Tutorial 3

Extracting Features from Graphs

Outline

- From Databases to Graphs
- Node Features
- Link Features
- Graph Features

Outline

- From Databases to Graphs
- Node Features
- Link Features
- Graph Features

Reminder: Heterogeneous Graph

- A graph which is: Labelled + Featured

$$G = (L_V, L_E, V, E, \lambda, d, \mathbf{X})$$

Vertex labels

Edge labels

Vertices

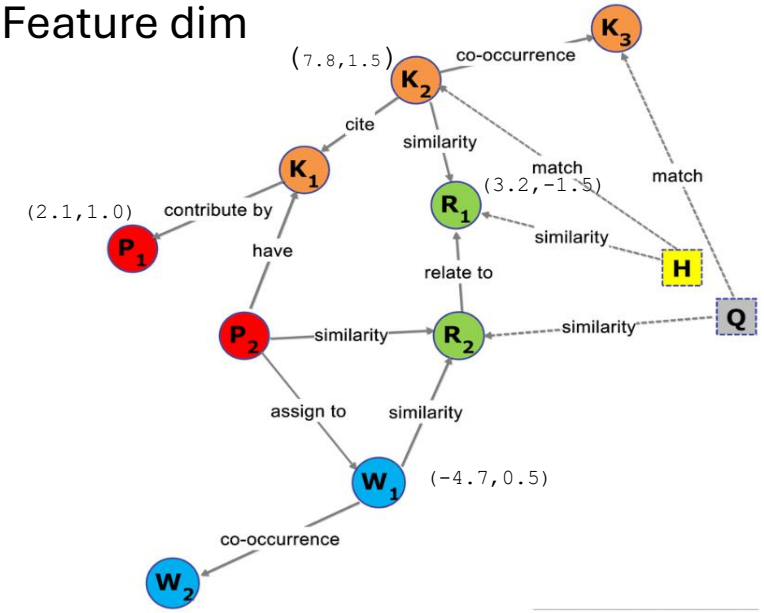
Edges
labeled:
$(u, label, v)$

Label mapping
$\lambda: V \to L_v$

$d: L_V \to \mathbb{N}$ Feature dim

Vertex Features

- Homogenous $\equiv G = (V, E)$



Heterogeneous Graph

Example:

$L_V = \{\text{Paper}, \text{Author}\}$ $L_E = \{\text{Cites}, \text{Owner}\}$

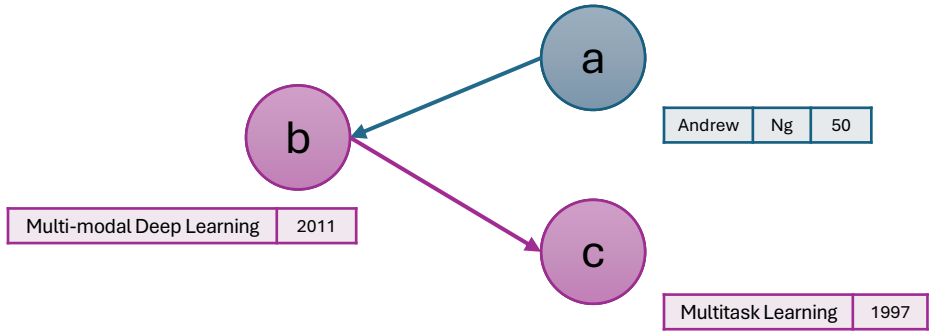
$$G = (L_V, L_E, V, E, \lambda, d, \mathbf{X})$$

$\mathbf{X}(a) = (\text{"Andrew"}, \text{"Ng"}, 55)$
$\mathbf{X}(b) = (\text{"Multi-modal Deep Learning"}, 2011)$
$\mathbf{X}(c) = (\text{"Multitask Learning"}, 1997)$

