Machine Learning for Link Prediction

Giacomo Fiumara

Table of contents

Introduction

Classical vs ML Approaches for Link Prediction

- **Classical heuristics** predict links using explicit formulas based only on graph topology.
- Examples include Common Neighbors, Jaccard Coefficient, Adamic-Adar, Preferential Attachment, Katz Index, and SimRank.
- These heuristics are transparent, easy to compute, and do not require training data.

Classical vs ML Approaches for Link Prediction

Limitations of classical heuristics

- Only use structural patterns; cannot combine with node or edge attributes.
- Can't learn complex patterns automatically; model is fixed regardless of data type.
- Performance is strongly dependent on network topology and may be poor on graphs that do not have high clustering or hub structure.

Classical vs ML Approaches for Link Prediction

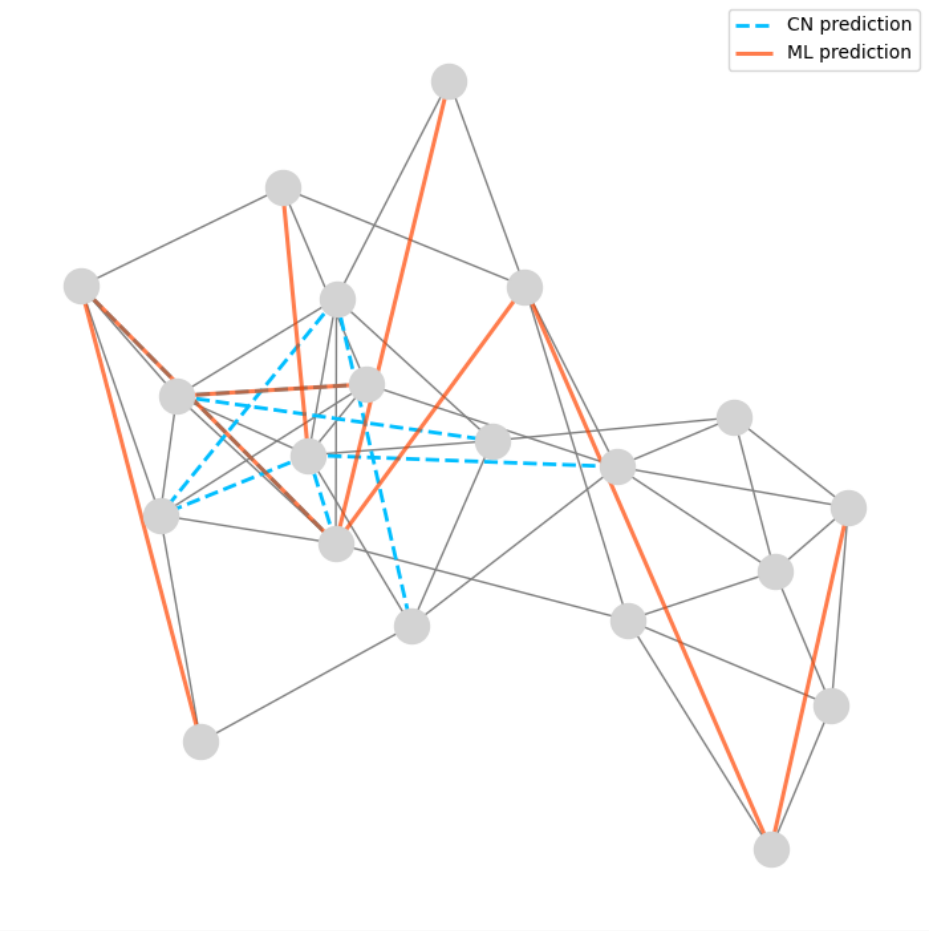
- **Machine Learning (ML) Approaches** leverage multiple features and learn optimal predictive combinations from data.
- ML models can incorporate classical heuristics as features, along with node/edge attributes and embedding-based signals.
- ML allows flexible modeling by training classifiers on labeled data (existing links and random non-existing links).

Classical vs ML Approaches for Link Prediction

Advantages of ML

- Learns from the actual data structure (adapts to domain specifics).
- Can easily incorporate node attributes, temporal information, and embeddings.
- Often achieves better performance than classical methods, especially for complex networks.

Classical vs ML: Visual Comparison Example



Comparing CN predictions and ML classifier output on the same graph.

Defining the ML Task for Link Prediction

- **Goal:** Given a network, predict which pairs of nodes may form (or should form) a link.
- ML approaches formulate link prediction as a supervised classification task.
- Each possible pair of nodes (x, y) is represented by a **feature vector** summarizing available information.

Defining the ML Task for Link Prediction

Common features

- Topological heuristics: Common Neighbors, Adamic-Adar, Jaccard, Preferential Attachment, Katz, etc.
- Node attributes: profile similarity, group membership, categorical features.
- Edge attributes: weights, timestamps, types (if available).
- Node embeddings: vector representations from algorithms like node2vec or DeepWalk.

Defining the ML Task for Link Prediction

- Feature vector for a pair (x, y):

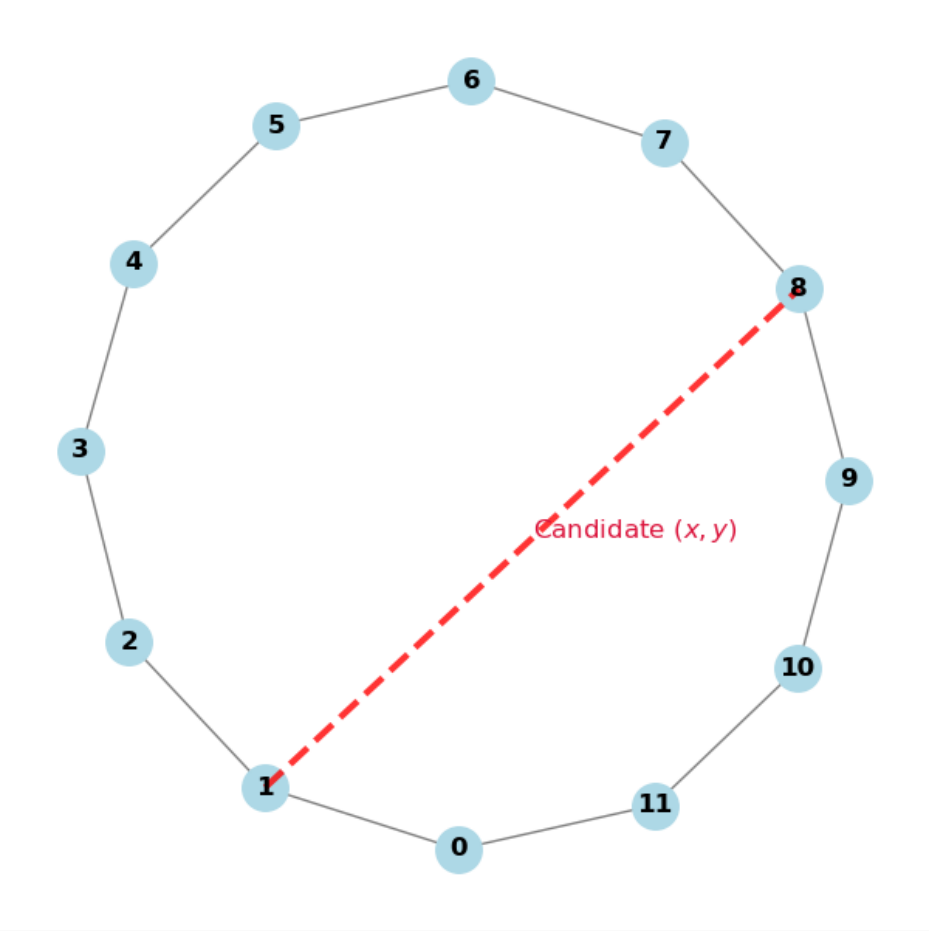
  $$X_{(x,y)} = [s_{CN}(x, y), s_{AA}(x, y), s_J(x, y), \text{attributes}, \text{embeddings}, ...]$$

- Ground truth label for (x, y):

  $$Y_{(x,y)} = \begin{cases} 1 & \text{if edge exists (or is hidden for testing)} \\ 0 & \text{otherwise} \end{cases}$$

- The ML model is trained to classify feature vectors $X_{(x,y)}$ as linked ($Y = 1$) or not linked ($Y = 0$).

ML Link Prediction Task: Feature Representation



For every candidate node pair, construct a feature vector from topological and attribute-based measures.

Train/Test Splitting in Link Prediction

- **Train/test splitting** is necessary to evaluate the ML model's ability to predict missing or future links.
- A common approach: randomly hide a fraction of true edges; these "hidden" links become the positive test set.
- Negative test samples are pairs of nodes not connected in the current network.

