The Henry and Marilyn Taub
**Faculty of Computer Science**

# Module 2: Structured Learning via Tabular Learning

StructML Course

Benny Kimelfeld
Technion – Israel Institute of Technology

Spring 2026

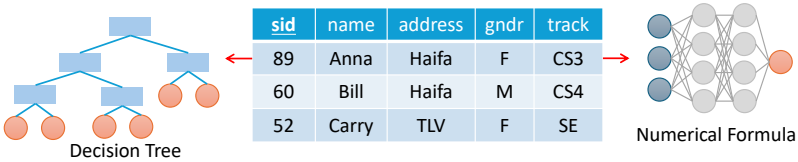
# Module Goals

- Understand the challenges involved in tabular ML
- Learn state-of-the-art ML over tabular data via decision trees
  - Specifically, the state-of-the-art XGBoost
  - (Later in the course, we will learn deep-learning approaches)
- Learn about generic feature generation over relational databases, specifically via SQL queries
  - Also, familiarize with a specific feature-generation system
- Learn popular techniques for feature generation for graph learning

# Outline

# Outline

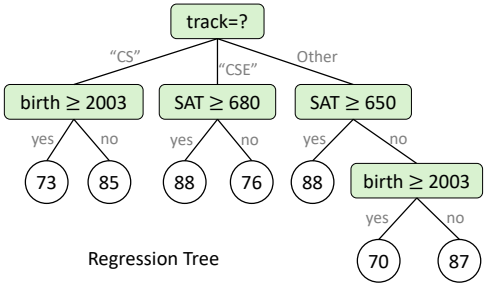
# Types of Tabular ML Models



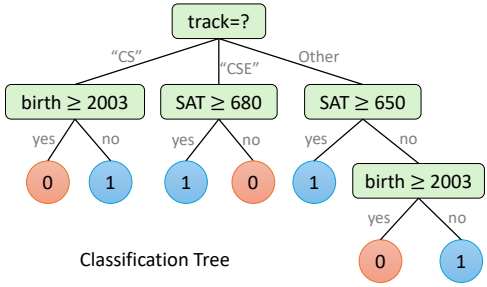

- Classic ML models over tabular data:
  - Algorithms over raw tuple values (typically based on *decision trees*)
  - Numerical formulas over numerical tuple encoding
    - Reduction to classic matrix-based ML
    - E.g., linear models (logistic regression, SVM, etc.)
- We will focus on ensembles of decision trees
  - State-of-the-art among the non-neural models (e.g., **XGBoost**)
  - Often outperforms neural alternatives [Shwartz-Ziv and Armon, 2022]

# Outline

# Illustration of Decision Trees




Regression Tree




Classification Tree

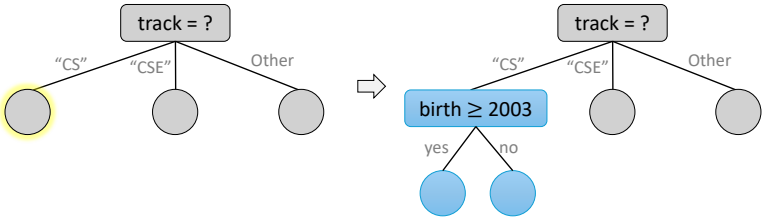
# Decision Trees

- Let $(\mathbf{A}, \mathbf{dom})$ be a tabular signature, $\mathbf{A} = A_1, \ldots, A_n$ ; $\mathbf{dom} = dom_1, \ldots, dom_n$
  - Recall: $\mathbf{A}$ is our sequence of attribute names, and $\mathbf{dom}$ is the sequence of their corresponding domains
  - Slight abuse of notation: identify $\mathbf{dom}$ with the product space $dom_1 \times \cdots \times dom_n$
- A *decision tree* over $(\mathbf{A}, \mathbf{dom})$ is a rooted tree where:
  - Each internal node $v$ with children $u_1, \ldots, u_d$ is associated with a mapping (condition) $\varphi_v : \mathbf{dom} \to \{u_1, \ldots, u_d\}$
  - Each leaf $u$ is associated with a target value $\nu_u \in \mathcal{Y}$
- In a (binary) classification tree, $\mathcal{Y} = \{0, 1\}$; in a regression tree, $\mathcal{Y} = \mathbb{R}$
- The model semantics is straightforward: given a tuple $\mathbf{t} \in \mathbf{dom}$, traverse the tree from the root to a leaf $u$, and output $\nu_u$
  - On every internal node $v$, apply $\varphi_v$ to the tuple $\mathbf{t}$ and go to the child $\varphi_v(\mathbf{t})$

# Learning Decision Trees: Main Idea

- Assume the training set $S = \{(\mathbf{t}_1, y_1), \ldots, (\mathbf{t}_n, y_n)\}$
- Starting with a single-node tree, we greedily iterate over every leaf $u$
  - In some order, e.g., BFS/DFS/...
- If a stopping criterion holds for $u$, then decide on the single predicted value for all rows that end up in $u$
  - E.g., common/average/median training value
  - The stopping condition states that, for the set of training rows that reach $u$, further case splitting is not worthwhile (e.g., overfits)
- Otherwise, consider every split condition $\varphi_u$ (from some class), choose one with a maximum gain, and *split*
  - I.e., add the corresponding new leaves as children of $u$

# Split without Regret




Tree growing is *greedy*: when we split, we do not go back

(Except for post-processing pruning, discussed later)

# Common Condition Types

- DTs restrict the space of split conditions to a collection of simple conditions
  - That, in particular, can be listed efficiently
- Such a condition is conventionally *unary*, based on a single attribute in **A**
  - As opposed to, e.g., $A_1 \geq A_3$ that involves two attributes
  - But nothing stops us from using (slightly) more complex conditions
- The common case of leaf splitting is by an *axis-aligned condition* $A_i \geq c$ for some constant $c \in dom_i$, assuming $dom_i$ is linearly ordered by "$\geq$"
  - Most importantly, a numerical attribute

# The Challenge of Categorical Values

What about an attribute $A$ with a categorical domain $dom$? Several options:

- For very few categories $c_1, \ldots, c_d$, we can *split by value*: $\varphi_{u_i}$ is "$A = c_i$"
  - Same effect as *one-hot encoding*, translating the attribute into $|dom|$ different numeric (1/0) columns
- *Category labeling* $e : dom \to \mathbb{N}$ (equivalently, apply an ordering)
  - Useful only if $dom$ is already naturally ordered (e.g., "low", "medium", "high")
- *Target statistics*: learn a numerical representation $e : dom \to \mathbb{R}$ of the categories, based on the target value $y$ of the learning
  - If our examples are $(\mathbf{t}_1, y_1), \ldots, (\mathbf{t}_n, y_n)$, then a TS can be $e(c) = \mathbb{E}[y \mid \mathbf{t}[A_i] = c]$
  - Problem: the encoding can leak information about the target value, risk of overfitting
  - Solutions proposed in **CatBoost** [Prokhorenkova et al., 2018]

# Conditions over Text Values

What about *text* attributes? Several options:

- Apply *text embedding* (encoding) into $\mathbb{R}^k$, and replace the attribute with the $k$ distinct numerical attributes
  - Problem: individual attributes make no sense
- Treat the text as a *bag of words* (unordered) and treat similarly to a *category-set attribute* (each attribute value includes multiple categories)
  - The condition applies to the weighted list of words, e.g., word $x$ occurs $\geq \tau$ times
  - Various techniques for splitting such values, e.g., [Guillame-Bert et al., 2020]
  - Not covered here

# Gain Functions

- Recall our training set $S = \{(\mathbf{t}_1, y_1), \ldots, (\mathbf{t}_n, y_n)\}$
- If $v$ is a node of the DT, we denote by $S[v]$ the set of training examples that reach $v$ (in the traversal from the root)
- The *gain* of a split $(v, \{u_1, \ldots, u_d\})$ is defined using an *error function* (cost), mapping $S[u]$ to a number $\mathrm{Err}(S[u])$:

$$\mathrm{gain}(v, \{u_1, \ldots, u_d\}) := \mathrm{Err}(S[v]) - \sum_{j=1}^{d} \mathrm{Err}(S[u_j])$$

- For $S' \subseteq S$, the value $\mathrm{Err}(S')$ penalizes $S'$ for being overly uniform (non-sensitive) in its predictions
- Next, we see some examples

# Common Error Functions

Different options for $\mathrm{Err}(S')$ for sharing a uniform decision:

- **Regression**
  - Let $\bar{y}$ be the *mean* of the $y_i$ ; let $y_{0.5}$ be the *median* of the $y_i$
  - Mean Squared Error: $\mathrm{Err}(S') := \frac{1}{|S'|} \sum_{(\mathbf{t},y) \in S'} (y - \bar{y})^2$ (same as *empirical variance*)
  - Mean Absolute Error: $\mathrm{Err}(S') := \frac{1}{|S'|} \sum_{(\mathbf{t},y) \in S'} |y - y_{0.5}|$
    - As theoretical justification, each of these corresponds to a conventional error measure, assuming that $u$ is a leaf and we choose the optimal decision $\nu_u$
- **Classification**
  - Let $S'_c := \{(\mathbf{t}, y) \in S' \mid y = c\}$ and $p_c = \frac{|S'_c|}{|S'|}$
    - That is, $p_c$ is the fraction of examples in $S'$ having the label $c$
  - Entropy: $\mathrm{Err}(S') := \sum_c p_c \cdot (-\log p_c)$
  - Gini: $\mathrm{Err}(S') := \sum_c p_c \cdot (1 - p_c)$

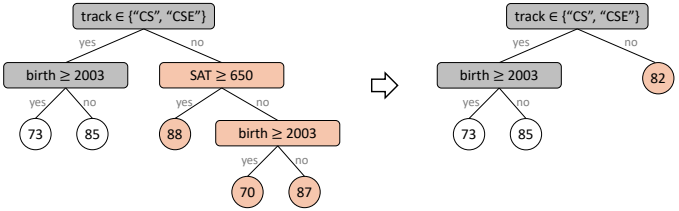
# Common Stopping Criteria

Common stopping criteria for node $v$ include the following:

- All examples in $S[v]$ have the *same label*
- *No split conditions* can separate $S[v]$ (e.g., all attributes used already)
- Too *low splitting gain* on every split (lower than a threshold)
- $S[v]$ has *too few examples* (avoid overfitting)
- A *maximum tree depth* has been reached (avoid overfitting)

# Pruning Post-Processing by Reduced Error Pruning

- Idea: make the tree smaller, by collapsing subtrees, to reduce overfitting




- Reduced Error Pruning: Use the *validation set* to measure the effect of pruning
  - Go over the nodes $v$ in some order, and estimate the effect of collapsing $T_v$ (the subtree rooted at $v$) into a single leaf
  - Very simple: if prediction accuracy does not go down, collapse
- Cost Complexity Pruning: next slide →

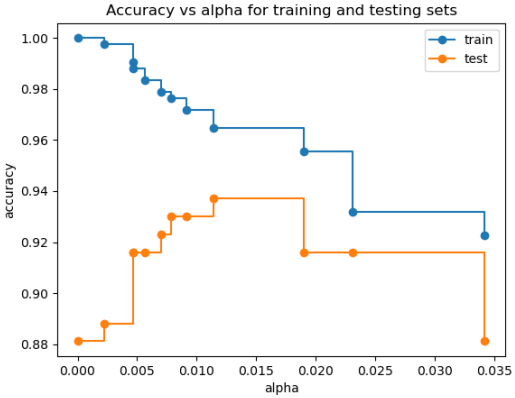
# Cost Complexity Pruning

Accounts for the *complexity* of the tree: prune a subtree $T_v$ rooted at node $v$ if

$$\frac{\mathrm{Err}(S[v]) - \sum_{u \in \mathrm{leaves}(T_v)} \mathrm{Err}(S[u])}{|\mathrm{leaves}(T_v)| - 1} < \alpha$$

for some $\alpha$ defined by the validation set

- cf. [Hastie et al., 2009b]
- Example from scikit-learn doc $\rightarrow$



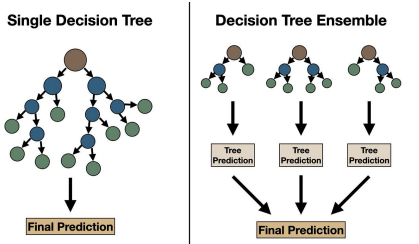
# Outline

# DT Ensembles

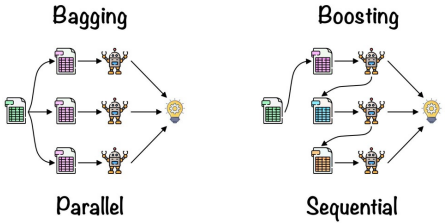
- *Tree ensembles* combine multiple learned trees to obtain better accuracy
- There are **many** ensemble models for decision trees (we list some)
- Main learning techniques for ensembles fall in the categories of *bagging* and *boosting*

# Bagging and Boosting

Evaluation:




Learning:

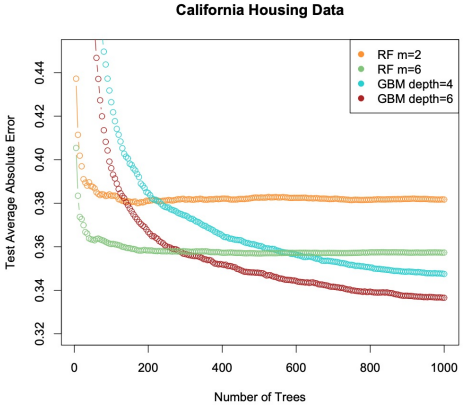



Pictures:

https://encord.com/blog/what-is-ensemble-learning/ ;

https://medium.com/data-science/10-decision-trees-are-better-than-1-719406680564

# Example Improvement on Housing Data




From the [Hastie et al., 2009a] book, comparing *random forests* (bagging) and *gradient boosting models* (boosting)

# Bagging via Random Forests

- Idea: learn multiple DTs $T_1, \ldots, T_n$ independently, combine
- The training of each $T_\ell$ involves randomness:
  - Each DT $T_\ell$ is trained using a *random sample* of the training data
    - Each referred to as a *bag*, hence the term "bagging"
  - For each node split, a *random subset of m attributes* (e.g., $m = \sqrt{p}$ out of a total of $p$ attributes) is selected for choosing the split condition from
- At inference time, apply all trees to the input
  - Classification: take the majority vote
  - Regression: take the average prediction

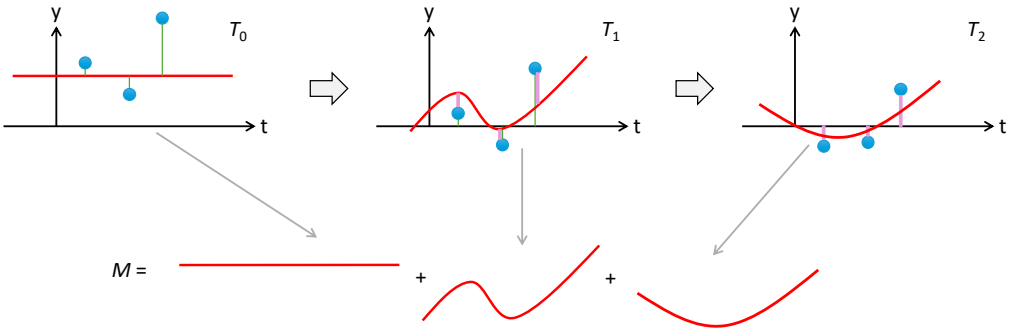
# Boosting by Reweighting

- Idea: learn $T_1, \ldots, T_k$ incrementally, each improving overall performance
- Specifically, learn $M_k(\mathbf{t}) = \gamma_0 + \gamma_1 T_1(\mathbf{t}) + \cdots + \gamma_k T_k(\mathbf{t})$, each $T_\ell$ a DT
- Learn the $\gamma_\ell$ and $T_\ell$ iteratively for $\ell = 1, \ldots, k$
- Before learning $T_\ell$, *re-weight* the examples: the higher the error of $M_{\ell-1}(\mathbf{t}_i)$ is, the higher weight gets $(\mathbf{t}_i, y_i)$
- Importantly, each $T_\ell$ aims to be accurate on the re-weighted training set independently of what the previous DTs did; the $\gamma_i$'s weight in the overall average
- Example: in **AdaBoost** the weight is $e^{-y_i \cdot M_{\ell-1}(\mathbf{t}_i)}$ where $y_i \in \{-1, 1\}$