$V = \{a, b, c\}$
$E = \{(a, \text{Owner}, b), (b, \text{Cites}, c)\}$

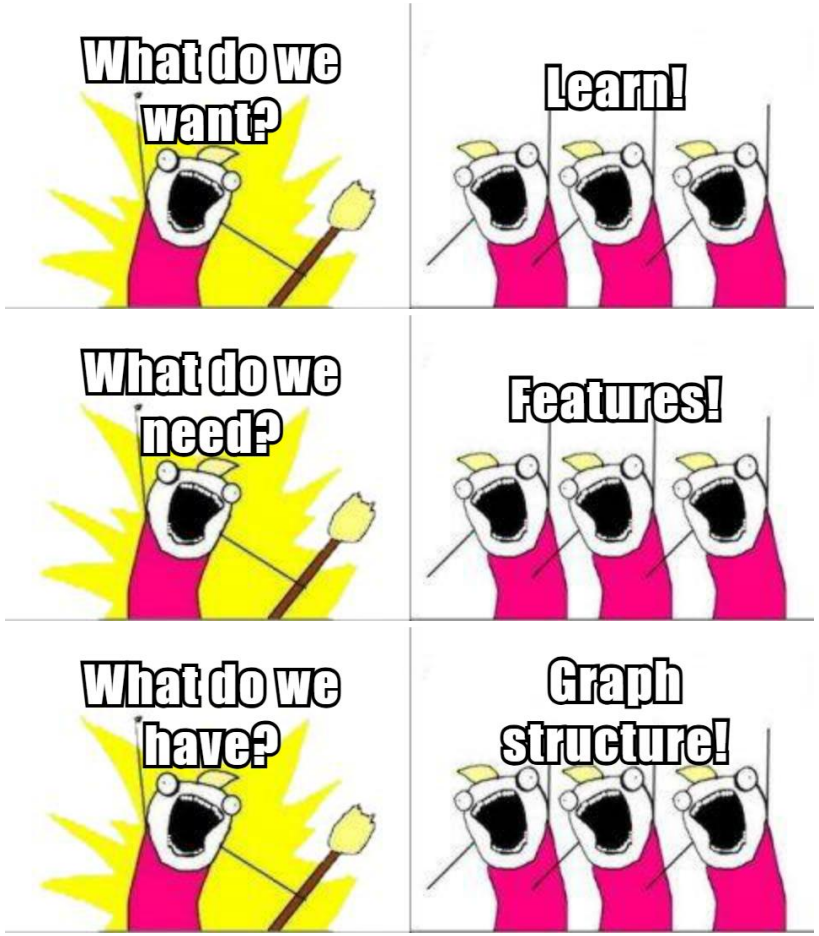
$\lambda(a) = \text{Author}$
$\lambda(b) = \text{Paper}$
$\lambda(c) = \text{Paper}$

$\text{d}(\text{Author}) = 3$
$\text{d}(\text{Paper}) = 2$



From Databases to Graphs

Today's Goal: Enrich features with structural information to improve learning



💡 *Idea*: Databases and graphs are tightly connected

- ✓ Represent the DB as a graph
- ✓ Capture features that encode graph structure
- ✓ Task agnostic
- ✓ Viable information not included in the DB

Features from Graphs – features extracted from graph structure

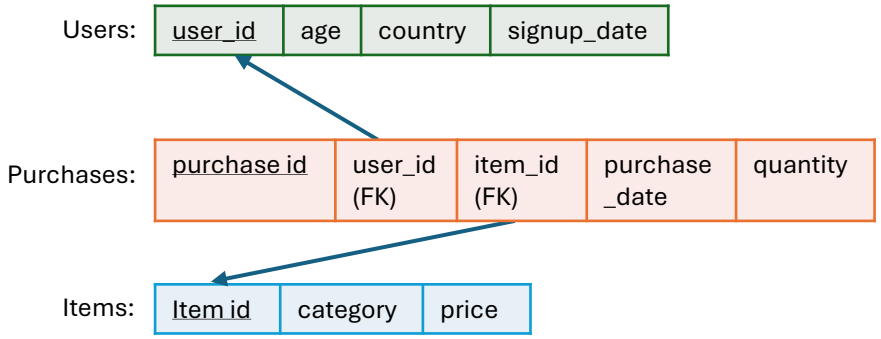
We will learn today main structural features of different kinds 👉🏻

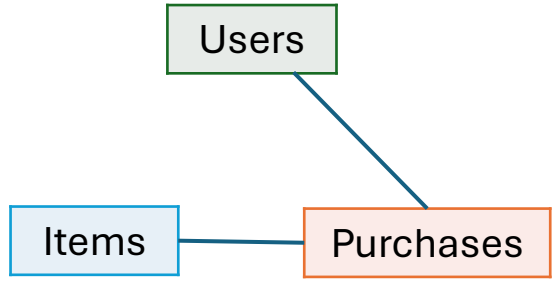
Converting DBs into Graphs

The input: **database** ⇒ *How to convert to a **graph**?*

Reminder - ACORA: schema-level conversion

- Nodes: relation names
- Edges: foreign keys





💡 ACORA is already capturing structural information!
But we can capture more...

Tuple-level Conversion to Graph

- Nodes: tuples (from all tables)
  - Label: relation name
  - Features: the tuple
- Edges: foreign keys

Label

Users

user_id	age	country	Signup date
1	25	Israel	2023-10-01
2	40	USA	2023-09-15

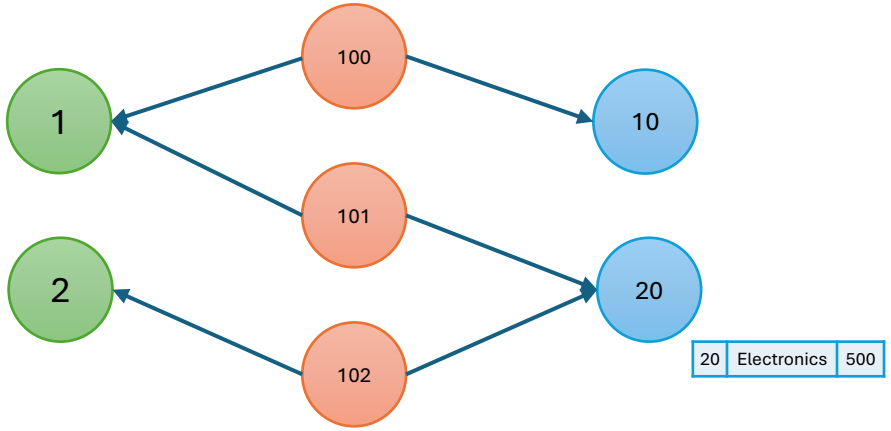
Items

Feature

Item id	category	price
10	Books	50
20	Electronics	500

Purchases

purchase id	User id (FK)	Item id (FK)	Purchase date	quantity
100	1	10	2024-01-01	1
101	1	20	2024-01-10	2
102	2	20	2024-01-05	1