Train/Test Splitting in Link Prediction

ML workflow

- **Training set:** All not-hidden edges (positive), and an equal or larger number of random non-edges (negative).
- **Test set:** Hidden edges (positive), plus sampled non-edges (negative).

Train/Test Splitting in Link Prediction

Other splitting strategies

- **Temporal splitting:** Use earlier time steps for training, later for testing (important for evolving networks).
- **Node-induced splitting:** Hold out links associated with a subset of nodes to simulate cold-start prediction.

Train/Test Splitting in Link Prediction

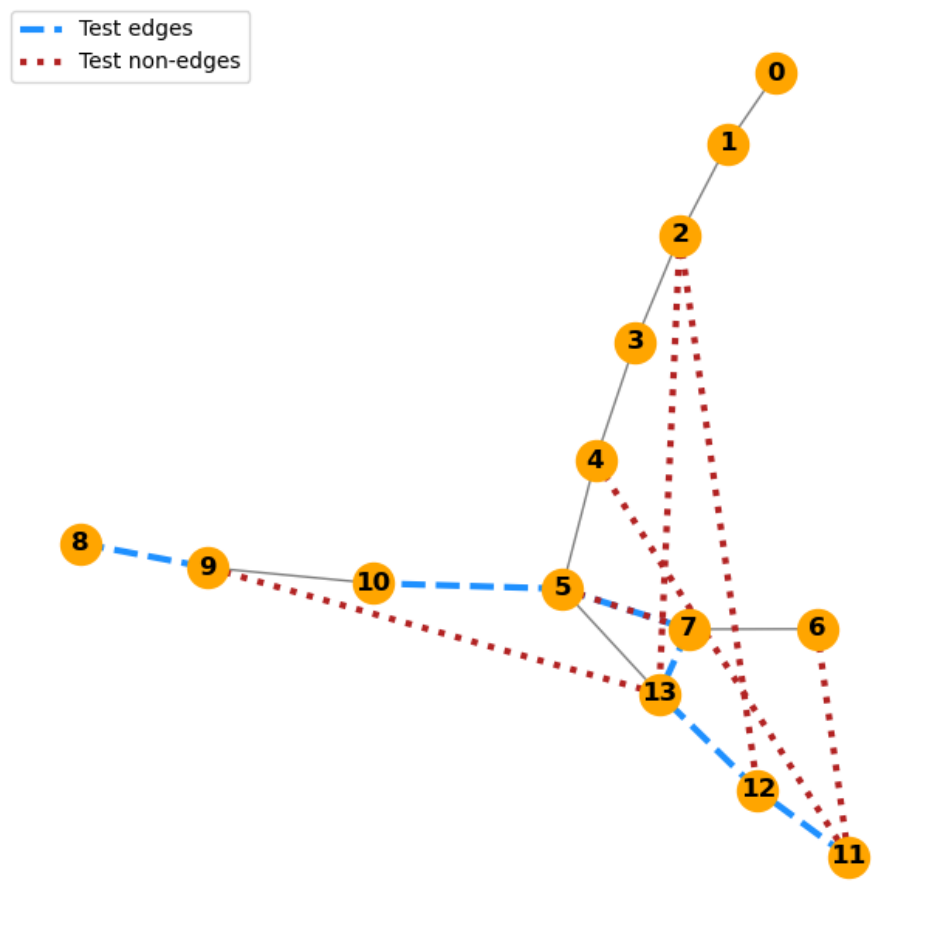
- Example: Let E_{train} be the set of observed edges, E_{test} be the set of hidden edges, N_{train}/N_{test} non-edges:

$$X_{train}, Y_{train} \qquad (E_{train} = 1, N_{train} = 0)$$

$$X_{test}, Y_{test} \qquad (E_{test} = 1, N_{test} = 0)$$

- It is crucial to ensure training and test sets do not overlap and reflect the prediction scenario of interest.

Train/Test Splitting: Visual Example



Original network with held-out test links and candidate non-edges.

Dimensionality Reduction: PCA and t-SNE

Why Dimensionality Reduction?

- High-dimensional data (e.g., node embeddings, gene expression, images) are hard to visualize or interpret directly.
- Dimensionality reduction projects data to a lower-dimensional space (usually 2D or 3D) while preserving key patterns.
- Visualization tools like PCA and t-SNE help:
    - Find clusters, outliers, and structure.
    - Interpret and communicate complex models.

Principal Component Analysis (PCA): Intuition

- **PCA** is a linear technique for dimensionality reduction and data visualization.
- Identifies directions (principal components) in data with the most variance.
- Projects data onto the first k components to obtain a summary with minimal information loss.
- Useful for projecting high-dimensional embeddings to 2D/3D for visualization.

PCA: Mathematical Description

- Given data matrix X (rows are samples, columns are features), center and compute the covariance:

$$S = \frac{1}{n-1} X^T X$$

- Find eigenvectors $\mathbf{w}_1, \mathbf{w}_2, \ldots$ of S: directions with highest variance.

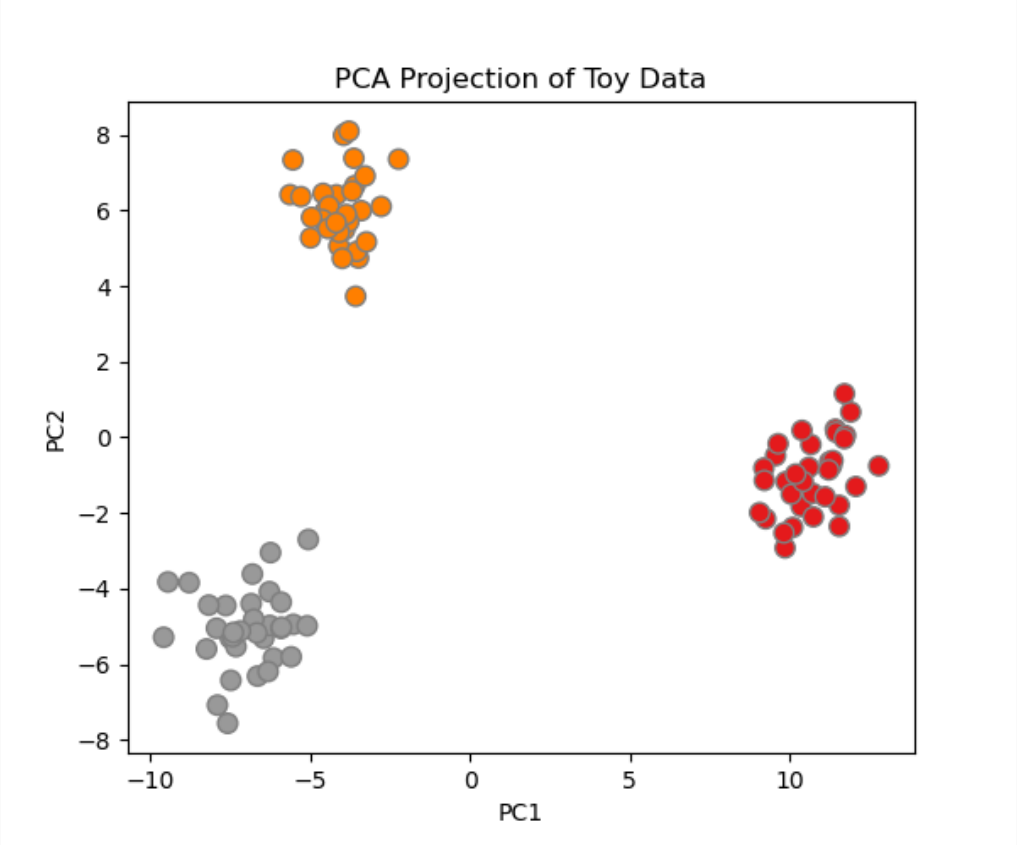
- Project data:

$$Z = X \cdot W_k$$

  where W_k contains the top k eigenvectors as columns.

- Result: each sample is now a k-dimensional vector ($k \ll D$).