# Idea of Gradient Boosting (GB)

- GB also learns an additive model $M_k(\mathbf{t}) = \gamma_0 + \gamma_1 T_1(\mathbf{t}) + \cdots + \gamma_k T_k(\mathbf{t})$ iteratively
- Let $(\mathbf{t}_1, y_1), \ldots, (\mathbf{t}_n, y_n)$ be the training examples
- At step $\ell$, the error (*residual*) at point $\mathbf{t}_i$ is $r_i = y_i - M_{\ell-1}(\mathbf{t}_i)$
- So, we train $T_\ell$ on $(\mathbf{t}_1, r_1), \ldots, (\mathbf{t}_n, r_n)$ instead of $(\mathbf{t}_1, y_1), \ldots, (\mathbf{t}_n, y_n)$
- To reduce noise coming from $T_\ell$, we also adapt the weight $\gamma_\ell \in \mathbb{R}$
  - E.g., exponential decay – in each step multiply the weight by some $\alpha \in (0, 1)$

# Illustration



# Using Actual Gradients

- Suppose that $z_i = M_{\ell-1}(\mathbf{t}_i)$ for each example $(\mathbf{t}_i, y_i)$
- We train $T_\ell$ so that each $z'_i = M_\ell(\mathbf{t}_i)$ will be closer than $z_i$ to $y_i$
  - And, consequently, $\text{Loss}(\mathbf{t}_i, y_i, z'_i)$ will be smaller than $\text{Loss}(\mathbf{t}_i, y_i, z_i)$
- We face the usual dilemma:
  - We want learning to be *efficient*, hence, liberal in changes when needed
  - We want to be *careful* not to diverge, hence, conservative when needed
- We use the idea of gradient descent: aggressive when the *impact* of changing $z_i$ is high, conservative when low
- How do we do that? Use the **gradient** of the loss function w.r.t. $z_i$

# Using Actual Gradients: More Formally

- Let $z_i = M_{\ell-1}(\mathbf{t}_i)$ and $r_i = y_i - z_i$
- Like GD, estimate the impact of improving $z_i$ on $\text{Loss}(\mathbf{t}_i, y_i, z_i)$ via differentiation:

$$r'_i := -\frac{\partial \text{Loss}(\mathbf{t}_i, y_i, z)}{\partial z}(z_i)$$

- Now, instead of training $T_\ell$ on $(\mathbf{t}_1, y_1), \ldots, (\mathbf{t}_n, y_n)$ , train it on $(\mathbf{t}_1, r'_1), \ldots, (\mathbf{t}_n, r'_n)$
- If Loss is *least squares*, $(y_i - \hat{y}_i)^2$, then $r'_i = 2r_i$ (simple derivative)
  - Hence, we get the GB learning we previously saw

# XGBoost

- State-of-the-art system for tabular ML, highly scalable [Chen and Guestrin, 2016]
- Advanced modeling and implementation of gradient boosting DTs
- DT $T_\ell$ aims to minimize the regularized loss (with hyper-parameters $\gamma$ and $\lambda$)

$$L(T) = \Big( \sum_{i=1}^{n} \text{Loss}\Big(y_i, M_{\ell-1}(\mathbf{t}_i) + T(\mathbf{t}_i)\Big)\Big) + \gamma|\text{leaves}(T)| + \lambda\|\text{weights}(T)\|_2^2$$

  - leaves($T$) is the set of leaves of $T$ ; weights($T$) are the numbers in the leaves

# XGBoost Tricks

$$L(T) = \Big( \sum_{i=1}^{n} \text{Loss}\Big(y_i, M_{\ell-1}(\mathbf{t}_i) + T(\mathbf{t}_i)\Big)\Big) + \gamma|\text{leaves}(T)| + \lambda\|\text{weights}(T)\|_2^2$$

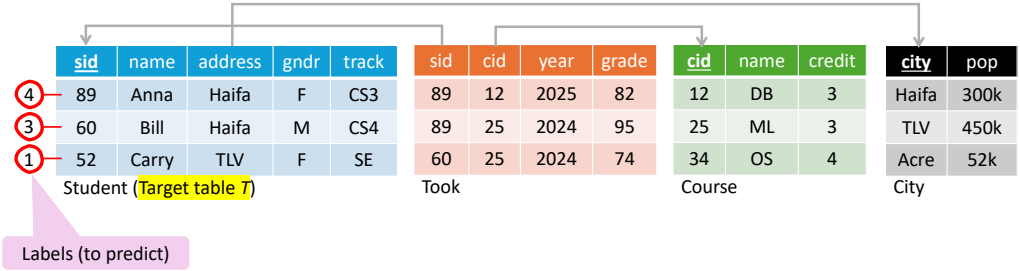
- *2nd Taylor expansion* of $L(T)$ translates to a gain function for node splitting
- *Attribute sampling* in node splitting (from Random Forests)
- Heuristics for *reducing the search space of splitting* thresholds
- Techniques for *batch processing*, *parallelization*, and *disk management*
- And many more

# Other Popular Implementations of Gradient Boosting

- **LightGBM** [Ke et al., 2017]
  - Optimization techniques similarly to XGBoost
  - Some unique behavior, e.g., choosing the *best leaf to expand* rather than all
- **CatBoost** [Prokhorenkova et al., 2018]
  - Target statistics for categorical data
  - To avoid leakage of target statistics: reveal training data incrementally alongside the boosting iteration

# Outline

# Task Example



# Reminder: Relational Schema & Database

Recall — 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)

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 **FK** and tuple $\mathbf{t} \in D(R)$ there is a tuple $\mathbf{s} \in D(S)$ such that $\mathbf{t}[\mathbf{C}] = \mathbf{s}[\text{key}(S)]$

# Setup

- We assume a relational schema $\mathcal{S} = (\mathcal{R}, \text{sig}, \text{key}, \textbf{FK})$
- We will focus on a column prediction task $P : \mathbf{D}_{\mathcal{S}} \times dom(T) \to \mathbb{R}$
  - $\mathbf{D}_{\mathcal{S}}$ is the class of all databases over $\mathcal{S}$
  - $T$ is called a *target table*
- Hence, $P$:
  - Takes as input a database $D$ and a tuple $\mathbf{t} \in D(T)$
  - Outputs a number $P(D, \mathbf{t})$
- For simplicity, we assume that $\text{key}(T)$ consists of a single attribute, denoted id
  - No loss of generality, just convenience

# Feature Table

- By *feature extraction* we refer to the task of mapping a given database $D$ over $\mathcal{S}$ into a table **F** over a tabular schema $(\mathbf{A}, \mathbf{dom})$, so that:
  - **A** includes the id attribute as a key (unique-value) attribute, and
  - $\mathbf{F}[\text{id}] = D(T)[\text{id}]$, that is, the tables **F** and $D(T)$ have the same set of keys
    - Hence, **F** and $D(T)$ have the same number of tuples
- **F** is called a *feature table*
- With **F** defined, we can learn a tabular model for **F** as in the previous part
  - E.g., XGBoost

# SQL Representation

- We will define every feature using an SQL query that creates a single-feature table (or view) $\mathbf{F}_i$ with the signature $(\mathrm{id}, A_i)$
- In the end, we apply *left outer join* to combine all single-feature tables into one big table $\mathbf{F}$ with the signature $(\mathrm{id}, A_1, \ldots, A_n)$

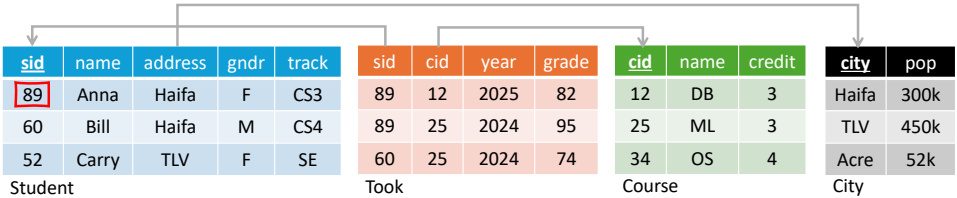
```
SELECT T.id, A_1, ..., A_n FROM
     T LEFT OUTER JOIN F_1 ON T.id = F_1.id
       LEFT OUTER JOIN F_2 ON T.id = F_2.id
     ...
       LEFT OUTER JOIN F_n ON T.id = F_n.id
```

# Recalling Left Outer Join

- *Left outer join* of relations $T$ and $S$, denoted $T \ltimes S$, consists of:
  1. All tuples $(\mathbf{t}, \mathbf{s})$ of $\mathbf{t} \in D(T)$ and $\mathbf{s} \in D(S)$ that satisfy the join condition; and
  2. All tuples $\mathbf{t} \in D(T)$ that join with **none** of the $\mathbf{s}$, padded with nulls
- *How do we handle null in ML?* We impute missing values
  - That is, replace nulls with some fake values from the domain
  - Can be a default value (e.g., average) or estimated based on the existing values ion the tuple
  - (Not covered here)

