Stanford CS224W: Traditional Methods for Machine Learning in Graphs

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu
Machine Learning Tasks: Review

- **Node-level** prediction
- **Link-level** prediction
- **Graph-level** prediction
Traditional ML Pipeline

- Design features for nodes/links/graphs
- Obtain features for all training data

![Diagram showing node, link, and graph features]

Node features: \( \in \mathbb{R}^D \)

Link features: \( \in \mathbb{R}^D \)

Graph features: \( \in \mathbb{R}^D \)
Traditional ML Pipeline

- **Train an ML model:**
  - Random forest
  - SVM
  - Neural network, etc.

- **Apply the model:**
  - Given a new node/link/graph, obtain its features and make a prediction

\[
\begin{align*}
x_1 & \rightarrow y_1 \\
\vdots & \\
x_N & \rightarrow y_N \\
x & \rightarrow y
\end{align*}
\]
Using effective features over graphs is the key to achieving good test performance. Traditional ML pipeline uses hand-designed features.

In this lecture, we overview the traditional features for:

- Node-level prediction
- Link-level prediction
- Graph-level prediction

For simplicity, we focus on undirected graphs.
Goal: Make predictions for a set of objects

Design choices:
- **Features**: $d$-dimensional vectors
- **Objects**: Nodes, edges, sets of nodes, entire graphs
- **Objective function**:
  - What task are we aiming to solve?
Machine Learning in Graphs

Machine learning in graphs:

- Given: $G = (V, E)$
- Learn a function: $f : V \rightarrow \mathbb{R}$

How do we learn the function?
Stanford CS224W:
Node-level Tasks and Features
Node-level Tasks

Node classification

ML needs features.
**Goal:** Characterize the structure and position of a node in the network:

- Node degree
- Node centrality
- Clustering coefficient
- Graphlets

Node feature
The degree $k_v$ of node $v$ is the number of edges (neighboring nodes) the node has.

- Treats all neighboring nodes equally.
Node Features: Node Centrality

- Node degree counts the neighboring nodes without capturing their importance.
- Node centrality $c_v$ takes the node importance in a graph into account.
- Different ways to model importance:
  - Eigenvector centrality
  - Betweenness centrality
  - Closeness centrality
  - and many others...
**Node Centrality (1)**

- **Eigenvector centrality:**
  - A node $v$ is important if surrounded by important neighboring nodes $u \in N(v)$.
  - We model the centrality of node $v$ as the sum of the centrality of neighboring nodes:

$$c_v = \frac{1}{\lambda} \sum_{u \in N(v)} c_u$$

$\lambda$ is some positive constant

- Notice that the above equation models centrality in a recursive manner. How do we solve it?
Node Centrality (1)

- **Eigenvector centrality:**
  - Rewrite the recursive equation in the matrix form.

\[
\frac{1}{\lambda} \sum_{u \in N(v)} c_u = \lambda c = Ac
\]

- \(\lambda\) is some positive constant

- We see that centrality is the **eigenvector**!

- The largest eigenvalue \(\lambda_{max}\) is always positive and unique (by Perron-Frobenius Theorem).

- The leading eigenvector \(c_{max}\) is used for centrality.

\[A_{uv} = 1 \text{ if } u \in N(v)\]

- \(A\): Adjacency matrix

\(c\): Centrality vector
Betweenness centrality:

A node is important if it lies on many shortest paths between other nodes.

$$c_v = \sum_{s \neq v \neq t} \frac{\text{#(shortest paths between } s \text{ and } t \text{ that contain } v)}{\text{#(shortest paths between } s \text{ and } t)}$$

Example:

$$c_A = c_B = c_E = 0$$
$$c_C = 3$$
$$c_D = 3$$

(A-C-B, A-C-D, A-C-D-E)

(A-C-D-E, B-D-E, C-D-E)
Closeness centrality:

A node is important if it has small shortest path lengths to all other nodes.

\[ c_v = \frac{1}{\sum_{u \neq v} \text{shortest path length between } u \text{ and } v} \]

Example:

\[ c_A = \frac{1}{(2 + 1 + 2 + 3)} = \frac{1}{8} \]

(A-C-B, A-C, A-C-D, A-C-D-E)

\[ c_D = \frac{1}{(2 + 1 + 1 + 1)} = \frac{1}{5} \]

