The Henry and Marilyn Taub
Faculty of Computer Science

Module 1: Fundamental Concepts in Structured ML

StructML Course

Benny Kimelfeld
Technion – Israel Institute of Technology

Spring 2026

Module Goals

- Understand the core data models of the course: *graphs*, *tables*, and *relational databases*
- Explore how relational databases can be analyzed using graph concepts and tools
- Develop familiarity with *key graph concepts* relevant to the course
- Review *fundamental machine learning* (ML) principles
- Introduce basic concepts in *ML for structured data*

Outline

1. Data Models
   - Graphs
   - Tables
   - Relational Databases
2. Databases as Hetero-Graphs
3. Required Graph Concepts
   - Neighborhood
   - Matrix Representations of Graphs
   - Weisfeiler-Leman Isomorphism Test
4. ML on Structured Data
   - General ML
   - Predictive Tasks
   - Training

Outline

1. Data Models
   - Graphs
   - Tables
   - Relational Databases
2. Databases as Hetero-Graphs
3. Required Graph Concepts
   - Neighborhood
   - Matrix Representations of Graphs
   - Weisfeiler-Leman Isomorphism Test
4. ML on Structured Data
   - General ML
   - Predictive Tasks
   - Training

Database: Collection of Tables

Student

sid	name	started	address	gender	track
89	Anna	2022	Haifa	F	CS3
60	Bill	2024	Haifa	M	CS4
52	Carry	2023	Tel-Aviv	F	SE

Take

sid	cid	year	semester
89	12	2025	1
89	25	2024	2
60	25	2024	2

Course

cid	name	credits	faculty
12	DB	3	CS
25	ML	3	CS
34	OS	4	ECS

Database: Collection of Tables **with Connected Tuples**

Student

sid	name	started	address	gender	track
89	Anna	2022	Haifa	F	CS3
60	Bill	2024	Haifa	M	CS4
52	Carry	2023	Tel-Aviv	F	SE

Take

sid	cid	year	semester
89	12	2025	1
89	25	2024	2
60	25	2024	2

Course

cid	name	credits	faculty
12	DB	3	CS
25	ML	3	CS
34	OS	4	ECS

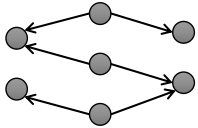
Our Data Models

- We focus on **machine learning over relational databases**
- A relational database consists of tables with *connected records* (rows)
- Hence, a natural representation of a database uses graphs
  - Where records serve as nodes
- Therefore, we will learn machine learning techniques over 3 data models: graphs, tables, and relational databases

Outline

1. Data Models
   - Graphs
   - Tables
   - Relational Databases
2. Databases as Hetero-Graphs
3. Required Graph Concepts
   - Neighborhood
   - Matrix Representations of Graphs
   - Weisfeiler-Leman Isomorphism Test
4. ML on Structured Data
   - General ML
   - Predictive Tasks
   - Training

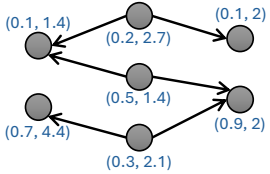
Basic Graphs



- *Directed graph*: (V, E) where V is a set of *nodes* and $E \subseteq V \times V$ is a set of *edges*
- *Undirected graph*: $E \subseteq \{\{u, v\} \subseteq V \mid u \neq v\}$

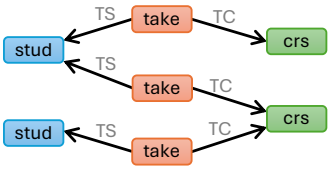
Unless stated explicitly, in this lecture we assume graphs are **directed**

Featured Graphs



- In a *featured graph*, every node is associated with a numerical *feature vector*
- Formally, a d-dimensional featured graph is a triple $(V, E, \mathbf{X})$ where
  - (V, E) is a graph
  - $\mathbf{X} : V \to \mathbb{R}^d$ associates a *feature vector* with every node
    - $\mathbf{X}$ is typically represented as a matrix in $\mathbb{R}^{|V| \times d}$, assuming a predefined ordering $v_1, v_2, \ldots$ over V
    - We often denote the vector $\mathbf{X}(v)$ by $\mathbf{x}_v$
- In rare cases, we may also discuss *edge featured* graphs, where each edge has a feature vector
  - (of some edge-feature dimension)

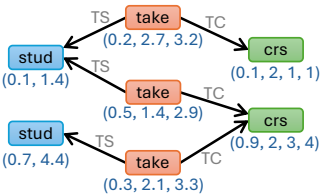
Labeled Graphs



- We often want the vertices and edges to be *labeled* (or *typed*)
- A *labeled graph* is a tuple $(L_V, L_E, V, E, \lambda)$ where:
  - L_V and L_E are sets of *node labels* and *edge labels*, respectively
  - V is the node set and $\lambda : V \to L_V$ assign a label (type) to each node
  - $E \subseteq V \times L_E \times V$ is a set of labeled edges (v_1, τ, v_2)
    - By a slight abuse of notation, we denote by $\lambda(e)$ the label τ of the edge $e = (v_1, \tau, v_2)$
- Important: we assume that edge labels are consistent with node labels: for all edges (v_1, τ, v_2) and (v_1', τ', v_2'):

$$\text{if } \tau = \tau', \text{ then } \lambda(v_1) = \lambda(v_1') \text{ and } \lambda(v_2) = \lambda(v_2')$$

Heterogeneous Graphs



- A *heterogeneous graph* (or *hetero-graph* for short) is a labeled and featured graph
- Importantly, nodes of different types may have features of different dimensions
- Formally, it is a tuple $(L_V, L_E, V, E, \lambda, d, \mathbf{X})$ where:
  - $(L_V, L_E, V, E, \lambda)$ is a labeled graph
  - $d : L_V \to \mathbb{N}$ associates a *feature dimension* with each node label
  - $\mathbf{X}$ is a function mapping each node v to a feature $\mathbf{x}_v \in \mathbb{R}^{d(\lambda(v))}$

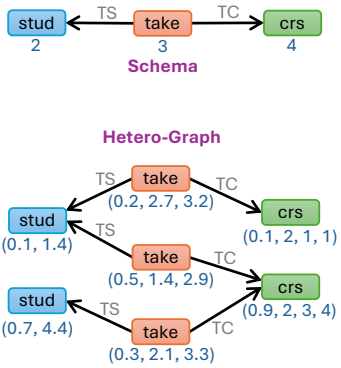
Type Consistency of Hetero-Graphs

- Two graphs $G = (L_V, L_E, V, E, \lambda, d, \mathbf{X})$ and $G = (L'_V, L'_E, V', E', \lambda', d', \mathbf{X}')$ are *type consistent* if:
  - They have the same sets of labels:

$$L_V = L'_V \quad \text{and} \quad L_E = L'_E$$

  - Every edge label is associated with the same in/out node labels in both graphs
  - $d = d'$, that is, every label $\tau \in L_V$ has the same feature dimension in both graphs
- In other words, two hetero-graphs are consistent if they have the same *schema*:

A Schema for Heterogeneous Graphs



- A *hetero-graph schema* is a tuple (L_v, L_e, M, δ) where:
  - L_v is a set of node labels
  - L_e is a set of edge labels
  - $M : L_e \to L_v \times L_v$ maps every edge label into a pair of node labels
  - $\delta : L_v \to \mathbb{N}$ is a dimension function
- A hetero-graph $G = (L_V, L_E, V, E, \lambda, d, \mathbf{X})$ over the schema (L_v, L_e, δ, M) satisfies:
  - $L_V = L_v$ and $L_E = L_e$
  - Every edge (u, τ, v) is such that $M(\tau) = (\lambda(u), \lambda(v))$
  - $d = \delta$

Outline

1. Data Models
   - Graphs
   - Tables
   - Relational Databases
2. Databases as Hetero-Graphs
3. Required Graph Concepts
   - Neighborhood
   - Matrix Representations of Graphs
   - Weisfeiler-Leman Isomorphism Test
4. ML on Structured Data
   - General ML
   - Predictive Tasks
   - Training