# Outline

# Example



| sid | name | address | gndr | track |
|---|---|---|---|---|
| 89 | Anna | Haifa | F | CS3 |
| 60 | Bill | Haifa | M | CS4 |
| 52 | Carry | TLV | F | SE |

Student

| sid | cid | year | grade |
|---|---|---|---|
| 89 | 12 | 2025 | 82 |
| 89 | 25 | 2024 | 95 |
| 60 | 25 | 2024 | 74 |

Took

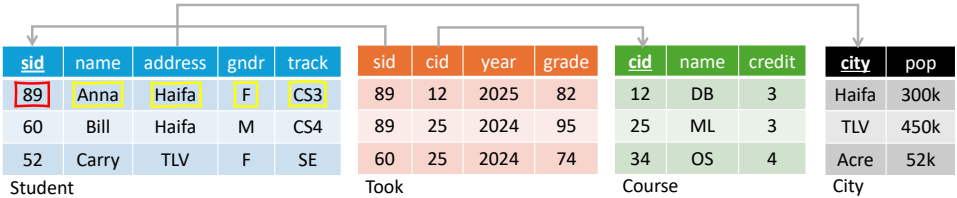
| cid | name | credit |
|---|---|---|
| 12 | DB | 3 |
| 25 | ML | 3 |
| 34 | OS | 4 |

Course

| city | pop |
|---|---|
| Haifa | 300k |
| TLV | 450k |
| Acre | 52k |

City

# Example



| sid | name | address | gndr | track |
|---|---|---|---|---|
| 89 | Anna | Haifa | F | CS3 |
| 60 | Bill | Haifa | M | CS4 |
| 52 | Carry | TLV | F | SE |

Student

| sid | cid | year | grade |
|---|---|---|---|
| 89 | 12 | 2025 | 82 |
| 89 | 25 | 2024 | 95 |
| 60 | 25 | 2024 | 74 |

Took

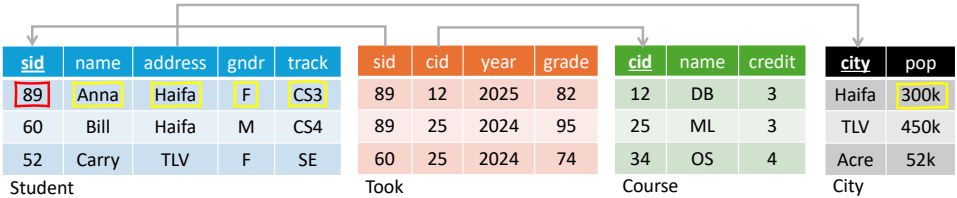
| cid | name | credit |
|---|---|---|
| 12 | DB | 3 |
| 25 | ML | 3 |
| 34 | OS | 4 |

Course

| city | pop |
|---|---|
| Haifa | 300k |
| TLV | 450k |
| Acre | 52k |

City

# Example



**Student**

| sid | name | address | gndr | track |
|---|---|---|---|---|
| 89 | Anna | Haifa | F | CS3 |
| 60 | Bill | Haifa | M | CS4 |
| 52 | Carry | TLV | F | SE |

**Took**

| sid | cid | year | grade |
|---|---|---|---|
| 89 | 12 | 2025 | 82 |
| 89 | 25 | 2024 | 95 |
| 60 | 25 | 2024 | 74 |

**Course**

| cid | name | credit |
|---|---|---|
| 12 | DB | 3 |
| 25 | ML | 3 |
| 34 | OS | 4 |

**City**

| city | pop |
|---|---|
| Haifa | 300k |
| TLV | 450k |
| Acre | 52k |

# Immediate Features

These are the simplest attributes: simply copy every attribute $A \neq \text{id}$ of $T$

- Ignoring the rest of the database

```
SELECT T.id, T.A AS A_i
FROM T
```

# Feature Using Forward References

- We can also copy values from the tables that $D(T)$ references via a foreign key
- Suppose that we have the FK $T[B] \sqsubseteq \text{key}(S)$
- Suppose that $\text{key}(S) = K$
- To each tuple $\mathbf{t} \in D(T)$ and attribute $A$ of $S$, we add as feature the value $\mathbf{s}[A]$ where $\mathbf{s} \in D(S)$ is the referenced tuple
  - That is, $\mathbf{s}[K] = \mathbf{t}[B]$
- In SQL:

```
SELECT T.id, S.A AS A_i
FROM T JOIN S ON T.B = S.K
```

# Chasing More References

- We can also continue to a tuple that $S$ references through some foreign key
- In general, we can use values from a relation $S_n$ such that:
  - $T$ references $S_1$ via $T[B] \sqsubseteq \text{key}(S_1)$
  - $S_1$ references $S_2$ via $S_1[B_1] \sqsubseteq \text{key}(S_2)$
  - $\vdots$
  - $S_{n-1}$ references $S_n$ via $S_{n-1}[B_{n-1}] \sqsubseteq \text{key}(S_n)$
- For example, for $n = 2$ it is the following SQL:

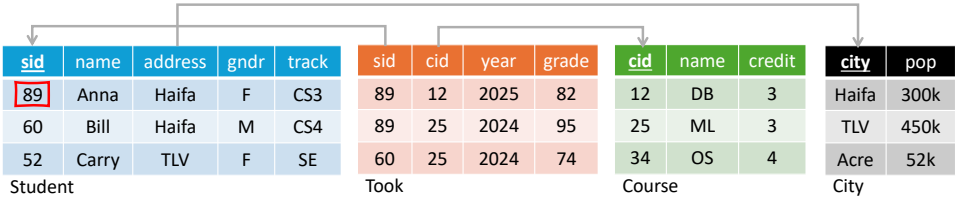
```
SELECT T.id, S_2.A AS A_i
FROM T JOIN S_1 ON T.B = S_1.K_1
       JOIN S_2 ON S_1.B_1 = S_2.K_2
```

# Outline

# Backward References

- What about the backward reference $S[B] \sqsubseteq \text{key}(T)$, pointing **to** $T$?
- Now a target tuple **t** can have multiple (possibly many) **s** tuples referencing it
- . . . but we still need to create *single scalars* from such references!
- To accommodate it, we utilize *aggregate functions* and *aggregate queries*

# Motivating Example



| sid | name | address | gndr | track |
|---|---|---|---|---|
| 89 | Anna | Haifa | F | CS3 |
| 60 | Bill | Haifa | M | CS4 |
| 52 | Carry | TLV | F | SE |

Student

| sid | cid | year | grade |
|---|---|---|---|
| 89 | 12 | 2025 | 82 |
| 89 | 25 | 2024 | 95 |
| 60 | 25 | 2024 | 74 |

Took

| cid | name | credit |
|---|---|---|
| 12 | DB | 3 |
| 25 | ML | 3 |
| 34 | OS | 4 |

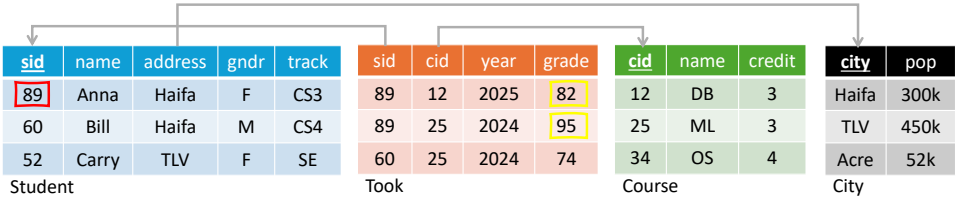
Course

| city | pop |
|---|---|
| Haifa | 300k |
| TLV | 450k |
| Acre | 52k |

City

Example feature: the *average grade* of the student

# Motivating Example



**Student**

| sid | name | address | gndr | track |
|---|---|---|---|---|
| 89 | Anna | Haifa | F | CS3 |
| 60 | Bill | Haifa | M | CS4 |
| 52 | Carry | TLV | F | SE |

**Took**

| sid | cid | year | grade |
|---|---|---|---|
| 89 | 12 | 2025 | 82 |
| 89 | 25 | 2024 | 95 |
| 60 | 25 | 2024 | 74 |

**Course**

| cid | name | credit |
|---|---|---|
| 12 | DB | 3 |
| 25 | ML | 3 |
| 34 | OS | 4 |

