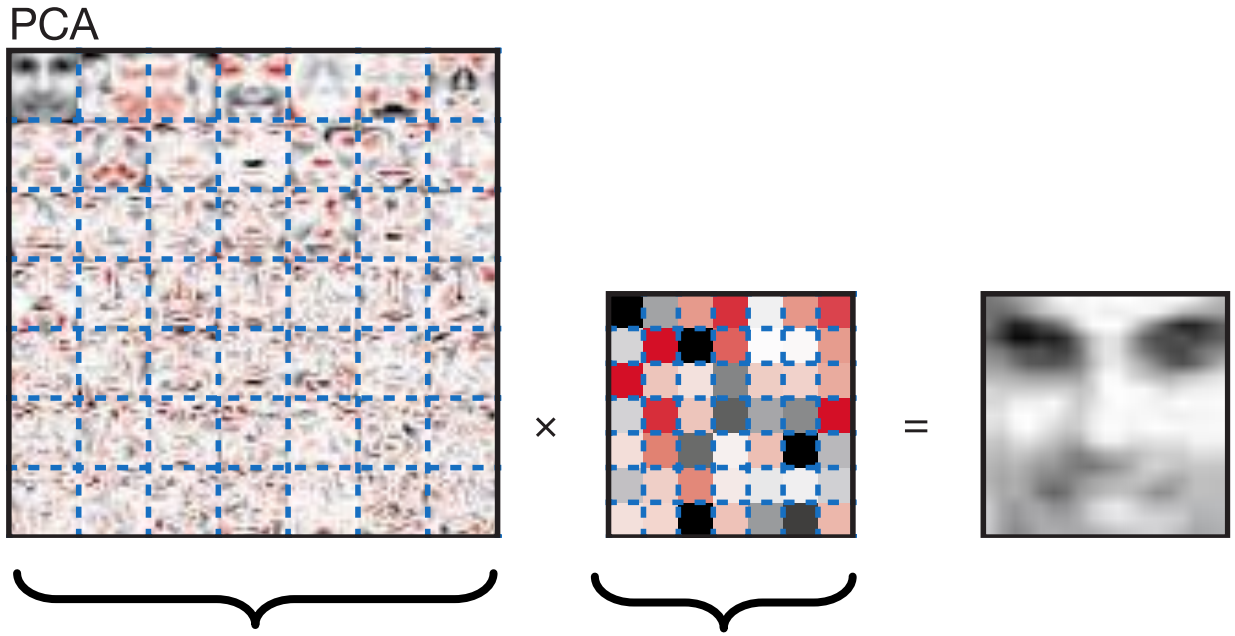
Comparison with PCA

$$\boldsymbol{W} \times \boldsymbol{V}_i^T = \text{reconstructed image } i$$



First non-negative matrix

A column of the second matrix



Eigenvectors from PCA
(the first few resemble faces, so we could call them "eigenfaces")

Low-dimensional representation
(weighting of eigenvectors)

Only additive composition (no negative values):
thus, localized features emerge (not the case for PCA)!

Lee & Seung, Learning the parts of objects by non-negative matrix factorization, Nature 1999

There are some problems though…

1. We had to provide "filler" values for the missing data, which adds bias…
2. Fully decomposing huge matrices is computationally expensive…
3. In particular, real data will be incredibly sparse, which SVD does not utilize…

To solve this, we will go back to 2006, when **Netflix posted a challenge to improve their recommender system** (back then, they borrowed DVDs).

Netflix challenge: Funk SVD (won the 3rd prize!)

To improve SVD, the idea is rather simple:
Instead of doing actual SVD, we just run an optimizer to find appropriate embeddings!

First, some notation:

- s_{ij}: score given by user i to movie j; $D = \{(i,j)\text{: we know } s_{ij}\}$
- $\boldsymbol{u}_i$: vector embedding of user i
- $\boldsymbol{m}_j$: vector embedding of movie j
- b_i: bias of user i (i.e., how critical is this user compared to others?)
- B_j: bias of movie j (i.e., how famous is this movie compared to others?)

Funk SVD then optimizes the following objective:

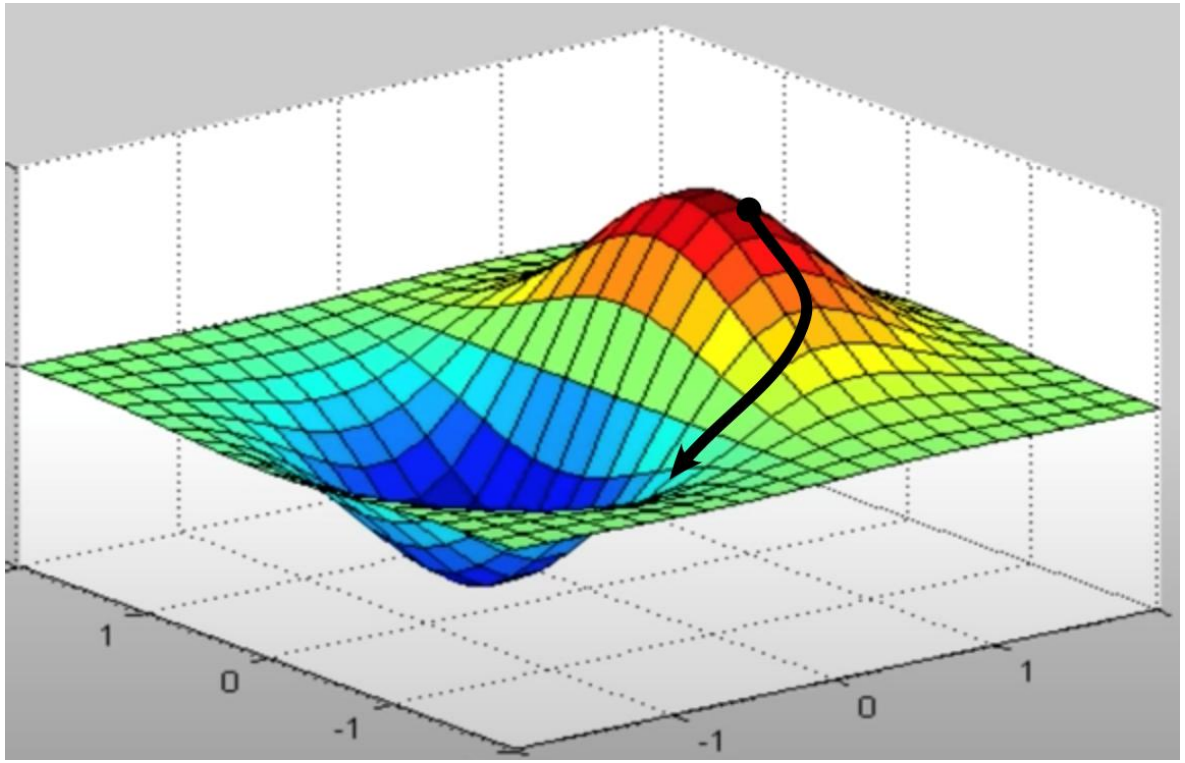
$$\min_{\boldsymbol{u},\boldsymbol{m},\boldsymbol{b},\boldsymbol{B}} L = \frac{1}{2}\sum_{i,j\in D}\left(s_{ij} - \boldsymbol{u}_i^T\boldsymbol{m}_j - b_i - B_j\right)^2 + \frac{1}{2}\sum_{i,j}\lambda\cdot\left(\|\boldsymbol{u}_i\|^2 + \|\boldsymbol{m}_j\|^2 + b_i^2 + B_j^2\right)$$

Matrix factorization, just like SVD!

Regularisation to avoid overfitting!

Hyperparameter we choose!

How do we optimize this? Gradient descent!



The concept is very simple:
calculate the gradient (direction of steepest slope) of L and make a step in this direction to update all parameters.

Basically: parameter updates = going downhill

$$u_i \leftarrow u_i - \eta \nabla_{u_i} L \qquad m_i \leftarrow m_i - \eta \nabla_{m_i} L$$

$$b_i \leftarrow b_i - \eta \nabla_{b_i} L \qquad B_i \leftarrow B_i - \eta \nabla_{B_i} L$$

step size or “learning rate”

Image: https://mriquestions.com/back-propagation.html, 2025

Update rule for Funk SVG

Our loss:

$$L = \frac{1}{2}\sum_{i,j\in D}\left(s_{ij} - \boldsymbol{u}_i^T\boldsymbol{m}_j - b_i - B_j\right)^2 + \frac{1}{2}\sum_{i,j}\lambda\cdot\left(\|\boldsymbol{u}_i\|^2 + \|\boldsymbol{m}_j\|^2 + b_i^2 + B_j^2\right)$$

For simplicity, we introduce the error:

$$\epsilon_{ij} = \begin{cases} s_{ij} - \boldsymbol{u}_i^T\boldsymbol{m}_j - b_i - B_j & \text{if } i,j \in D \\ 0 & \text{else} \end{cases}$$

(actually, the error is undefined as we have no ground truth)

We then get:

$$\boldsymbol{u}_k \leftarrow \boldsymbol{u}_k + \eta\left(\sum_j \epsilon_{kj}\,\boldsymbol{m}_j - \lambda\boldsymbol{u}_k\right) \qquad \boldsymbol{m}_k \leftarrow \boldsymbol{m}_k + \eta\left(\sum_j \epsilon_{jk}\,\boldsymbol{u}_j - \lambda\boldsymbol{m}_k\right)$$

$$b_k \leftarrow b_k + \eta\left(\sum_j \epsilon_{kj} - \lambda b_k\right) \qquad B_k \leftarrow B_k + \eta\left(\sum_j \epsilon_{jk} - \lambda B_k\right)$$

Let's mimic the Netflix challenge!

We use the **MovieLens-100**k dataset:
100,000 ratings from 1000 users on 1700 movies. We use a 100x400 subset here.

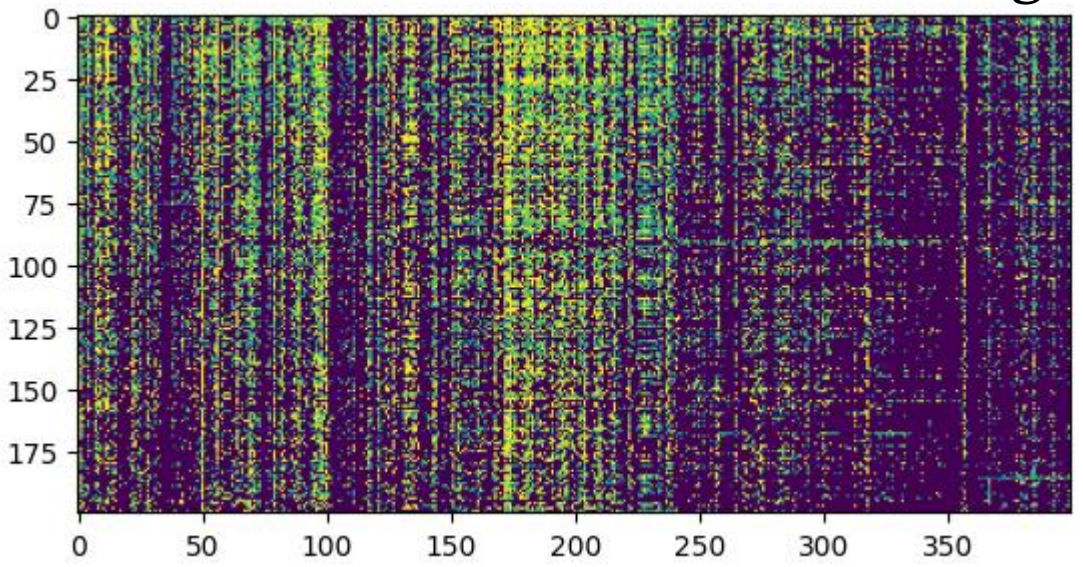
Movies →

Users ↓

Movie 1	Movie 2	Movie 3	…
0.5	?	0.2	0
?	1	0	0.2
0	0	1	?

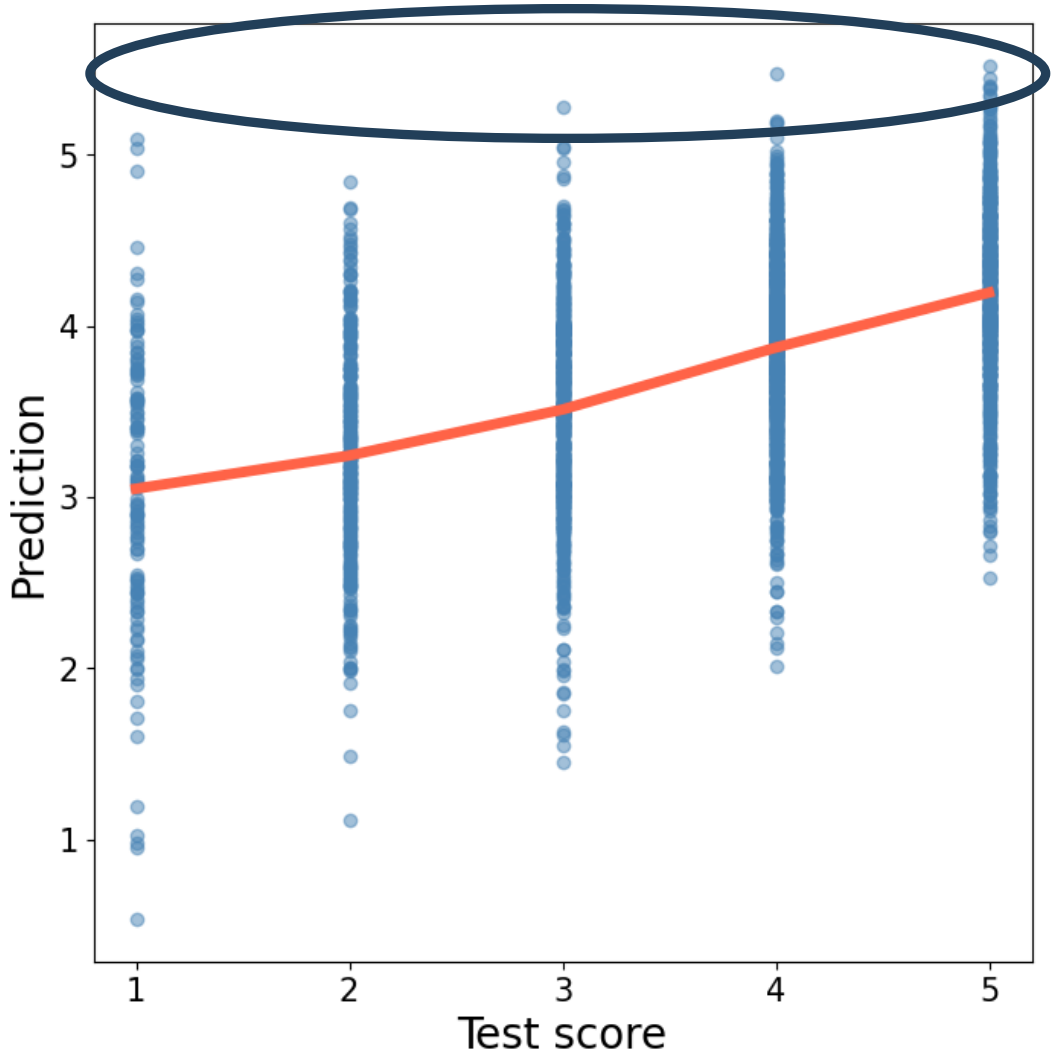
Scores given by users

Table = Matrix = Image



To test, we hold-out some of the scores and see how well our model predicts them (with the mean-squared error!)

Performance of Funk SVD on hold-out data



Our result is not perfect, but we extracted the **trend**!

- The model really struggles with overfitting…
- If we are **only interested in recommending matching movies**, those with high test score are at the top of the list!

To achieve this, we only used the score matrix!

- We did not use any other information, such as the structure of the bipartite graph.
- We did not use any meta-information.
- Everything was done using linear models.

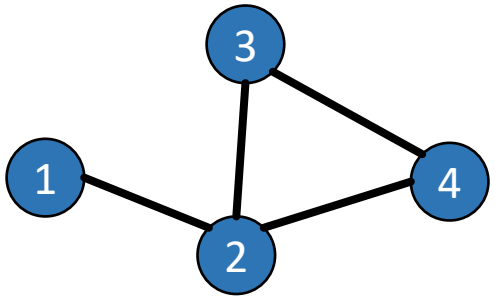
→ We will look at these next

How can we embed nodes just using graph structure?

Basically: nodes with a similar “neighbourhood” should be mapped to similar vectors.

? Does that sound familiar? Any idea how we could achieve this?