(D-C-A, D-B, D-C, D-E)
Node Features: Clustering Coefficient

- Measures how connected \( v' \)'s neighboring nodes are:

\[
e_v = \frac{\text{#(edges among neighboring nodes)}}{\binom{k_v}{2}} \in [0,1]
\]

#(node pairs among \( k_v \) neighboring nodes)

- **Examples:**

\[
\begin{align*}
\text{Example 1: } & \quad e_v = 1 \\
\text{Example 2: } & \quad e_v = 0.5 \\
\text{Example 3: } & \quad e_v = 0
\end{align*}
\]
**Observation:** Clustering coefficient counts the \#(triangles) in the ego-network

\[ e_v = 0.5 \]

3 triangles (out of 6 node triplets)

- We can generalize the above by counting \#(pre-specified subgraphs, i.e., **graphlets**).
**Graphlets:** Rooted connected non-isomorphic subgraphs:

- 2-node graphlet: \( G_0 \)
- 3-node graphlets: \( G_1, G_2 \)
- 4-node graphlets: \( G_3, G_4, G_5, G_6, G_7, G_8 \)
- 5-node graphlets: \( G_9, G_{10}, G_{11}, G_{12}, G_{13}, G_{14}, G_{15}, G_{16}, G_{17}, G_{18}, G_{19} \)

There are equations, but not simple. Here are the links showing the actual numbers:

1. Graphlets (only considering connected graphs) [http://oeis.org/A001349]
2. Graphlets (also considering non-connected) [http://oeis.org/A000088]

Generalization to directed graphs: [https://arxiv.org/pdf/1511.01964.pdf]
Node Features: Graphlets

- **Graphlet Degree Vector (GDV)**: Graphlet-base features for nodes
  - **Degree** counts $\#(\text{edges})$ that a node touches
  - **Clustering coefficient** counts $\#(\text{triangles})$ that a node touches.
  - **GDV** counts $\#(\text{graphlets})$ that a node touches
Graphlet Degree Vector (GDV): A count vector of graphlets rooted at a given node.

Example:

Graphlet instances:

GDV of node \( v \): 
\[ a, b, c, d \]
\[ [2, 1, 0, 2] \]
Node Features: Graphlets

- Considering graphlets on 2 to 5 nodes we get:
  - Vector of 73 coordinates is a signature of a node that describes the topology of node's neighborhood.
  - Captures its interconnectivities out to a distance of 4 hops.

- Graphlet degree vector provides a measure of a node’s local network topology:
  - Comparing vectors of two nodes provides a more detailed measure of local topological similarity than node degrees or clustering coefficient.
Node-Level Feature: Summary

- We have introduced different ways to obtain node features.

- They can be categorized as:
  - Importance-based features:
    - Node degree
    - Different node centrality measures
  - Structure-based features:
    - Node degree
    - Clustering coefficient
    - Graphlet count vector
Node-Level Feature: Summary

- **Importance-based features**: capture the importance of a node in a graph
  - **Node degree**: Simply counts the number of neighboring nodes
  - **Node centrality**: Models the importance of neighboring nodes in a graph
    - Different modeling choices: eigenvector centrality, betweenness centrality, closeness centrality

- Useful for predicting influential nodes in a graph
  - **Example**: predicting celebrity users in a social network
Node-Level Feature: Summary

- **Structure-based features**: Capture topological properties of local neighborhood around a node.
  - **Node degree**: Counts the number of neighboring nodes
  - **Clustering coefficient**: Measures how connected neighboring nodes are
  - **Graphlet degree vector**: Counts the occurrences of different graphlets

- **Useful for predicting a particular role a node plays in a graph**:
  - **Example**: Predicting protein functionality in a protein-protein interaction network.
Discussion

Different ways to label nodes of the network:

Node features defined so far would allow to distinguish nodes in the above example

However, the features defined so far would not allow for distinguishing the above node labelling

All results for all tasks are statistically significant. Two rounds of optimization are required. This parameter is set to unity.

DeepWalk have a parameter (approximation in negative sampling [26]. Hence, keeping everything else the same, we switch to negative sampling in DeepWalk which is also the de facto implementation language (C/C++/Python) since it is secondary to the core algorithm. In order to create fair and reproducible comparisons, we use parameters that satisfy all benchmarks optimize using a stochastic gradient descent algorithm. In doing so, we obtain labels from the hallmark gene sets [21] and represent them as binary vectors.