PCA: Example on a Toy Dataset

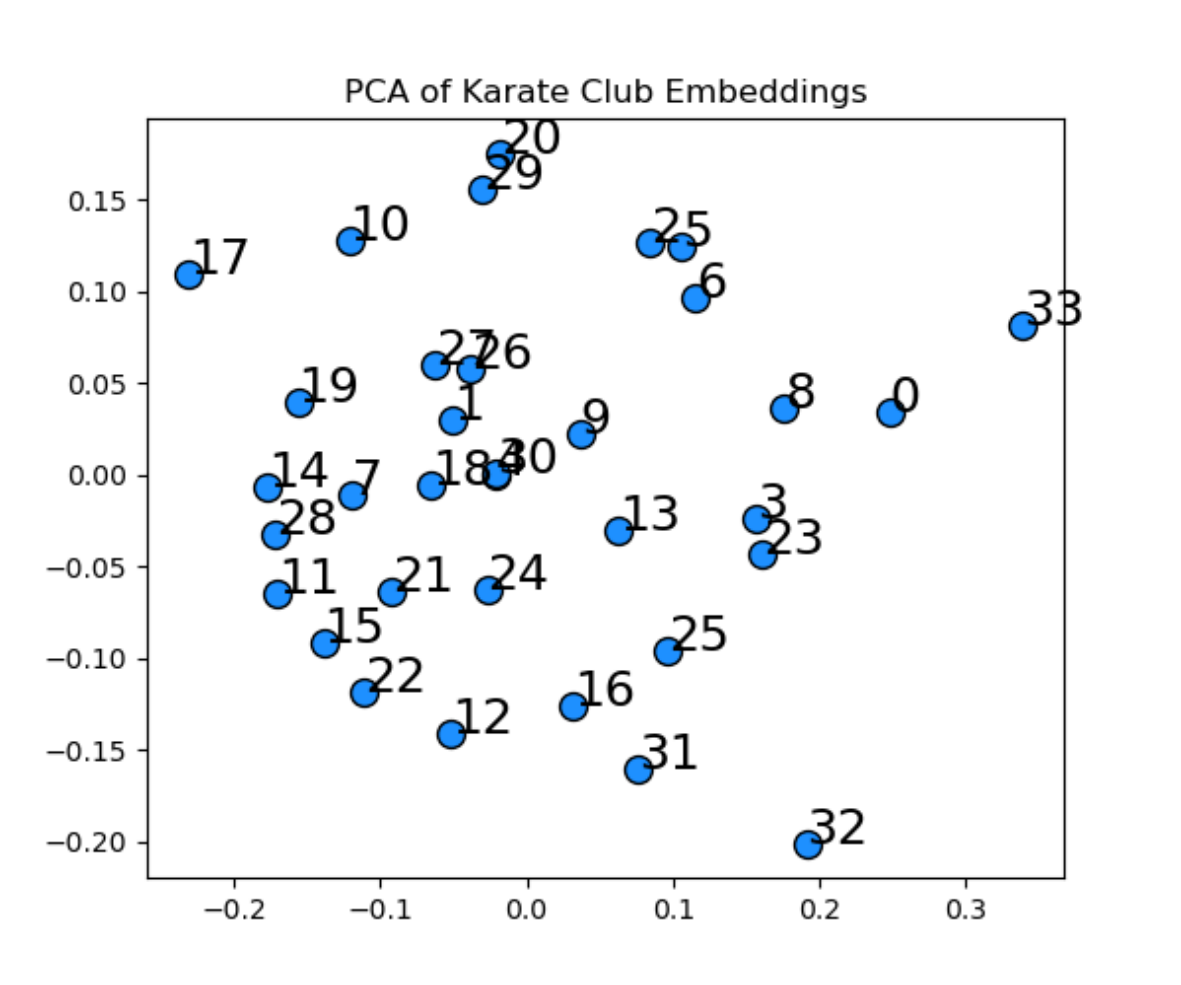


Data projected onto main axes of variance in 2D.

PCA for Embedding Visualization

- PCA can be used to project node embeddings (e.g., from node2vec/DeepWalk) to 2D for direct visualization.
- Global structure (relative distances, spread) is largely preserved.
- Good for revealing clusters and overall geometric relationships in high dimensions.

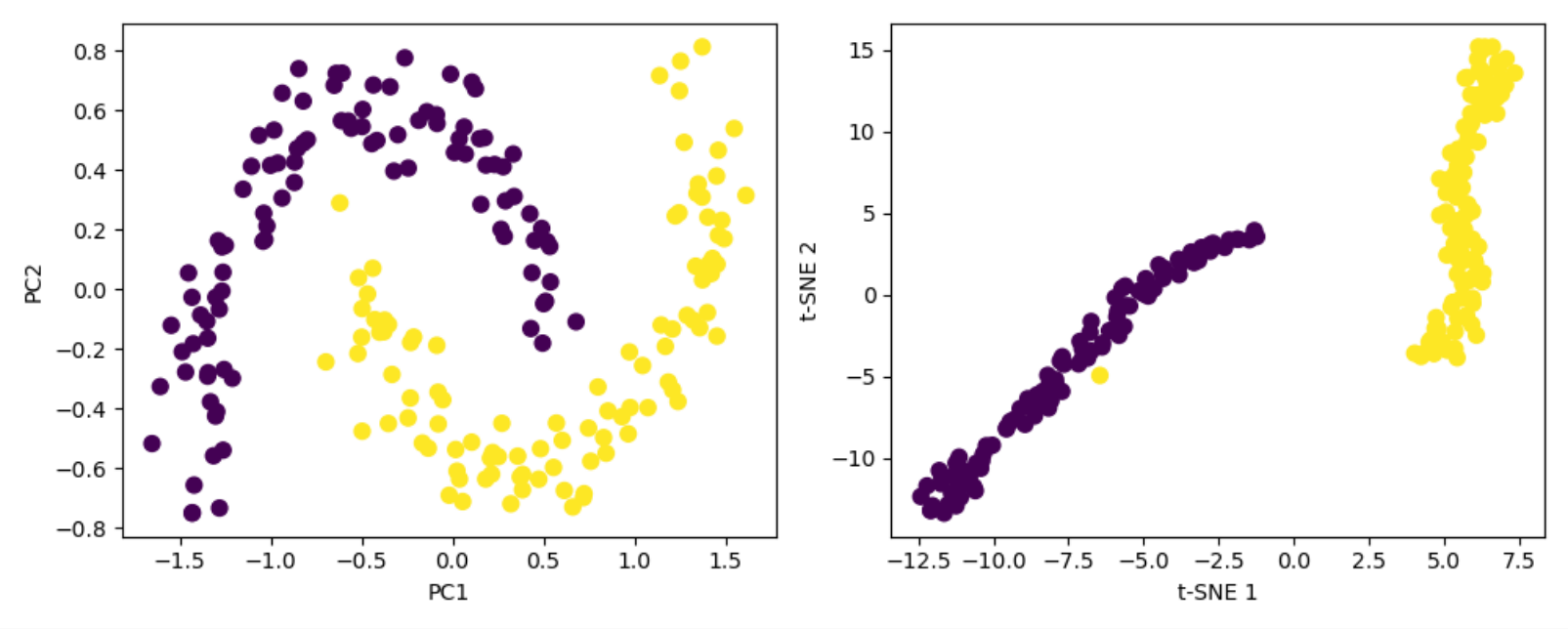
PCA for Embedding Visualization



Why Do We Need t-SNE? Limitations of Linear Methods

- Linear methods like PCA are interpretable and preserve global structure, but they may fail to reveal complex, nonlinear patterns.
- In many real datasets—including network embeddings—clusters and groups are arranged in nonlinear manifolds that PCA cannot separate.
- Example: the "two moons" or "Swiss roll" datasets have clear geometric structure that PCA projects poorly, but nonlinear methods reveal easily.
- **t-SNE** is designed to discover and visualize such nonlinear structure by preserving local neighborhoods rather than global distances.

Example: PCA Fails on Nonlinear Data



t-SNE: The Core Idea

- **t-SNE** stands for **t-Distributed Stochastic Neighbor Embedding**.
- The algorithm models similarity between data points in high-dimensional space and then finds a low-dimensional (2D/3D) embedding that preserves these similarities.
- Key principle: **Points that are neighbors in high dimensions should remain neighbors in low dimensions.**
- Unlike PCA, t-SNE is **nonlinear** and does not preserve global distances—only local structure matters.

t-SNE Step 1: High-Dimensional Similarities

- In the original high-dimensional space, compute pairwise similarities between points using a **Gaussian kernel**:

$$p_{j|i} = \frac{\exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma_i^2}\right)}{\sum_{k \neq i} \exp\left(-\frac{\|x_i - x_k\|^2}{2\sigma_i^2}\right)}$$

- Here, $p_{j|i}$ is the probability that point i "picks" point j as a neighbor in high dimensions.
- σ_i is a bandwidth parameter (often set via a target "perplexity"—roughly the expected number of neighbors).
- Smaller distances yield higher $p_{j|i}$; distant points have near-zero probability.