**City**

| city | pop |
|---|---|
| Haifa | 300k |
| TLV | 450k |
| Acre | 52k |

Example feature: the *average grade* of the student

# Aggregate Functions

- An *aggregate function* Agg takes as input a bag of tuples and outputs a scalar
- There are a few common aggregate functions: Min, Max, Sum, Avg, Median, Count, Count-Distinct
  - All functions assume a single column, except for Count and Count-Distinct
  - Sum, Avg, and Median assume a numeric domain
  - When the result is undefined (e.g., Avg($\emptyset$)), the result is *null*
  - Min and Max assume that the domain is linearly ordered (e.g., numbers by the natural order, strings by lexicographic order)
- Every database brand supports a list of built-in aggregate functions beyond the SQL standard; see, e.g.,
  - `https://www.postgresql.org/docs/9.5/functions-aggregate.html`
  - `https://dev.mysql.com/doc/refman/8.4/en/aggregate-functions.html`

# Example from MySQL Documentation

| Name | Description |
|---|---|
| AVG() | Return the average value of the argument |
| BIT_AND() | Return bitwise AND |
| BIT_OR() | Return bitwise OR |
| BIT_XOR() | Return bitwise XOR |
| COUNT() | Return a count of the number of rows returned |
| COUNT(DISTINCT) | Return the count of a number of different values |
| GROUP_CONCAT() | Return a concatenated string |
| JSON_ARRAYAGG() | Return result set as a single JSON array |
| JSON_OBJECTAGG() | Return result set as a single JSON object |
| MAX() | Return the maximum value |
| MIN() | Return the minimum value |
| STD() | Return the population standard deviation |
| STDDEV() | Return the population standard deviation |
| STDDEV_POP() | Return the population standard deviation |
| STDDEV_SAMP() | Return the sample standard deviation |
| SUM() | Return the sum |
| VAR_POP() | Return the population standard variance |
| VAR_SAMP() | Return the sample variance |
| VARIANCE() | Return the population standard variance |

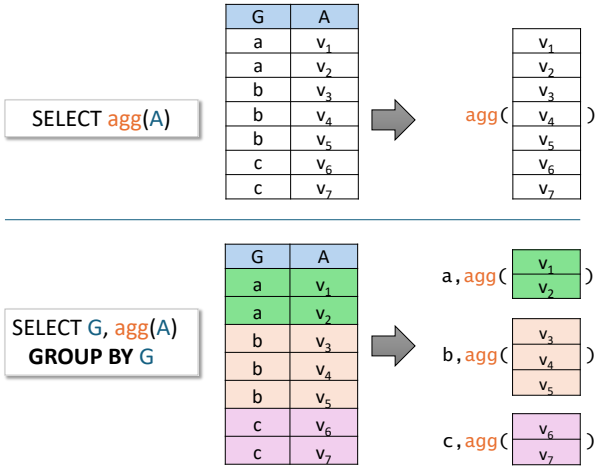
# Aggregate Queries

We create features from *aggregate queries* of the following form:

```
SELECT G, Agg(A)
FROM [some join that involves attributes G and A]
GROUP BY G
```

- $G$ is a *grouping attribute*
  - In our case, $G$ will be $T.\mathrm{id}$ (the identifier of the target row)
- Semantics:
  1. Apply the join
  2. Group together tuples with the same $G$ value
  3. For each group $g$, apply Agg to its $A$ column $C_g$ and return the tuple $(g, \mathrm{Agg}(C_g))$

# Grouping-Aggregate Queries



# Using Backward References

Considering $S[B] \sqsubseteq \text{key}(T)$:

- Choose:
  - An aggregation function Agg
  - An aggregrated attribute $A$ of $S$
- For each tuple $\mathbf{t}$ in $D(T)$, aggregate the values $\mathbf{s}[A]$ of the tuples $\mathbf{s}$ in $D(S)$ that reference $\mathbf{t}$

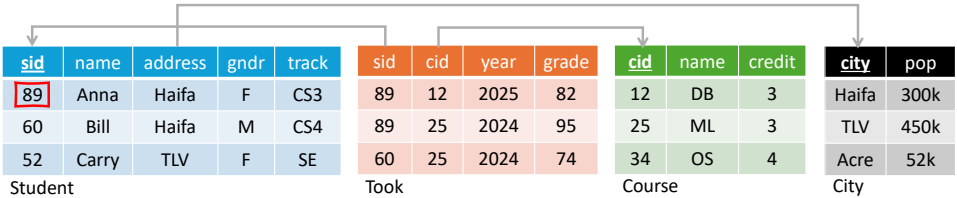
```
SELECT T.id, Agg(S.A) AS A_i
FROM T JOIN S ON T.K = S.B
GROUP BY T.id
```

# Chaining References

- In fact, with aggregation, we can extract features from arbitrary joins
- For example, we can follow a chain of multiple references:

```
SELECT T.id, Agg(S_n.A) AS A_i
FROM T JOIN S_1 ON T.B = S_1.E_1
       JOIN S_2 ON S_1.B_1 = S_2.E_2
       ...
       JOIN S_n ON S_{n-1}.B_{n-1} = S_n.E_n
GROUP BY T.id
```

# Example



**Student**

| sid | name | address | gndr | track |
|---|---|---|---|---|
| 89 | Anna | Haifa | F | CS3 |
| 60 | Bill | Haifa | M | CS4 |
| 52 | Carry | TLV | F | SE |

**Took**

| sid | cid | year | grade |
|---|---|---|---|
| 89 | 12 | 2025 | 82 |
| 89 | 25 | 2024 | 95 |
| 60 | 25 | 2024 | 74 |

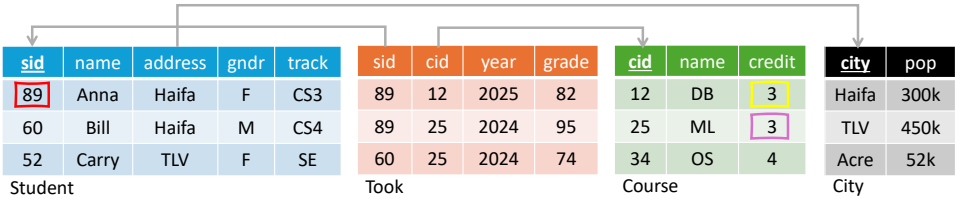
**Course**

| cid | name | credit |
|---|---|---|
| 12 | DB | 3 |
| 25 | ML | 3 |
| 34 | OS | 4 |

**City**

| city | pop |
|---|---|
| Haifa | 300k |
| TLV | 450k |
| Acre | 52k |

# Example



| sid | name | address | gndr | track |
|---|---|---|---|---|
| 89 | Anna | Haifa | F | CS3 |
| 60 | Bill | Haifa | M | CS4 |
| 52 | Carry | TLV | F | SE |

Student

| sid | cid | year | grade |
|---|---|---|---|
| 89 | 12 | 2025 | 82 |
| 89 | 25 | 2024 | 95 |
| 60 | 25 | 2024 | 74 |

Took

| cid | name | credit |
|---|---|---|
| 12 | DB | 3 |
| 25 | ML | 3 |
| 34 | OS | 4 |

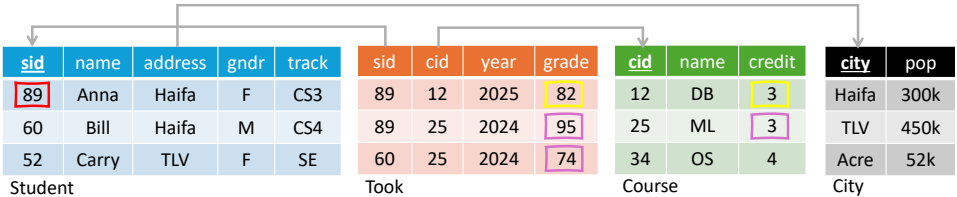
Course

| city | pop |
|---|---|
| Haifa | 300k |
| TLV | 450k |
| Acre | 52k |

City

- *Sum of credits* of the student

# Example



**Student**

| sid | name | address | gndr | track |
|---|---|---|---|---|
| 89 | Anna | Haifa | F | CS3 |
| 60 | Bill | Haifa | M | CS4 |
| 52 | Carry | TLV | F | SE |

**Took**

| sid | cid | year | grade |
|---|---|---|---|
| 89 | 12 | 2025 | 82 |
| 89 | 25 | 2024 | 95 |
| 60 | 25 | 2024 | 74 |

**Course**

| cid | name | credit |
|---|---|---|
| 12 | DB | 3 |
| 25 | ML | 3 |
| 34 | OS | 4 |

**City**

| city | pop |
|---|---|
| Haifa | 300k |
| TLV | 450k |
| Acre | 52k |

- *Sum of credits* of the student
- *Average of all grades* in the courses that the student took