🔨 We will see more conversion methods in the future

Converting DBs into Graphs in Code

1. Load Data

```
import pandas as pd

users     = pd.read_csv("users.csv")
items     = pd.read_csv("items.csv")
purchases = pd.read_csv("purchases.csv")
```

2. Initialize a graph

```
import networkx as nx
G = nx.Graph()
```

3. Add a node for each tuple

```
G.add_nodes_from([(("User", user_id),
                   {"label": "User"}) for user_id in users['user_id'])
G.add_nodes_from([(("Item", item_id),
                   {"label": "Item"}) for item_id in items[‘item_id']])
G.add_nodes_from([(("Purchase", purchase_id),
                   {"label": "Purchase"}) for purchase_id in purchases[‘purchase_id']])
```

4. Add edges using foreign keys (FK)

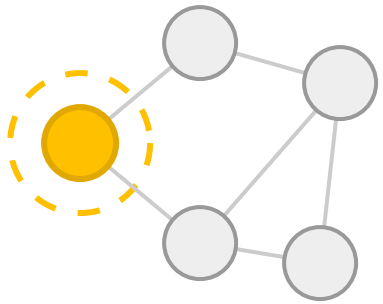
```
user_edges = [(("User", user_id), ("Purchase", purchase_id))
   for user_id, purchase_id in zip(purchases['user_id'], purchases['purchase_id'])]
item_edges = [(("Item", item_id), ("Purchase", purchase_id))
   for item_id, purchase_id in zip(purchases['item_id'], purchases['purchase_id'])]

G.add_edges_from(user_edges + item_edges)
```

Types of Tasks on Graphs

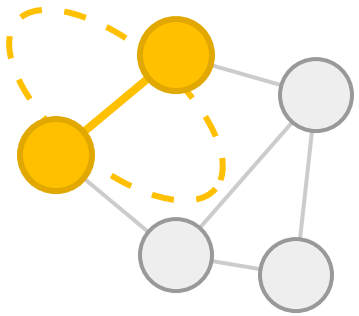
We can categorize learning tasks on graphs into different levels:

Node-level



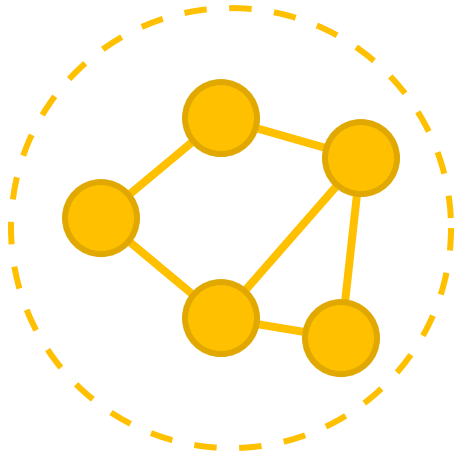
- Document Classification
- Fraud Detection

Link-level



- Friend/Item Recommendation
- Drug-Drug Interaction

Graph-level

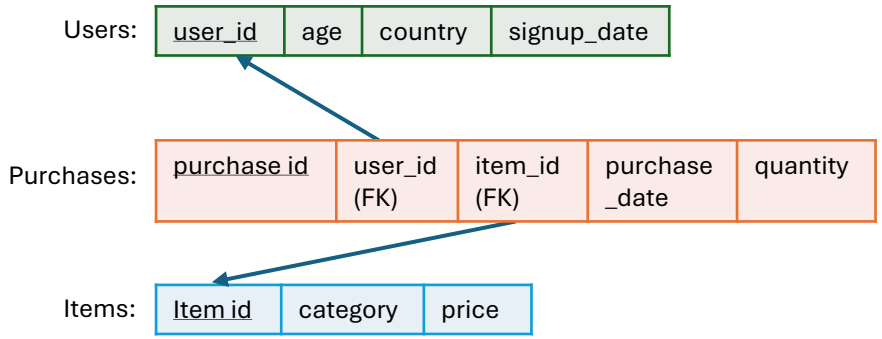


- Molecular Property Prediction
- Program Vulnerability

💡 Similarly, we define levels of features over graph

Task Definition

Schema



Our task for today

predict whether a user will buy an item

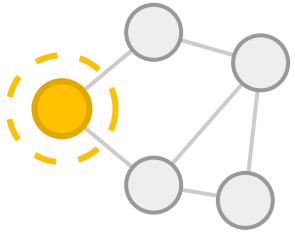
(link-level)

Outline

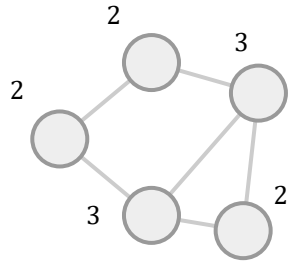
- From Databases to Graphs
- Node Features
- Link Features
- Graph Features