Perplexity: Controlling Neighborhood Size

- **Perplexity** is a hyperparameter (e.g., 5, 30, 50) that controls the expected size of each point's neighborhood.
- For each point i, we set σ_i such that the entropy of the conditional distribution $p_{j|i}$ equals log(perplexity).
- Typical range: 5–50. Higher perplexity considers larger neighborhoods (more global structure); lower perplexity focuses on very local structure.
- Choose perplexity based on dataset size: a rule of thumb is perplexity $\approx \sqrt{n}$, where n is the number of samples.

t-SNE Step 2: Low-Dimensional Similarities

- In the low-dimensional space (2D or 3D), place points $y_1, \ldots, y_n$ and define similarities using a **heavy-tailed Student-t distribution**:

$$q_{j|i} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq i}(1 + \|y_i - y_k\|^2)^{-1}}$$

- Why Student-t? It has heavier tails than a Gaussian, which helps avoid overcrowding of points in 2D (the "crowding problem").
- The form $(1 + \|y_i - y_j\|^2)^{-1}$ means even moderately distant points in 2D have non-negligible similarity.

t-SNE Step 3: Minimize KL Divergence

- t-SNE finds $y_1, \ldots, y_n$ that minimize the Kullback-Leibler (KL) divergence between high-dimensional and low-dimensional similarity distributions:

$$\mathrm{KL}(P \| Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

- Here,

$$p_{ij} = \frac{1}{2}(p_{j|i} + p_{i|j})$$

and

$$q_{ij} = \frac{1}{2}(q_{j|i} + q_{i|j})$$

are symmetrized similarities.

t-SNE Step 3: Minimize KL Divergence

- Minimization is done via gradient descent (or adaptive methods like Adam).
- Small KL means the 2D layout preserves neighborhood structure: if i and j are close in high-D, they remain close in 2D.

Gradient of KL Divergence

- The gradient of KL with respect to y_i is:

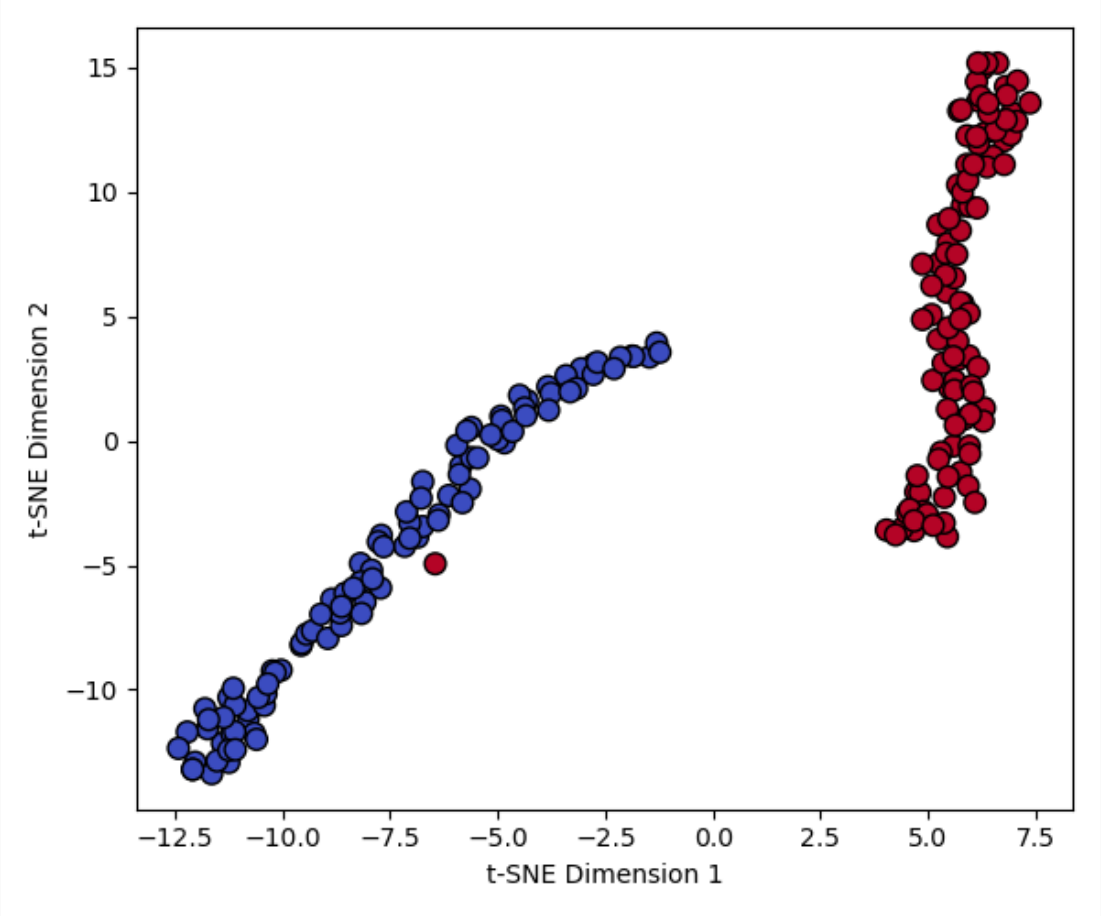
$$\frac{\partial \mathrm{KL}}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + \|y_i - y_j\|^2)^{-1}$$

- Intuition: if $p_{ij} > q_{ij}$ (points are neighbors in high-D but far in low-D), the gradient pulls y_i and y_j together.
- If $p_{ij} < q_{ij}$ (unrelated in high-D, close in low-D), the gradient pushes them apart.
- Optimization alternates gradient updates with "momentum" and learning rate annealing.

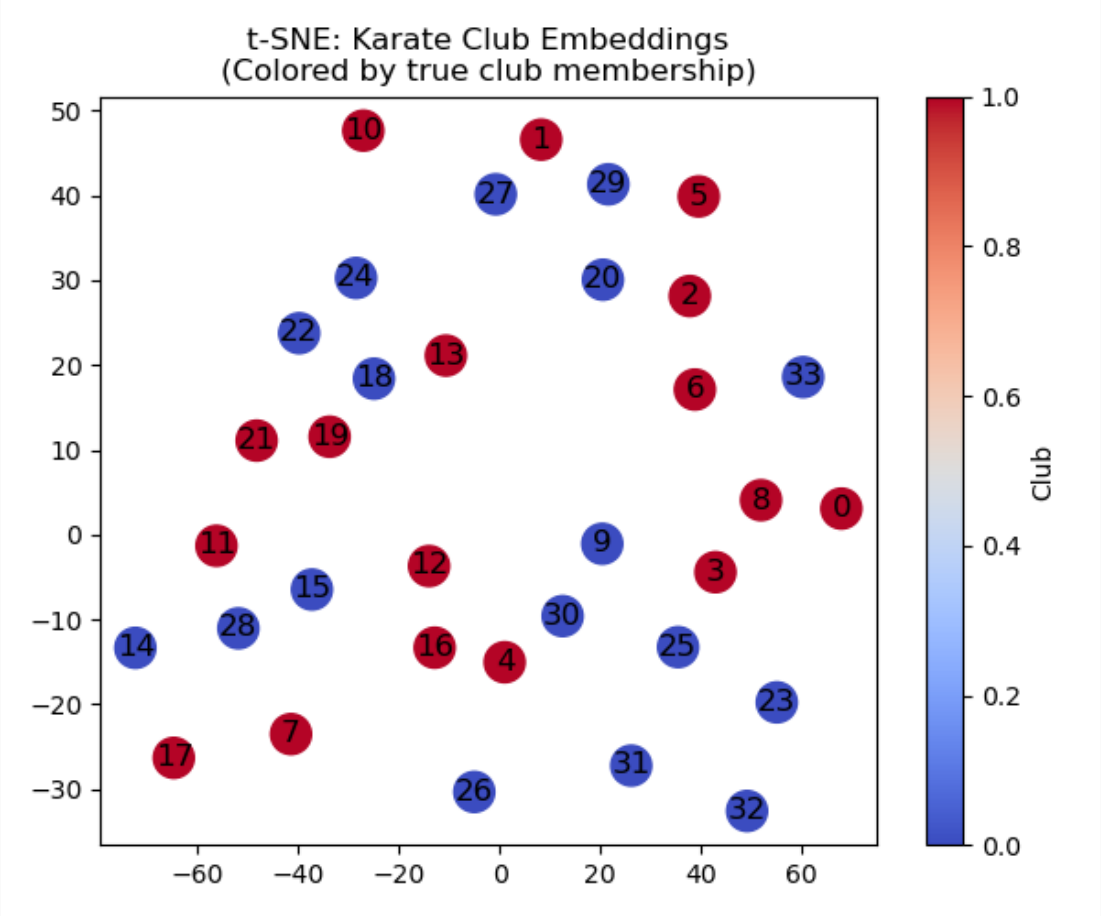
t-SNE: The Complete Algorithm

1. **Input:** High-dimensional data $X \in \mathbb{R}^{n \times d}$, perplexity $\texttt{perp}$, number of iterations T, learning rate η.
2. Compute high-D pairwise similarities p_{ij} (Gaussian kernel, adjust σ_i for each point to match perplexity).
3. Initialize $y_i \in \mathbb{R}^2$ randomly (e.g., from $\mathcal{N}(0, 10^{-4}I)$).
4. For iteration $t = 1, \ldots, T$:
   - 4.1 Compute low-D similarities q_{ij} (Student-t).
   - 4.2 Compute KL divergence and its gradient.
   - 4.3 Update y_i via gradient descent with momentum.
5. **Output:** Low-dimensional embeddings $Y = \{y_1, \ldots, y_n\}$.