Similar idea than for Diffusion Maps: **We perform random walks on the graph!**



Adjacency matrix:

$$\boldsymbol{A} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

Degree matrix: $D_{ii} = \sum_j A_{ij}$

Transition matrix:

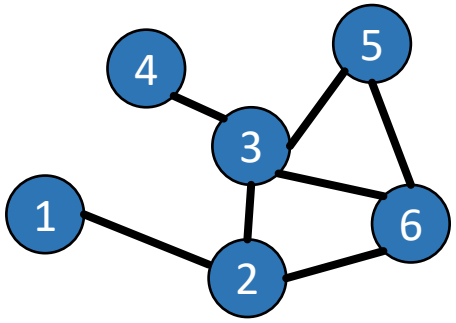
$$\boldsymbol{M} = \boldsymbol{D}^{-1}\boldsymbol{A}$$

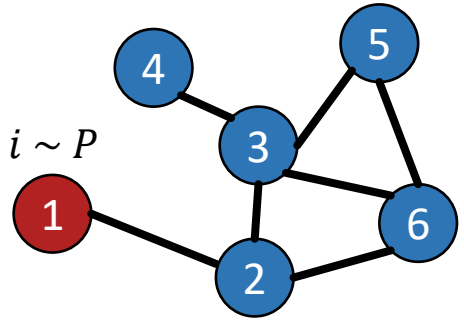
M_{ij}: prob. to jump from node i to node j

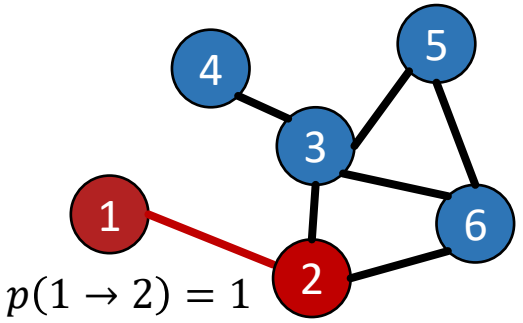
Perozzi et al., Deepwalk: Online Learning of Social Representations (2014)

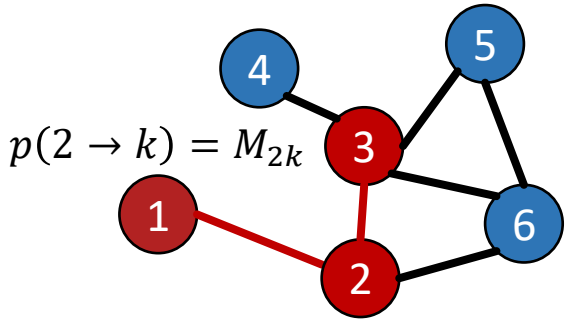
First step of DeepWalk: create "dataset" of random walks

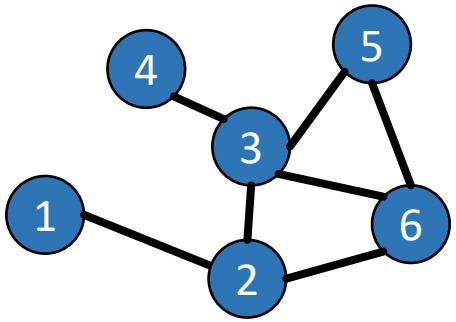
1. Select a random node $P_i = \frac{D_{ii}}{\mathrm{Tr}(\boldsymbol{D})}$
2. Starting from the sampled node, sample a walk of length L using $\boldsymbol{M}$
3. Repeat 1.-2. n times

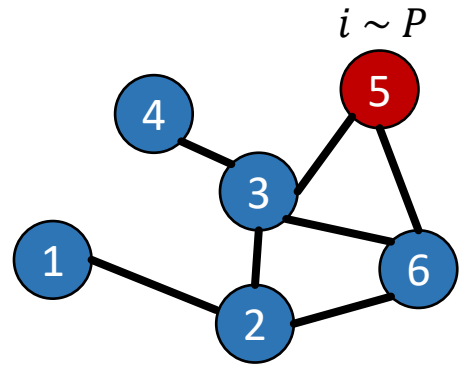


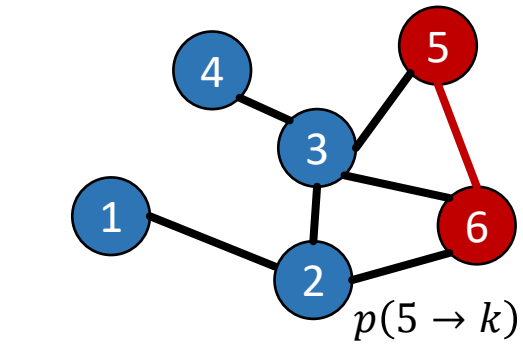


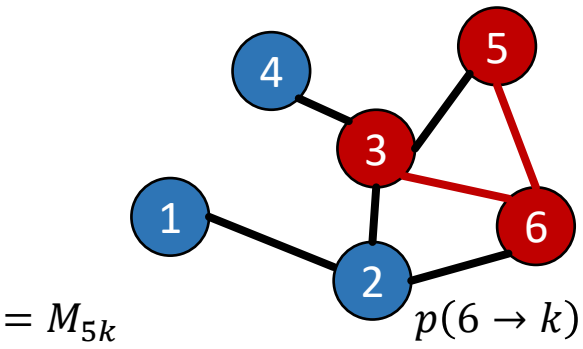








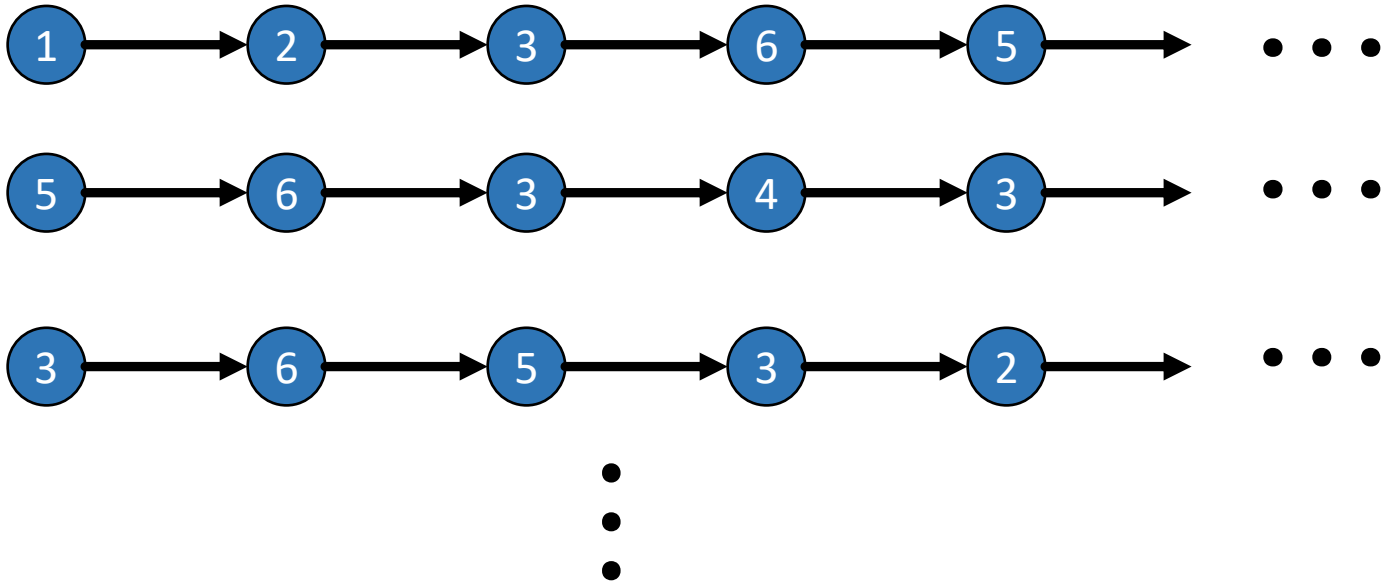




Perozzi et al., Deepwalk: Online Learning of Social Representations (2014)

Second step: calculating embeddings

We now have several sequences of vertices!



Embeddings are found using the Skip-gram algorithm.

Perozzi et al., Deepwalk: Online Learning of Social Representations (2014)

Intermezzo: Skip-gram

Skip-gram is a word-embedding method. It aims at embedding each word to a vector, such that words that occur together within a context window of $\pm T$ words have similar embeddings!

Example: context window of size $T = 3$ around "the"

This is a test sentence to illustrate **the** idea and ensure you understand it.

Example: do "illustrate" and "understand" co-occur in this sentence? **NO**

This is a test sentence to **illustrate** the idea and ensure you understand it.

We define: $P(w, c) = \frac{e^{v_w^{\mathrm{T}} \cdot v_c}}{\sum_{(a,b)} e^{v_a^{\mathrm{T}} \cdot v_b}}$ = the probability of w and c co-occurring!

Problem: the denominator sums over all words… this can become unwieldy!

Mikolov et al., Distributed Representations of Words and Phrases and their Compositionality (2013)

Intermezzo: Skip-gram

Instead, we define: $P(w, c) = \sigma(\boldsymbol{v}_w^{\mathrm{T}} \cdot \boldsymbol{v}_c)$ with $\sigma(x) = (1 + e^{-x})^{-1}$

The embeddings are found by maximizing $\log\left(\sigma\left(\boldsymbol{v}_w^{\mathrm{T}} \cdot \boldsymbol{v}_c\right)\right) + k\ \mathrm{E}_{b \sim P_N}(\log \sigma(-\boldsymbol{v}_w^{\mathrm{T}} \cdot \boldsymbol{v}_b))$

Increase probability for observed (w, c) pairs!

To avoid the normalization factor, generate k "negative samples" that should have low probability!
These are sampled from

$$P_N(b) = \frac{\#b}{\#pairs}$$

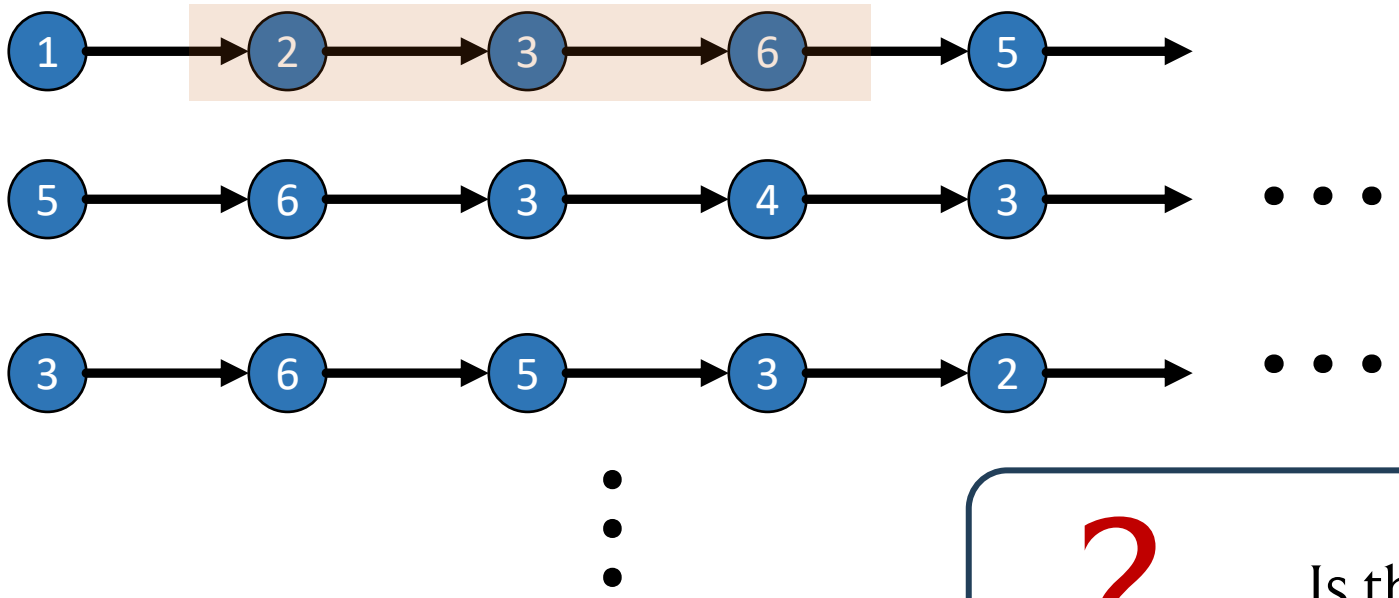
Obtained from the text data!
Basically: fraction of times words b occurs in the word pairs extracted from the text.

Note: we could even use different embeddings for the "target" word and "context" words: $\boldsymbol{v}_w^T \cdot \boldsymbol{u}_c$

Idea: replace words by … nodes in our random walks!

Instead, we define: $P(w, c) = \sigma(\boldsymbol{v}_w^{\mathrm{T}} \cdot \boldsymbol{v}_c)$ with $\sigma(x) = (1 + e^{-x})^{-1}$

The embeddings are found by maximizing $\log\left(\sigma\left(\boldsymbol{v}_w^{\mathrm{T}} \cdot \boldsymbol{v}_c\right)\right) + k\, \mathrm{E}_{b \sim P_N}(\log \sigma(-\boldsymbol{v}_w^{\mathrm{T}} \cdot \boldsymbol{v}_b))$



$P(w, c)$ = probability to see node w and node c co-occurring!

? Is this related to matrix factorization?

DeepWalk implicitly factorizes a matrix we know

DeepWalk is equivalent to factorizing the matrix:

$$S_{wc} = \log\left(\frac{P(w,c)}{P(w)\cdot P(c)}\right) - \log k = \log\left(\frac{\mathrm{Tr}(D)}{T}\left(\sum_{r=1}^{T} \boldsymbol{M}^r\right)\boldsymbol{D}^{-1}\right) - \log k$$

with

$P(w,c)$: frequency of (w,c) co-occurring in random walks.

$P(x)$: frequency of node x occurring in random walks.

k: number of negative samples.

This is also known as pointwise mutual information!

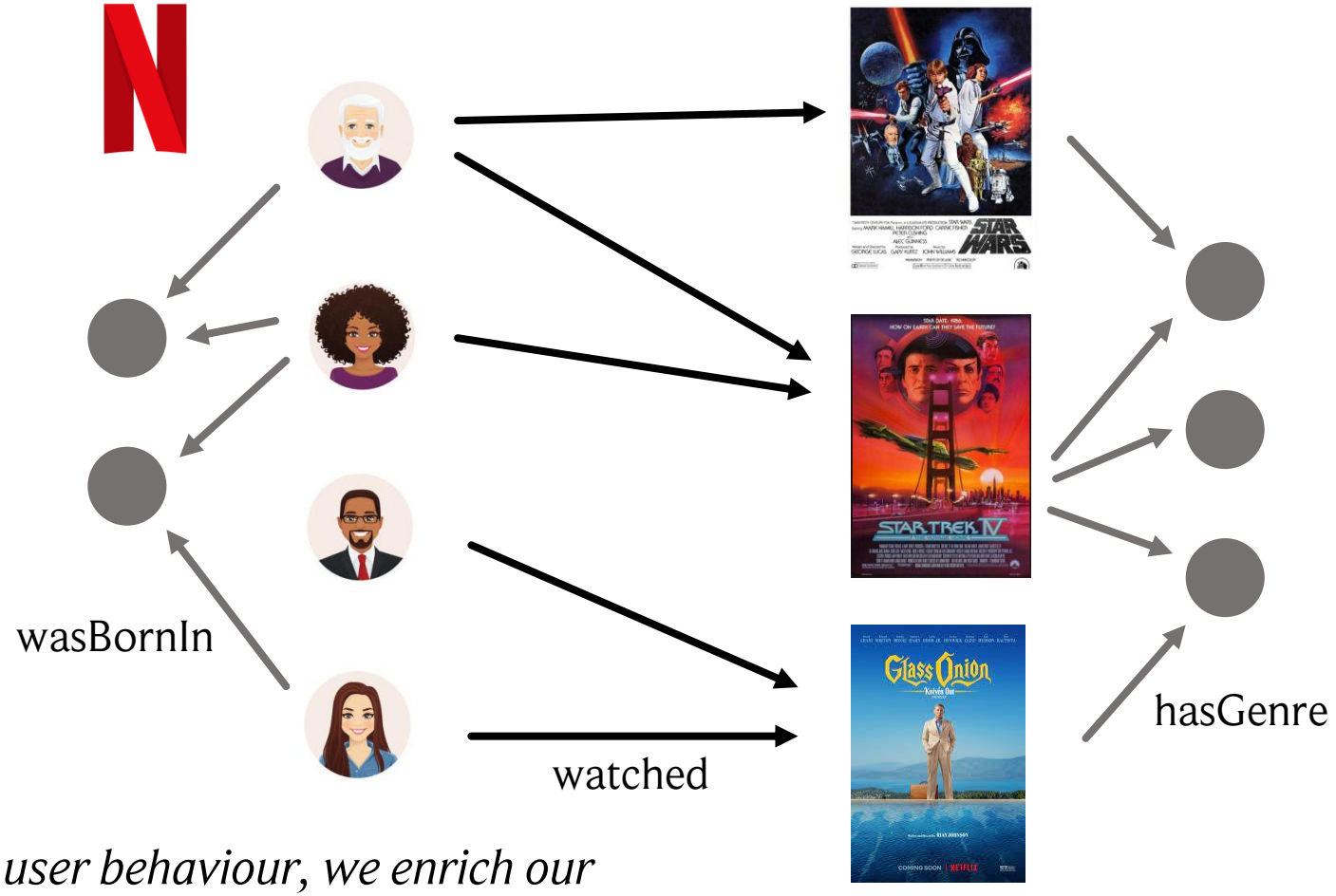
Proof: see handwritten notes

This is also true for many other embedding algorithms!