The parameter settings used for all benchmarks are in line with typical settings (p,q) = 4, 1, 0.2581, 0.0681, 22.3, and the optimization is run for a single epoch. (Following prior work [34], we use spectral clustering.)

In the multi-label classification setting, every node is assigned to multiple labels. In our experiments, we focus on the primary label. In order to create a more challenging problem, we use hierarchical softmax, which approximates the softmax probabilities. However, hierarchical softmax is inefficient when compared with negative sampling. This is because it involves matrix factorization, which is computationally intensive. Our experiments stand out to incorporate information from network neighborhoods beyond 2-neighbors.

The learned node feature representations are evaluated in settings that equalize for runtime. In doing so, we are able to compare the performance of different algorithms. Table 2 summarizes the results for multilabel classification on BlogCatalog, PPI (Homo sapiens) and Wikipedia word coocurrence datasets.

Two tasks under consideration:

1. Classification: The network has 3,890 nodes, 76,584 edges and 40 different labels. The labels represent the biological state of proteins in a protein interaction network.
2. Clustering: The network has 10,312 nodes, 333,983 edges and 39 different labels. The labels are obtained from the Hallmark gene sets [21].

The data provided by the bloggers. The network has 10,312 nodes, 184,812 edges and 40 different labels. The labels represent blogger interests inferred through the meta-Catalog, PPI (Homo sapiens) and Wikipedia word coocurrence datasets.

BlogCatalog [44]: This is a network of social relationships where nodes represent bloggers and edges represent comments, email interactions, and blogrolls. The network has 10,312 nodes, 184,812 edges and 40 different labels. The labels are determined by the meta-Catalog, PPI (Homo sapiens) and Wikipedia word coocurrence datasets.

In the multi-label classification setting, every node is assigned to multiple labels. In our experiments, we focus on the primary label. In order to create a more challenging problem, we use hierarchical softmax, which approximates the softmax probabilities. However, hierarchical softmax is inefficient when compared with negative sampling. This is because it involves matrix factorization, which is computationally intensive. Our experiments stand out to incorporate information from network neighborhoods beyond 2-neighbors.

The learned node feature representations are evaluated in settings that equalize for runtime. In doing so, we are able to compare the performance of different algorithms. Table 2 summarizes the results for multilabel classification on BlogCatalog, PPI (Homo sapiens) and Wikipedia word coocurrence datasets.

Two tasks under consideration:

1. Classification: The network has 3,890 nodes, 76,584 edges and 40 different labels. The labels represent the biological state of proteins in a protein interaction network.
2. Clustering: The network has 10,312 nodes, 333,983 edges and 39 different labels. The labels are obtained from the Hallmark gene sets [21].
Stanford CS224W: Link Prediction Task and Features
The task is to predict **new links** based on existing links.

At test time, all node pairs (no existing links) are ranked, and top $K$ node pairs are predicted.

The key is to design features for a pair of nodes.
Two formulations of the link prediction task:

1) Links missing at random:
   - Remove a random set of links and then aim to predict them

2) Links over time:
   - Given $G[t_0, t'_0]$ a graph on edges up to time $t'_0$, output a ranked list $L$ of links (not in $G[t_0, t'_0]$) that are predicted to appear in $G[t_1, t'_1]$

Evaluation:
   - $n = |E_{new}|$: # new edges that appear during the test period $[t_1, t'_1]$
   - Take top $n$ elements of $L$ and count correct edges
Link Prediction via Proximity

- **Methodology:**
  - For each pair of nodes \((x,y)\) compute score \(c(x,y)\)
    - For example, \(c(x,y)\) could be the number of common neighbors of \(x\) and \(y\)
  - Sort pairs \((x,y)\) by the decreasing score \(c(x,y)\)
  - Predict top \(n\) pairs as new links
  - See which of these links actually appear in \(G[t_1, t'_1]\)
Link-Level Features: Overview

- Distance-based feature
- Local neighborhood overlap
- Global neighborhood overlap

Diagram with nodes A, B, C, D, E, F, G, H connected in a graph.
Distance-Based Features

Shortest-path distance between two nodes

- Example:

- However, this does not capture the degree of neighborhood overlap:
  - Node pair $(B, H)$ has 2 shared neighboring nodes, while pairs $(B, E)$ and $(A, B)$ only have 1 such node.