# Counting Join Matches

- A well-studied class of features involves testing for, or counting, joins of different connection patterns (a.k.a. *conjunctive queries*) [Kimelfeld and Ré, 2018]
- For example, the following query counts directed triangles that start from a node in the graph described by the database with the relations $T(\mathrm{id})$ and $E(\mathrm{src}, \mathrm{tgt})$
  - Both src and tgt are foreign-key references to id

```
SELECT T.id, Count(*) AS A_i
FROM T JOIN E AS E1 ON T.id = E1.src
       JOIN E AS E2 ON E1.tgt = E2.src
       JOIN E AS E3 ON E2.tgt = E3.src AND E3.tgt = E1.src
GROUP BY T.id
```

- This can also be done by extracting *graphlet features* from the graph that represents the database, as explained later in the module

# Outline

# The ACORA Algorithm

- The **ACORA**[1] system [Perlich and Provost, 2006] constructs features automatically from a relational database
- Building many join expressions gradually
  - Not only foreign-key references, but rather arbitrary equalities between attributes
  - Deploys a routine to determine when two tables are *joinable* on a given attribute
- Deploys a collection of aggregate functions
  - Checks whether a given aggregate function makes sense for a given attribute
- Generates many features, many likely to be useless; deploys *feature selection*
  - We discuss it later

[1] Stands for Automated Construction of Relational Attributes

# ACORA Pseudo Code

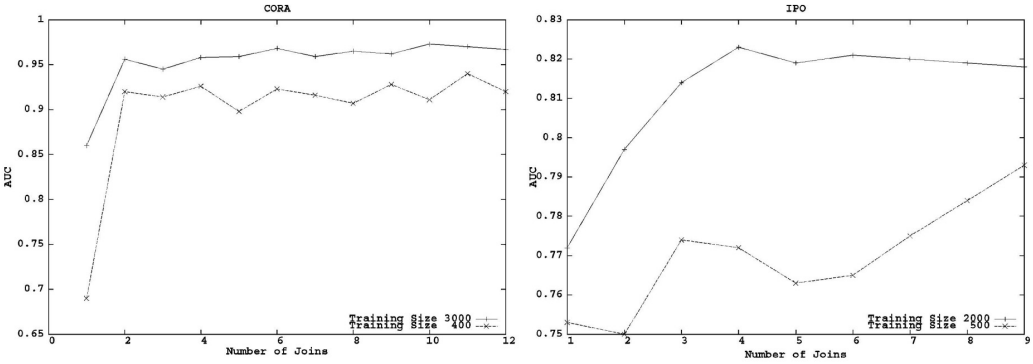
**ACORA Algorithm**

**Input:** The domain specification (tables, attributes, types, and an equality relation over types), and a database $RDB$ including a target table $T$ with labeled training objects $t$ and unlabeled test cases.

1. Read specification and build domain graph $\mathcal{G}$
2. Initialize breadth-first list $\mathcal{L}$ with target table: $\mathcal{L} = \{T\}$
3. Initialize feature table $F$ =non-identifier attributes($T$)
4. Loop
5. $Q$ = First($\mathcal{L}$)
6. Foreach table $R$ in $RDB$ related to $Q$ in $\mathcal{G}$ through some identifiers $(Q.l, R.k)$
7. $J$ = Join $R$ and $Q$ under the condition $R.k$=$Q.l$
8. Foreach attribute $R.j, j \neq k$ (Section 3.3)
9. Foreach target observation $t$
10. Foreach applicable aggregation operator $\mathcal{A}$
11. Construct $\mathcal{A}(\mathcal{B}_{R.j}(t))$ where $\mathcal{B}_{R.j}(t)$ is the bag of values of attribute $R.j$ related to target entity $t$.
12. End Foreach
13. Append aggregates $\mathcal{A}$ as attributes to $F$
14. End Foreach
15. Append $J$ to queue $\mathcal{L}$
16. End Foreach
17. if (stopping criterion) GOTO Select Features
18. End Foreach
19. End Loop
20. Select Features $SF$ from $F$
21. Build propositional model from $SF$

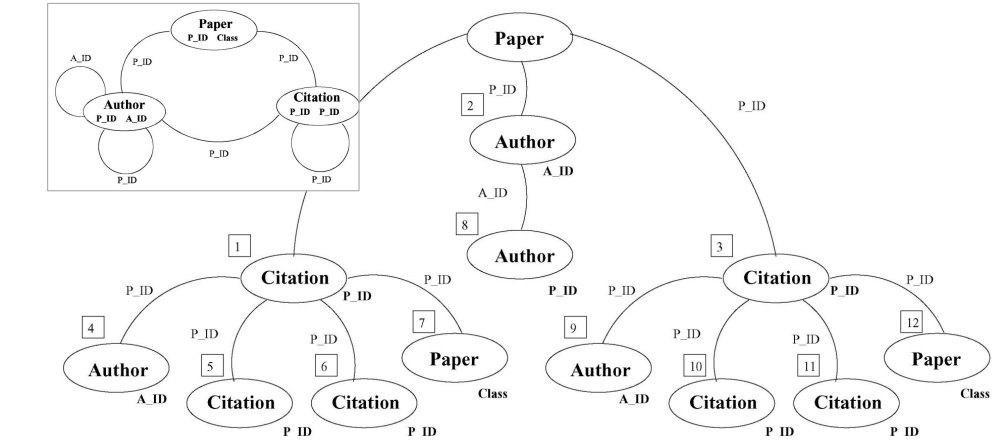
[Perlich and Provost, 2006]

# Few Joins Suffice




[Perlich and Provost, 2006]

# Join Search on the CORA (Citation) Dataset




[Perlich and Provost, 2006]

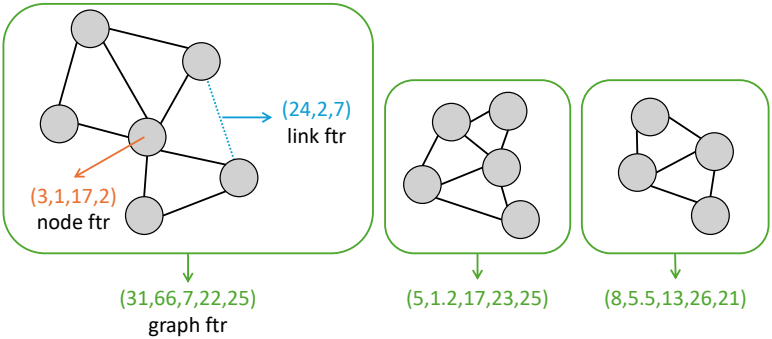
# Techniques for Feature Selection

**Feature selection** is the task of choosing a small subset of the columns of a feature table that should suffice for (and even improve) the prediction task at hand

- *Filter* methods
  - Test each column individually, measure its contribution to accuracy, leave the top-$k$
  - Examples: Information theory w.r.t. training target (e.g., mutual info / info gain), statistical correlation with the training target (e.g., Fisher's, Pearson's, ANOVA)
- *Wrapper* methods
  - Re-train a model iteratively to test (with cross validation) the benefit of features: forward selection (start from nothing and extend), backward elimination (start with all and reduce), exhaustive (check every subset)
- *Embedded* methods
  - Train a model and rely on its regularization capabilities to eliminate useless features
- This is related to, but not the same as, *dimensionality reduction*
  - Examples: *PCA* on the vector matrix, *autoencoder* via a neural network

# Outline

# Illustration: Features from Graphs



# Types of Features from Graphs

- In this part, we consider a graph $G = (V, E)$
- We discuss a collection of generic node features of 3 kinds:
  - *Node features* map every node in $V$ to a feature space
  - *Link features* map every pair of nodes in $V \times V$ to a feature space
  - *Graph features* map the whole graph $G$ to a feature space
- For simplicity, we assume that $G$ is *undirected*
  - Usually, the ideas extend easily to directed graphs, but more details are needed to distinguish between incoming and outgoing edges
- We focus on *structural features* that are defined only based on the graph structure, without accounting for labels or any other external information

# Outline

# Centrality Measures

- *Centrality measures* are scalar features that aim to quantify the importance of nodes $v \in V$ to the other nodes
- Simplest is the node's *degree* — the number of nodes it connects to:

$$\deg(v) := \left|\{u \mid \{u, v\} \in E\}\right|$$

- Next, we review more interesting examples

# Betweenness

*Betweenness centrality* measures how important a node is in connecting others:

$$\mathrm{bw}(v) := \sum_{u \neq v \neq u'} \frac{|\mathrm{SP}(u, v, u')|}{|\mathrm{SP}(u, u')|}$$

where $\mathrm{SP}(u, u')$ is the set of shortest paths from $u$ to $u'$, and $\mathrm{SP}(u, v, u')$ is the subset of such paths that visit $v$