Qiu et al., Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec (2018)

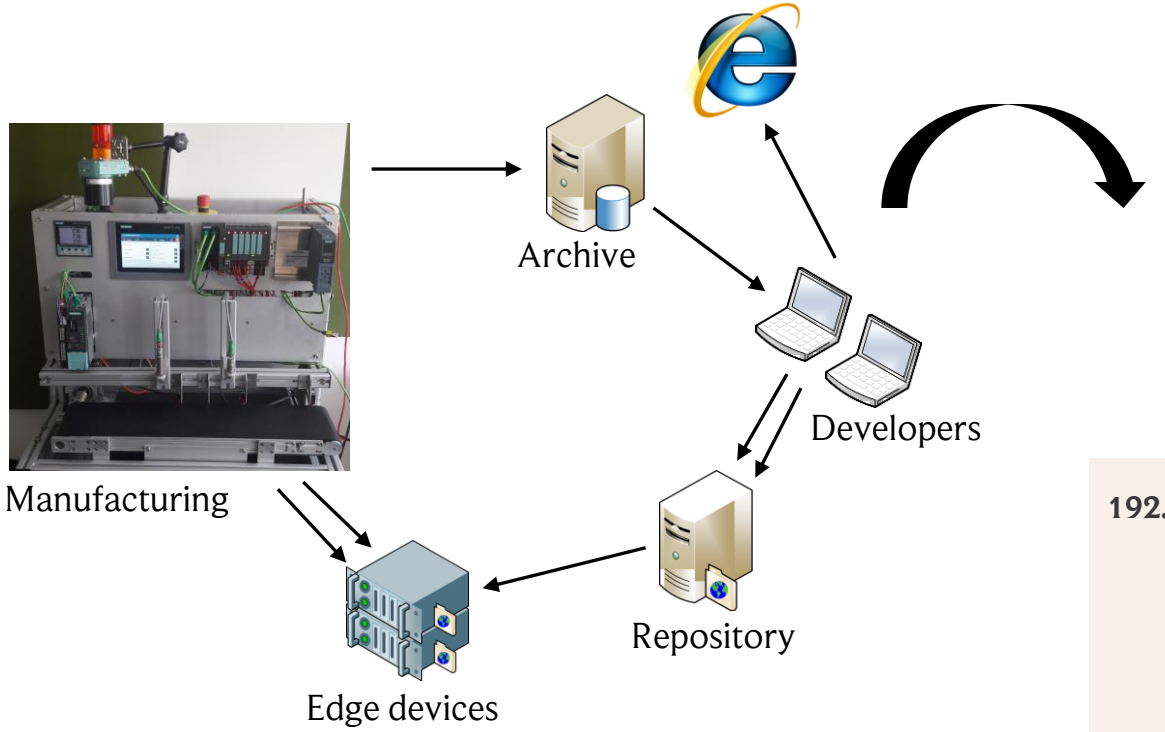
Ok, so far we only looked at non-relational data

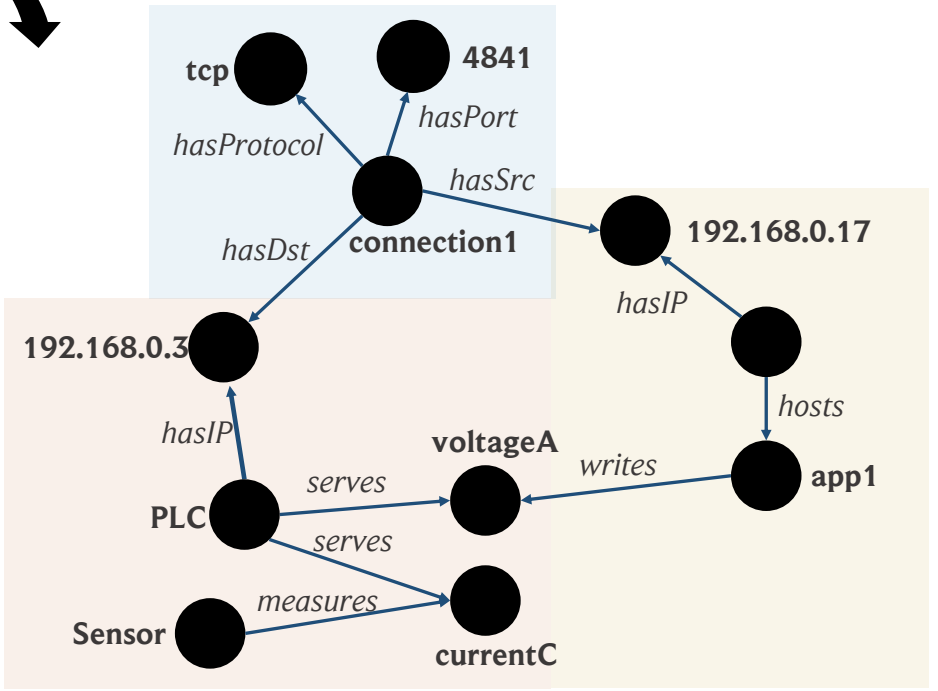
Relational data can be represented as a knowledge graph:
edges have different types, and nodes and edges can have “features”



For instance, in addition to user behaviour, we enrich our graph with meta information about users and movies!

Another example: industrial automation factory

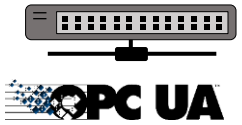




static engineering data

+ dynamic network data

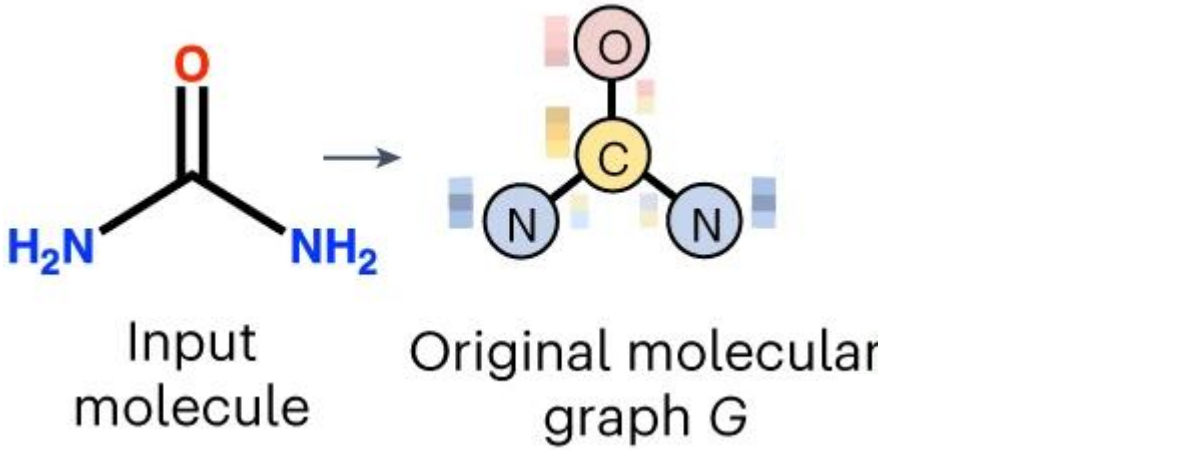
+ dynamic app data



192.168.0.3	*hasType*	**ipAddress**
192.168.0.17	*hasType*	**ipAddress**
edgePC	*hasType*	**device**
edgePC	*hasIP*	**192.168.0.17**
edgePC	*hosts*	**app1**
app1	*hasType*	**industrialApp**
…	…	…

*Note: here we actually turned things around! We **built a recommender system to find anomalies** — anomalies = links we would never recommend!*

Another example: molecules!



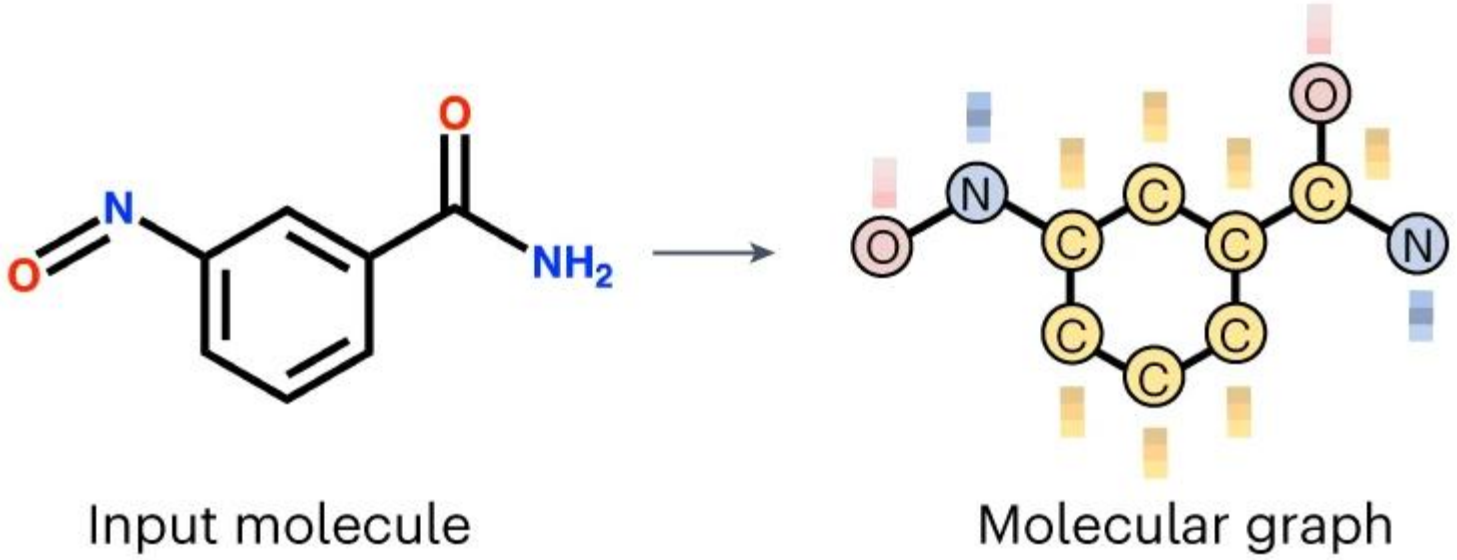
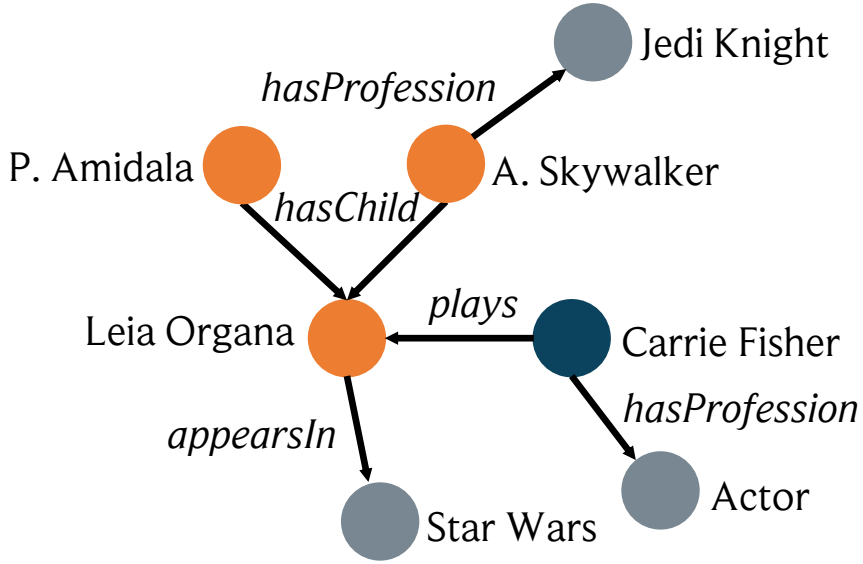


Image credit: Yin Fang et al., Knowledge graph-enhanced molecular contrastive learning with functional prompt (Nature Machine Intelligence, 2023)

In-depth view with a simple example

Raw knowledge graph:



Alternative representation: triples (subject — predicate — object)

Carrie Fisher	*hasProfession*	Actor
Carrie Fisher	*plays*	Leia Organa
P. Amidala	*hasChild*	Leia Organa
…	…	…

What can we learn from this type of data?

Our recommender system setting!

Link prediction: predict new links!

Carrie Fisher	*appearsIn*	Star Wars	?
A. Skywalker	*hasChild*	Carrie Fisher	?

Node classification

Carrie Fisher	Real or Fictional	?
A. Skywalker		

Graph classification

Molecule	HasProperty	?

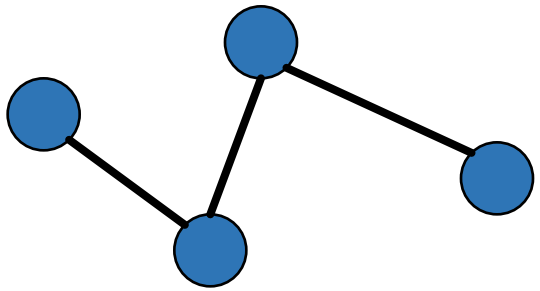
There are some challenges though…!

Looking at real world knowledge graphs, they often have

- thousands to millions of nodes
- that are only very sparsely connected.

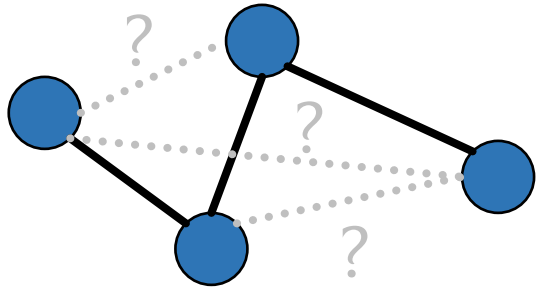
There are two ways how we can treat this situation!

Closed-world assumption (CWA)



Edges missing is the ground-truth.

Open-world assumption (OWA)

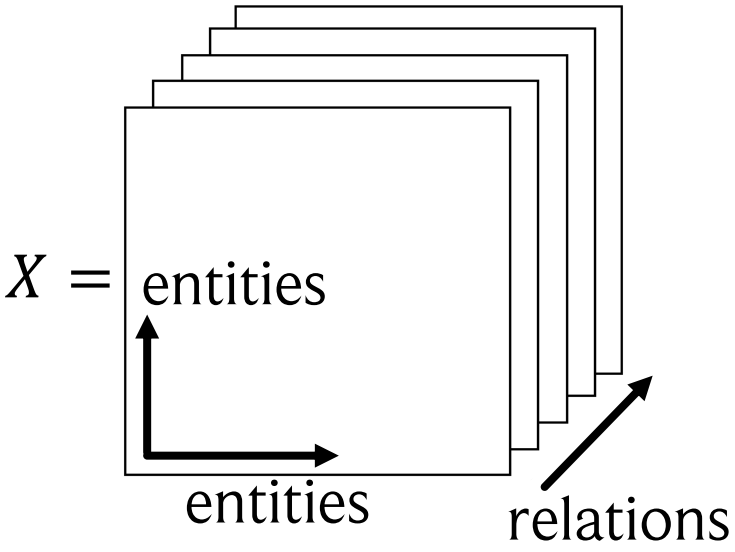


We don't know if edges should be missing or not.

We want to **score unknown edges**, so we go with **OWA**!

RESCAL: representing knowledge graphs as tensors

Our knowledge graph is basically a list of triplets (s, p, o), which we can encode in a tensor as follows:



$$X_{ijk} = \begin{cases} 1 & \text{if the link (i,j,k) is part of our graph} \\ 0 & \text{otherwise} \end{cases}$$

This is the same idea we used for user-movie rankings! X_{ij} would be that, we just added one more dimension for different **edge types**!

Hence, we can use the same idea! **Factorize the tensor using low-rank elements**!