Tables

- A *tabular signature* $\mathcal{T}$ is pair $(\mathbf{A}, \mathbf{dom})$ where:
  - $\mathbf{A} = A_1, \ldots, A_n$ is a sequence of *attribute names* (or just *attributes* for short)
  - $\mathbf{dom} = dom_1, \ldots, dom_n$ is a sequence of *domains* corresponding to $A_1, \ldots, A_n$, respectively
    - Each dom_i is a finite or infinite set
    - We may write $dom(A_i)$ instead of dom_i
- A *table* (or *relation*) r over $\mathcal{T}$ is a finite subset of $dom_1 \times \cdots \times dom_n$
  - Note: we use set semantics (duplicate tuples disallowed)
- Each element $\mathbf{t} \in r$ is called a *tuple* (and sometimes a *row* or a *record*)

Domains:

[10-99]	text	[0-10]	CS, ECS, Math,...
cid	name	credits	faculty
12	DB	3	CS
25	ML	3	CS
34	OS	4	ECS

Projection

- If $\mathbf{t} = (a_1, \ldots, a_n) \in r$ is a tuple, then we may refer to each a_i as $\mathbf{t}.A_i$ or $\mathbf{t}[A_i]$
- If $\mathbf{B} = B_1, \ldots, B_\ell$ is a sequence of attributes from the signature, then we define

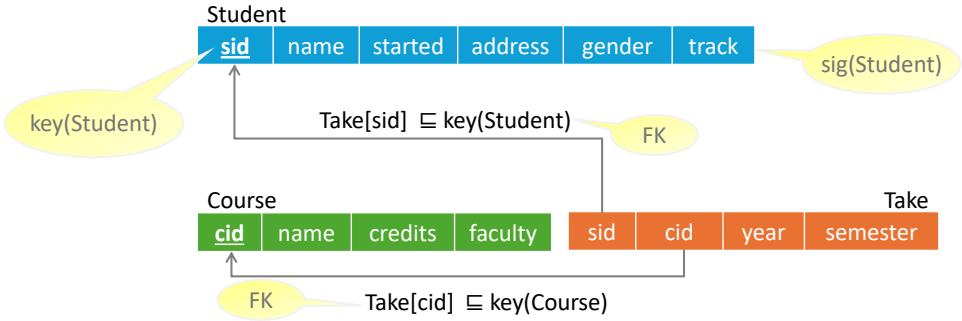
$$\mathbf{t}[\mathbf{B}] := (\mathbf{t}.B_1, \ldots, \mathbf{t}.B_\ell)$$

That is, $\mathbf{t}[\mathbf{B}]$ is the *projection* of $\mathbf{t}$ to $\mathbf{B}$

Outline

1. Data Models
   - Graphs
   - Tables
   - Relational Databases
2. Databases as Hetero-Graphs
3. Required Graph Concepts
   - Neighborhood
   - Matrix Representations of Graphs
   - Weisfeiler-Leman Isomorphism Test
4. ML on Structured Data
   - General ML
   - Predictive Tasks
   - Training

Example of a Relational Schema



Definition of a Relational Schema

A *relational schema* $\mathcal{S}$ is a tuple $(\mathcal{R}, \text{sig}, \text{key}, \mathbf{FK})$ where:

- $\mathcal{R}$ is a finite set of *relation names*
- sig maps each relation name R to a tabular signature $\text{sig}(R)$
- key maps every relation name R into a *primary key*, which is a sequence of attributes from $\text{sig}(R)$
- **FK** is a set of *foreign-key constraints* (*FK*s for short)

Relational Schema Notation

- An FK over the schema $\mathcal{S} = (\mathcal{R}, \text{sig}, \text{key}, \mathbf{FK})$ has the form $R[\mathbf{C}] \sqsubseteq \text{key}(S)$:
  - R and S are relation names in $\mathcal{R}$
  - $\mathbf{C}$ is a sequence of attributes from $\text{sig}(R)$
  - $\mathbf{C}$ and $\text{key}(S)$ have the same length
- We sometimes say that a relation R "has no key" to mean that the $\text{key}(R)$ consists of all attributes
  - Which is very different from "$\text{key}(R) = \emptyset$" (meaning what?)
- We denote by $dom(R)$ the domain $dom_1 \times \cdots \times dom_n$ of a relation name $R \in \mathcal{R}$ with $\text{sig}(R) = (\mathbf{A}, \mathbf{dom})$ where $\mathbf{dom} = dom_1, \ldots, dom_n$

Databases

Student

sid	name	started	address	gender	track
89	Anna	2022	Haifa	F	CS3
60	Bill	2024	Haifa	M	CS4
52	Carry	2023	Tel-Aviv	F	SE

Take

sid	cid	year	semester
89	12	2025	1
89	25	2024	2
60	25	2024	2

Course

cid	name	credits	faculty
12	DB	3	CS
25	ML	3	CS
34	OS	4	ECS

- Let $\mathcal{S} = (\mathcal{R}, \text{sig}, \text{key}, \mathbf{FK})$ be a database schema
- A *database* D over $\mathcal{S}$ maps every $R \in \mathcal{R}$ to a table $D(R)$ over $\text{sig}(R)$ so that:
  - Primary keys are respected: for every relation name R, no two distinct tuples $\mathbf{t}_1, \mathbf{t}_2 \in D(R)$ satisfy $\mathbf{t}_1[\text{key}(R)] = \mathbf{t}_2[\text{key}(R)]$
  - FKs are respected: for every $R[\mathbf{C}] \sqsubseteq \text{key}(S)$ in $\mathbf{FK}$ and tuple $\mathbf{t} \in D(R)$ there is a tuple $\mathbf{s} \in D(S)$ such that $\mathbf{t}[\mathbf{C}] = \mathbf{s}[\text{key}(S)]$

Database Facts

- If D is a database over $\mathcal{S}$ and $\mathbf{t} \in D(R)$ for some $R \in \mathcal{R}$, then we call the expression $R(\mathbf{t})$ a *fact* (of D)
  - For example, $\text{Take}(89, 12, 2025, 1)$ is a fact
- We may identify a database as *the set of its facts*
  - For example, $D_1 \subseteq D_2$ means that every relation of D_1 is contained in the corresponding relation of D_2

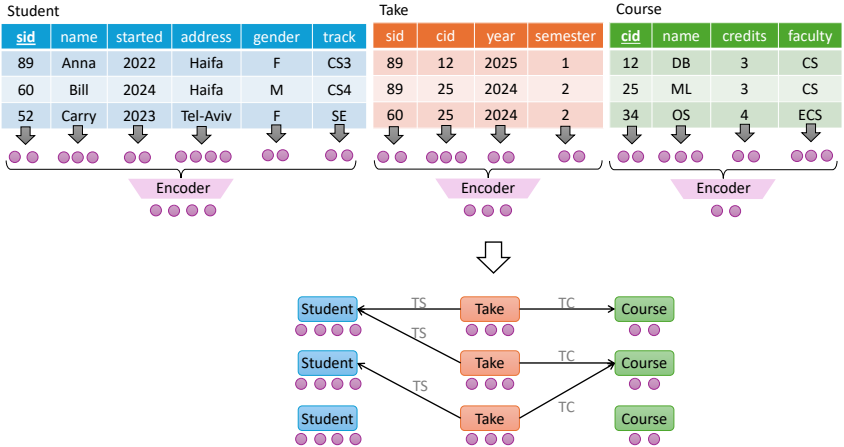
Outline

1. Data Models
   - Graphs
   - Tables
   - Relational Databases
2. Databases as Hetero-Graphs
3. Required Graph Concepts
   - Neighborhood
   - Matrix Representations of Graphs
   - Weisfeiler-Leman Isomorphism Test
4. ML on Structured Data
   - General ML
   - Predictive Tasks
   - Training

Databases as Heterogeneous Graphs

- Various ways have been proposed for translating relational databases into graphs
- We follow a common conversion
  - E.g., [Robinson et al., 2024] [Wang et al., 2025] [Fey et al., 2024]
  - Sometimes referred to as *Row2Node* [Wang et al., 2024]

Illustration



Value Encoding

- The conversion (like many ML models over tables/databases) requires a representation of each tuple as a numerical vector
- To this aim, we first encode each cell as a numerical vector
- Cell encoding is typically determined by the modality (i.e., the type) of the domain of the cell value
  - E.g., text, numbers, categories, timestamps, images
- Off-the-shelf encoders provided in ML libraries
  - E.g., pytorch frame[1] and scikit-learn preprocessing[2]