Node Features

Features targeted at nodes



Examples:

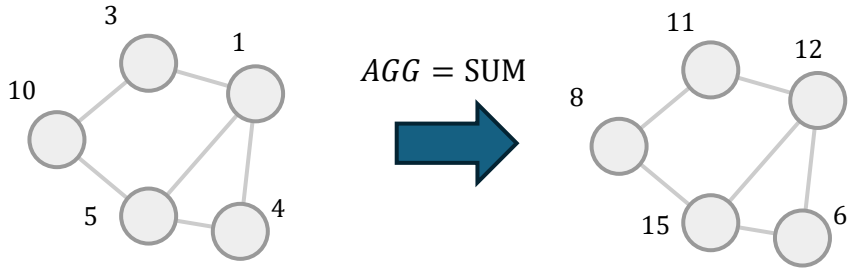


Degree (number of neighbors): $f(v) = |N(v)|$

- $N(v) = \{u \mid (v,u) \in E\}$ – the neighbors of v in the graph

Neighbor aggregation:

- $f(v) \coloneqq AGG\big(x_u : u \in N(v)\big)$



Node Features

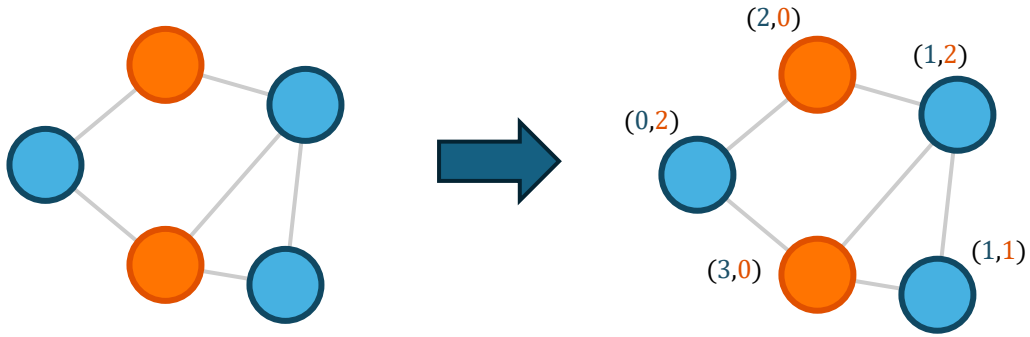
Node degree can be generalized into a “labeled node-degree” in heterogenous graphs:

Given a type $\tau \in L_V$,

- The degree of τ-labelled neighbors: $d_\tau(v) = |\{u \in N(v) \mid \lambda(u) = \tau\}|$

Can gather for all node labels noted $L_V = \{\tau_1, \dots, \tau_k\}$ into a vector:

$$f(v) = \left(d_{\tau_1}(v), d_{\tau_2}(v), \dots, d_{\tau_k}(v)\right)$$



Enables counting separately connected users from connected items

Degree Feature in Code

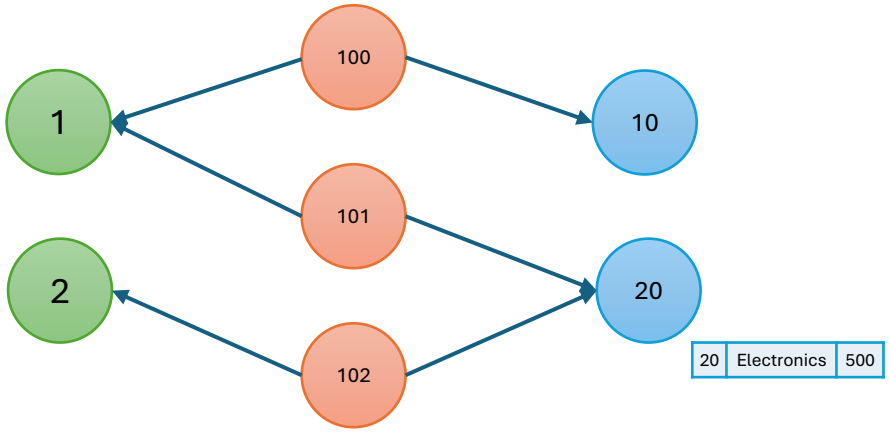
```
target_df = users.merge(items, how='cross’)

degree_dict = dict(G.degree())

target_df['user_degree'] = target_df['user_id'].apply(
    lambda user_id: degree_dict.get(("User", user_id))
)

target_df['item_degree'] = target_df['item_id'].apply(
    lambda item_id: degree_dict.get(("Item", item_id))
)
```

Cartesian Product



Items

item id	category	price
10	Books	50
20	Electronics	500

Users

user_id	age	country	signup date
1	25	Israel	2023-10-01
2	40	USA	2023-09-15

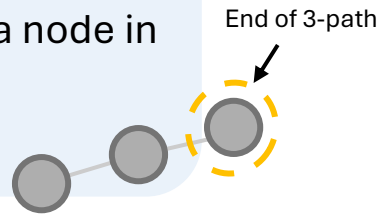
user_id	item id	age	country	signup date	category	price	user_degree	item_degree
1	10	25	Israel	2023-10-01	Books	50	2	1
2	10	40	USA	2023-09-15	Books	50	1	1
1	20	25	Israel	2023-10-01	Electronics	500	2	2
2	20	40	USA	2023-09-15	Electronics	500	1	2

Graph Degree Vector (GDV)

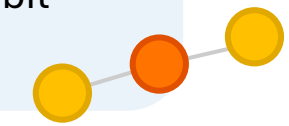
"Degree" is just the count of times a node takes part in an edge

⇒ Can generalize "edge" to any structure - e.g., a triangle, a 4-node path, etc.