# Closeness

*Closeness centrality* aims to measure how close a node is to other nodes:

$$\mathrm{cl}(v) := \left( \sum_{u \neq v} \mathrm{distance}(v, u) \right)^{-1}$$

where $\mathrm{distance}(v, u)$ is the length of the shortest path between $v$ and $u$

# Eigenvector Centrality

- Idea:
  - We wish to assign to each node $v$ a centrality score $s(v)$, stating how "important" $v$ is in the network
  - Suppose that we require the normalization $\sum_v s(v) = 1$, and the importance of a node is determined by the sum of importances of its neighbors:

$$s(v) = \frac{1}{\lambda} \sum_{\{u,v\} \in E} s(u)$$

- Let $V = \{v_1, \ldots, v_n\}$, denote $\mathbf{s} = (s_1, \ldots, s_n)^\top$ where $s_i = s(v_i)$
- Let $\mathbf{A}$ be the adjacency matrix
- Then $\mathbf{s}$ and $\lambda$ correspond to an eigenvector and eigenvalue: $\mathbf{As} = \lambda \mathbf{s}$
- *Which $\lambda$ and $\mathbf{s}$ should we use?*

# The Perron-Frobenius Theorem

Can we always find such $\lambda$ and $\mathbf{s}$? **Yes!** And they are essentially unique!

**Theorem (Perron-Frobenius)**

*Let $\mathbf{A}$ be a matrix with nonnegative entries. There is a (unique) real eigenvalue $\lambda \geq 0$ with nonnegative (left and right) eigenvectors, such that $\lambda \geq |\lambda'|$ for every other eigenvalue. Moreover, if $\mathbf{A}$ is the adjacency matrix of a connected graph, then $\lambda > 0$ and its eigenvector is positive and unique (up to normalization).*

# Centrality via the Perron-Frobenius Theorem

- Hence, the eigenvector of the Perron-Frobenius theorem guarantees a specific scoring function $s(v)$ for connected graphs
- For disconnected graphs, we can take the unique $s$ for every connected component
- We call the entry $s(v)$ the *eigenvector centrality* of the node $v$
- There are known approximation algorithms for the greatest eigenvalue/eigenvector of a matrix, e.g., *power iteration*
- In the case of *directed graphs*, we take the unique Perron-Frobenius eigenvectors of the strongly connected components
  - The strong version of the theorem applies to strongly connected graphs

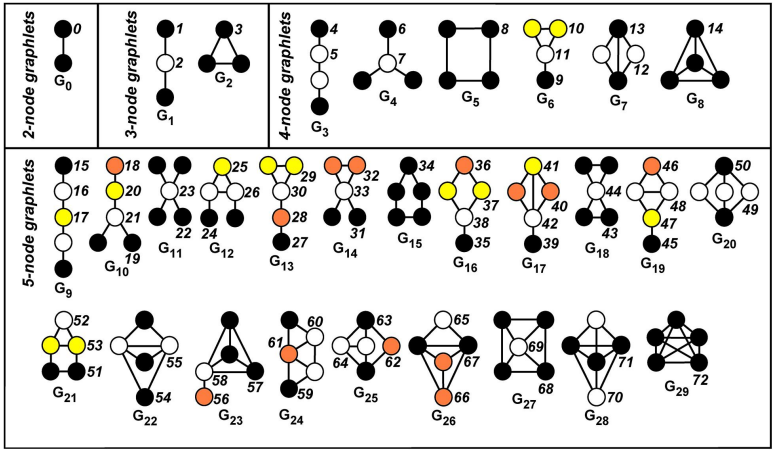
# Orbits (for Graphlet Features)

- *Graphlet features* extract node features based on the subgraph structures that involve the node
- To define these, we need the concept of an *orbit*
- Let $h$ be a graph (directed or undirected)
- An *automorphism* on $h$ is an isomorphism from $h$ to itself
- Call two nodes $v$ and $u$ of $h$ *similar* if there is an automorphism that maps $v$ to $u$
- It is easy to verify that similarity is an equivalence relationship
- Every equivalence class is called an *orbit* of $h$

# Graphlets

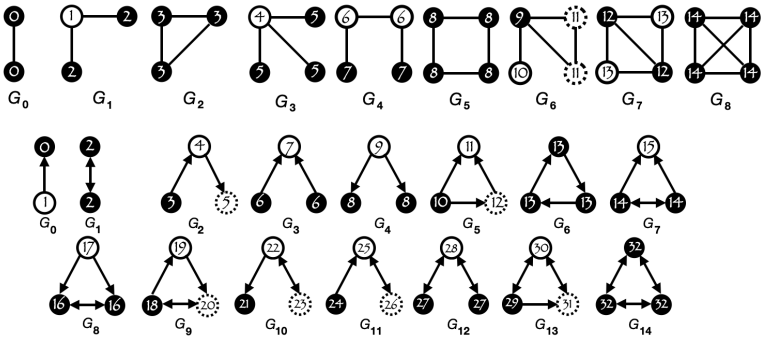
- A *pattern* is a (small) connected labeled graph $h$ with nodes labeled by their orbits
- A *pointed pattern* consists of a pattern $h$ and an orbit $o$ of $h$
- We denote it by $h^{(o)}$
- Let $G$ be a graph and $h^{(o)}$ a pointed pattern
- An $h^{(o)}$-*graphlet* of $G$ has the form $H^{(v)}$ where:
  - $H$ is an *induced subgraph* of $G$
    - That is, a subgraph consisting of a set of nodes and all edges between them
  - $v$ is a node of $G$
  - There is an *isomorphism* $\varphi$ from $H$ to $h$ where $\varphi(v)$ has the orbit $o$

# Graphlets up to 5 Nodes (Undirected)




[Finotelli et al., 2021]

# Graphlets up to 3 Nodes (Undirected & Directed)




[Silva et al., 2023]

# Graphlet Degree Vector (GDV)

- Let $\mathbf{H} = (h_1^{(o_1)}, \ldots, h_p^{(o_p)})$ be a sequence of pointed patterns
  - E.g., all different pointed patterns of 2-5 nodes
- Given a graph $G$ with a node $v$, the GDV of $v$ w.r.t. $\mathbf{H}$ is the $p$-length vector $(n_1, \ldots, n_p)$ where

$$n_i := |\{H \mid \ H^{(v)} \text{ is an } h_i^{(o_i)}\text{- graphlet of } G\}|$$

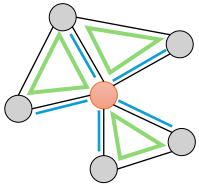
- For later use, we denote $n_i$ as $\text{GDV}_i(v)$
- This generalizes the usual *degree* definition, which is the special case where $p = 1$ and $h_1$ is an edge
- Another example is the *clustering coefficient* $\rightarrow$

# Clustering Coefficient

- The *clustering coefficient* of a node measures how connected its immediate neighborhood is (e.g., how close its friends are)
- Formally, it is the fraction of connected pairs among the node's neighbors:

$$\mathrm{cc}(v) = \frac{\left|\{\{u, u'\} \in E \mid u, u' \in N_v\}\right|}{\binom{|N_v|}{2}}$$

- Can be computed from the counts of the edge and triangle graphlets



# Weisfeiler-Lehman as Node Features

- Consider $k$-steps of 1-WL on a graph $G = (V, E)$:

$\underline{\text{WL}(V, E)}$:

1. Initialize $\kappa_0(v) = 0$ for all $v \in V$
2. For all $i = 1, \ldots, k$, for all $v \in V$ do:
   $$\kappa_i(v) = \textsc{Hash}\big(\kappa_{i-1}(v), \{\!\{\kappa_{i-1}(u) \mid u \in N(v)\}\!\}\big)$$

- We can obtain from WL node features by simply writing down the sequence $(\kappa_1(v), \ldots, \kappa_k(v))$
  - You can further apply 1-hot encoding to obtain a bit vector

# Summary: Node Features

- Centrality:
  - Degree
  - Betweenness
  - Closeness
  - Eigenvector centrality
- Graphlets
- WL

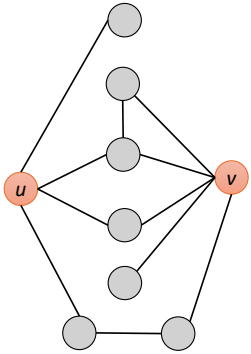
# Outline

# Link Features

- We will look at several examples $F(u, v)$ of features for a pair $u, v$ of nodes
- Note: $u$ and $v$ are **not** necessarily connected by an edge
  - In fact, an important use case is that of predicting missing links
- As usual, we give several features that can be used together (e.g., via concatenation to the vector feature)
- For a start, useful features for the link are the node features for its endspoints:

$$\text{nodeftrs}(u, v) := F(u) \oplus F(v)$$

(where $\oplus$ denotes the vector concatenation operation)