Example: t-SNE on Two Moons



t-SNE on a Network Embedding: Karate Club



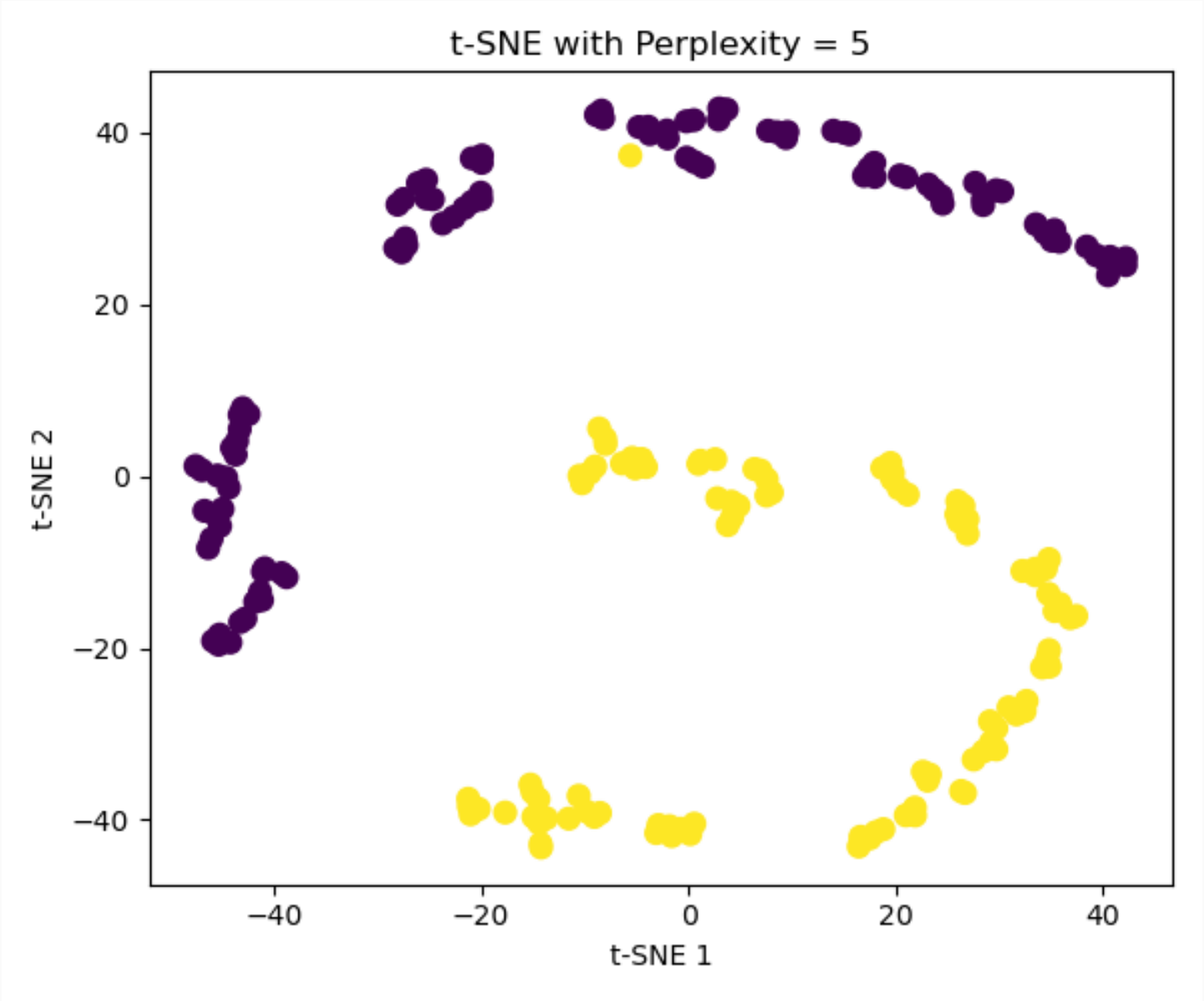
Advantages of t-SNE

- **Reveals nonlinear structure:** Clusters and manifolds are visualized clearly.
- **Local neighborhood preservation:** Points that are neighbors in high-D remain neighbors in 2D.
- **Handles complex geometries:** Swiss roll, intertwined spirals, and other complex manifolds are straightened and separated.
- **Intuitive interpretation:** Close points in the plot are similar; distant points are different (locally).

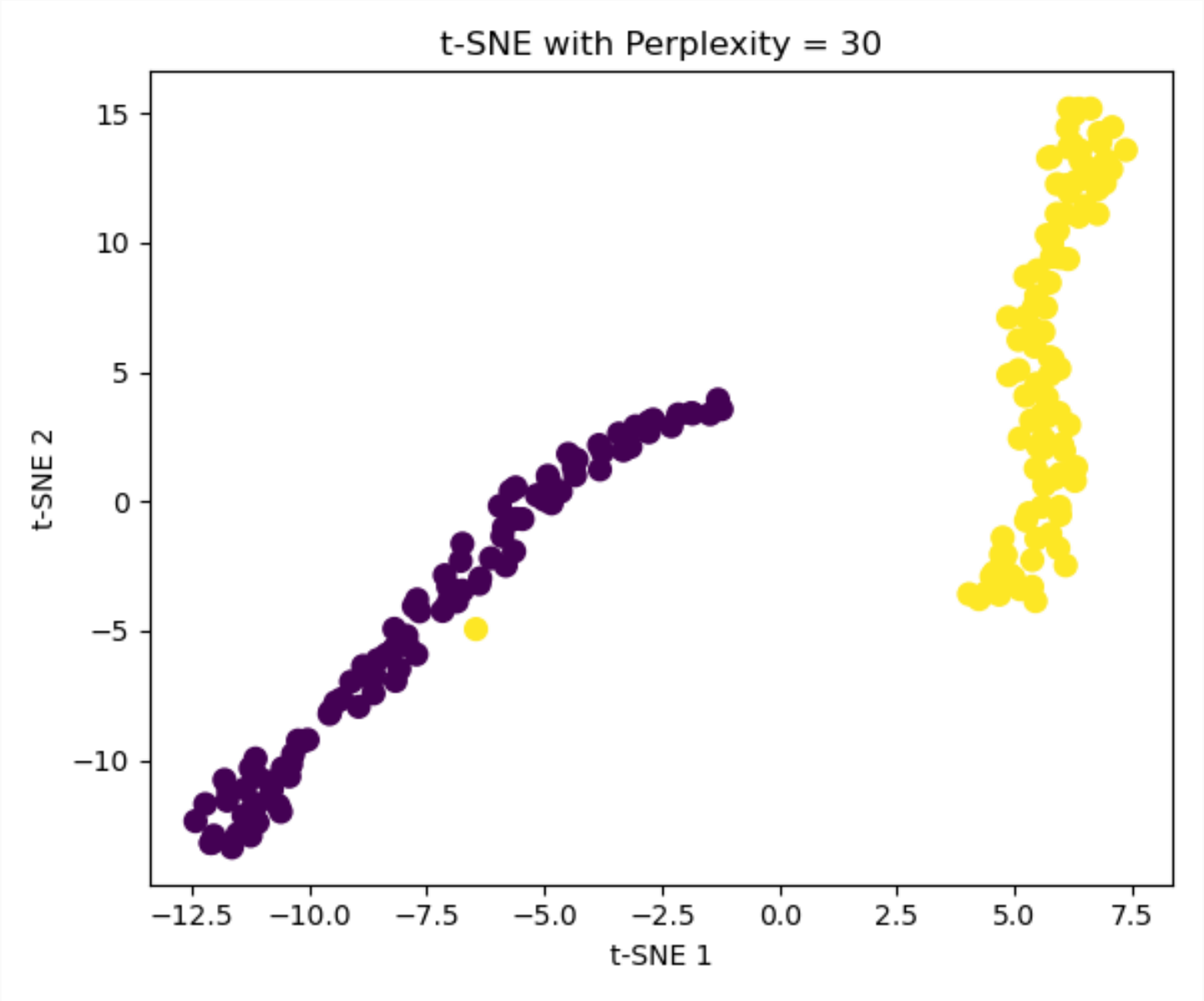
Limitations

- **Global structure is not preserved:** Distances between far-apart clusters are not meaningful.
- **Computationally expensive:** $O(n^2)$ time and memory for n points; approximations for large datasets.
- **Hyperparameter sensitive:** Perplexity choice affects results; no single "right" perplexity for all datasets.
- **Non-deterministic:** Different random initializations and optimization runs can yield different (but topologically similar) layouts.