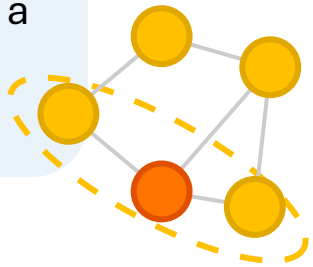
Orbit
The structural role of a node in the structure



Pattern
The structure, with the nodes labeled by their orbit



Graphlet
The pattern as it appears in a bigger graph

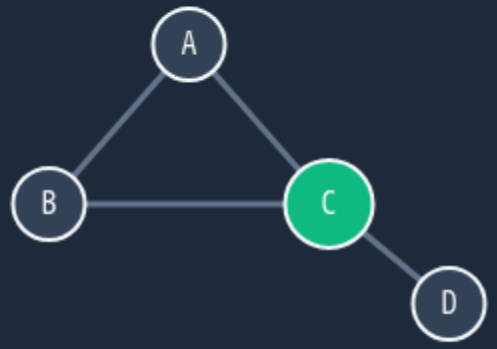


GDV
Counting a node's participation across graphlets

$(2, \quad 1, \quad 1)$

GDV Example

Target: Node C



C acts as a central hub in this "Kite" graph. Let's build its signature using up to 3-node orbits.

Orbit Counting

Orbit 0 (Degree) 3

Edges (C,A), (C,B), (C,D).

Orbit 1 (End of 3-path) 2

C-A-B is a triangle, not an induced path.

Orbit 2 (Mid of 3-path) 2

Induced paths A-C-D and B-C-D. C is middle.

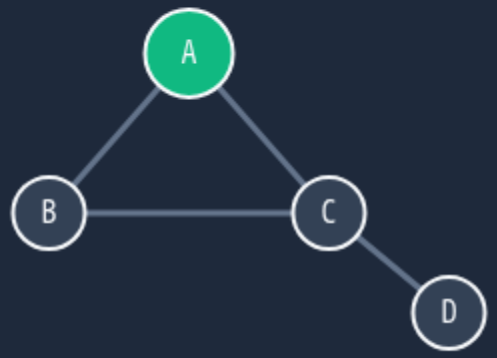
Orbit 3 (Triangle) 1

Triangle formed by A, B, and C.

```
GDV(C) = [3, 2, 2, 1]
```

GDV Example

Target: Node A



Same graph, but targeting Node A. Notice how its structural role is different from Node C.

Orbit Counting

Orbit 0 (Degree) 2

Edges (A,B) and (A,C).

Orbit 1 (End of 3-path) 3

Path A-C-D is induced. Node A is at the end.

Orbit 2 (Mid of 3-path) 1

Path B-A-C is NOT induced (edge B-C exists).