# Features Based on Common Neighbors

- Useful for link prediction in social networks
- Number of common neighbors: $C(u, v) := |N(v) \cap N(u)|$
  - Can be normalized by the total number of neighbors to penalize highly connected nodes, e.g., the *Jaccard coefficient*: $|N(v) \cap N(u)|/|N(v) \cup N(u)|$
- Adamic–Adar index:

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

  - Idea: # common neighbors, but the contribution of each is scaled down by its degree—penalize if the common neighbor has many neighbors regardless of $u$ and $v$

# Mere Distance

- Distance: $D(u, v) := \text{distance}(u, v)$
- That is, the length of the shortest path between $u$ and $v$
- Limitations:
  - What if we want to distinguish between *many* vs. *few* shortest paths?
  - And what about paths slightly longer than the shortest path?
  - Addressed by the *Katz centrality index* $\rightarrow$

# The Katz Centrality Index

- Katz centrality: $\text{Katz}(u, v) := \sum_{\ell=1}^{\infty} \beta^{\ell} |\text{paths}_{\ell}(u, v)|$ where:
  - $\text{paths}_{\ell}(u, v)$ is the set of paths of length $\ell$ between $u$ and $v$
  - $0 < \beta < 1$
- Measures the influence of an actor within a social network [Katz, 1953]
- Idea: the score benefits from many paths, but the impact diminishes with the length of the paths
- *But how can we compute it? Does it even converge?*

# Computing Katz Centrality

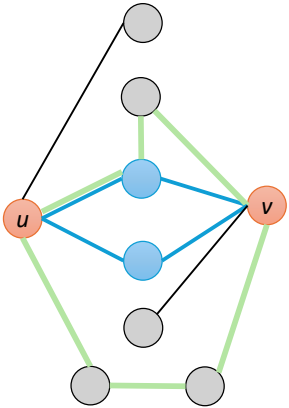
## How to compute Katz?

- Let $\mathbf{A}$ be the adjacency matrix
- In $\mathbf{A}^\ell$, the $(i,j)$ entry is the number of paths of length $\ell$ between $v_i$ and $v_j$
- Hence, Katz $= \sum_{\ell=1}^{\infty} \beta^\ell \mathbf{A}^\ell$
  - Katz stands here for the matrix with the entries Katz$(v_i, v_j)$
- If $\mathbf{M}$ is a square matrix with eigenvalues smaller than one (in absolute value), then $\sum_{\ell=0}^{\infty} \mathbf{M}^\ell = (\mathbf{I} - \mathbf{M})^{-1}$ where $\mathbf{I}$ is the identity matrix; hence (with $\beta\mathbf{A}$ as $\mathbf{M}$):

$$\text{Katz} = (\mathbf{I} - \beta\mathbf{A})^{-1} - \mathbf{I}$$

Hence, for convergence, we require $0 < \beta < 1/\lambda$ where $\lambda$ is the largest eigenvalue of the adjacency matrix $\mathbf{A}$

# Summary: What are the link features here?



# Outline

# Graph Features

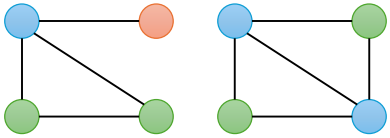
- We discuss some generic techniques of producing graph-level feature vectors
- We again assume undirected graphs, but all techniques generalize naturally to directed and labeled graphs

# Degree Distribution

- A simple numerical vector that describes the graph is a *degree distribution*:

$$F(G) = (d_0(G), d_1(G), \ldots, d_k(G))$$

for some number $i$, where $d_i(G)$ is the number of nodes of $G$ with the degree $i$




- The next features we describe extend and generalize degree distributions

# Graphlet Degree Distribution (GDD)

- The degree distribution states, for every number $j$, the number of nodes that participate in precisely $j$ edges
- Assume $\mathbf{H} = (h_1^{(o_1)}, \ldots, h_p^{(o_p)})$ is a sequence of pointed patterns
- GDD states, for every pointed pattern (not just edge) in $\mathbf{H}$, and number $j$, the number of nodes that participate in precisely $j$ occurrences of the pattern
- For a bound $k$, the GDD of a graph is a $(p \times k)$-matrix that contains, in the entry $(i, \ell)$ the number of nodes $v$ with $\mathrm{GDV}_i(v) = \ell$
  - Recall: $\mathrm{GDV}_i(v) = |\{H \mid H^{(v)} \text{ is an } h_i^{(o_i)}\text{- graphlet of } G\}|$
- Various feature vectors can be obtained from this matrix
  - For example, *vectorization* concatenates the rows, leaving all information intact
  - Other techniques take a weighted average of the degrees of every pattern

# Weisfeiler-Lehman Color Distribution (WL Kernel)

WL$(V, E)$:

1. Initialize $\kappa_0(v) = 0$ for all $v \in V$
2. For all $i = 1, \ldots, k$, for all $v \in V$ do:
$$\kappa_i(v) = \textsc{Hash}\big(\kappa_{i-1}(v), \{\!\{\kappa_{i-1}(u) \mid u \in N(v)\}\!\}\big)$$

- Note: color sets from different iterations are pairwise disjoint
  - Proof by induction on $i$
- Note:
  - The number of nodes who received the color 0 is $|V|$ (since all nodes get it)
  - Each color $c$ at iteration $i = 1$ corresponds to a unique degree of a node
    - Hence, the count distribution for the level-1 colors is the degree distribution
- As a generalization, the 1-WL vector uses a predefined sequence $(c_1, \ldots, c_\ell)$ of colors, and assigns to the graph the vector $(n_1, \ldots, n_\ell)$ where $n_j$ is the number of nodes that were assigned the color $c_j$

# Key Takeaways from Module 2

- Tabular ML is closest to textbook ML once we encode values as numerical vectors
- Boosted variants of tree ensembles provide strong baselines for learning over tabular data
  - A notable example is XGBoost
- In relational databases, we can generate features automatically using compositions of projection, joins, and aggregation
  - A notable example is ACORA
- There are well-established techniques for extracting graph features based only on the graph structure
  - 3 types of features: node features, link features, whole graph features
  - For hetero-graphs, we can combine these with the given node/edge features

# The End

Many thanks for helping with the slides:

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

# References I



Chen, T. and Guestrin, C. (2016).
Xgboost: A scalable tree boosting system.
In *KDD*, pages 785–794. ACM.


Finotelli, P., Piccardi, C., Miglio, E., and Dulio, P. (2021).
A graphlet-based topological characterization of the resting-state network in healthy people.
*Frontiers in neuroscience*, 15:665544.


Guillame-Bert, M., Bruch, S., Mitrichev, P., Mikheev, P., and Pfeifer, J. (2020).
Modeling text with decision forests using categorical-set splits.
*CoRR*, abs/2009.09991.


Hastie, T., Tibshirani, R., Friedman, J., et al. (2009a).
The elements of statistical learning.


Hastie, T., Tibshirani, R., and Friedman, J. H. (2009b).
*The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*.
Springer Series in Statistics. Springer.

# References II



Katz, L. (1953).
A new status index derived from sociometric analysis.
*Psychometrika*, 18(1):39–43.


Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T. (2017).
Lightgbm: A highly efficient gradient boosting decision tree.
In *NIPS*, pages 3146–3154.


Kimelfeld, B. and Ré, C. (2018).
A relational framework for classifier engineering.
*ACM Trans. Database Syst.*, 43(3):11:1–11:36.


Perlich, C. and Provost, F. J. (2006).
Distribution-based aggregation for relational learning with identifier attributes.
*Mach. Learn.*, 62(1-2):65–105.

# References III



Prokhorenkova, L. O., Gusev, G., Vorobev, A., Dorogush, A. V., and Gulin, A. (2018).
Catboost: unbiased boosting with categorical features.
In *NeurIPS*, pages 6639–6649.


Shwartz-Ziv, R. and Armon, A. (2022).
Tabular data: Deep learning is not all you need.
*Inf. Fusion*, 81:84–90.


Silva, M. E., Gaunt, R. E., Ospina-Forero, L., Jay, C., and House, T. (2023).
Comparing directed networks via denoising graphlet distributions.
*Journal of Complex Networks*, 11(2):cnad006.