Nickel et al., A Three-Way Model for Collective Learning on Multi-Relational Data (2011)

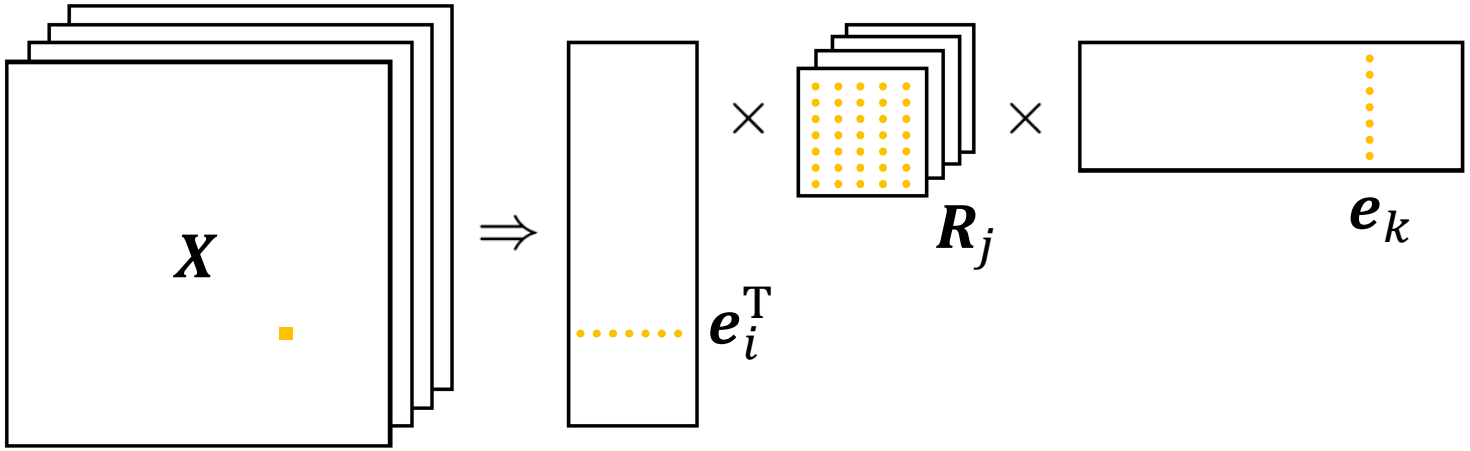
RESCAL: Tensor factorization

Our approach for doing this looks incredibly similar to SVD, NMF, Funk SVD, etc.

$$X_{ijk} \approx \boldsymbol{e}_i^{\mathrm{T}} \boldsymbol{R}_j \boldsymbol{e}_k$$

$\boldsymbol{e}_i$: vector embedding of node i

$\boldsymbol{R}_j$: matrix embedding of edge type j



How do we get the embeddings?

Minimize $\sum_{(i,j,k)\in KG} \left\| X_{ijk} - \boldsymbol{e}_i^{\mathrm{T}} \boldsymbol{R}_j \boldsymbol{e}_k \right\|^2$, e.g., using gradient descent.

all triplets in our knowledge graph KG

Do you see any problem with this approach?

RESCAL: Negative sampling

How do we get the embeddings?

Minimize $\sum_{(i,j,k)\in KG} \left\| X_{ijk} - \boldsymbol{e}_i^{\mathrm{T}} \boldsymbol{R}_j \boldsymbol{e}_k \right\|^2$, e.g., using gradient descent.

! This is **trivially solved** by finding embeddings that result in $\boldsymbol{e}_i^{\mathrm{T}} \boldsymbol{R}_j \boldsymbol{e}_k = 1$ for all i,j,k!

But we can fix this by adding "negative samples" to the mix:

sample from uniform distr.

$$\text{Minimize} \sum_{(i,j,k)\in KG} \left\| X_{ijk} - \boldsymbol{e}_i^{\mathrm{T}} \boldsymbol{R}_j \boldsymbol{e}_k \right\|^2 + \sum_{l=1}^{n_s} \left\| \boldsymbol{e}_r^{\mathrm{T}} \boldsymbol{R}_j \boldsymbol{e}_k \right\|^2 \Big|_{r\sim P} + \sum_{l=1}^{n_o} \left\| \boldsymbol{e}_i^{\mathrm{T}} \boldsymbol{R}_j \boldsymbol{e}_r \right\|^2 \Big|_{r\sim P}$$

So, for every true triplet, **corrupt either the subject or object** to obtain "counter"-examples that should be scored with 0. We only sample a few $(n_s + n_o)$ to not slip into the CWA!

Taking some steps back…

For RESCAL, we decompose our tensor into **a product of vectors and matrices**: $X_{ijk} \approx \boldsymbol{e}_i^{\mathrm{T}} \boldsymbol{R}_j \boldsymbol{e}_k$

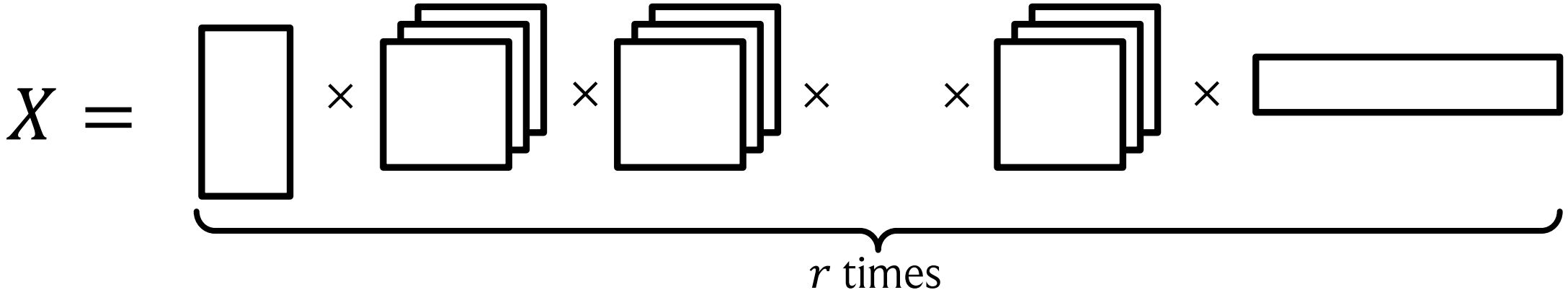
This is an example of a **tensor train** decomposition!

Given a tensor $X \in \mathbb{R}^{n_1 \times \cdots \times n_r}$, a tensor train decomposition of X has the form:

$$X_{i_1 \ldots i_r} = \sum_{\alpha_1, \ldots, \alpha_{r-1}} V^{(1)}_{i_1 \alpha_1} V^{(2)}_{\alpha_1 i_2 \alpha_2} \ldots V^{(r-1)}_{\alpha_{r-2} i_{r-1} \alpha_{r-1}} V^{(r)}_{\alpha_{r-1} i_r}$$

In case of RESCAL: Node embeddings (i_1 / i_r = subject and predicate node) and relation embeddings (i_2 = edge type)

Schematically, it looks like this:



Can we always find a tensor train decomposition?

Yes! This is called the **Tensor Train (TT) SVD**. We will only motivate it briefly here.

First, lets define the **flattening / matricization** of a tensor X:

$$X_{i_1 \dots i_r} \rightarrow X_{i_1, i_2 \dots i_r}$$

Basically: we single out one dimension and flatten the rest into a second, single dimension, ***turning the tensor into a matrix****!*

Then we can find a Tensor decomposition by iteratively applying flattening and SVD:

$$X_{i_1 \dots i_r} \rightarrow X_{i_1 \, , \, i_2 \dots i_r} = \boldsymbol{U}^{(1)} \boldsymbol{\Sigma}^{(1)} \boldsymbol{V}^{(1)} \quad \text{with} \quad \boldsymbol{U}^{(1)} \in \mathbb{R}^{n_1 \times r_1}, \boldsymbol{\Sigma}^{(1)} \boldsymbol{V}^{(1)} \in \mathbb{R}^{r_1 \times i_2 \times \cdots \times i_r}$$

$$\left(\boldsymbol{\Sigma}^{(1)} \boldsymbol{V}^{(1)}\right)_{\alpha_2 i_2 \dots i_r} \rightarrow \left(\boldsymbol{\Sigma}^{(1)} \boldsymbol{V}^{(1)}\right)_{\alpha_2 i_2 \, , \, i_3 \dots i_r} = \boldsymbol{U}^{(2)} \boldsymbol{\Sigma}^{(2)} \boldsymbol{V}^{(2)}$$

and so on…

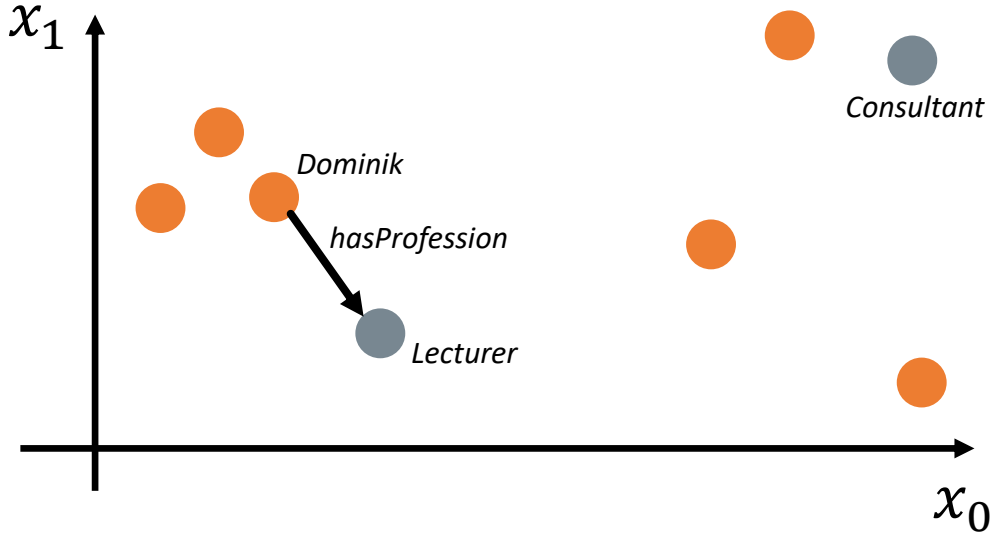
Oseledets & Trytyshnikov, Breaking the Curse of Dimensionality, Or How to Use SVD in Many Dimensions (2009)

Beyond factorization: translational embeddings (TransE)

Next to factorization, translational models are very famous.
A classic one is **TransE** (with lots of variations existing!)

The idea: entities are points, and relations are translations between these points.

Example:



Our score function then is:

$X_{spo} = \left\| \boldsymbol{e}_s + \boldsymbol{r}_p - \boldsymbol{e}_o \right\|^L$ for $L \in \mathbb{N}$

where a low value means that the triplet (s, p, o) has a high likelihood to be true.

We usually find the embeddings using a **margin loss**, i.e., we demand that true triplets have far lower score than negative samples, with a gap γ separating them (similar idea as for SVMs!).

There is a problem!

With our factorization approach, we have so far ignored one problem…

? What if, after factorization, a new user joins for whom no embedding exists?

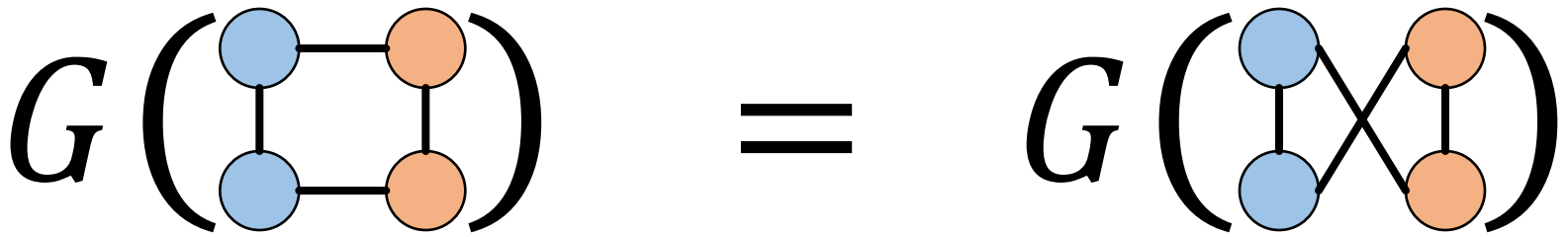
Clearly one can just refactor the new matrix that includes the user…
we need to refactor often in production, as our model does not generalize to unseen “nodes”…
Is there another way?

Inductive models: entering graph neural networks

Graph neural networks are a type of model that can deal with this!

We will focus on the **"graph"** part and treat neural networks in more detail later.
So for now, a neural network is simply a **non-linear function** $f_{\boldsymbol{\theta}}(x)$ with continuous parametrization $\boldsymbol{\theta}$.

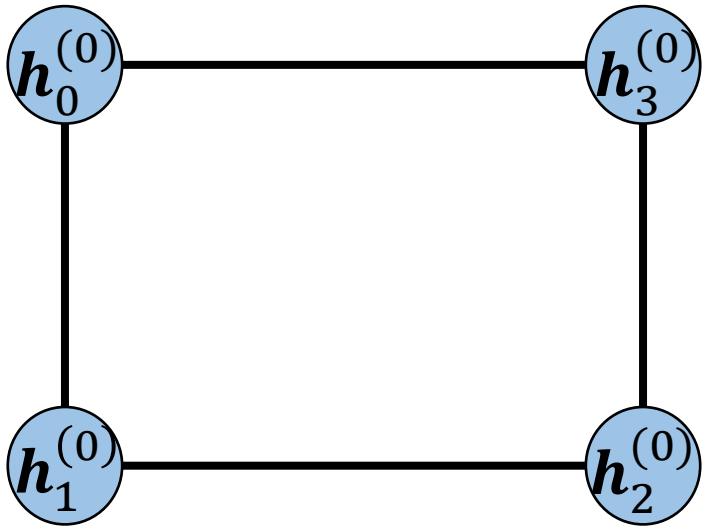
A graph neural network (GNN) G is designed to respect the symmetries of graph structures: a GNN is invariant to the order of nodes of the graph (**permutation invariance**).



The three steps of graph neural networks

How is this achieved?

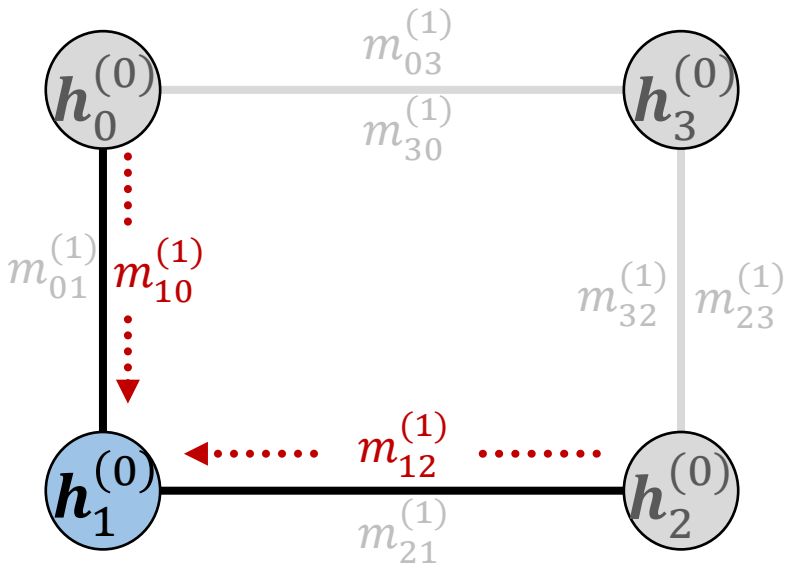
First, every node has a feature vector $\boldsymbol{h}_i^{(0)}$.
A GNN performs three steps to update the node features:



The three steps of graph neural networks

How is this achieved?

First, every node has a feature vector $\boldsymbol{h}_i^{(0)}$.
A GNN performs three steps to update the node features:



"Messages" m are being passed between nodes.
Meaning node 1 learns about the features of its neighbours!

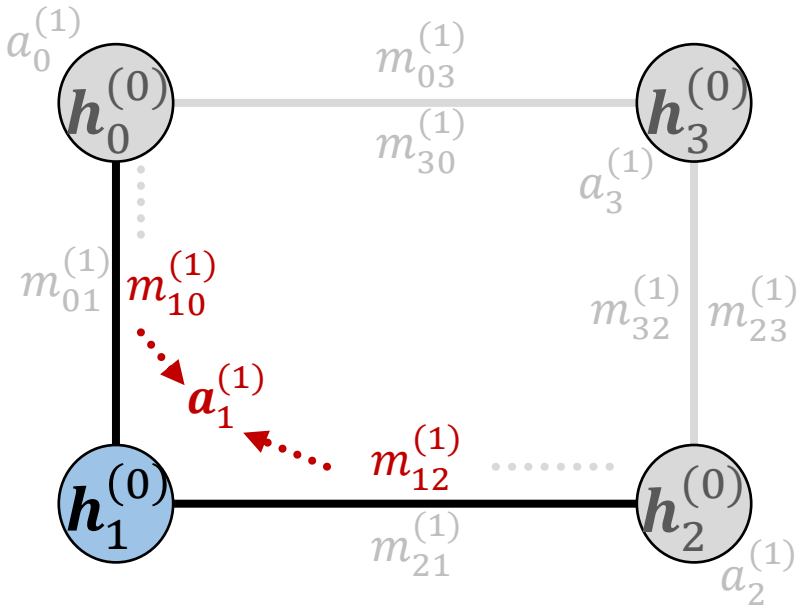
1. Message passing along edges:

$$\boldsymbol{m}_{ij}^{(l)} \leftarrow \mathrm{MSG}\left(\boldsymbol{h}_i^{(l-1)}, \boldsymbol{h}_j^{(l-1)}\right)$$

The three steps of graph neural networks

How is this achieved?