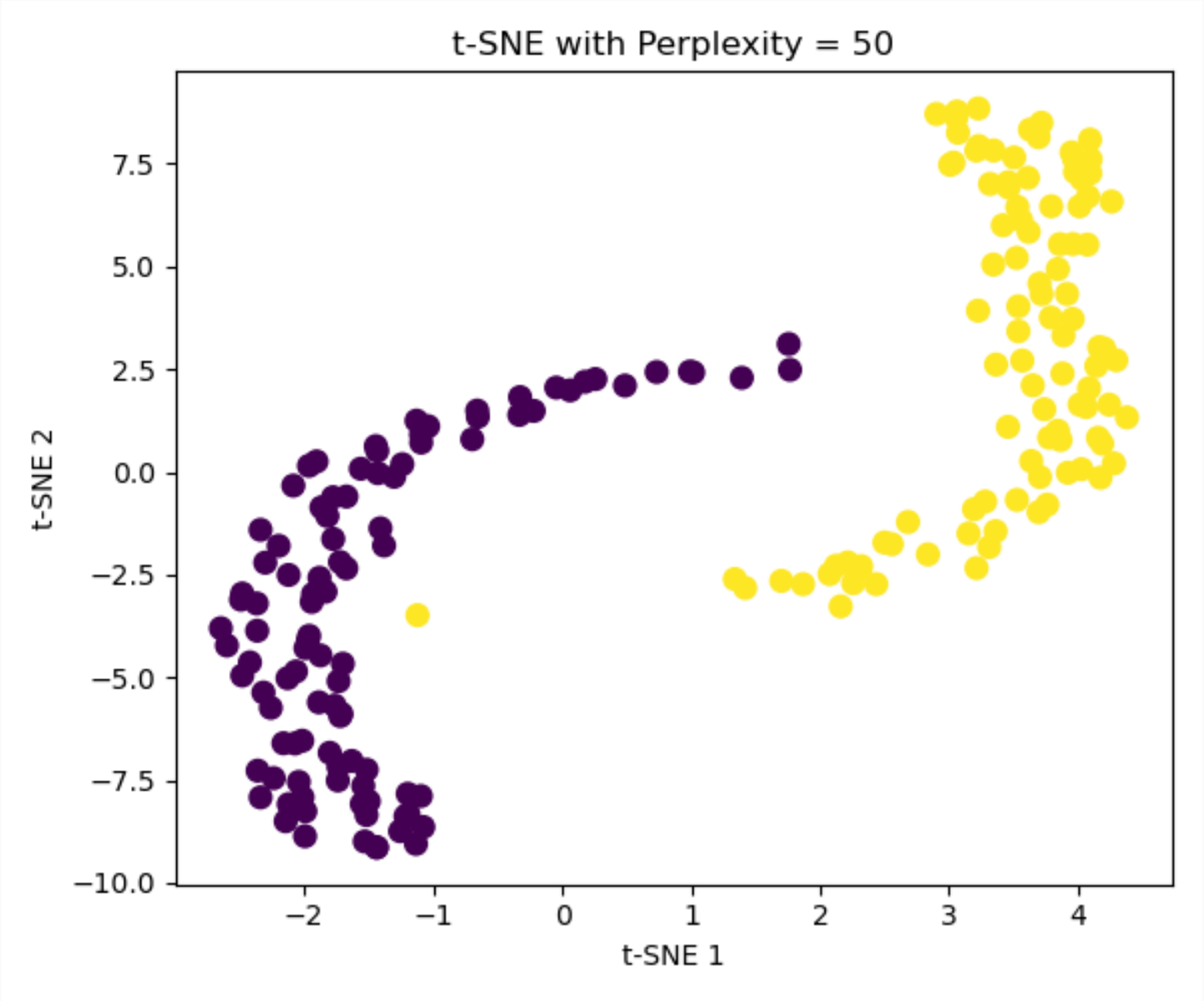
Perplexity Comparison 1/3



Perplexity Comparison 2/3



Perplexity Comparison 3/3



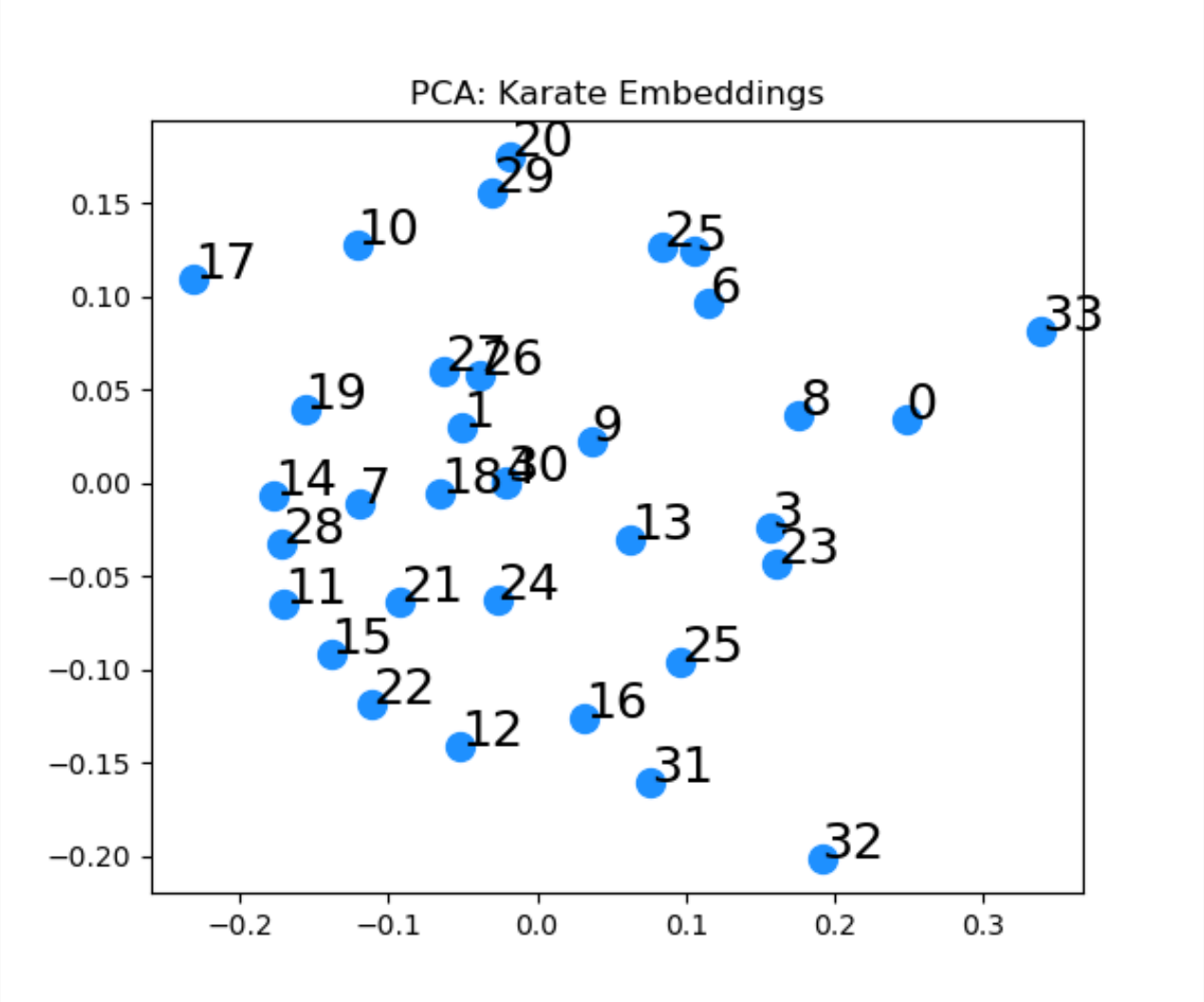
Practical Recommendations

- Use t-SNE when interested in discovering clusters and local neighborhoods.
- Set perplexity between 5 and 50; for most datasets, 30 is a good starting point.
- Always visualize with multiple perplexities to ensure stability.
- Do NOT interpret global distances or cluster separation sizes as meaningful—focus on local groupings.
- Combine t-SNE with other visualizations (PCA, UMAP, original network layout) for a complete picture.