[1] https://pytorch-frame.readthedocs.io/en/latest/index.html
[2] https://scikit-learn.org/stable/api/sklearn.preprocessing.html

Examples of Encodings

- A *categorical* value can be encoded by numbering categories, by one-hot encoding, by embedding each category into a learned feature vector, etc.

sid	degree-level	cat#	one-hot	embed
89	UG	1	(1,0,0)	(0.41,0.2,0.35)
60	masters	2	(0,1,0)	(0.98,0.55,0.59)
26	UG	1	(1,0,0)	(0.41,0.2,0.35)
52	doctorate	3	(0,0,1)	(0.88,0.42,0.4)
56	masters	2	(0,1,0)	(0.98,0.55,0.59)

- A *numerical* value x can undergo a scaling w.r.t. the dataset (e.g., to obtain numbers in $[0, 1]$), or a linear encoding $(w_1 x, w_2 x, \dots)$ where the w_i are weights
  - Predefined weights or learned as part of training

Examples of Encodings (cont'd)

- A *textual* value can be assigned an embedding from a text model
  - E.g., a transformer that is either *fully precomputed* or with *learned parameters*
- A *timestamp* value can be encoded simply via the Epoch converter, or (more commonly) as an encoding of the time's individual features, e.g., month, day, day-of-the-week, hour, minute, using direct or cyclic encoding

Timestamp (UTC)	Epoch (sec)	Extracted components	Circular (hour)
2026-01-09 00:00:00	1767916800	h = 0, d = 9, m = 1, wd=6	$(\sin 0,\ \cos 0)$
2026-01-09 06:00:00	1767938400	h = 6, d = 9, m = 1, wd=6	$(\sin \frac{\pi}{2},\ \cos \frac{\pi}{2})$
2026-01-09 18:00:00	1767981600	hour = 18, d= 9, m = 1, wd=6	$(\sin \frac{3\pi}{2},\ \cos \frac{3\pi}{2})$

Tuple Encoding

- Let $(\mathbf{A}, \mathbf{dom})$ be a tabular signature, $\mathbf{A} = A_1, \ldots, A_n$ and $\mathbf{dom} = dom_1, \ldots, dom_n$
- An *encoding scheme* Enc for $(\mathbf{A}, \mathbf{dom})$ associates with every column $i = 1, \ldots, n$:
  - A dimension $d_i \in \mathbb{N}$
  - A value encoding function $e_i : dom_i \to \mathbb{R}^{d_i}$
- A tuple $\mathbf{t}$ is encoded as a concatenation of its value embeddings:

$$\text{Enc}(\mathbf{t}) := e_1(\mathbf{t}[A_1]) \oplus \cdots \oplus e_n(\mathbf{t}[A_n]) \in \mathbb{R}^{d_1 + \cdots + d_n}$$

- We may also have another encoding layer that further processes the result
  - (e.g., lowering the dimension)

Encoding Scheme for a Relational Schema

An *encoding scheme* Enc for a relational schema $(\mathcal{R}, \text{sig}, \text{key}, \mathbf{FK})$ associates with each relation name R an encoding scheme Enc_R for $\text{sig}(R)$

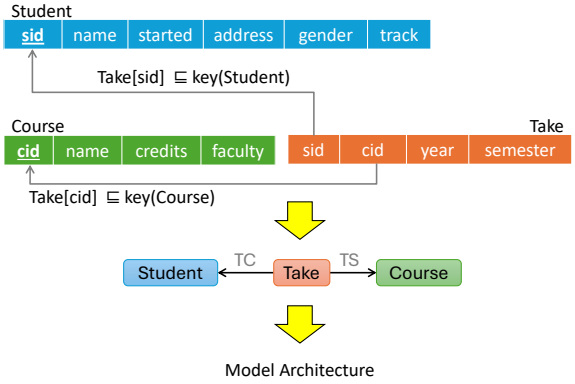
- Hence, the encoding of a tuple $\mathbf{t}$ in $D(R)$ is $\text{Enc}_R(\mathbf{t})$

The Common Conversion

- Let $\mathcal{S} = (\mathcal{R}, \text{sig}, \text{key}, \mathbf{FK})$ be a relational schema and:
  - D be a database over $\mathcal{S}$
  - Enc be an encoding scheme for $\mathcal{S}$
- The hetero-graph $\mathcal{G}_D = (L_V, L_E, V, E, \lambda, \mathbf{X})$ is defined as follows:
  - Every fact $R(\mathbf{t})$ becomes a node v with $\lambda(v) = R$ and $\mathbf{x}_v = \text{Enc}_R(\mathbf{t})$
  - For every FK $\Phi = R[\mathbf{C}] \sqsubseteq \text{key}(R')$ and facts $R(\mathbf{t})$ and $R'(\mathbf{t}')$ such that $R(\mathbf{t})$ references $R'(\mathbf{t}')$, we have the edge $(R(\mathbf{t}), \Phi, R'(\mathbf{t}'))$
  - In words, for every reference in the database, we add a corresponding edge labeled by the corresponding FK
- Note: if D and D' are databases of the same schema $\mathcal{S}$, then the hetero-graphs $\mathcal{G}_D$ and $\mathcal{G}_{D'}$ are type consistent, i.e., have the same hetero-schema

Why are Schemas Important to ML?

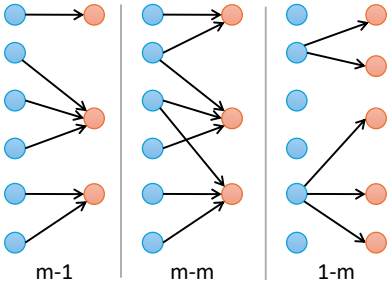
- A common approach to ML over databases is to learn a model for a specific relational schema $\mathcal{S} = (\mathcal{R}, \text{sig}, \text{key}, \textbf{FK})$
- And, a common methodology is by converting the database into a hetero-graph and learn a model for the corresponding hetero-graph schema (L_v, L_e, M, d)



Mapping a Database Schema to a Hetero-Graph Schema

- Let:
  - $\mathcal{S} = (\mathcal{R}, \text{sig}, \text{key}, \mathbf{FK})$ be a relational schema
  - Enc be an encoding scheme for $\mathcal{S}$
- The schema of the hetero-graphs obtained from $\mathcal{S}$ is (L_v, L_e, M, δ) where:
  - $L_v := \mathcal{R}$
  - $L_e := \mathbf{FK}$
  - M maps every edge label (FK) $\Phi = R[\mathbf{C}] \sqsubseteq \text{key}(S)$ into the pair (R, S)
    - That is, Φ connects an R-node to an S-node
  - δ maps every relation name $R \in \mathcal{R}$ into the output dimension of Enc_R

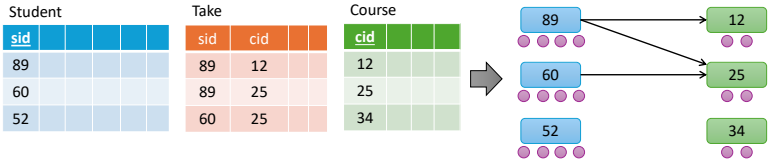
Many-to-One vs. Many-to-Many



- Importantly, FKs represent many-to-one relationships among nodes: a node can have **at most one outgoing edge** of a given type (FK)
- Hence, the out-degree of a node is bounded by the number of FKs from the node's relation
- This restriction implies that we construct only *certain types of graphs*
- Graph learning can be highly sensitive to the kind of conversion
- Hence, other conversions studied

Row2N/E

- Another conversion [Wang et al., 2024] allows to designate some relationships with multiple FKs as converting directly into edges
- Specifically, every pair of FKs from R represents an edge type in some direction

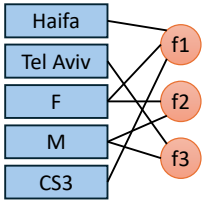


- Which direction? For common ML models, this is *not* important
- What about features of the connecting table?
  - These can be encoded as *edge features*

Cells as Nodes

Another example of a conversion: every *cell value* and every *fact* become a node, and each fact connects to its values [Cappuzzo et al., 2020]

	address	gender	track
f1	Haifa	F	CS3
f2	Haifa	M	CS4
f3	Tel-Aviv	F	SE



Conversion in the Course

Throughout the course, we will assume the Row2Node (FK-to-edge) conversion

- Which is most studied in the literature of deep learning over relational databases