First, every node has a feature vector $\boldsymbol{h}_i^{(0)}$.
A GNN performs three steps to update the node features:



1. Message passing along edges:

$$\boldsymbol{m}_{ij}^{(l)} \leftarrow \mathrm{MSG}\left(\boldsymbol{h}_i^{(l-1)}, \boldsymbol{h}_j^{(l-1)}\right)$$

2. Aggregation:

neighbours of node i

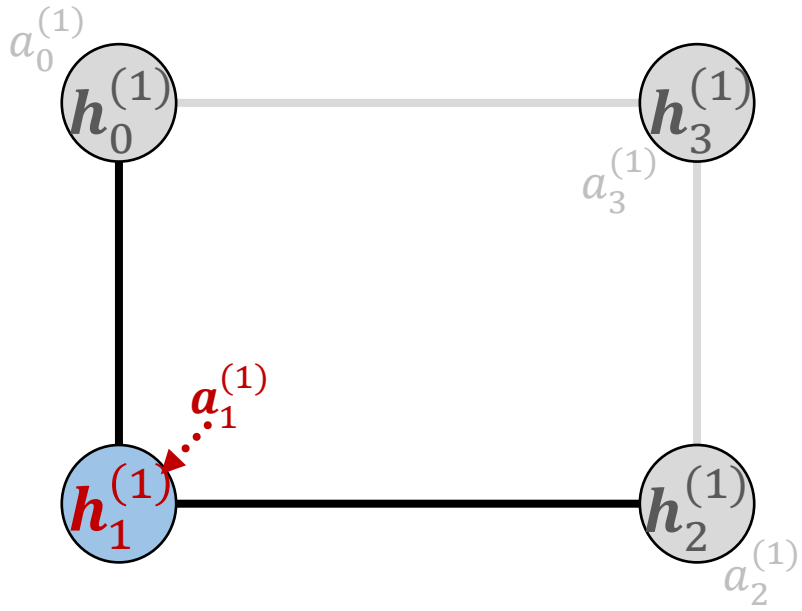
$$\boldsymbol{a}_i^{(l)} \leftarrow \mathrm{AGG}\left(\left\{\boldsymbol{m}_{ij}^{(l)} \mid j \in \mathcal{N}(i)\right\}\right)$$

Take the messages of all neighbours and "aggregate" them into one vector!

The three steps of graph neural networks

How is this achieved?

First, every node has a feature vector $\boldsymbol{h}_i^{(0)}$.
A GNN performs three steps to update the node features:



Assign each node a new feature using the aggregated message.
Each node feature now contains information about its 1-hop neighbourhood!

1. Message passing along edges:

$$\boldsymbol{m}_{ij}^{(l)} \leftarrow \mathrm{MSG}\left(\boldsymbol{h}_i^{(l-1)}, \boldsymbol{h}_j^{(l-1)}\right)$$

2. Aggregation:

neighbours of node i

$$\boldsymbol{a}_i^{(l)} \leftarrow \mathrm{AGG}\left(\left\{\boldsymbol{m}_{ij}^{(l)} \mid j \in \mathcal{N}(i)\right\}\right)$$

3. Update node features:

$$\boldsymbol{h}_i^{(l+1)} \leftarrow \mathrm{UPD}\left(\boldsymbol{h}_i^{(l)}, \boldsymbol{a}_i^{(l)}\right)$$

The three steps of graph neural networks

How is this achieved?

First, every node has a feature vector $\boldsymbol{h}_i^{(0)}$.
A GNN performs three steps to update the node features:

MSG, **UPD** are usually neural networks.

AGG is different: it has to be permutation invariant, e.g., $\mathrm{AGG}(x, y) = \mathrm{AGG}(y, x)$

E.g., a simple sum or average satisfies this!

1. Message passing along edges:

$$\boldsymbol{m}_{ij}^{(l)} \leftarrow \mathrm{MSG}\left(\boldsymbol{h}_i^{(l-1)}, \boldsymbol{h}_j^{(l-1)}\right)$$

2. Aggregation:

neighbours of node i

$$\boldsymbol{a}_i^{(l)} \leftarrow \mathrm{AGG}\left(\left\{\boldsymbol{m}_{ij}^{(l)} \mid j \in \mathcal{N}(i)\right\}\right)$$

3. Update node features:

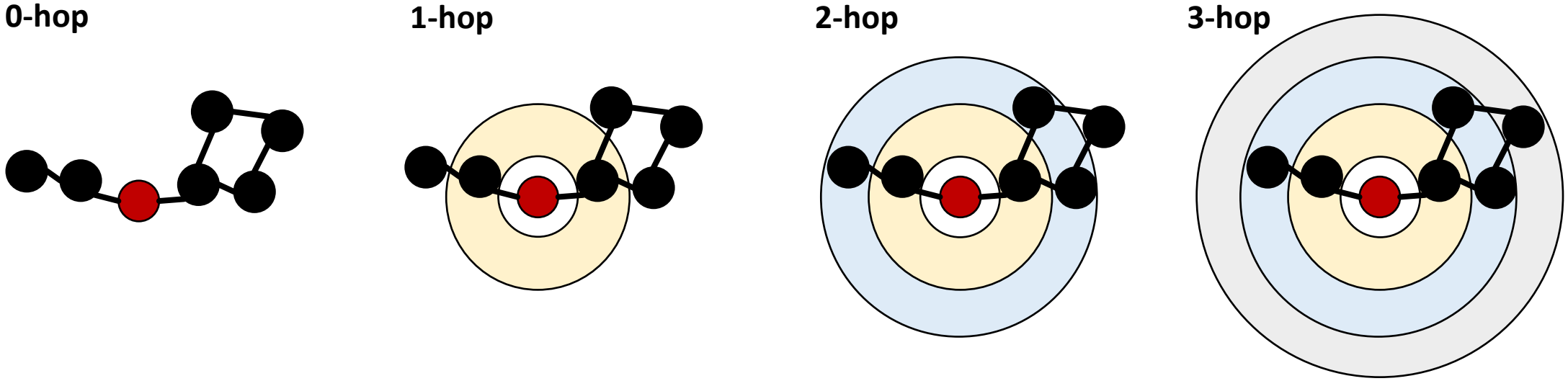
$$\boldsymbol{h}_i^{(l+1)} \leftarrow \mathrm{UPD}\left(\boldsymbol{h}_i^{(l)}, \boldsymbol{a}_i^{(l)}\right)$$

The full graph neural network

Doing these steps once is **“1 layer”** of a graph neural network.

We usually repeat this k times. After k iterations, the features of nodes contain information about other nodes **that are at most k-hops (links) away** from them!

Thus, through these iterations **we “enrich” the node features** using local aggregation!



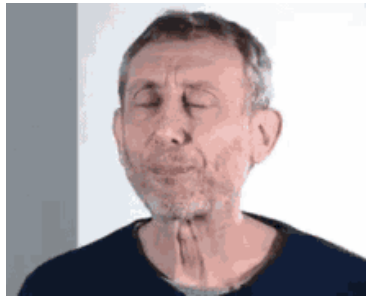
The full graph neural network

After k steps, one then uses a permutation invariant **READOUT** function that takes the aggregated features $\boldsymbol{h}_i^{(k)}$ as input and produces the final output.

Note that this way, the **whole GNN is permutation invariant**.
In fact, operations are **only performed between nodes**, independent of the number of nodes.

Thus, we can apply a GNN to **any** graph with similar structure
(e.g., same number of initial node features, same number of edge types, etc.)!

In the recommender system setting, that means we can handle new content without retraining / refactorizing everything…!



Expressivity of graph neural networks

What can we learn using graph neural networks?

There is a very simple result on that!

Let ϕ be a graph neural network and G_1, G_2 two non-isomorphic graphs.

If $\phi(G_1) \neq \phi(G_2)$, then the Weisfeiler-Lehman test also decides that G_1, G_2 are not isomorphic.

In other terms: GNNs are at most as good as the Weisfeiler-Lehman (WL) test at distinguishing graphs. If the WL test cannot distinguish two graphs, then the GNN cannot distinguish them either, producing the same output for both!

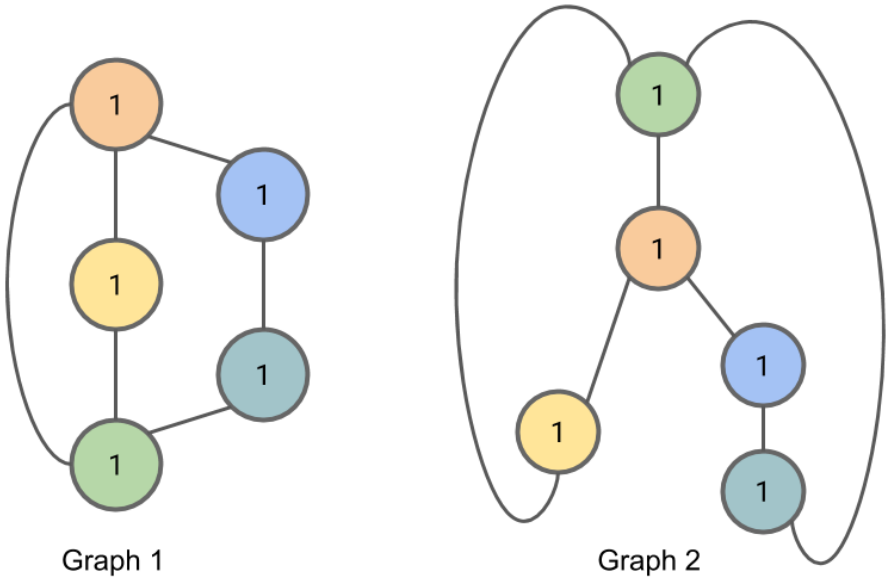
So, what is the Weisfeiler-Lehman test?

The 1-dimensional Weisfeiler-Lehman (WL) test

Problem: You have two graphs G_1 and G_2, and you would like to know if they are the same. Of course, the nodes are labelled differently, so we have to find an isomorphic (1-to-1) mapping that assigns each node in G_1 its counterpart in G_2.

There is currently no algorithm that solves this!
But the WL can be used to figure out, in many cases, **if** two graphs are **not** isomorphic.

How it works:



Step 0: Initialize both graphs the same way

Assign features to all nodes. Here, we use a compressed representation by just labelling nodes that have the same features (here, we use the label 1).

If all nodes are identical, they start all with the same label.

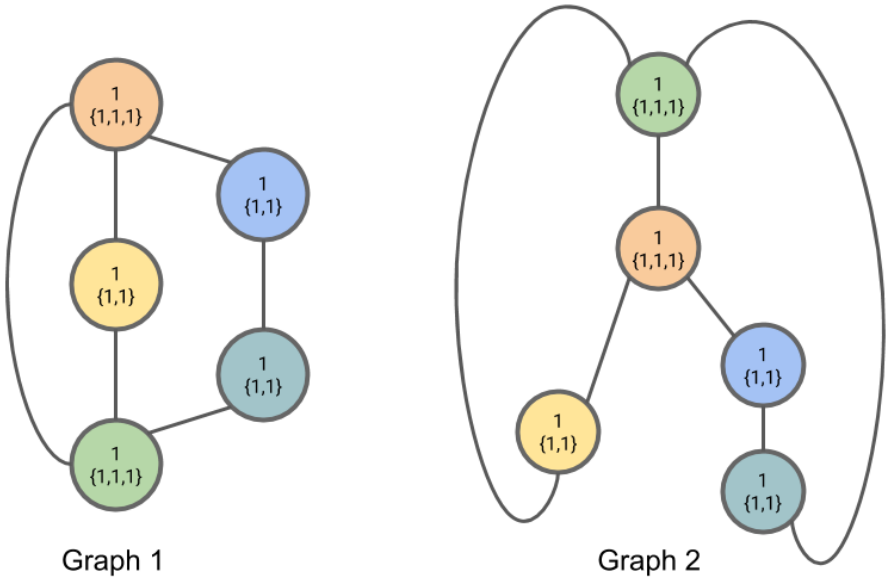
Image credit: https://davidbieber.com/post/2019-05-10-weisfeiler-lehman-isomorphism-test/

The 1-dimensional Weisfeiler-Lehman (WL) test

Problem: You have two graphs G_1 and G_2, and you would like to know if they are the same. Of course, the nodes are labelled differently, so we have to find an isomorphic (1-to-1) mapping that assigns each node in G_1 its counterpart in G_2.

There is currently no algorithm that solves this!
But the WL can be used to figure out, in many cases, **if** two graphs are **not** isomorphic.

How it works:



Step 1: Message and aggregate

For each node, look at neighbouring nodes and aggregate a multiset of their features.

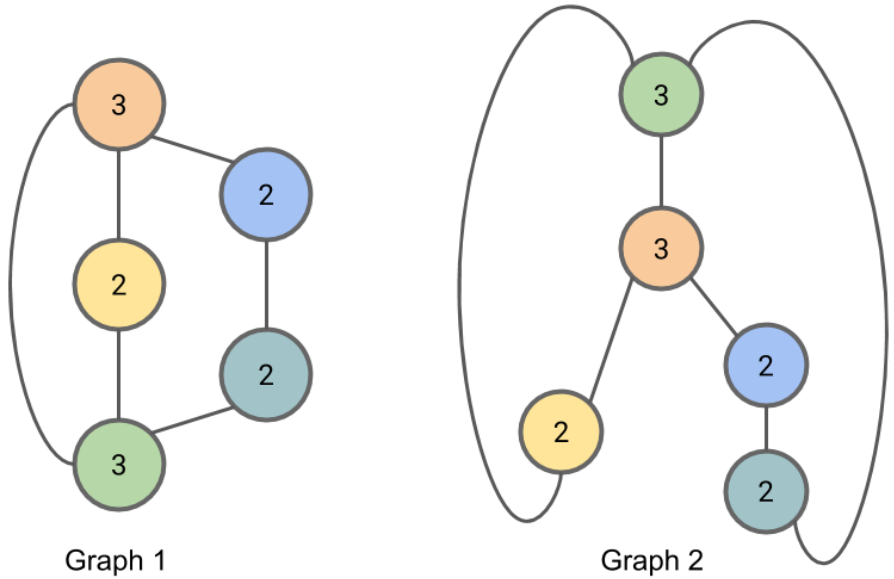
Image credit: https://davidbieber.com/post/2019-05-10-weisfeiler-lehman-isomorphism-test/

The 1-dimensional Weisfeiler-Lehman (WL) test

Problem: You have two graphs G_1 and G_2, and you would like to know if they are the same. Of course, the nodes are labelled differently, so we have to find an isomorphic (1-to-1) mapping that assigns each node in G_1 its counterpart in G_2.

There is currently no algorithm that solves this!
But the WL can be used to figure out, in many cases, **if** two graphs are **not** isomorphic.

How it works:



Step 2: Update

Map all multisets to new labels. This has to be injective: different multisets are always mapped to different labels!

Here, we mapped:

$$\{1,1\} \rightarrow 2, \qquad \{1,1,1\} \rightarrow 3$$

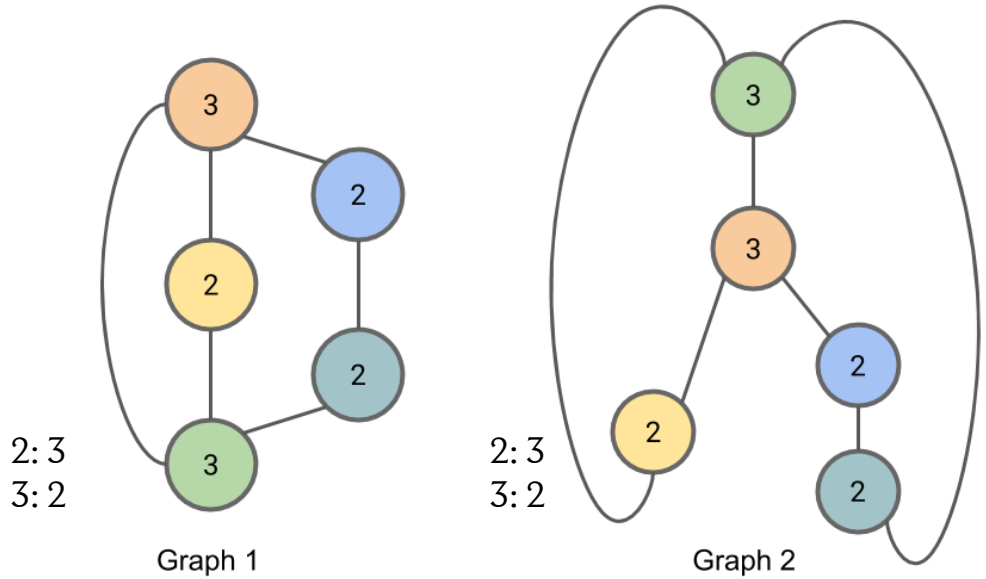
Image credit: https://davidbieber.com/post/2019-05-10-weisfeiler-lehman-isomorphism-test/

The 1-dimensional Weisfeiler-Lehman (WL) test

Problem: You have two graphs G_1 and G_2, and you would like to know if they are the same. Of course, the nodes are labelled differently, so we have to find an isomorphic (1-to-1) mapping that assigns each node in G_1 its counterpart in G_2.