Orbit 3 (Triangle) 1

Triangle formed by A, B, and C.

```
GDV(A) = [2, 3, 1, 1]
```

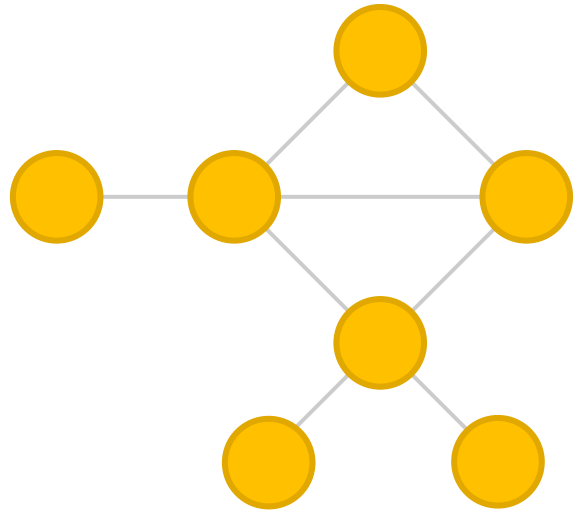
Weisfeiler Lehman

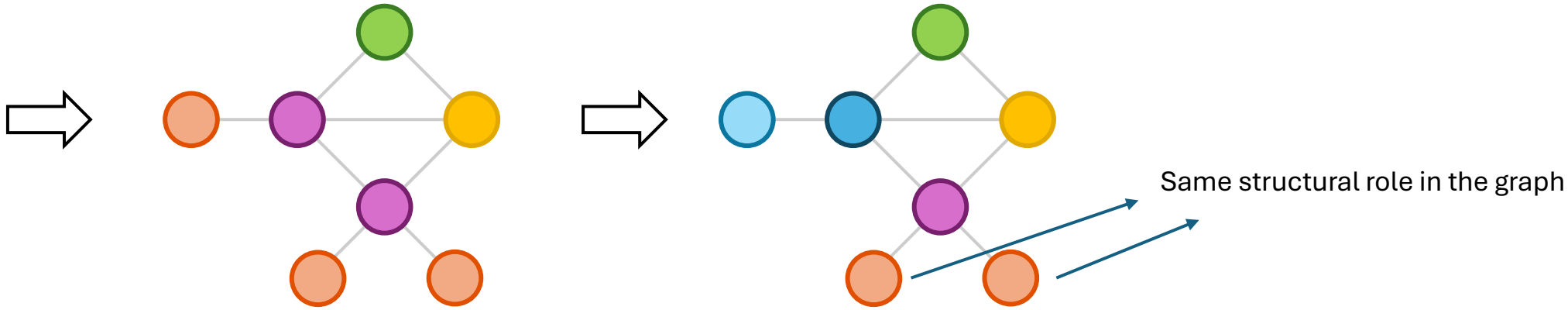
💡 Originally, WL is used for testing isomorphism. But it can also be used as a node feature!

Given $k \in \mathbb{N}$ (steps):

1. Initialize $\kappa_0(v) = 0$ for all $v \in V$
2. For all $i = 1, \ldots, k$ and for all nodes $v \in V$:

$$\kappa_i(v) = HASH(\kappa_{i-1}(v), \{\{\kappa_{i-1}(u) \mid u \in N(v)\}\})$$

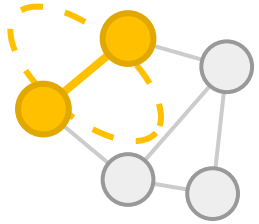




Outline

- From Databases to Graphs
- Node Features
- Link Features
- Graph Features

Link Features



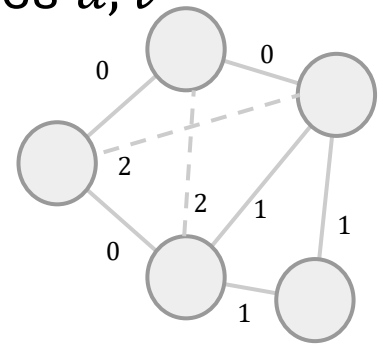
Features for couples of nodes

- Useful for edge prediction tasks
- Example: the existence of an edge, e.g., friends in social networks

Examples:

Common neighbors (CN): number of shared neighbors between two nodes u, v

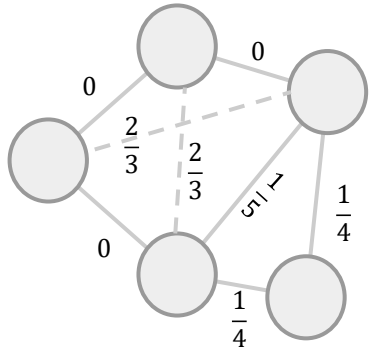
$$CN(u,v) = |N(u) \cap N(v)|$$



- Can be extended to number of shared neighbors that also fulfill some condition
- Can be generalized into labelled-CN similarly to labelled node degree

Jaccard similarity – normalized overlap between neighbors:

$$J(u,v) = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}$$



- Jaccard adjusts to the node size – eliminate centrality

Adamic Adar

Link feature quantifying that nodes are more likely to connect if they share common neighbors

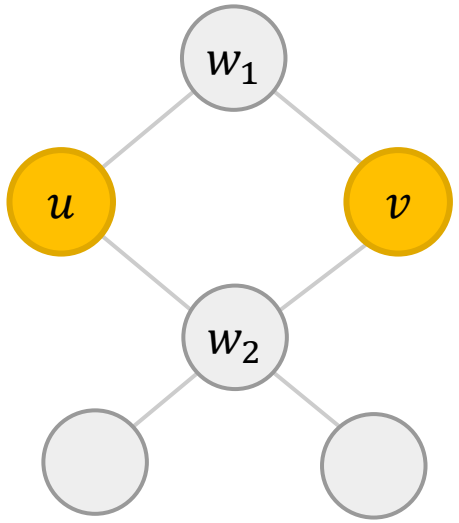
- but weights the rarity of those neighbors:

$$AA(u,v) = \sum_{w \in N(u) \cap N(v)} \frac{1}{\log|N(w)|}$$

- A shared "niche" friend (low degree) provides a strong, specific signal of a potential connection.
- A shared "hub" provides very little connection evidence.

Adamic Adar Example

$$AA(u,v) = \sum_{w \in N(u) \cap N(v)} \frac{1}{\log|N(w)|}$$



Contribution Breakdown

Shared Neighbors { W1, W2 }

These form the length-2 paths between U and V.

W1 Contribution 1.44

1 / log(2) ≈ 1.44. High reward for low degree.

W2 Contribution 0.72

1 / log(4) ≈ 0.72. Lower reward for higher degree.