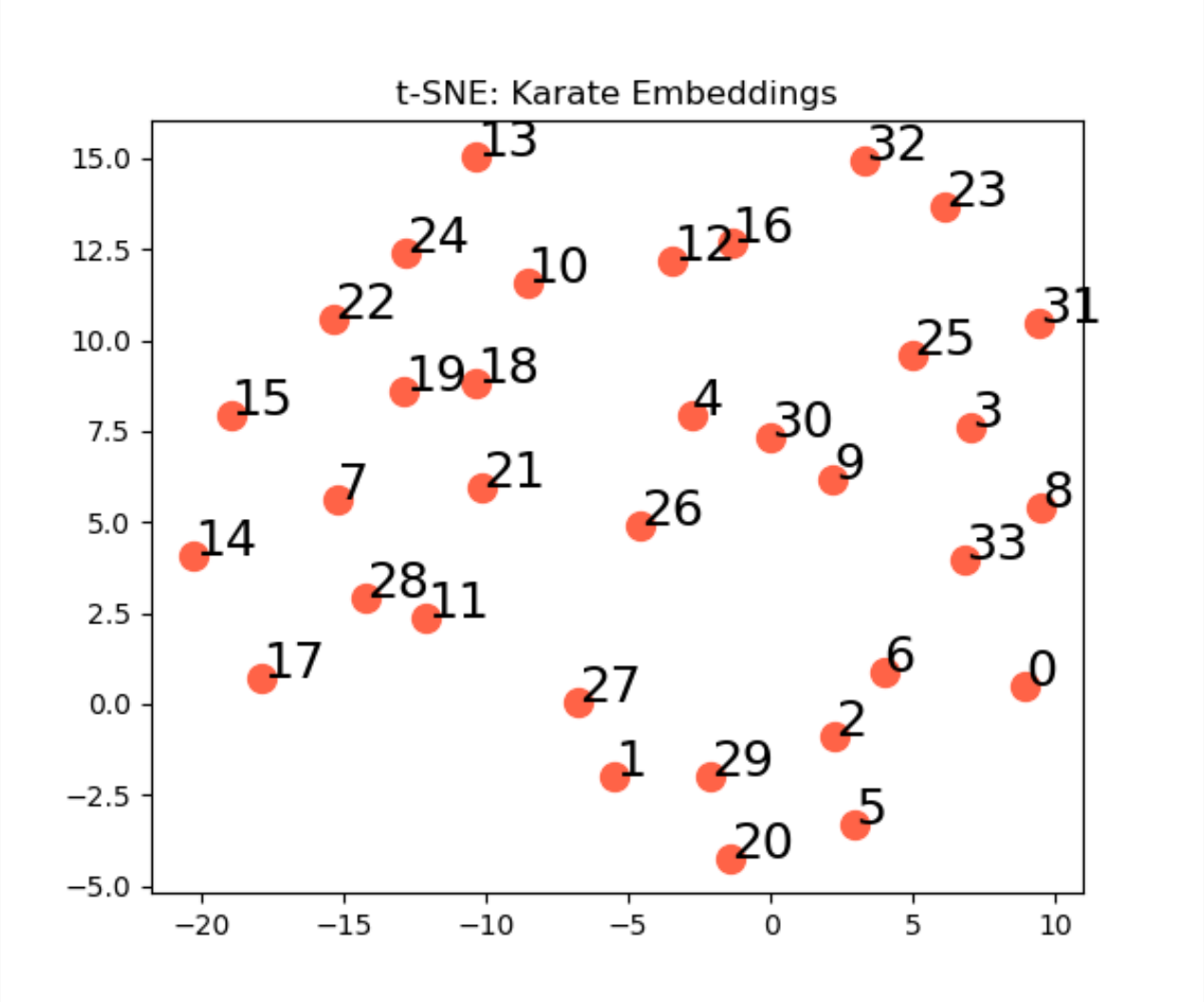
PCA vs t-SNE: Visualization Comparison

- **PCA preserves global structure**: relative distances in 2D reflect high-dimensional distances.
- **t-SNE preserves local neighborhoods**: clusters are faithfully preserved, but global layout can be stretched or compressed.

PCA vs t-SNE: Visualization Comparison



PCA vs t-SNE: Visualization Comparison



Random Walks on Networks

What is a Random Walk?

- A **random walk** is a stochastic process that starts at a node and, at each step, moves to a random neighbor.
- Formally, given a current node v with neighbors

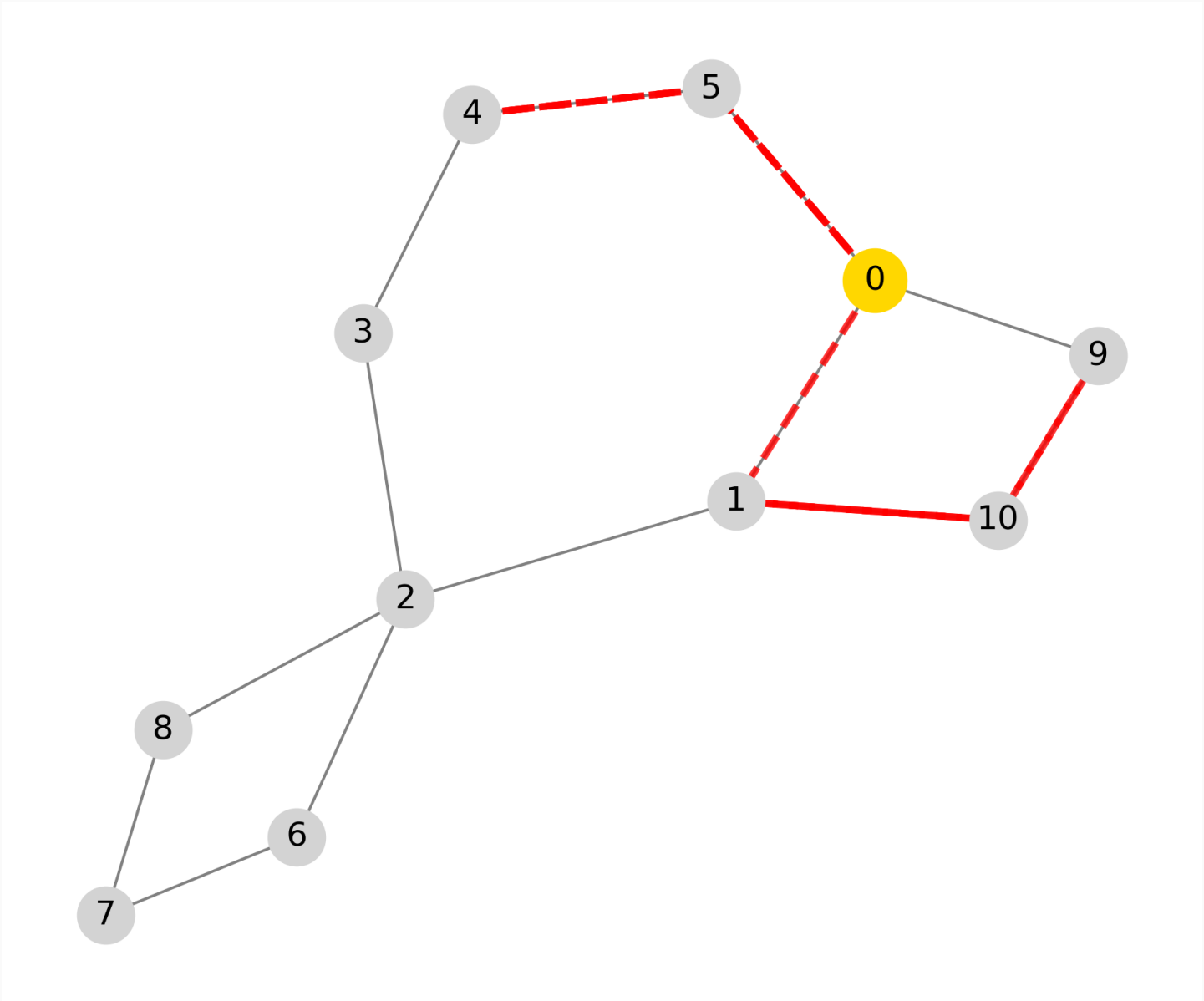
$$\Gamma(v) = \{u_1, u_2, \ldots, u_k\}$$

- The next node is chosen uniformly at random:

$$P(\text{next} = u_i \mid \text{current} = v) = \frac{1}{|\Gamma(v)|}$$

- A walk of length L is a sequence of nodes: $(v_0, v_1, v_2, \ldots, v_L)$ where each transition is governed by this random process.