\[
S_{BH} = S_{BE} = S_{AB} = 2 \\
S_{BG} = S_{BF} = 3
\]
Captures # neighboring nodes shared between two nodes \( v_1 \) and \( v_2 \):

- **Common neighbors**: \( |N(v_1) \cap N(v_2)| \)
  - Example: \( |N(A) \cap N(B)| = |\{C\}| = 1 \)

- **Jaccard’s coefficient**: \( \frac{|N(v_1) \cap N(v_2)|}{|N(v_1) \cup N(v_2)|} \)
  - Example: \( \frac{|N(A) \cap N(B)|}{|N(A) \cup N(B)|} = \frac{|\{C\}|}{|\{C,D\}|} = \frac{1}{2} \)

- **Adamic-Adar index**: 
  \[ \sum_{u \in N(v_1) \cap N(v_2)} \frac{1}{\log(k_u)} \]
  - Example: \( \frac{1}{\log(k_C)} = \frac{1}{\log 4} \)
Limitation of local neighborhood features:

- Metric is always zero if the two nodes do not have any neighbors in common.

However, the two nodes may still potentially be connected in the future.

Global neighborhood overlap metrics resolve the limitation by considering the entire graph.
Katz index: count the number of paths of all lengths between a given pair of nodes.

Q: How to compute #paths between two nodes?

Use powers of the graph adjacency matrix!
### Computing #paths between two nodes

- **Recall:** \( A_{uv} = 1 \) if \( u \in N(v) \)
- Let \( P^{(K)}_{uv} = \) #paths of length \( K \) between \( u \) and \( v \)
- We will show \( P^{(K)} = A^k \)
- \( P^{(1)}_{uv} = \) #paths of length 1 (direct neighborhood) between \( u \) and \( v = A_{uv} \)

\[
P^{(1)}_{12} = A_{12}
\]

\[
A = \begin{pmatrix}
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 \\
\end{pmatrix}
\]
How to compute $P_{uv}^{(2)}$?

**Step 1:** Compute #paths of length 1 between each of $u$’s neighbor and $v$

**Step 2:** Sum up these #paths across $u$’s neighbors

$P_{uv}^{(2)} = \sum_i A_{ui} \cdot P_{iv}^{(1)} = \sum_i A_{ui} \cdot A_{iv} = A_{uv}^2$

### Example

<table>
<thead>
<tr>
<th>Node 1’s neighbors</th>
<th>#paths of length 1 between Node 1’s neighbors and Node 2</th>
<th>$P_{12}^{(2)} = A_{12}^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A^2 = \begin{pmatrix} 0 &amp; 1 &amp; 0 &amp; 1 \ 1 &amp; 0 &amp; 0 &amp; 1 \ 0 &amp; 0 &amp; 0 &amp; 1 \ 1 &amp; 1 &amp; 1 &amp; 0 \end{pmatrix}$</td>
<td>$\begin{pmatrix} 0 &amp; 1 &amp; 0 &amp; 1 \ 1 &amp; 0 &amp; 0 &amp; 1 \ 0 &amp; 0 &amp; 0 &amp; 1 \ 1 &amp; 1 &amp; 1 &amp; 0 \end{pmatrix}$</td>
<td>$\begin{pmatrix} 2 &amp; 1 &amp; 1 &amp; 1 \ 1 &amp; 2 &amp; 1 &amp; 1 \ 1 &amp; 1 &amp; 1 &amp; 0 \ 1 &amp; 1 &amp; 0 &amp; 3 \end{pmatrix}$</td>
</tr>
</tbody>
</table>
Katz index: count the number of paths of all lengths between a pair of nodes.

How to compute #paths between two nodes?

Use adjacency matrix powers!

- $A_{uv}$ specifies #paths of length 1 (direct neighborhood) between $u$ and $v$.
- $A_{uv}^2$ specifies #paths of length 2 (neighbor of neighbor) between $u$ and $v$.
- And, $A_{uv}^l$ specifies #paths of length $l$. 
Katz index between $v_1$ and $v_2$ is calculated as

Sum over all path lengths

$$S_{v_1v_2} = \sum_{l=1}^{\infty} \beta^l A^l_{v_1v_2}$$

#paths of length $l$ between $v_1$ and $v_2$

$0 < \beta < 1$: discount factor

Katz index matrix is computed in closed-form:

$$S = \sum_{i=1}^{\infty} \beta^i A^i = (I - \beta A)^{-1} - I,$$

= $\sum_{i=0}^{\infty} \beta^i A^i$

by geometric series of matrices
Distance-based features:
- Uses the shortest path length between two nodes but does not capture how neighborhood overlaps.