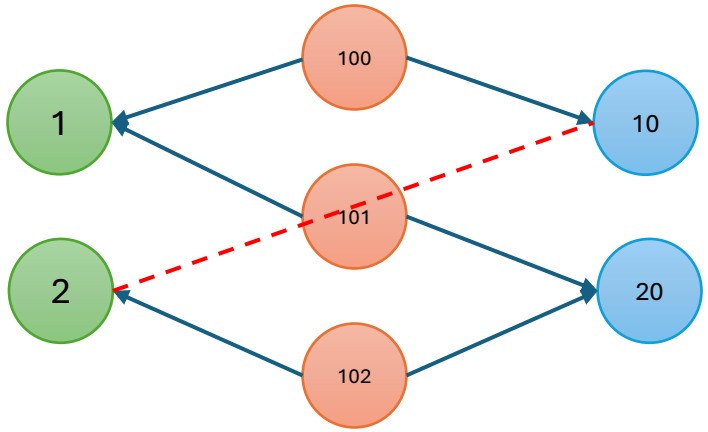
AA(U,V) = 1.44 + 0.72 = 2.16

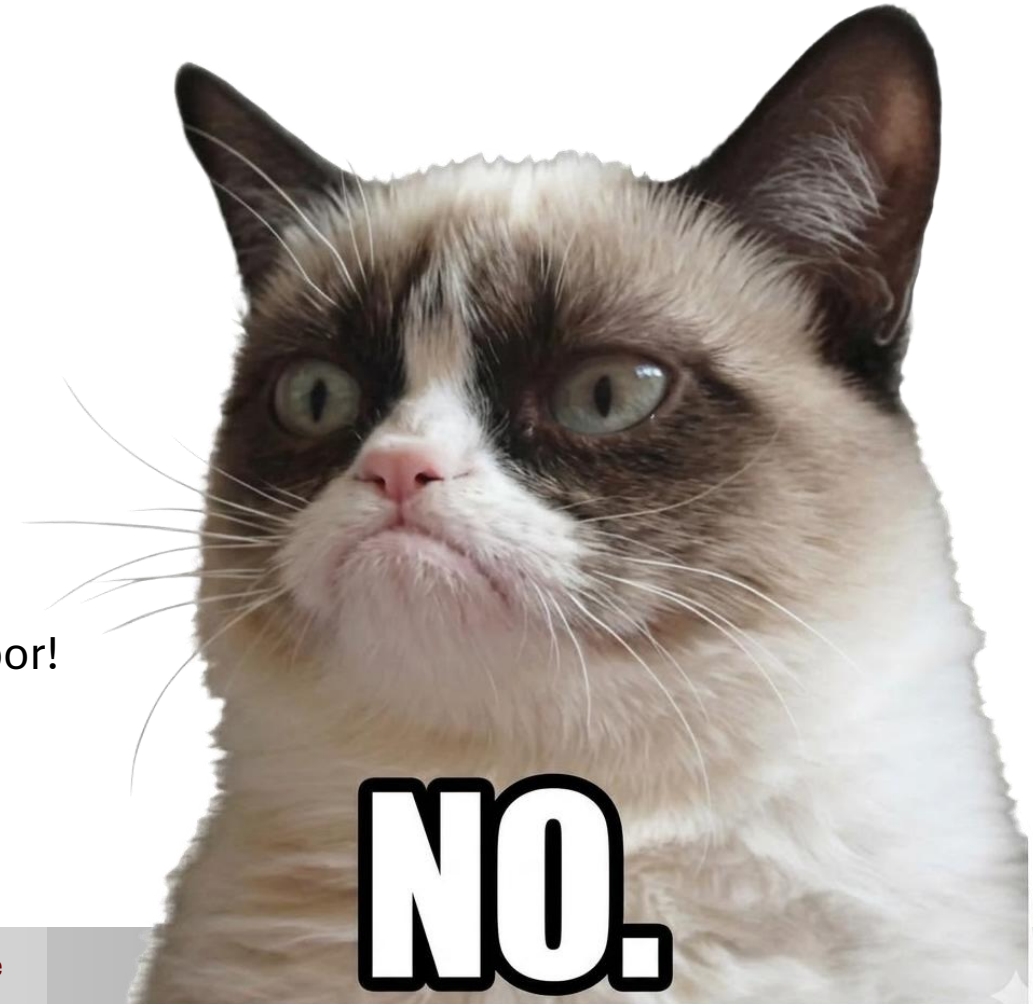
$AA(w_1, w_2) = 2.88 > AA(u,v) = 2.16 \quad \Rightarrow$

⭐ The Celebrity Effect: Bigger hubs ⇒ Less indicative

Next Task's Feature

Can we use Adamic Adar for our task? *(will a user buy an item)*





⇒ Non-existing user-item couples do not share a neighbor!
⇒ Also CN and Jaccard can't help
⇒ Then what can we use?

Katz Index

Measures how strongly two nodes are connected by considering paths of all lengths

$$\mathrm{Katz}(u,v) \coloneqq \sum_{\ell=1}^{\infty} \beta^{\ell} |\mathrm{paths}_{\ell}(u,v)| = \left((I - \beta A)^{-1} - I\right)$$

- $0 < \beta < 1$
  - Dictates how far the node has a sight of
    - high $\beta \Rightarrow$ longer paths impact
    - low $\beta \Rightarrow$ shorter paths impact

- Global: Unlike Adamic Adar, Katz measures **all possible paths** exponentially penalized by length
  - Hence, far nodes can still be measured