Outline

1 Data Models
- Graphs
- Tables
- Relational Databases

2 Databases as Hetero-Graphs

3 Required Graph Concepts
- Neighborhood
- Matrix Representations of Graphs
- Weisfeiler-Leman Isomorphism Test

4 ML on Structured Data
- General ML
- Predictive Tasks
- Training

Outline

1. Data Models
   - Graphs
   - Tables
   - Relational Databases
2. Databases as Hetero-Graphs
3. Required Graph Concepts
   - Neighborhood
   - Matrix Representations of Graphs
   - Weisfeiler-Leman Isomorphism Test
4. ML on Structured Data
   - General ML
   - Predictive Tasks
   - Training

Neighbor Notation

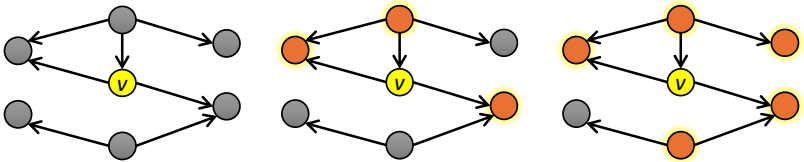
- Let G be a graph and v be a node of G
- We denote by:
  - $\text{Nodes}(G)$ the set of nodes of G
  - $N_G(v)$ the set of *neighbors* of v, that is, the nodes u of G that share an edge (in either direction) with v
    - Note: we follow the convention that $v \notin N_G(v)$
  - $N_G^{\text{in}}(v)$ the set of neighbors u of v via an *incoming* edge (u, v) into v
  - $N_G^{\text{out}}(v)$ the set of neighbors u of v via an *outgoing* edge (v, u) from v
- We usually remove the subscript G when G is clear from the context
  - E.g., $N(v)$ instead of $N_G(v)$

Node Degrees

Degrees are defined accordingly:

$$\deg_G(v) := |N_G(v)| \qquad \deg_G^{\text{in}}(v) := |N_G^{\text{in}}(v)| \qquad \deg_G^{\text{out}}(v) := |N_G^{\text{out}}(v)|$$

k-Hop Neighborhood



0-hop neighborhood 1-hop neighborhood 2-hop neighborhood

- Let $G = (V, E)$ be a graph and v be a node of G
- For $k \in \mathbb{N}$, the *k-hop neighborhood of v*, denoted $N_G^k(v)$, is the set of nodes in all undirected paths of length at most k, starting from v
  - That is, $N_G^k(v)$ consists of all nodes of distance at most k from v
- Note: v belongs to the k-hop neighborhood of v

Outline

1. Data Models
   - Graphs
   - Tables
   - Relational Databases
2. Databases as Hetero-Graphs
3. Required Graph Concepts
   - Neighborhood
   - Matrix Representations of Graphs
   - Weisfeiler-Leman Isomorphism Test
4. ML on Structured Data
   - General ML
   - Predictive Tasks
   - Training

Matrix Representations of Graphs

- It is very often convenient to represent a graph as a matrix
  - It allows for the extraction of interesting properties of the graph
  - Importantly, *spectral analysis* of graphs refers to the study of a graph via the spectrum—eigenvalues and eigenvectors—of matrix representations
  - It allows us to represent ML models on graphs as learnable numerical formulas over matrices (and tensors)
- We will encounter various matrices throughout the course; we define them here

The Adjacency Matrix

- Most importantly, the *adjacency matrix* of an undirected graph $G(V, E)$ with $V = \{v_1, \ldots, v_n\}$ is the matrix $\mathbf{A}$ with

$$\mathbf{A}_{i,j} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \in E \\ 0 & \text{otherwise} \end{cases}$$

  - The degree of a node is obtained by summing either its row or its column
- For a directed graph replace $\{v_i, v_j\}$ with (v_i, v_j)
  - The *in-degree* of a node is obtained by summing its column
  - The *out-degree* by summing its row

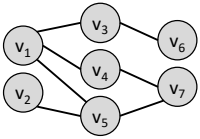
Degree and Laplacian Matrices

- Two other matrices commonly associated with an undirected graph $G(V, E)$ where $V = \{v_1, \ldots, v_n\}$: the *degree matrix* and the *graph Laplacian*
  - Various variants exist for directed graphs
- The degree matrix $\mathbf{D} \in \mathbb{R}^{n \times n}$ is a diagonal matrix:

$$\mathbf{D}_{i,j} := \begin{cases} \deg(v_i) & \text{if } i = j\,; \\ 0 & \text{otherwise.} \end{cases}$$

  - We can similarly define the *out-degree* matrix and the *in-degree* matrix
- The graph Laplacian is defined by $\mathbf{L} = \mathbf{D} - \mathbf{A}$
- When G is undirected, each of $\mathbf{A}$, $\mathbf{D}$ and $\mathbf{L}$ are *symmetric*

Example



A

0	0	1	1	1	0	0
0	0	0	0	1	0	0
1	0	0	0	0	1	0
1	0	0	0	0	0	1
1	1	0	0	0	0	1
0	0	1	0	0	0	0
0	0	0	1	1	0	0

D

3	0	0	0	0	0	0
0	1	0	0	0	0	0
0	0	2	0	0	0	0
0	0	0	2	0	0	0
0	0	0	0	3	0	0
0	0	0	0	0	1	0
0	0	0	0	0	0	2

L

3	0	-1	-1	-1	0	0
0	1	0	0	-1	0	0
-1	0	2	0	0	-1	0
-1	0	0	2	0	0	-1
-1	-1	0	0	3	0	-1
0	0	-1	0	0	1	0
0	0	0	-1	-1	0	2

Outline

1. Data Models
   - Graphs
   - Tables
   - Relational Databases
2. Databases as Hetero-Graphs
3. Required Graph Concepts
   - Neighborhood
   - Matrix Representations of Graphs
   - Weisfeiler-Leman Isomorphism Test
4. ML on Structured Data
   - General ML
   - Predictive Tasks
   - Training

Graph Isomorphism

- $G = (V, E)$ and $G' = (V', E')$ are *isomorphic* if there is a bijection $\varphi : V \to V'$ such that for all $v, u \in V$ it holds that $(v, u) \in E$ if and only if $(\varphi(v), \varphi(u)) \in E'$
  - In the case of an undirected graph: $\{v, u\} \in E$ if and only if $\{\varphi(v), \varphi(u)\} \in E'$
- Note: two graphs G_1 and G_2 with adjacency matrices $\mathbf{A}_1$ and $\mathbf{A}_2$ are isomorphic iff there is a *permutation matrix* $\mathbf{P}$ such that $\mathbf{A_1} = \mathbf{P}\mathbf{A_2}\mathbf{P}^\top$
  - A *permutation matrix* is a matrix with precisely one 1-cell in each row and in each column, and 0 everywhere else
  - Idea: first multiplication permutes the rows, second permutes the columns
- Graph isomorphism is not known to be solvable in polynomial time
  - It is in NP, but conventionally believed to be easier than NP-hard problems
  - In contrast, it is NP-complete to decide whether a graph G is *isomorphic to a subgraph* of another graph G'

Why is Graph Isomorphism Important to us?

- Consider a graph G and two nodes v and u of G
- If $N^k(v)$ and $N^k(u)$ induce isomorphic subgraphs (mapping v to u), then *any algorithm* that looks only at k-hops cannot distinguish between v and u
  - No matter how "deep" or "full-of-parameters" the algorithm is
- As we will see in the course, this will also be true for an important class of algorithms (GNNs), even if we replace isomorphism with a *weaker* notion that is easy to compute (in contrast to isomorphism), namely **WL**
  - An idea from the 1960s, nowadays found very useful in graph ML design & analysis
- This will shed light on the expressiveness and limitations of the families of functions we learn over graphs

Weisfeiler-Leman (WL) Isomorphism Test (Intuition)

- Suppose that we wish to quickly prove that G_1 and G_2 are **not** isomorphic
  - If the node sets of G_1 and G_2 have different sizes, we're done
  - Otherwise, group the nodes by their degree; if any two corresponding groups have different sizes, then we're done
  - Otherwise, further group nodes by the collection of the degrees of their neighbors
  - ...