Local neighborhood overlap:
- Captures how many neighboring nodes are shared by two nodes.
- Becomes zero when no neighbor nodes are shared.

Global neighborhood overlap:
- Uses global graph structure to score two nodes.
- Katz index counts #paths of all lengths between two nodes.
Stanford CS224W: Graph-Level Features and Graph Kernels
Graph-Level Features

- **Goal:** We want features that characterize the structure of an entire graph.

- **For example:**

```
A -- B -- D -- E
|    |    |    |
C    D    E
|    |    |
|    |    |
|    |    |
H    G    F
```
Kernel methods are widely-used for traditional ML for graph-level prediction.

Idea: Design kernels instead of feature vectors.

A quick introduction to Kernels:

- Kernel $K(G, G') \in \mathbb{R}$ measures similarity b/w data
- Kernel matrix $K = \left(K(G, G')\right)_{G,G'}$ must always be positive semidefinite (i.e., has positive eigenvals)
- There exists a feature representation $\phi(\cdot)$ such that $K(G, G') = \phi(G)^T \phi(G')$
- Once the kernel is defined, off-the-shelf ML model, such as kernel SVM, can be used to make predictions.
Graph Kernels: Measure similarity between two graphs:

- Graphlet Kernel [1]
- Weisfeiler-Lehman Kernel [2]
- Other kernels are also proposed in the literature (beyond the scope of this lecture)
  - Random-walk kernel
  - Shortest-path graph kernel
  - And many more...

Goal: Design graph feature vector $\phi(G)$

Key idea: Bag-of-Words (BoW) for a graph

- Recall: BoW simply uses the word counts as features for documents (no ordering considered).
- Naïve extension to a graph: Regard nodes as words.
- Since both graphs have 4 red nodes, we get the same feature vector for two different graphs...

$$\phi(\begin{array}{c} \text{red nodes} \end{array}) = \phi(\begin{array}{c} \text{red nodes} \end{array})$$
Graph Kernel: Key Idea

What if we use Bag of node degrees?

Deg1: • Deg2: • Deg3: •

\[ \phi(G) = \text{count}(G) = [1, 2, 1] \]

\[ \phi(G) = \text{count}(G) = [0, 2, 2] \]

- Both Graphlet Kernel and Weisfeiler-Lehman (WL) Kernel use Bag-of-* representation of graph, where * is more sophisticated than node degrees!

Obtains different features for different graphs!
Graphlet Features

- **Key idea**: Count the number of different graphlets in a graph.

- **Note**: Definition of graphlets here is slightly different from node-level features.

- The two differences are:
  - Nodes in graphlets here do **not need to be connected** (allows for isolated nodes)
  - The graphlets here are not rooted.
  - Examples in the next slide illustrate this.
Graphlet Features

Let $G_k = (g_1, g_2, \ldots, g_{n_k})$ be a list of graphlets of size $k$.

- For $k = 3$, there are 4 graphlets.
  - $g_1$
  - $g_2$
  - $g_3$
  - $g_4$

- For $k = 4$, there are 11 graphlets.

Shervashidze et al., AISTATS 2011
Graphlet Features

- Given graph $G$, and a graphlet list $G_k = (g_1, g_2, ..., g_{n_k})$, define the graphlet count vector $f_G \in \mathbb{R}^{n_k}$ as

$$(f_G)_i = \#(g_i \subseteq G) \text{ for } i = 1, 2, ..., n_k.$$
Graphlet Features

- Example for \( k = 3 \).

\[
G_f = \begin{pmatrix}
1 \\
3 \\
6 \\
0
\end{pmatrix}^T
\]
Given two graphs, $G$ and $G'$, graphlet kernel is computed as

$$K(G, G') = f_G^T f_{G'}$$

**Problem:** if $G$ and $G'$ have different sizes, that will greatly skew the value.

**Solution:** normalize each feature vector

$$h_G = \frac{f_G}{\text{Sum}(f_G)} \quad K(G, G') = h_G^T h_{G'}$$
Limitations: Counting graphlets is expensive!