Generating Katz Index Feature

Graph→Adjacency Matrix

```
A = nx.to_numpy_array(G, nodelist=node_list)
beta = 0.1
I = np.eye(G.number_of_nodes())
katz_matrix = inv(I - beta * A) – I

user2idx = {node[1]: idx for idx, node in enumerate(node_list) if node[0] == "User"}
item2idx = {node[1]: idx for idx, node in enumerate(node_list) if node[0] == "Item"}

target_df['katz_score'] = target_df.apply(
   lambda row: katz_matrix[user2idx[row['user_id']], item2idx[row['item_id']]],
   axis=1
)
```

Katz Index Calculation

Map user/item id↦ Adj index

Add to target table

Optimization opportunity:

```
import networkit as nk

tuple_to_int_map =
      {node: i for i, node in
       enumerate(G.nodes())}
G_int = nx.relabel_nodes(G,
          tuple_to_int_map)

nk_graph =
      nk.nxadapter.nx2nk(G_int)

ki =
nk.linkprediction.KatzIndex(nk_graph,
                            5, 0.1)
```

💡 *Tip*: Always look for existing implementations

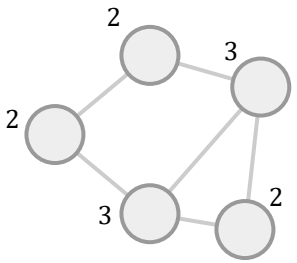
Outline

- From Databases to Graphs
- Node Features
- Link Features
- Graph Features

Graph Features

Features of a whole graph

Examples:



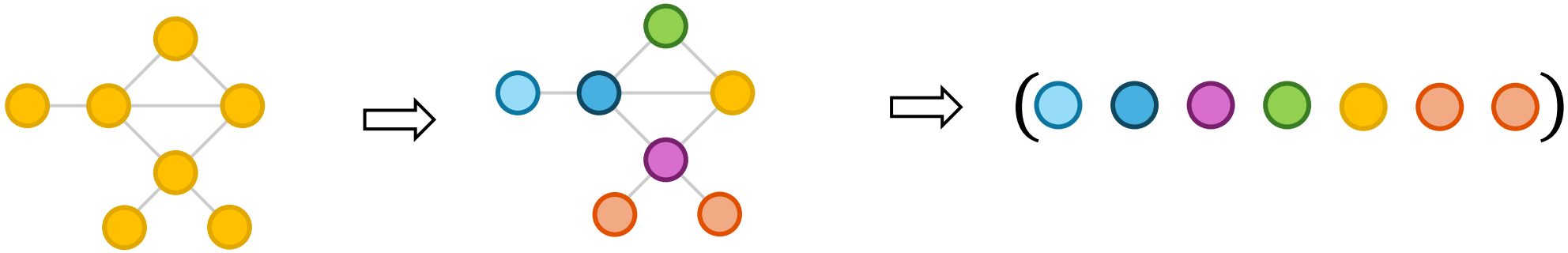
(2,2,3,3,2)

Graph Degree Distribution (GDD)

- A vector representing the degrees of each node in the graph

Weisfeiler-Lehman Color Distribution (WL Kernel)

- A distribution of the colors of each node given by WL



```
wl_dist = nx.weisfeiler_lehman_subgraph_hashes(G, node_attr='label', iterations=2)
```

Let's Train!

```python
from sklearn.neural_network import MLPClassifier
...
train_mask = target_df['split'] == 'train'
test_mask = target_df['split'] == 'test'

X_train = target_df[train_mask][features]
y_train = target_df[train_mask]['label']
X_test = target_df[test_mask][features]
y_test = target_df[test_mask]['label']

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

clf = MLPClassifier(hidden_layer_sizes=(64, 32), activation='relu', solver='adam', max_iter=500, random_state=42)
clf.fit(X_train_scaled, y_train)

predictions_proba = clf.predict_proba(X_test_scaled)[:, 1]
predictions_binary = clf.predict(X_test_scaled)

print("\n========================================")
print(" ✅ Full Pipeline Complete! Model Results:")
print(f"AUC-ROC Score : {roc_auc_score(y_test, predictions_proba):.3f}")
print(f"Accuracy      : {accuracy_score(y_test, predictions_binary):.3f}")
print("========================================")
```

Traditional ML library

Precalculated train/test split

Actual split of target DF

Normalization

Define model & train

Predict

Print results

Final DF schema

user_id	item id	age	country	signup date	category	price	user_degree	item_degree	katz_index	label

```
========================================
✅ Full Pipeline Complete! Model Results:
AUC-ROC Score : 0.612
Accuracy      : 0.579
========================================
```

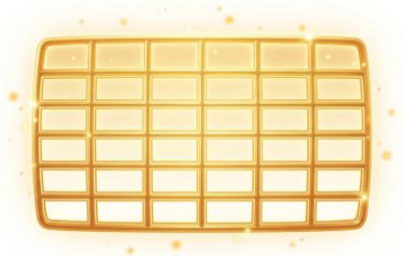
Big Picture

Input Database

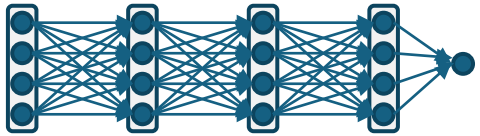
Graph Representation

Feature Engineering

Features From Graphs



Big Table



Learning Algorithm

Prediction