- The Weisfeiler-Leman isomorphism test [Weisfeiler and Leman, 1968] applies this process repeatedly until the classification cannot be further refined
- In each round, nodes v are being "colored" by a natural number $\kappa(v) \in \mathbb{N}$ that represents the group description (e.g., "all nodes of degree 3")
- Referred to as *WL*; also *1-WL* (k-WL colors k-sequences of nodes)

Undirected WL

- WL is based on a hash (coloring) function $\textsc{Hash}$ that takes as input a node color $c \in \mathbb{N}$ and a bag B of neighbor colors, and returns a color $\textsc{Hash}(c, B) \in \mathbb{N}$
  - By a *bag* we mean a set where each element is assigned a multiplicity
- Crucially, $\textsc{Hash}(c, B)$ is assumed to be *injective*:

$$\textsc{Hash}(c, B) = \textsc{Hash}(c', B') \text{ iff } c = c' \text{ and } B = B'$$

- Two colorings $\kappa : V \to \mathbb{N}$ and $\kappa' : V \to \mathbb{N}$ for a graph $G = (V, U)$ are *equivalent* if they define the same groups; that is:

$$\text{For all } v, u \in V \text{ we have } \kappa(v) = \kappa(u) \text{ iff } \kappa'(v) = \kappa'(u)$$

The Undirected WL(V, E) Procedure

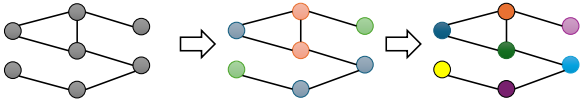
1. Initialize $\kappa_0(v) = 0$ for all $v \in V$
2. Having constructed κ_i, define κ_{i+1} as follows for all nodes v:

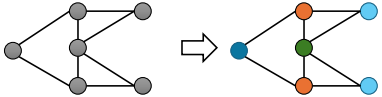
$$\kappa_{i+1}(v) = \textsc{Hash}\big(\kappa_i(v), \{\!\{\kappa_i(u) \mid u \in N(v)\}\!\}\big)$$

3. Stop when κ_{i+1} and κ_i are equivalent

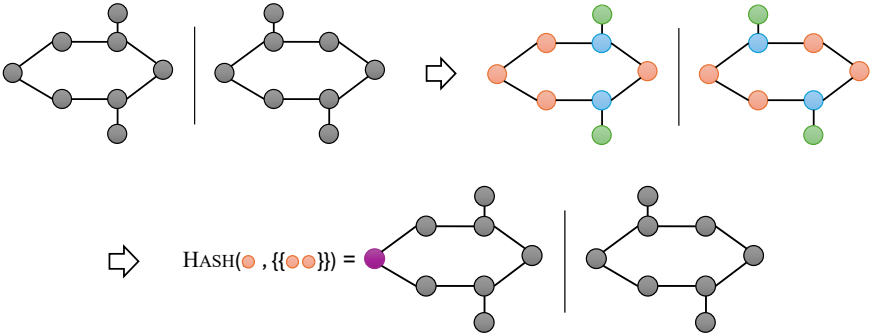
Note: by $\{\!\{\kappa_i(u) \mid u \in N(v)\}\!\}$ we mean the *bag* (rather than *set*) of colors $\kappa_i(u)$

Examples

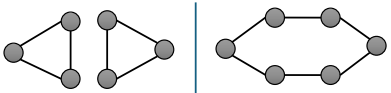




WL for Isomorphism Testing



Non-isomorphic, but WL cannot tell:



Understanding WL I

Theorem

If κ_{i+1} and κ_i are equivalent and we continue WL indefinitely, then κ_j and κ_i are equivalent for every $j \geq i$

- Hence, the stopping criterion terminates the process when reaching a stable state
- **Proof idea:** The state at step $i+1$ is "equivalent" to the state at step i, so HASH will be called with similar inputs, up to recoloring, and will thus produce similar outputs, hence κ_{i+2} is equivalent to κ_{i+1}, and so on (induction)

Understanding WL II

Lemma

For all $i \geq 0$ and nodes $v, u \in V$, if $\kappa_i(v) \neq \kappa_i(u)$, then $\kappa_{i+1}(v) \neq \kappa_{i+1}(u)$.

- Hence, the grouping induced by κ_{i+1} is a refinement (i.e., obtained by splitting groups) of the grouping κ_i
- **Proof idea:** From the ith step on, $\textsc{Hash}(c, B)$ will be called with two different colors c, and hence produce two different new colors c'

Understanding WL III

Theorem

WL terminates after at most $|V|$ steps

- Hence, we can execute WL efficiently
- **Proof idea:** Due to the previous lemma, every step strictly refines the previous grouping, and we can have at most $|V|$ groups

Understanding WL IV

Theorem

Let φ be an isomorphism between the graphs $G = (V, E)$ and $G' = (V', E')$. If we run WL in parallel over G and G', then $\kappa_i(v) = \kappa_i(\varphi(v))$ for all $i \geq 0$.

- Hence, a necessary condition for G and G' to be isomorphic is that each τ_i produces the same number of occurrences of every color
- **Proof idea:** Straightforward induction on the step number

Directed and Labeled WL

- Directed: (e.g., [Rossi et al., 2023])

$$\kappa_{i+1}(v) = \textsc{Hash}\Big(\kappa_i(v)\,,\ \{\!\{\kappa_i(u) \mid u \in N^{\text{in}}(v)\}\!\},\{\!\{\kappa_i(u) \mid u \in N^{\text{out}}(v)\}\!\}\Big)$$

- Labeled: $\kappa_0(v) = \lambda(v)$
  - Identifying labels as natural numbers
- Featured & labeled (heterogeneous): $\kappa_0(v) = \textsc{Hash}'(\lambda(v), \mathbf{x}_v)$
  - For some perfect hashing function $\textsc{Hash}'$
  - We need to assume that $\mathbf{x}_v$ comes from a *countable domain* (e.g., $\mathbb{Z}^d$) for such $\textsc{Hash}'$ to exist

That's it! The lemma and theorems continue to hold

Outline

1. Data Models
   - Graphs
   - Tables
   - Relational Databases
2. Databases as Hetero-Graphs
3. Required Graph Concepts
   - Neighborhood
   - Matrix Representations of Graphs
   - Weisfeiler-Leman Isomorphism Test
4. ML on Structured Data
   - General ML
   - Predictive Tasks
   - Training

Outline

1. Data Models
   - Graphs
   - Tables
   - Relational Databases
2. Databases as Hetero-Graphs
3. Required Graph Concepts
   - Neighborhood
   - Matrix Representations of Graphs
   - Weisfeiler-Leman Isomorphism Test
4. ML on Structured Data
   - General ML
   - Predictive Tasks
   - Training

Core ML Task

- We have a space $\mathcal{X}$ of *examples* (*instances*), a *target space* $\mathcal{Y}$
- We have an unknown function $f : \mathcal{X} \to \mathcal{Y}$ that we wish to mimic
- For that, we are getting a set S of labeled examples (x_i, y_i) where $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$, representing values of f on different inputs $(y_i = f(x_i))$
- We have a hypothesis space $\mathbf{H}$ of functions (models) $h : \mathcal{X} \to \mathcal{Y}$
  - $\mathbf{H}$ does not necessarily include f
- Our goal is to find a hypothesis $h \in \mathbf{H}$ that is closest to f
- To this end, we assume that the labeled examples of S are chosen randomly
  - (e.g., i.i.d.)