There is currently no algorithm that solves this!
But the WL can be used to figure out, in many cases, **if** two graphs are **not** isomorphic.

How it works:



Repeat step 2 and 3

We do this k times or until the distribution of labels does not change between steps.

If at any step, the multiset of all labels of G_1 and G_2 is different, we stop and the test decides that the two graphs are not isomorphic. **Otherwise, the test is inconclusive**.

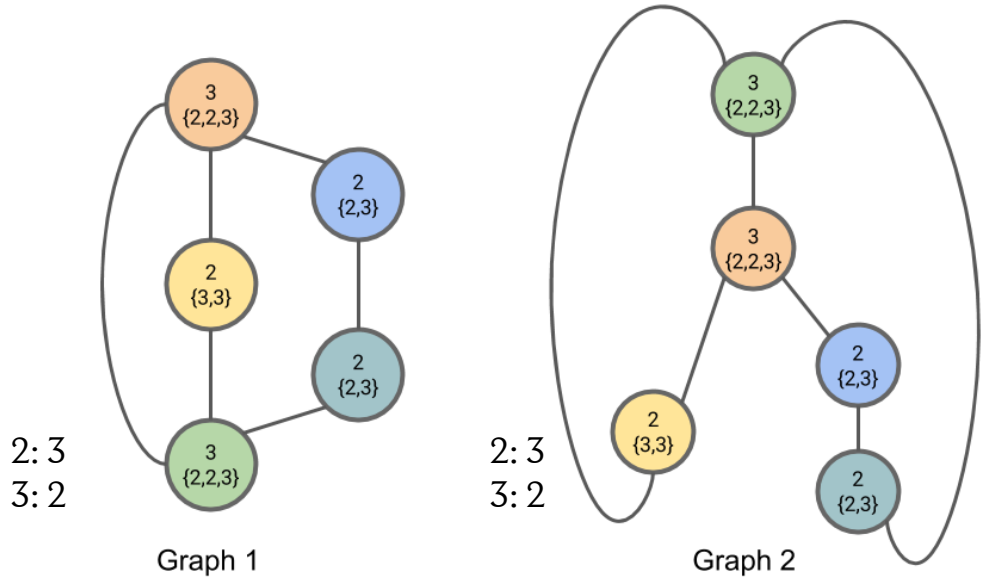
Image credit: https://davidbieber.com/post/2019-05-10-weisfeiler-lehman-isomorphism-test/

The 1-dimensional Weisfeiler-Lehman (WL) test

Problem: You have two graphs G_1 and G_2, and you would like to know if they are the same. Of course, the nodes are labelled differently, so we have to find an isomorphic (1-to-1) mapping that assigns each node in G_1 its counterpart in G_2.

There is currently no algorithm that solves this!
But the WL can be used to figure out, in many cases, **if** two graphs are **not** isomorphic.

How it works:



Repeat step 2 and 3

We do this k times or until the distribution of labels does not change between steps.

If at any step, the multiset of all labels of G_1 and G_2 is different, we stop and the test decides that the two graphs are not isomorphic. **Otherwise, the test is inconclusive**.

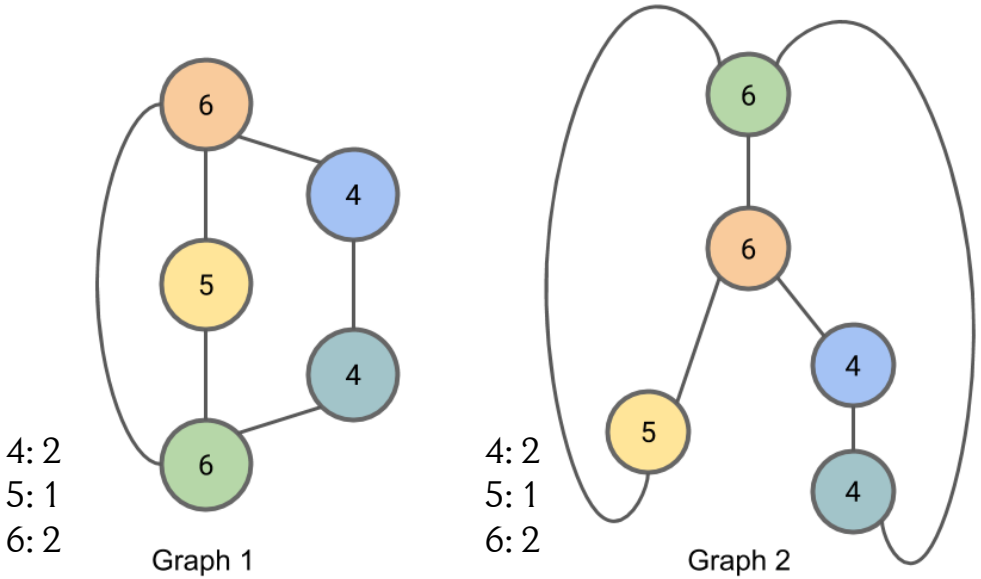
Image credit: https://davidbieber.com/post/2019-05-10-weisfeiler-lehman-isomorphism-test/

The 1-dimensional Weisfeiler-Lehman (WL) test

Problem: You have two graphs G_1 and G_2, and you would like to know if they are the same. Of course, the nodes are labelled differently, so we have to find an isomorphic (1-to-1) mapping that assigns each node in G_1 its counterpart in G_2.

There is currently no algorithm that solves this!
But the WL can be used to figure out, in many cases, **if** two graphs are **not** isomorphic.

How it works:



Repeat step 2 and 3

We do this k times or until the distribution of labels does not change between steps.

If at any step, the multiset of all labels of G_1 and G_2 is different, we stop and the test decides that the two graphs are not isomorphic. **Otherwise, the test is inconclusive**.

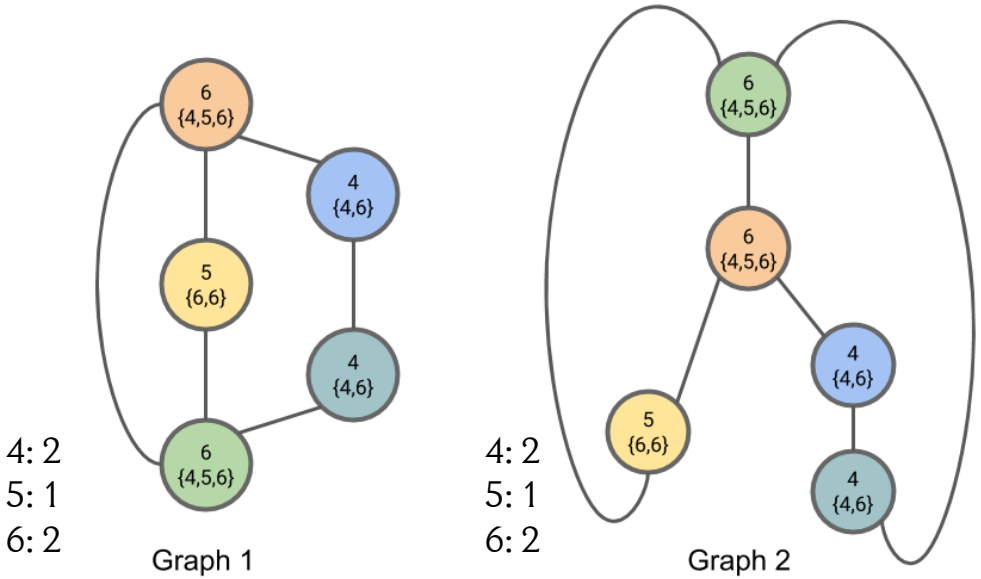
Image credit: https://davidbieber.com/post/2019-05-10-weisfeiler-lehman-isomorphism-test/

The 1-dimensional Weisfeiler-Lehman (WL) test

Problem: You have two graphs G_1 and G_2, and you would like to know if they are the same. Of course, the nodes are labelled differently, so we have to find an isomorphic (1-to-1) mapping that assigns each node in G_1 its counterpart in G_2.

There is currently no algorithm that solves this!
But the WL can be used to figure out, in many cases, **if** two graphs are **not** isomorphic.

How it works:



Repeat step 2 and 3

We do this k times or until the distribution of labels does not change between steps.

If at any step, the multiset of all labels of G_1 and G_2 is different, we stop and the test decides that the two graphs are not isomorphic. **Otherwise, the test is inconclusive**.

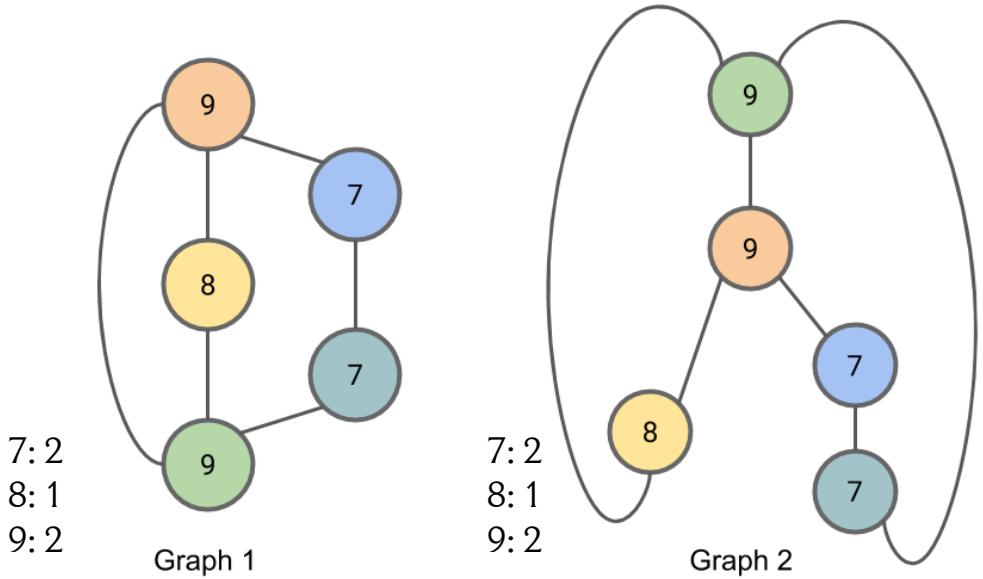
Image credit: https://davidbieber.com/post/2019-05-10-weisfeiler-lehman-isomorphism-test/

The 1-dimensional Weisfeiler-Lehman (WL) test

Problem: You have two graphs G_1 and G_2, and you would like to know if they are the same. Of course, the nodes are labelled differently, so we have to find an isomorphic (1-to-1) mapping that assigns each node in G_1 its counterpart in G_2.

There is currently no algorithm that solves this!
But the WL can be used to figure out, in many cases, **if** two graphs are **not** isomorphic.

How it works:



Repeat step 2 and 3

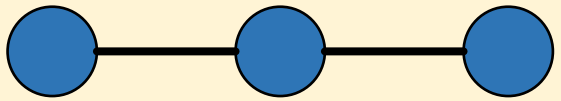
We do this k times or until the distribution of labels does not change between steps.

If at any step, the multiset of all labels of G_1 and G_2 is different, we stop and the test decides that the two graphs are not isomorphic. **Otherwise, the test is inconclusive**.

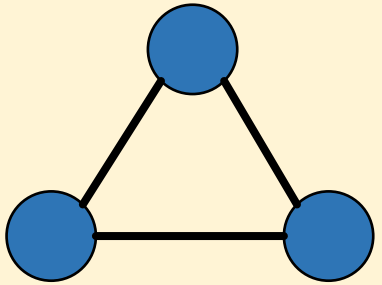
Image credit: https://davidbieber.com/post/2019-05-10-weisfeiler-lehman-isomorphism-test/

Example 1: when it works

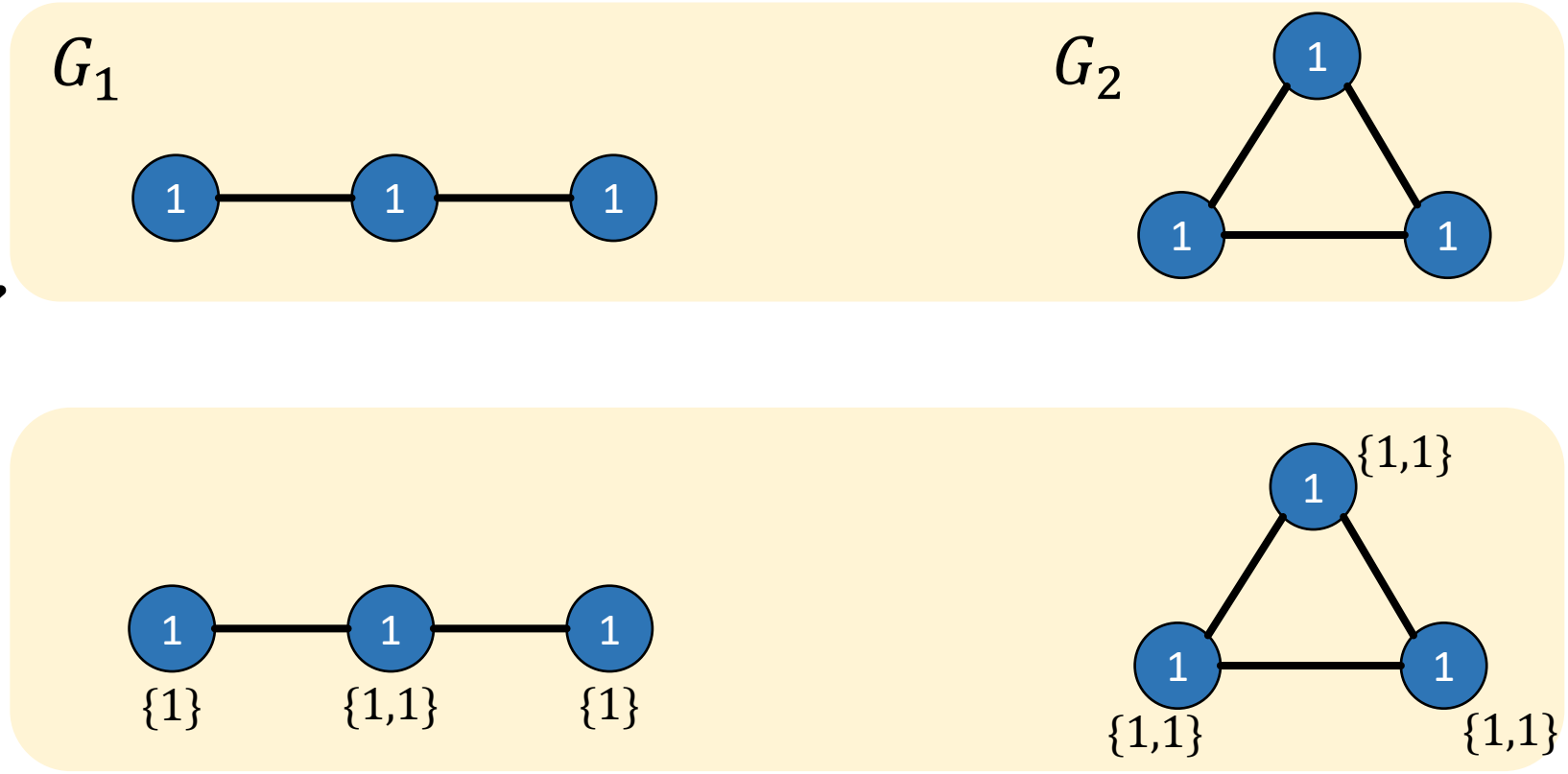
G_1



G_2



Example 1: when it works



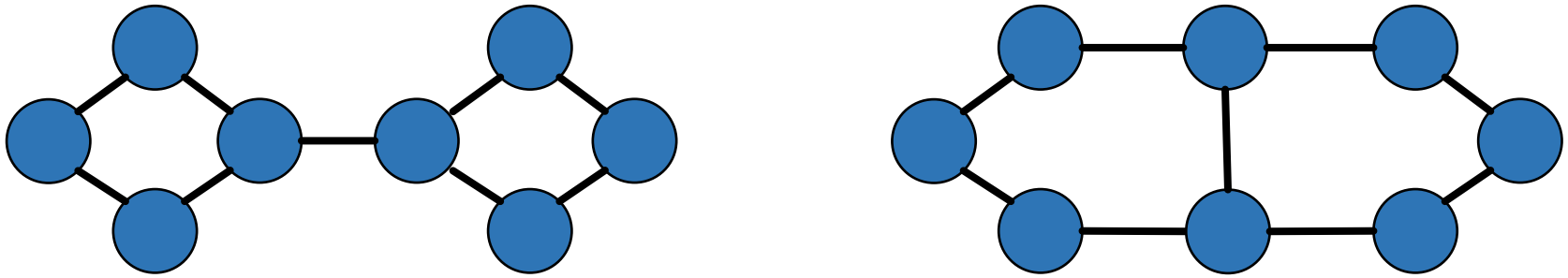
Label distribution

G_1 2: 2, 3: 1 G_2 3: 3

Graphs are different!