- Counting size-$k$ graphlets for a graph with size $n$ by enumeration takes $n^k$.
- This is unavoidable in the worst-case since subgraph isomorphism test (judging whether a graph is a subgraph of another graph) is NP-hard.
- If a graph’s node degree is bounded by $d$, an $O(nd^{k-1})$ algorithm exists to count all the graphlets of size $k$.

Can we design a more efficient graph kernel?
Goal: design an efficient graph feature descriptor $\phi(G)$

Idea: use neighborhood structure to iteratively enrich node vocabulary.

- Generalized version of Bag of node degrees since node degrees are one-hop neighborhood information.

Algorithm to achieve this:

Color refinement
**Color Refinement**

- **Given:** A graph $G$ with a set of nodes $V$.
  - Assign an initial color $c^{(0)}(v)$ to each node $v$.
  - Iteratively refine node colors by
    $$c^{(k+1)}(v) = \text{HASH} \left( \{ c^{(k)}(v), \{ c^{(k)}(u) \}_{u \in N(v)} \} \right),$$
    where \text{HASH} maps different inputs to different colors.
  - After $K$ steps of color refinement, $c^{(K)}(v)$ summarizes the structure of $K$-hop neighborhood.
Example of color refinement given two graphs

- Assign initial colors

- Aggregate neighboring colors
Example of color refinement given two graphs

- Aggregated colors

- Hash aggregated colors

Hash table

<table>
<thead>
<tr>
<th>Color</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,1</td>
<td>2</td>
</tr>
<tr>
<td>1,11</td>
<td>3</td>
</tr>
<tr>
<td>1,111</td>
<td>4</td>
</tr>
<tr>
<td>1,1111</td>
<td>5</td>
</tr>
</tbody>
</table>
Example of color refinement given two graphs

- **Aggregated colors**

- **Hash aggregated colors**
Color Refinement (4)

Example of color refinement given two graphs

- Aggregated colors

- Hash aggregated colors

Hash table:

<table>
<thead>
<tr>
<th>Color</th>
<th>-&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,4</td>
<td>6</td>
</tr>
<tr>
<td>2,5</td>
<td>7</td>
</tr>
<tr>
<td>3,44</td>
<td>8</td>
</tr>
<tr>
<td>3,45</td>
<td>9</td>
</tr>
<tr>
<td>4,245</td>
<td>10</td>
</tr>
<tr>
<td>4,345</td>
<td>11</td>
</tr>
<tr>
<td>5,2244</td>
<td>12</td>
</tr>
<tr>
<td>5,2344</td>
<td>13</td>
</tr>
</tbody>
</table>
After color refinement, WL kernel counts number of nodes with a given color.

$$\phi(\cdot)$$

Colors

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

Counts

$$= [6, 2, 1, 2, 1, 0, 2, 1, 0, 0, 2, 1]$$
The WL kernel value is computed by the inner product of the color count vectors:

\[
K(\mathbf{\phi}(\mathbf{G}_1), \mathbf{\phi}(\mathbf{G}_2)) = \phi(\mathbf{G}_1)^T \phi(\mathbf{G}_2) = 49
\]
WL kernel is **computationally efficient**

- The time complexity for color refinement at each step is linear in #(edges), since it involves aggregating neighboring colors.

- When computing a kernel value, only colors appeared in the two graphs need to be tracked.
  - Thus, #(colors) is at most the total number of nodes.

- Counting colors takes linear-time w.r.t. #(nodes).

- In total, time complexity is **linear in #(edges)**.
Graph-Level Features: Summary

- **Graphlet Kernel**
  - Graph is represented as **Bag-of-graphlets**
  - Computationally expensive

- **Weisfeiler-Lehman Kernel**
  - Apply $K$-step color refinement algorithm to enrich node colors
    - Different colors capture different $K$-hop neighborhood structures
  - Graph is represented as **Bag-of-colors**
  - Computationally efficient
  - Closely related to Graph Neural Networks (as we will see!)
Today’s Summary

- **Traditional ML Pipeline**
  - Hand-crafted feature + ML model

- **Hand-crafted features for graph data**
  - **Node-level:**
    - Node degree, centrality, clustering coefficient, graphlets
  - **Link-level:**
    - Distance-based feature
    - local/global neighborhood overlap
  - **Graph-level:**
    - Graphlet kernel, WL kernel