Common Types of ML Tasks

- ML tasks traditionally classified into different types, based on the target space $\mathcal{Y}$
  - Binary classification: $\mathcal{Y} = \{0, 1\}$
  - Multi-class classification: $\mathcal{Y} = \{0, \ldots, K-1\}$ for $K \geq 2$
  - Multi-label classification: $\mathcal{Y} = \mathcal{P}(\{0, \ldots, K-1\})$ – each x mapped to a *set* of labels
  - Regression: $\mathcal{Y} = \mathbb{R}$
  - Multivariate regression: $\mathcal{Y} = \mathbb{R}^k$
- Commonly, a binary classification model h is learned as a regression model h' (e.g., a probability function $h' : \mathcal{X} \to [0, 1]$), followed by thresholding:

$$h(x) := \begin{cases} 0 & \text{if } h'(x) < \tau; \\ 1 & \text{otherwise.} \end{cases}$$

- We will also see multi-class classification via multivariate regression

Training

- *Training* is the process of computing the candidate $h \in \mathbf{H}$, given S
- Conventionally, this is done by adopting a *loss function* $\text{Loss}(S, h)$ over the set

$$\{(x_i, y_i, h(x_i)) \mid (x_i, y_i) \in S\},$$

  stating how far off h is
- We then apply an algorithm for finding h that minimizes $\text{Loss}(S, h)$
- For example $\text{Loss}(S, h) := \sum_{(x_i, y_i) \in S} \|y_i - h(x_i)\|$ when $(\mathcal{Y}, \|\cdot\|)$ is a metric space

Parametric Models

- Most prominent examples of ML are cases where **H** is of the form $\{h_{\boldsymbol{\theta}} \mid \boldsymbol{\theta} \in \mathbb{R}^k\}$ where $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_q)$ is a sequence of *numerical parameters*
  - Also called *weights* or *coefficients*
  - Same number q of parameters for every model in **H**
- Hence, the goal is to compute the best $\boldsymbol{\theta}$
- But there are other models, called *non-parametric*, where different hypotheses differ in their entire form, not (just) the parameters
  - Most importantly (to us), *decision trees*
  - Also, *k-Nearest-Neighbor*

Regularization

- A typical addition is a component that introduces a bias toward "preferred" models; specifically, *regularization* penalizes h for being complex
- Rationale: complex functions might *overfit*, that is, hang on to nuances of the examples
  - Simplicity forces the choice of h to grasp the big picture and better generalize

Common Loss Functions for Numerical Parametric Prediction

- Denote $\hat{y}_i = h(x_i)$, the predicted value (aiming to be close to y_i)
- $\text{Loss}(S, h_{\boldsymbol{\theta}}) := \text{Err}(S, h_{\boldsymbol{\theta}}) + \lambda \cdot \text{Reg}(h_{\boldsymbol{\theta}})$ (where λ tunes the level of regularization)

Error functions: (for regression)

- Mean Squared Error: $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
- Mean Absolute Error: $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$
- Huber Loss: $\frac{1}{n}\sum_{i=1}^{n} L(y_i - \hat{y}_i)$ where

$$L(r) = \begin{cases} \frac{1}{2}r^2 & \text{if } |r| < \delta; \\ \delta(|r| - \frac{1}{2}\delta) & \text{otherwise.} \end{cases}$$

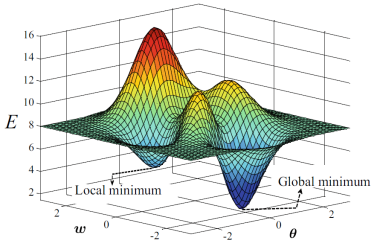
  - Combination between MSE (on small values) and MAE (on large values)

- Cross Entropy: (for binary/multiclass) $-\frac{1}{n}\sum_{i=1}^{n}\sum_{c} y_{i,c}\log(\hat{y}_{i,c})$
  - $y_{i,c}$ indicates whether y_i is labeled c ($y_{i,c} = 1$) or not ($y_{i,c} = 0$)
  - Assumes $0 < \hat{y}_{i,c}$

Regularization terms:

- L1 (Lasso regression): $\sum_{\theta_i \in \boldsymbol{\theta}} |\theta_i|$
- L2 (Ridge regression): $\sum_{\theta_i \in \boldsymbol{\theta}} \theta_i^2$

Fitting via Optimization

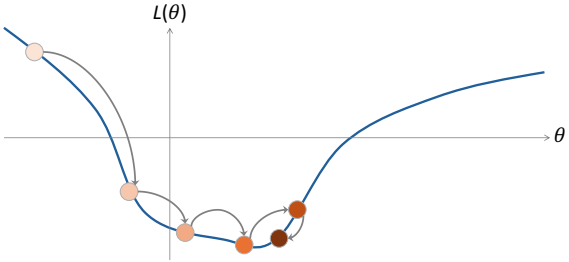


- Importantly, we translate ML into an algorithmic optimization problem
- Some traditional ML algorithms solve it exactly, e.g., *Support Vector Machines*
- Yet, for complex models with many parameters, training on large volumes of data, a guaranteed optimum is typically computationally intractable, and impractical

Gradient-Based Optimization

Hence, we use heuristic techniques that are successful in practice

- And oftentimes prove their quality analytically under assumptions
- The most common techniques are *gradient-based* methods



Gradient Descent

- Denote $L(\boldsymbol{\theta}) := \text{Loss}(S, h_{\boldsymbol{\theta}})$
- Assume that L is differentiable, and we can compute the gradient:

$$\nabla L(\boldsymbol{\theta}) := \left(\frac{\partial L}{\partial \theta_1}(\boldsymbol{\theta}), \ldots, \frac{\partial L}{\partial \theta_q}(\boldsymbol{\theta}) \right)$$

- *Gradient descent* (GD) iteratively updates $\boldsymbol{\theta}$ via steps of the form

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \eta \cdot \nabla L(\boldsymbol{\theta})$$

where $\eta \in \mathbb{R}_+$ is a positive *step size* (or *learning rate*)

- Until a stopping condition is reached

Problems with Vanilla GD

- **Costly gradient computation**: We may have too many examples to compute the gradient on the whole training set S on each step
  - Optimization is often done in batch-oriented hardware (e.g., GPUs, to be discussed later in the course), and $\nabla L(\boldsymbol{\theta})$ may be too big to fit when S is large
- **Anomalous convergence**: Which learning rate η to use?
  - With a large η, GD can get way off due to crude jumps
  - With a small η, GD can get stuck in local minima and generally converge too slowly
  - The fundamental issue is that the learning rate η is fixed, no matter where we are in the learning process

Stochastic Variants

- Efficiency & randomness introduced by the *stochastic* variants of GD
- *Stochastic GD* (SGD): pick a random sample $(x_i, y_i) \in S$, typically without replacement, and update only based on this example!

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \eta \cdot \nabla L_i(\boldsymbol{\theta}) \quad \text{where} \quad L_i(\boldsymbol{\theta}) := \text{Loss}(x_i, y_i, h_{\boldsymbol{\theta}})$$

- *Mini-batch SGD*: pick a random *mini-batch* $B \subseteq S$, without replacement, and update only based on this batch:

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \eta \cdot \nabla L_B(\boldsymbol{\theta}) \quad \text{where} \quad L_B(\boldsymbol{\theta}) := \text{Loss}(B, h_{\boldsymbol{\theta}})$$

  - Every *epoch* is a round that completes all batches (and then we can go again)

Adam Optimizer [Kingma and Ba, 2015]

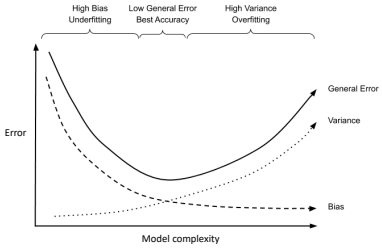
- State-of-the-art form of GD with *adaptive learning rates* that combines known approaches of momentum and RMSProp (Root Mean Square Propagation)
- Maintains a balance between efficient convergence and stability by incorporating an exponentially weighted *moving average of past gradients*
- Past gradients diminish their impact in time (forgotten)

Specification of the Adam Optimizer

- Start with $m_0 = v_0 = g_0 = 0$
- Let t denote the update number, and g_t the gradient of $\boldsymbol{\theta}$ at time t
- *1st & 2nd moments* (for hyper-parameters β_1 and β_2):
  - $m_t = g_t \cdot (1 - \beta_1) + m_{t-1} \cdot \beta_1$ (mean estimate)
  - $v_t = g_t^2 \cdot (1 - \beta_2) + v_{t-1} \cdot \beta_2$ (variance estimate)
- *Bias correction:* $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$; $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$
  - Diminishing amplification
- *Parameter update:* $\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$ for hyper-parameters α and ϵ
- Common defaults: $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