Example 2: when it doesn't work

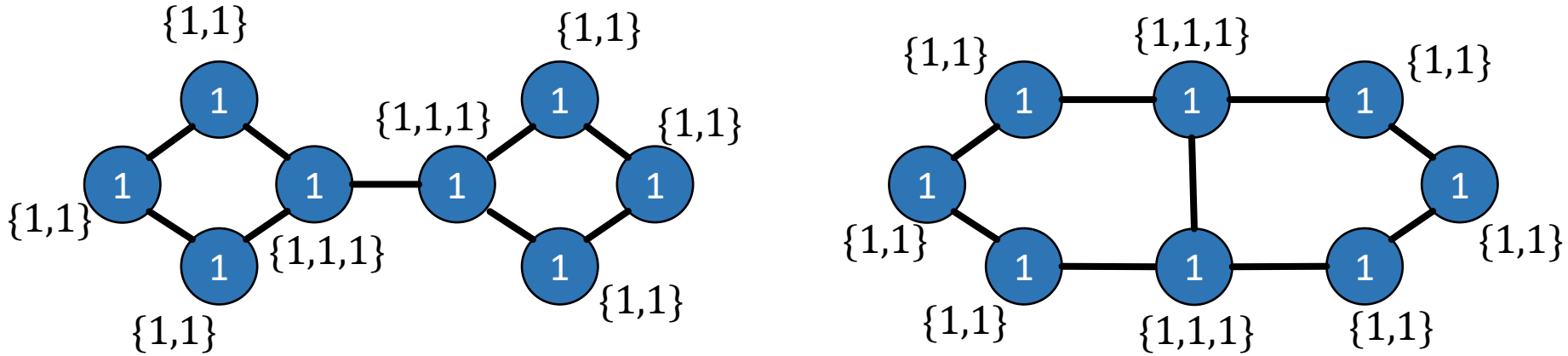
These two graphs are clearly not identical, as we can cut the left one into two separate parts with only one cut, while the right one requires at least two cuts to do so! Let's run 1-WL and see if it agrees!



Example 2: when it doesn't work

These two graphs are clearly not identical, as we can cut the left one into two separate parts with only one cut, while the right one requires at least two cuts to do so! Let's run 1-WL and see if it agrees!

Message + aggregate



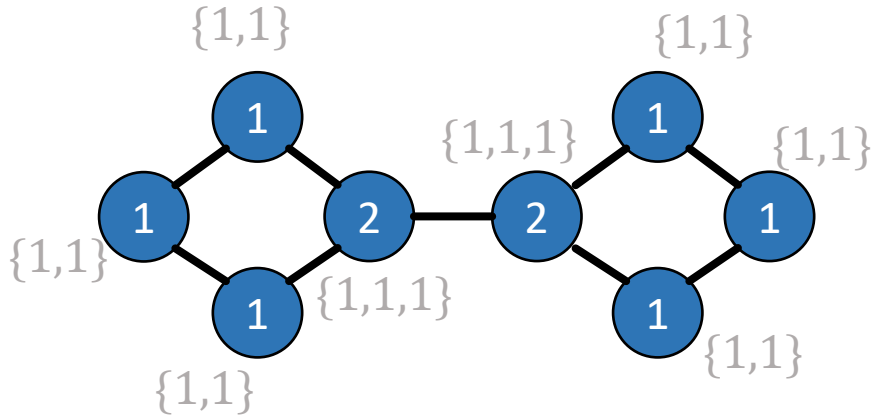
Example 2: when it doesn't work

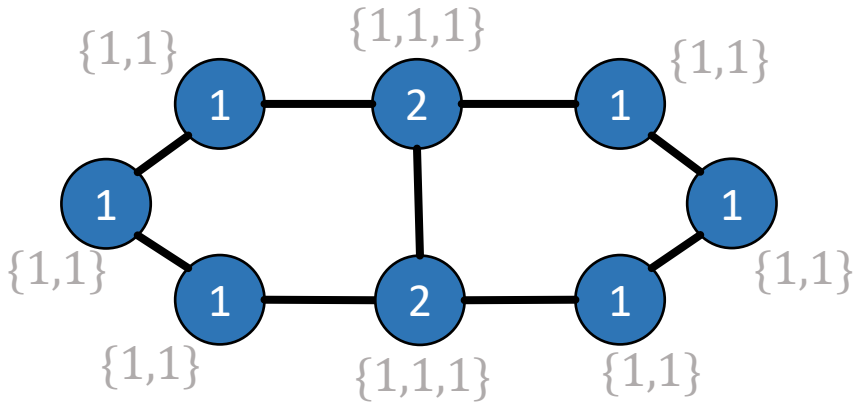
These two graphs are clearly not identical, as we can cut the left one into two separate parts with only one cut, while the right one requires at least two cuts to do so! Let's run 1-WL and see if it agrees!

Update

$\{1,1\} \rightarrow 1$

$\{1,1,1\} \rightarrow 2$





Check stopping condition:

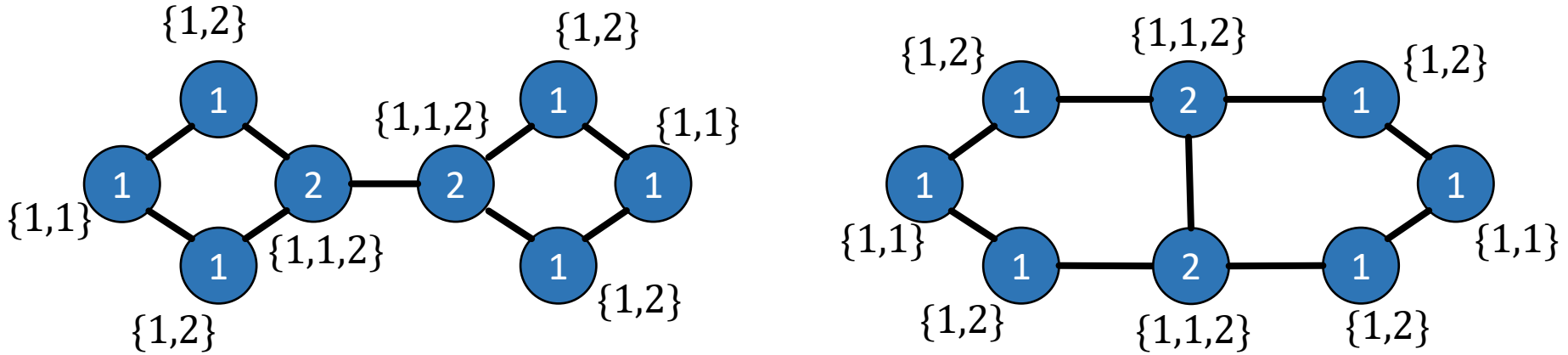
Graph 1: 1: 6, 2: 2 Graph 1: 1: 6, 2: 2

Identical, so we continue…

Example 2: when it doesn't work

These two graphs are clearly not identical, as we can cut the left one into two separate parts with only one cut, while the right one requires at least two cuts to do so! Let's run 1-WL and see if it agrees!

Message + aggregate

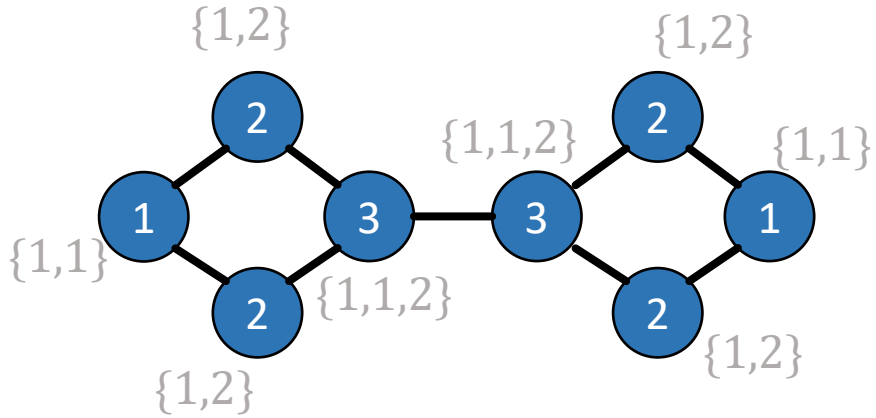


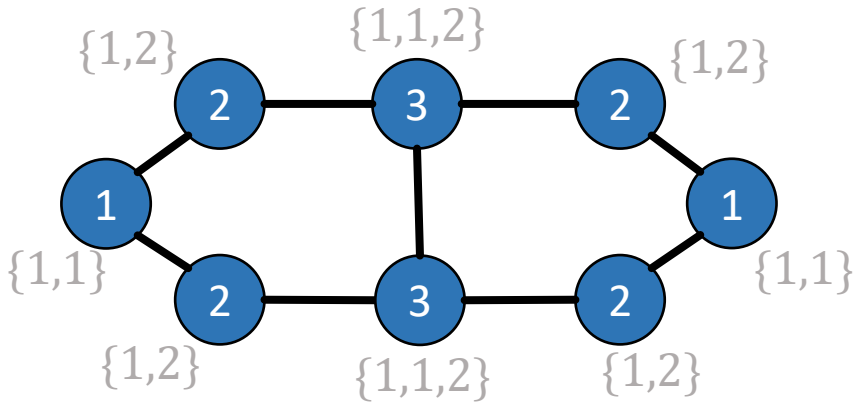
Example 2: when it doesn't work

These two graphs are clearly not identical, as we can cut the left one into two separate parts with only one cut, while the right one requires at least two cuts to do so! Let's run 1-WL and see if it agrees!

Update

$\{1,1\} \rightarrow 1$
$\{1,2\} \rightarrow 2$
$\{1,1,2\} \rightarrow 3$





Check stopping condition:

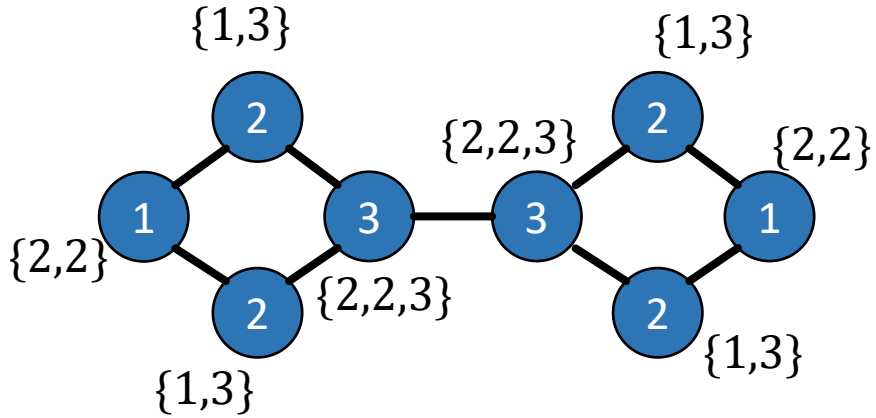
	1: 1		1: 1
Graph 1:	2: 4	Graph 1:	2: 4
	3: 1		3: 1

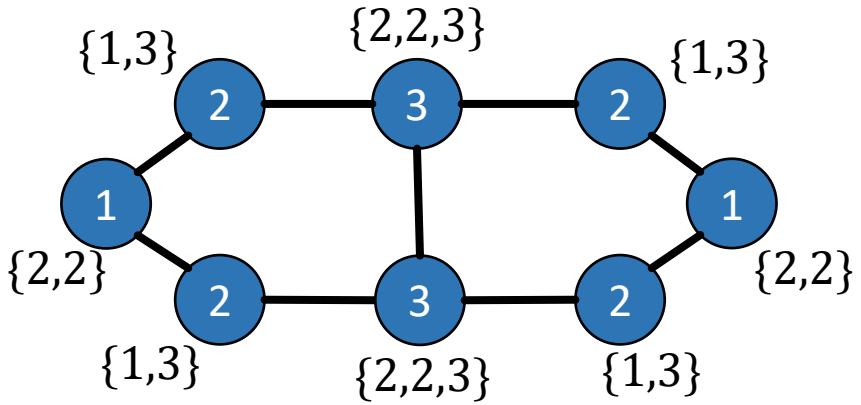
Identical, so we continue…

Example 2: when it doesn't work

These two graphs are clearly not identical, as we can cut the left one into two separate parts with only one cut, while the right one requires at least two cuts to do so! Let's run 1-WL and see if it agrees!

Message + aggregate





Example 2: when it doesn't work

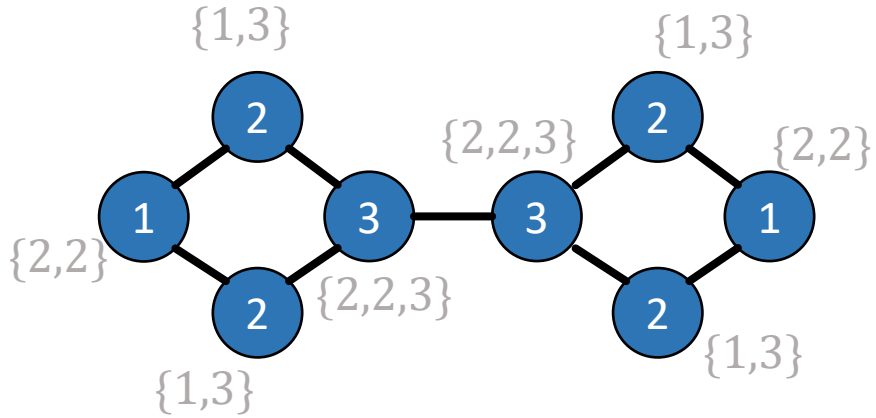
These two graphs are clearly not identical, as we can cut the left one into two separate parts with only one cut, while the right one requires at least two cuts to do so! Let's run 1-WL and see if it agrees!

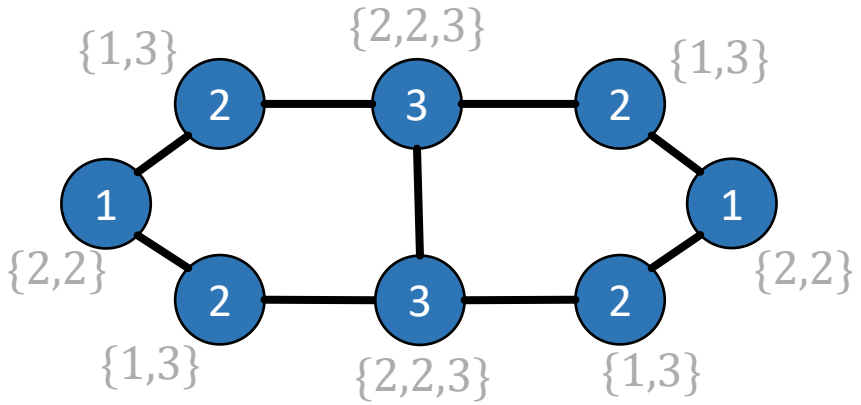
Update

$\{2,2\} \rightarrow 1$

$\{1,3\} \rightarrow 2$

$\{2,3,3\} \rightarrow 3$





Check stopping condition:

	1: 1		1: 1
Graph 1:	2: 4	Graph 1:	2: 4
	3: 1		3: 1

*Identical to the results we got for the last step, so 1-WL **failed** to distinguish the two graphs!*

Prove that GNNs distinguish graphs at most as good as 1-WL

Let ϕ be a graph neural network and G_1, G_2 two non-isomorphic graphs.

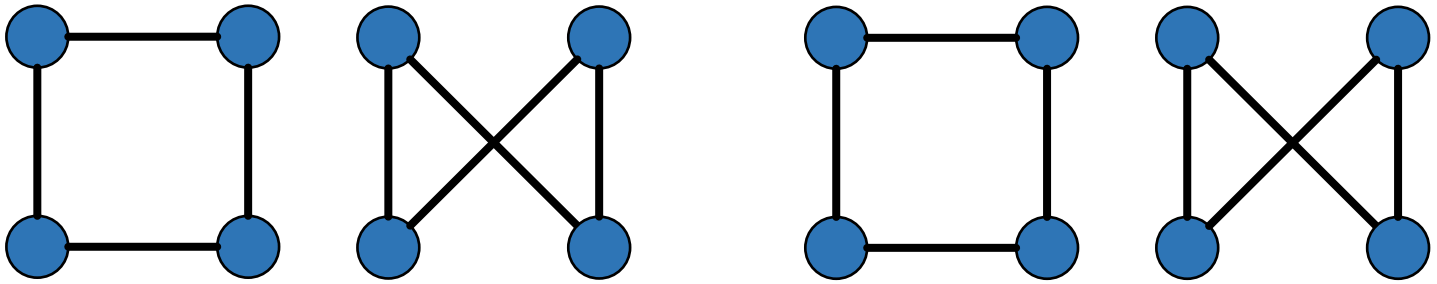
If $\phi(G_1) \neq \phi(G_2)$, then the Weisfeiler-Lehman test also decides that G_1, G_2 are not isomorphic.