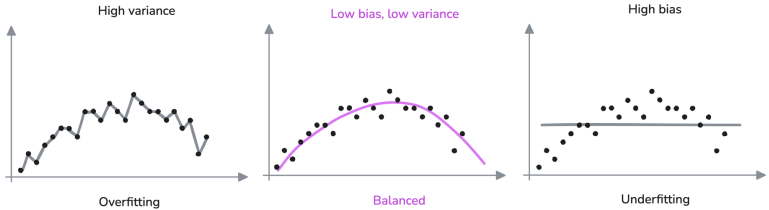
Fitting Risks

- Standard challenges in ML:
  - Under-fitting: Fail to fit the training examples, bad performance during training
  - Over-fitting: Fit well the training set, but generalize poorly to instances outside
- These can be viewed via the *bias-variance tradeoff* of our setting—hypothesis class **H** and learning algorithm
  - Bias: How biased are we towards wrong/simplistic assumptions (e.g., too simple hypotheses)?
    - *High bias means that we can poorly fit our samples*
  - Variance: How sensitive are we to the specifics of the samples (e.g., too complex hypotheses)?
    - *High variance means that we can get highly different things on different sample sets*



https://www.ml-science.com/

Managing the Bias-Variance Tradeoff



- To handle under-fitting, we use more complex models (e.g., additional layers with more parameters), apply additional epochs, use better SGD update strategies
- To handle over-fitting, we use better regularization, and control our ML development using proper *data splitting*

[2] Figure from https://briefgenai.com/blog/fundamentals/bias-variance-under-over-fitting

Data Splitting

- Conventional ML splits the data into three parts:

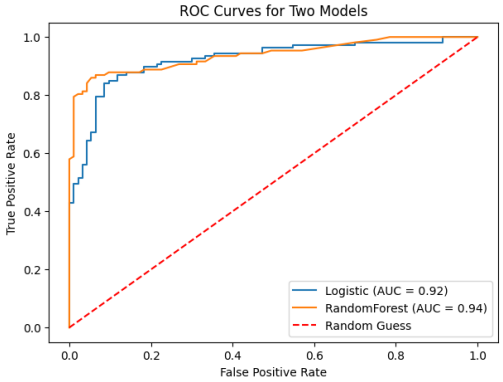
  1. Train data **2.** Validation data **3.** Test data

- The main training process (e.g., SGD) is applied to the **training data**
- **Validation data** is used for *hyper-parameter tuning* (e.g., GD step size, number of layers, regularization weights), *model selection* (in case we train multiple models and choose the best), and *performance monitoring* (e.g., estimating how well we are during SGD) ; not "processed" — just validates
- The **test data** is used for reporting the predicted quality of the model
- Another common split is a holdout data that is allowed to be seen only at the very end (and, once seen, becomes invalid)
  - This is to avoid overfitting of the whole ML development (by the programmers)

Quality Evaluation of a Learned Model

- How do we measure the quality of a model over the test set?
- We can use the error component of the loss function (or some other loss function)
- Important case: threshold-based **binary classifier**
- We would like to measure not just the result (with a specific threshold), but also the ability of the score to support good classification
- A popular measure is the *area under the curve of the receiver operating characteristic* (**ROC-AUC**)
- ROC-AUC is the probability that a randomly chosen "yes" (1) instance is ranked higher than a randomly chosen "no" (0) instance
  - For example, ROC-AUC of a *random score* is 0.5
- The name comes from the *ROC curve*:

Visual Interpretation of ROC-AUC (and why *AUC*?)



https://www.geeksforgeeks.org/machine-learning/auc-roc-curve/

- Sort the 0-instances by decreasing score on the x-axis: $x_1, \ldots, x_n$
- Each x_i defines a *false positive rate*: the portion of 0-instances scored at least as x_i, namely i/n
- For each x_i, mark its y-value as the ratio of *true positives* (i.e., % of 1-instances scored above x_i) — called the ROC curve
- Our probability is the *area under the curve*, i.e., the probability that a random point is placed under the curve

Outline

1. Data Models
   - Graphs
   - Tables
   - Relational Databases
2. Databases as Hetero-Graphs
3. Required Graph Concepts
   - Neighborhood
   - Matrix Representations of Graphs
   - Weisfeiler-Leman Isomorphism Test
4. ML on Structured Data
   - General ML
   - Predictive Tasks
   - Training

Predictive Tasks on Graphs: Generic Form

- A general class of functions we learn over graphs have the shape

$$F : \mathbf{G} \times \mathbf{V}^k \to \mathbb{R}^\ell$$

where:

  - $\mathbf{G}$ is a class of graphs
  - $\mathbf{V}$ is the domain of graph nodes
  - $F(G, \mathbf{v})$ is defined only when $\mathbf{v}$ is a sequence of k nodes from G

- In the case of featured graphs, we require that all graphs in $\mathbf{G}$ have the same feature dimension d
- In the case of hetero-graphs, we require that all graphs in $\mathbf{G}$ are type consistent
- Next, we see examples

Predictive Tasks on Graphs: Node Tasks

- Node classification: $\mathbf{G} \times \mathbf{V} \to \{0, 1\}$
  - For example, student's graduation within four years (students represented as nodes)
  - This variant is *binary classification*; *multiclass classification* has the form $\mathbf{G} \times \mathbf{V} \to \{0, \ldots, K-1\}$
- Node regression: $\mathbf{G} \times \mathbf{V} \to \mathbb{R}$
  - For example, student's average grade (or #credit points) by the end of the year
- Node embedding: $\mathbf{G} \times \mathbf{V} \to \mathbb{R}^k$
  - For example, in *protein folding* we have $\mathbf{G} \times \mathbf{V} \to \mathbb{R}^3$ where each node represents an amino acid, and we wish to predict its position in a folded state
  - Often used to facilitate link prediction
- Node clustering: $\mathbf{G} \times \mathbf{V} \to \mathbb{N}$
  - $F(G, v) = i$ means that v is assigned to cluster number i
  - Also known as *community detection*

Predictive Tasks on Graphs: beyond Nodes

- Link prediction: $\mathbf{G} \times \mathbf{V}^2 \to \{0, 1\}$
  - For example, predict which student will take which course
- Graph-level tasks: classification: $\mathbf{G} \to \{0, 1\}$, regression $\mathbf{G} \to \mathbb{R}$, graph embedding $\mathbf{G} \to \mathbb{R}^k$, graph clustering $\mathbf{G} \to \mathbb{N}$
  - Example: Is a molecule toxic (classification)? What is the level of toxicity (regression)?
  - More examples: Embed molecules on a screen so that similar ones are close to each other, and vice versa (embedding); group molecules by their behavior (clustering)
- . . . and other variations, e.g., link regression, predict multi-ary relationships, etc.

Predictive Tasks on Tables

- Let $(\mathbf{A}, \mathbf{dom})$ be a tabular signature, $\mathbf{A} = A_1, \ldots, A_n$, $\mathbf{dom} = dom_1, \ldots, dom_n$
- Row regression/classification: $F : dom_1 \times \cdots \times dom_n \to \mathbb{R}^\ell$ (classic ML)
- Column prediction: Given a table r with the column A_i missing, complete the missing column
  - The domain dom_i of A_i can be Boolean (binary classification), categorical (multiclass classification), or numerical (regression)
- Missing data imputation: Given a table r' with several cells missing, guess the completion r of r'
- Data cleaning: Given a table r' with errors in some cells, guess which cells have wrong values, and guess a correction r of r'
- Entity resolution: Given a table r, guess rows refer to the same real-world entity (a.k.a. *de-duplication*)

Predictive Tasks on Databases

- Let $\mathcal{S} = (\mathcal{R}, \text{sig}, \text{key}, \textbf{FK})$ be a database schema
- Denote by $\mathbf{D}_{\mathcal{S}}$ the set of all databases over $\mathcal{S}$
- Column prediction / missing data imputation / cleaning: Same as table tasks on a designated *target relation* $T \in \mathcal{R}$, given the entire database D (not just $D(T)$)
- Entity linking/matching: given a database D and two relation names $R, S \in \mathcal{R}$, guess which tuples of R are linked to which tuples of S
  - $F : \mathbf{D}_{\mathcal{S}} \times dom(R) \times dom(S) \to \{0, 1\}$
  - E.g., which Customer rows will purchase which Product rows
- Relational forecasting: database changes over time—D_t is the database at time t; fix a numerical query $Q : \mathbf{D}_{\mathcal{S}} \to \mathbb{R}$; predict the answer $Q(D_t)$ at a future time t
  - For example, predict the October 2026 #sales for customers from Berlin

A Note about Invariance of Models

- An important property of the class of models we learn is *invariance* to aspects of the database that should not matter to us
  - And are expected to be changed without implication on the model
- Examples:
  - Order of columns in each relation
  - Order of the tuples
  - Opaque keys that we use as artificial identifiers
  - *Anything else?*
    - For example, one might want invariance to the *attribute names*, another may not
- We need to be careful in how we define *invariance*; for example, column reordering will change the order of encoders—might break "legitimate" invariant models

Schema-Level Tasks

- *Schema-level tasks* predict concepts that apply to the schema and not a specific database, but can be learned using specific databases
- Schema matching: Given two schemas $\mathcal{S}$ and $\mathcal{S}'$, determine which pairs of attributes A and A' from the two refer to the same information (e.g., a person's name, a company's location)
- Query learning: Given answers to a query (e.g., SQL), guess what the query is
- Text-to-SQL: Given a schema $\mathcal{S}$, translate a textual description of a query into an SQL query

Schema-level tasks are not covered in the course!