For simplicity, we will only sketch the proof here.

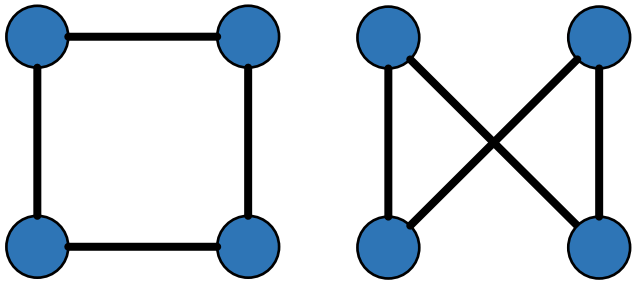
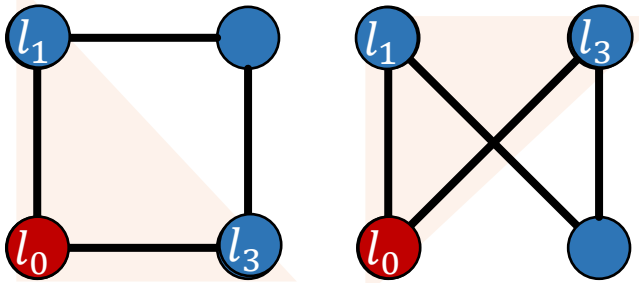
Main idea: assume we have two graphs G_1, G_2 and a GNN ϕ.
We assume that $\phi(G_1) \neq \phi(G_2)$, but that the 1-WL test does not distinguish the graphs.

We will show that this leads to a contradiction!

Prove that GNNs distinguish graphs at most as good as 1-WL

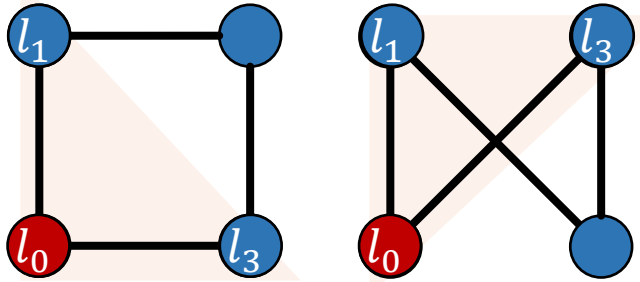
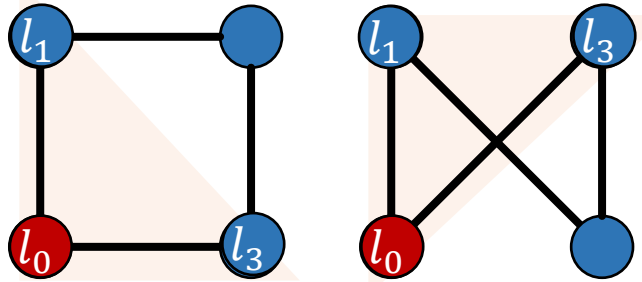
Iteration	GNN view	1-WL view
1		

Prove that GNNs distinguish graphs at most as good as 1-WL

Iteration	GNN view	1-WL view
1		 $\{l_0, \{l_1, l_3\}\}$

1. 1-WL inconclusive
→ if we pick a node in G_1, we can find a similar node in G_2!

Prove that GNNs distinguish graphs at most as good as 1-WL

Iteration	GNN view	1-WL view
1	 $\{l_0, \{l_1, l_3\}\}$	 $\{l_0, \{l_1, l_3\}\}$

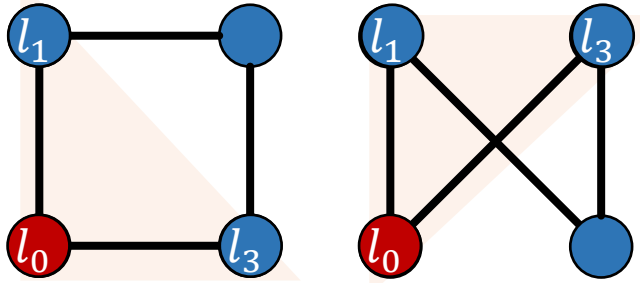
1. 1-WL inconclusive
→ if we pick a node in G_1, we can find a similar node in G_2!

2. GNN & 1-WL start with the same features
→ same is true for GNN features!

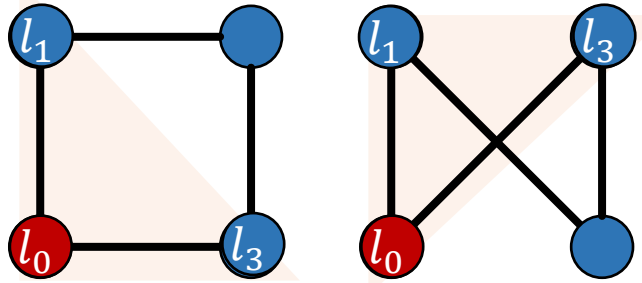
Prove that GNNs distinguish graphs at most as good as 1-WL

Iteration **GNN view** **1-WL view**

1

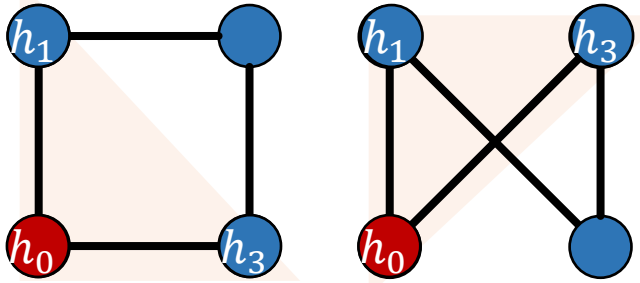


$\{l_0, \{l_1, l_3\}\}$

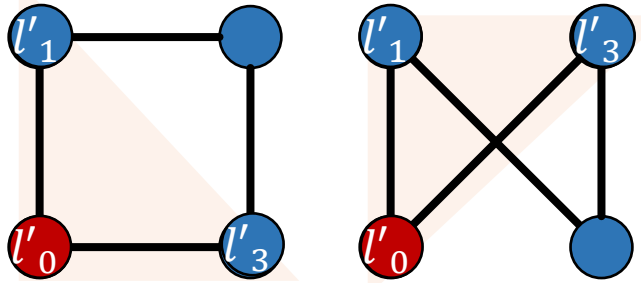


$\{l_0, \{l_1, l_3\}\}$

k



$\{h_0, \{h_1, h_3\}\}$



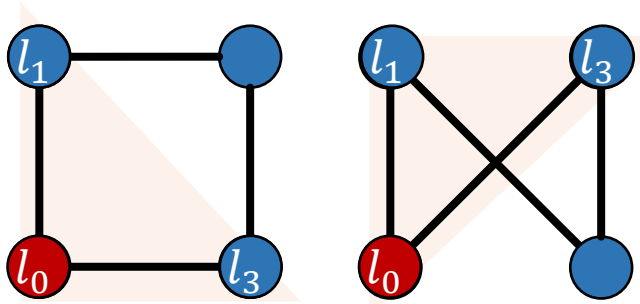
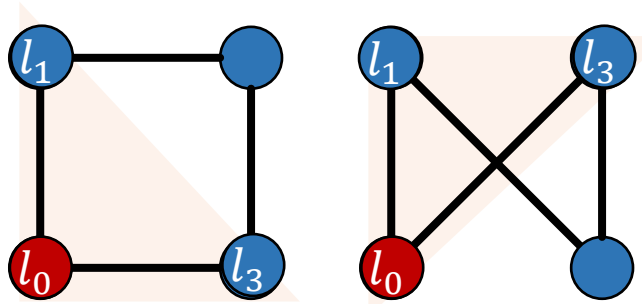
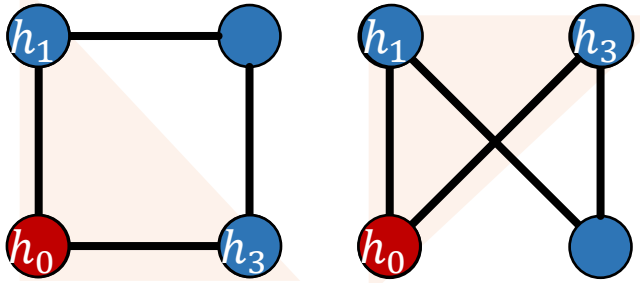
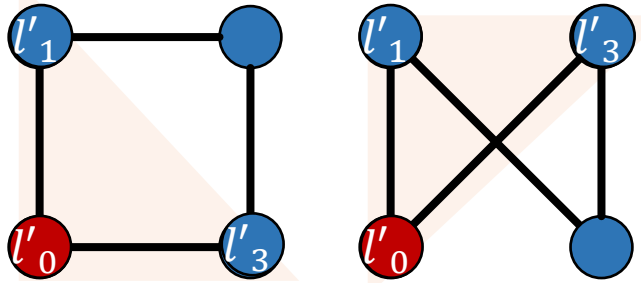
$\{l'_0, \{l'_1, l'_3\}\}$

1. 1-WL inconclusive
→ if we pick a node in G_1, we can find a similar node in G_2!

2. GNN & 1-WL start with the same features
→ same is true for GNN features!

3. Assume the same is true at iteration k!

Prove that GNNs distinguish graphs at most as good as 1-WL

Iteration	GNN view	1-WL view
1	 $\{l_0, \{l_1, l_3\}\}$	 $\{l_0, \{l_1, l_3\}\}$
k	 $\{h_0, \{h_1, h_3\}\}$	 $\{l'_0, \{l'_1, l'_3\}\}$

1. 1-WL inconclusive
→ if we pick a node in G_1, we can find a similar node in G_2!

2. GNN & 1-WL start with the same features
→ same is true for GNN features!

3. Assume the same is true at iteration k!

How is node 0 updated in graph 1 and graph 2?

$$G_1: \boldsymbol{h}_0^{(k+1)} = \mathrm{UPD}\left(\mathrm{AGG}\left(\boldsymbol{h}_0, \mathrm{MSG}(\boldsymbol{h}_1, \boldsymbol{h}_3)\right)\right)$$
$$G_2: \boldsymbol{h}_0^{(k+1)} = \mathrm{UPD}\left(\mathrm{AGG}\left(\boldsymbol{h}_0, \mathrm{MSG}(\boldsymbol{h}_1, \boldsymbol{h}_3)\right)\right)$$

The same! And because of 1., this is **true for all nodes**!

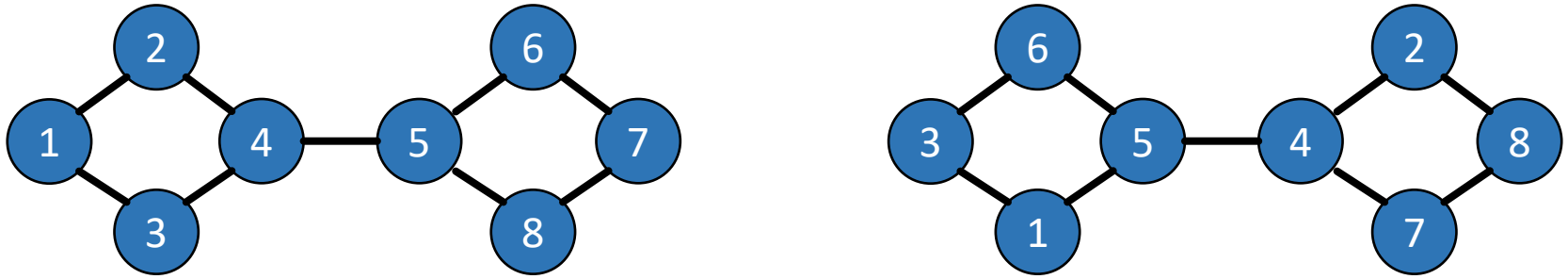
→ As READOUT is permutation invariant, we thus have: $\phi(G_1) = \phi(G_2)$, a **contradiction**!

Graph neural networks as good (or better) than 1-WL?

Yes! The way we defined GNNs, they are as powerful as 1-WL if the MSG+AGG, UPD, and READOUT are injective functions!
Then the GNN always maps different multisets to different labels, just like 1-WL.

But can GNNs go beyond 1-WL? Yes!

By tracking node identities, so we can distinguish "identical" neighbourhoods that are in different locations in the graph. Thus, we attach a unique "identifier" (e.g., one-hot coded vector $\boldsymbol{e}_i$) to each node label ($\boldsymbol{h}_i^{(0)} \oplus \boldsymbol{e}_i$), basically giving them names :)



Problem: Now our GNN is not permutation invariant anymore, meaning that it does not generalize to different graphs! :(

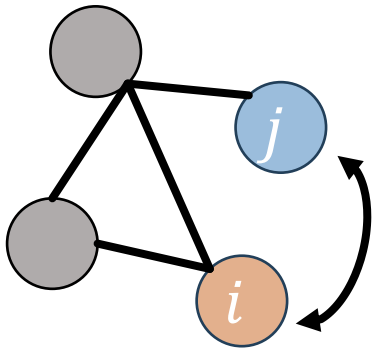
Matrix (factorization) Reloaded

Can we give nodes unique "identifiers" while keeping permutation invariance?

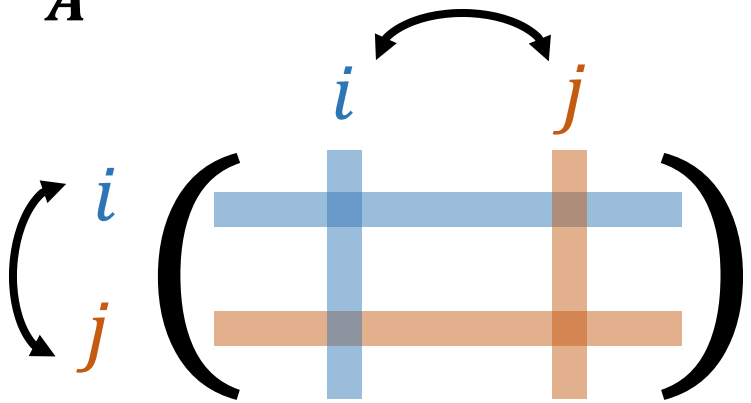
Assume our graph has adjacency matrix $\boldsymbol{A}$
with singular value decomposition $\boldsymbol{A} = \boldsymbol{U}\,\boldsymbol{\Sigma}\boldsymbol{V}^{\mathrm{T}}$

Permuting nodes in the graph = permuting columns and rows of $\boldsymbol{A}$**!**
This leads to a permutation of rows in $\boldsymbol{U}$.

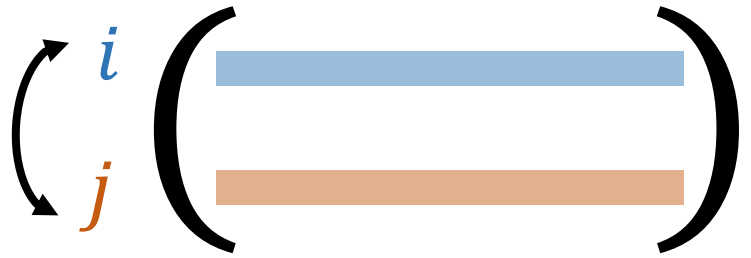
Graph



$\boldsymbol{A}$



$\boldsymbol{U}$



Matrix (factorization) Reloaded

If all singular values are distinct, then the eigenvectors (columns of $\boldsymbol{U}$) are unique up to the sign.

Thus, we do the following: we randomly multiply the columns of $\boldsymbol{U}$ by ± 1, and then use the rows of this matrix as identifiers for our nodes!

$$\phi(G, \boldsymbol{h}_i^{(\mathbf{0})} \oplus (\boldsymbol{U\Gamma})_i^{\mathrm{T}}) \quad \text{with} \quad \Gamma_{ii} \sim \mathrm{Uniform}(\{-1,1\}), \Gamma_{ij} = 0 \text{ if } i \neq j$$

Then $E\left(\phi\left(G, \boldsymbol{h}_i^{(\mathbf{0})} \oplus (\boldsymbol{U\Gamma})_i^{\mathrm{T}}\right)\right) = E\left(\phi\left(G', \boldsymbol{h}_i^{(\mathbf{0})} \oplus (\boldsymbol{U\Gamma})_i^{\mathrm{T}}\right)\right)$ if G, G' are isomorphic.

Note: We can also use other decompositions than SVD, e.g., of the graph Laplacian ***L*** *(**Diffusion Maps**, **DeepWalk**, …)*