Outline

1. Data Models
   - Graphs
   - Tables
   - Relational Databases
2. Databases as Hetero-Graphs
3. Required Graph Concepts
   - Neighborhood
   - Matrix Representations of Graphs
   - Weisfeiler-Leman Isomorphism Test
4. ML on Structured Data
   - General ML
   - Predictive Tasks
   - Training

Transductive vs. Inductive Learning in Graphs

- Consider a node-classification task on graphs
- We distinguish between two types of *learning problems*:
  - Inductive: What we considered so far, where we learn a model $F : \mathbf{G} \times \mathbf{V} \to \{0, 1\}$ that can be applied to unseen graphs
    - For example, we learn the *weights* of a general graph model (e.g., a GNN)
  - Transductive: We have a fixed graph $G = (V, E)$, treated as our *context*, for both training and evaluation time—train on labeled nodes, evaluate on the rest
    - This means that we actually wish to learn a function $F_G : V \to \{0, 1\}$
    - For example, we classify based on *actual node embedding* that we learn for G, and know nothing about new nodes
- Importantly, we can apply an **inductive** learning method on a **transductive** setting by providing it with the same graph G
- Same applies to all other graph-learning tasks (e.g., link prediction)

3 Levels of Relational Learning

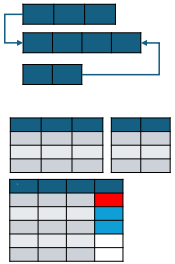
When learning over relational databases (or hetero-graphs), we can consider *three modes of generalization* (considering row-level tasks):

1. Transductive learning: Fixed schema, fixed database, unknown tuple label
   - Example: data imputation
2. Inductive learning: Fixed schema, unknown database, unknown tuple label
   - Example: column prediction
3. Foundation models: Unknown schema, unknown database, unknown tuple label

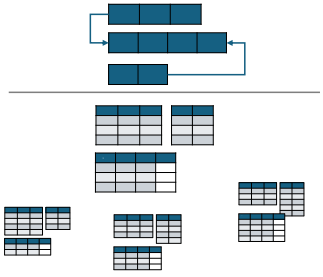
In the course, we will focus mainly on the *inductive setting*, and toward the end discuss *foundation models*

Generalization Modes: Transductive, Inductive, Foundation

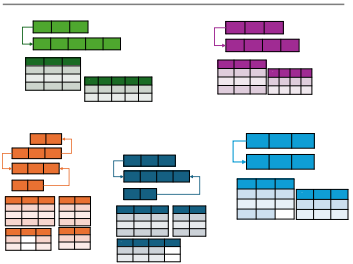
Transductive model:
Known schema, known task, known DB



Inductive model:
Known schema, known task, unknown DB



Foundation model:
Unknown schema, unknown task, unknown DB



Training an Inductive Model in Transductive Conditions

- Consider a learning task on structured data, where we wish to learn a function $f(D, e)$ on an input instance that consists of a *data structure* D and an *entity* e
  - D is a graph or a database
  - e is a node, a row, a fact, a sequence of nodes, etc.
- In an inductive ML setting, we need to train on many examples (D_i, e_i, y_i)
  - This is the case, for instance, when D represents a molecule
- In practice, we often train in a *transductive setting* on a *single* D with many entities, that is, examples (D, e_i, y_i) that share the same D
  - But we still want an inductive model, since the structure D changes in time (e.g., new customers, etc.), and we want it to keep being applicable

Substructures in Mini-Batching

- When D is large, training steps on (D, e_i, y_i) cannot consume it fully
- To handle large structures, mini-batching often involves sampling from D a small sub-structure D_B
  - Hence, SGD is applied to the (D_B, e_i, y_i) of the mini-batch B
- For example, on graph structures, we can sample walks from *k-hop neighborhoods* of sample entities e_i [Hamilton et al., 2017], or take clusters [Chiang et al., 2019]

Conclusion: We commonly learn an inductive model on one big structure by translating the setup into a classic inductive setting with many small structures

Timestamp-Based Training

- When we learn to predict a future event, it is convenient to use timestamps for training
- Specifically, we assume that each fact f of a database D is associated with a timestamp $ts(f)$ that represents the creation time of the the fact
- Timestamps are then used in multiple ways:
  1. *Data splitting*: training data is the oldest, validation data is newer, and test data is the latest
  2. *Structure sampling* for mini-batches may avoid using facts that are newer than the tested target node
  3. The timestamp itself is used as an additional *attribute* of the fact and, consequently, encoded into the feature vector

Key Takeaways from Module 1

- We focus on 3 types of data structures: tables, graphs, and relational databases
- We use a natural translation of a relational database into a hetero-graph
- WL is a generic graph technique for gathering graph information about each node; necessary but insufficient condition for isomorphism
- Many types of ML tasks on each of the three data types
- Generalization modes: transductive (one database), inductive (one schema, input database), foundation models (input schema, input database)

The End

Many thanks for helping with the slides:

- **Shunit Agmon** (Technion)
- **Boaz Berger** (Technion)
- **Ilias Fountalis** (RelationalAI)

References I


Cappuzzo, R., Papotti, P., and Thirumuruganathan, S. (2020).
Creating embeddings of heterogeneous relational datasets for data integration tasks.
In *SIGMOD Conference*, pages 1335–1349. ACM.


Chiang, W., Liu, X., Si, S., Li, Y., Bengio, S., and Hsieh, C. (2019).
Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks.
In *KDD*, pages 257–266. ACM.


Fey, M., Hu, W., Huang, K., Lenssen, J. E., Ranjan, R., Robinson, J., Ying, R., You, J., and Leskovec, J. (2024).
Position: Relational deep learning - graph representation learning on relational databases.
In *ICML*. OpenReview.net.


Hamilton, W. L., Ying, Z., and Leskovec, J. (2017).
Inductive representation learning on large graphs.
In *NIPS*, pages 1024–1034.

References II


Kingma, D. P. and Ba, J. (2015).
Adam: A method for stochastic optimization.
In *ICLR (Poster)*.


Robinson, J., Ranjan, R., Hu, W., Huang, K., Han, J., Dobles, A., Fey, M., Lenssen, J. E., Yuan, Y., Zhang, Z., He, X., and Leskovec, J. (2024).
Relbench: A benchmark for deep learning on relational databases.
In *NeurIPS*.


Rossi, E., Charpentier, B., Giovanni, F. D., Frasca, F., Günnemann, S., and Bronstein, M. M. (2023).
Edge directionality improves learning on heterophilic graphs.
In *LoG*, volume 231 of *Proceedings of Machine Learning Research*, page 25. PMLR.


Wang, M., Gan, Q., Wipf, D., Zhang, Z., Faloutsos, C., Zhang, W., Zhang, M., Cai, Z., Li, J., Mao, Z., Song, Y., Tang, J., Zhang, Y., Yang, G., Lei, C., Qin, X., Li, N., Zhang, H., Wang, Y., and Zhang, Z. (2024).
4dbinfer: A 4d benchmarking toolbox for graph-centric predictive modeling on rdbs.
In *NeurIPS*.

References III


Wang, Y., Wang, X., Gan, Q., Wang, M., Yang, Q., Wipf, D., and Zhang, M. (2025).
Griffin: Towards a graph-centric relational database foundation model.
CoRR, abs/2505.05568.


Weisfeiler, B. and Leman, A. (1968).
The reduction of a graph to canonical form and the algebra which appears therein.
nti, Series, 2(9):12–16.