Simple Example: Random Walk on a Toy Graph



Why Random Walks Explore Network Structure

- Random walks naturally spend more time in densely-connected regions (communities) because there are more paths to traverse within dense areas.
- Conversely, sparsely-connected regions are visited less frequently because there are fewer paths.
- If two nodes are in the same community (densely connected), they are more likely to co-occur in random walks.
- Random walks thus provide a way to **sample neighborhoods** that reflects network structure without explicitly computing distances or metrics.
- This is important for methods like DeepWalk and node2vec.

Mathematical Properties of Random Walks

- **Stationary distribution:** For a connected, undirected graph, a random walk converges to a stationary distribution $\pi(v) \propto \deg(v)$ (degree distribution).
- This means: in the long run, the probability of being at node v is proportional to its degree. High-degree nodes are visited more often.
- **Mixing time:** The number of steps needed to approach the stationary distribution. Fast mixing means the walk explores the graph quickly.
- **Commute time:** The expected time to go from node u to node v and back. Short commute times indicate u and v are in the same community.

- These properties explain why random walk based embeddings

Generating Random Walks: Algorithm

```python
import networkx as nx
import numpy as np

def random_walk(G, start_node, walk_length, seed=None):
    if seed is not None:
        rng = np.random.default_rng(seed)
    else:
        rng = np.random.default_rng()

    walk = [start_node]
    current = start_node

    for x in range(walk_length):
        neighbors = list(G.neighbors(current))
        if not neighbors:
            break
        current = rng.choice(neighbors)
        walk.append(current)

    return walk
```

Generating Random Walks: Algorithm

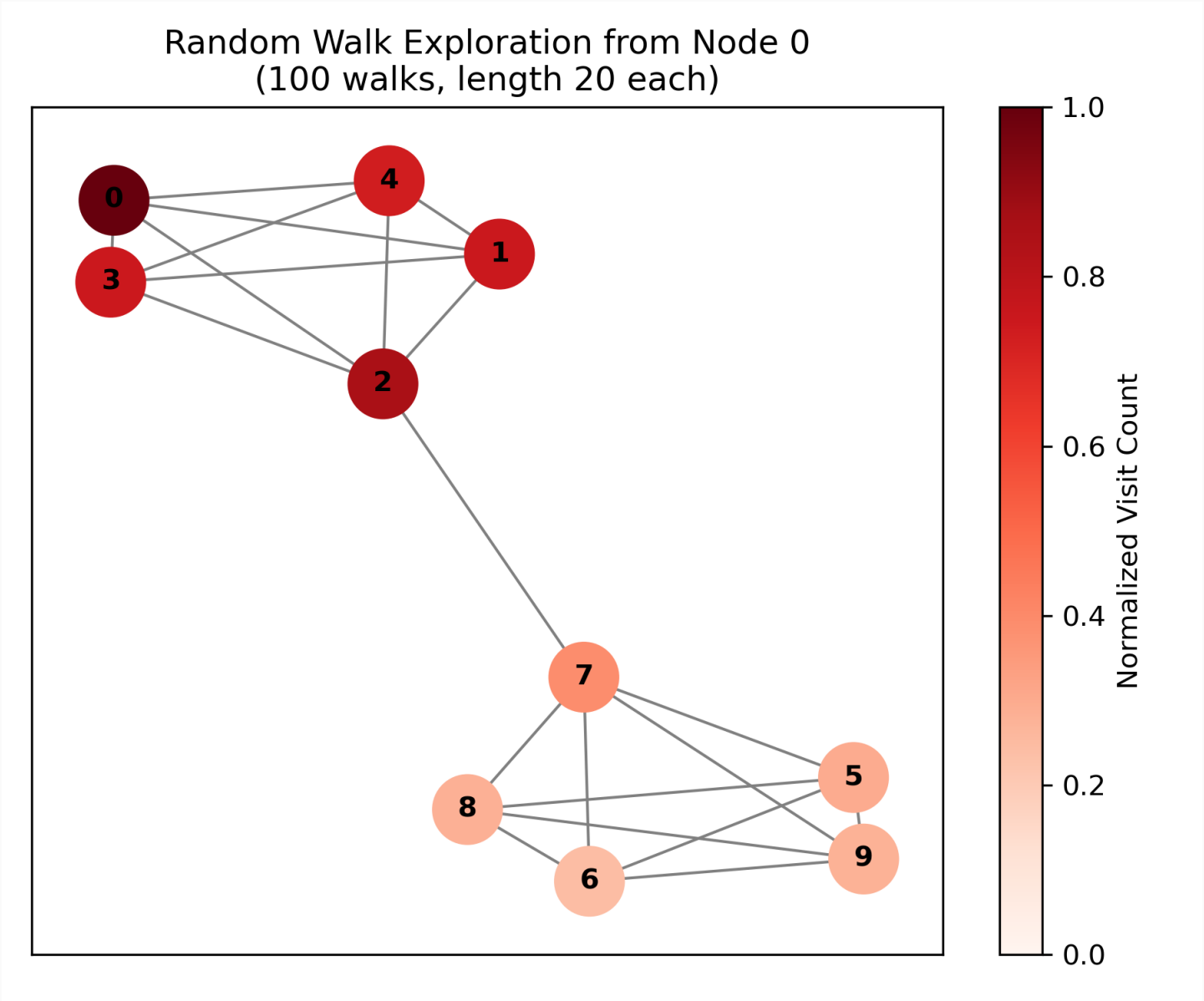
```python
#G = nx.cycle_graph(6)  # nodes 0..5 in a ring
G = nx.karate_club_graph()

w = random_walk(G, start_node=0, walk_length=10, seed=42)
print("Random walk:", w)

def generate_walks(G, num_walks_per_node=5, walk_length=10, seed=None):
    walks = []
    rng = np.random.default_rng(seed)
    for node in G.nodes():
        for _ in range(num_walks_per_node):
            s = int(rng.integers(0, 1e9))
            walks.append(random_walk(G, start_node=node, walk_length=walk_length, seed=s))
    return walks

walks = generate_walks(G, num_walks_per_node=3, walk_length=8, seed=123)
for i, w in enumerate(walks[:5]):
    print(f"Walk {i}:", w)
```

```
Random walk: [0, 2, 27, 24, 27, 23, 33, 9, 33, 14, 32]
Walk 0: [0, 17, 0, 21, 1, 13, 3, 0, 12]
Walk 1: [0, 3, 1, 7, 2, 0, 17, 1, 17]
Walk 2: [0, 10, 5, 10, 4, 0, 6, 16, 5]
Walk 3: [1, 3, 7, 0, 8, 33, 26, 33, 31]
Walk 4: [1, 7, 2, 0, 11, 0, 19, 0, 1]
```

Multiple Random Walks: Sampling Neighborhoods at Scale

- One random walk from a node provides a single sample of its neighborhood.
- Many walks from each node sample different neighborhoods and capture differences in network structure.
- Each walk explores a different path, discovering neighbors at different distances and in different communities.
- Longer walks explore farther, capturing global structure. Shorter walks stay more local, focusing on immediate neighborhoods.
- A collection of walks of various lengths captures a rich, multi-scale view of the network neighborhood.

Random Walks Visualized: Distribution of Visited Nodes



From Walks to Embeddings

- If node u and node v often appear together in random walks, they should have similar embeddings.
- This is the foundation of embedding methods:
  - Generate walks to collect neighborhoods.
  - Apply a language model (skip-gram) to learn embeddings from walk sequences.
  - The language model sees walks as sentences and learns to represent co-occurring nodes as similar vectors.
- This two-step approach (walks + skip-gram) is exactly what DeepWalk and node2vec do.

Word2Vec: Learning Word Embeddings from Text

What is Word2Vec?

- **Word2Vec** learns a real-valued vector embedding for each word in a vocabulary.

- Key insight: *Words that appear in similar contexts should have similar vectors.*

- Training signal comes from the text itself:
  - Look at a word and its neighbors in a sliding window.
  - Define a prediction task (context $\rightarrow$ word, or word $\rightarrow$ context).
  - Learn embeddings that are good at this task.

- Result: word vectors capture semantic and syntactic relationships.

The Two Flavors: CBOW vs Skip-Gram

- CBOW (Continuous bag of words) and Skip-Gram are **shallow neural networks** with the same architecture.
- They differ in the **direction of prediction**:

	CBOW	**Skip-Gram**
Input	Context words	Target word
Output	Target word	Context words
Direction	context $\rightarrow$ target	target $\rightarrow$ context

- Despite the swap, the learned embeddings capture the same kind of similarity.

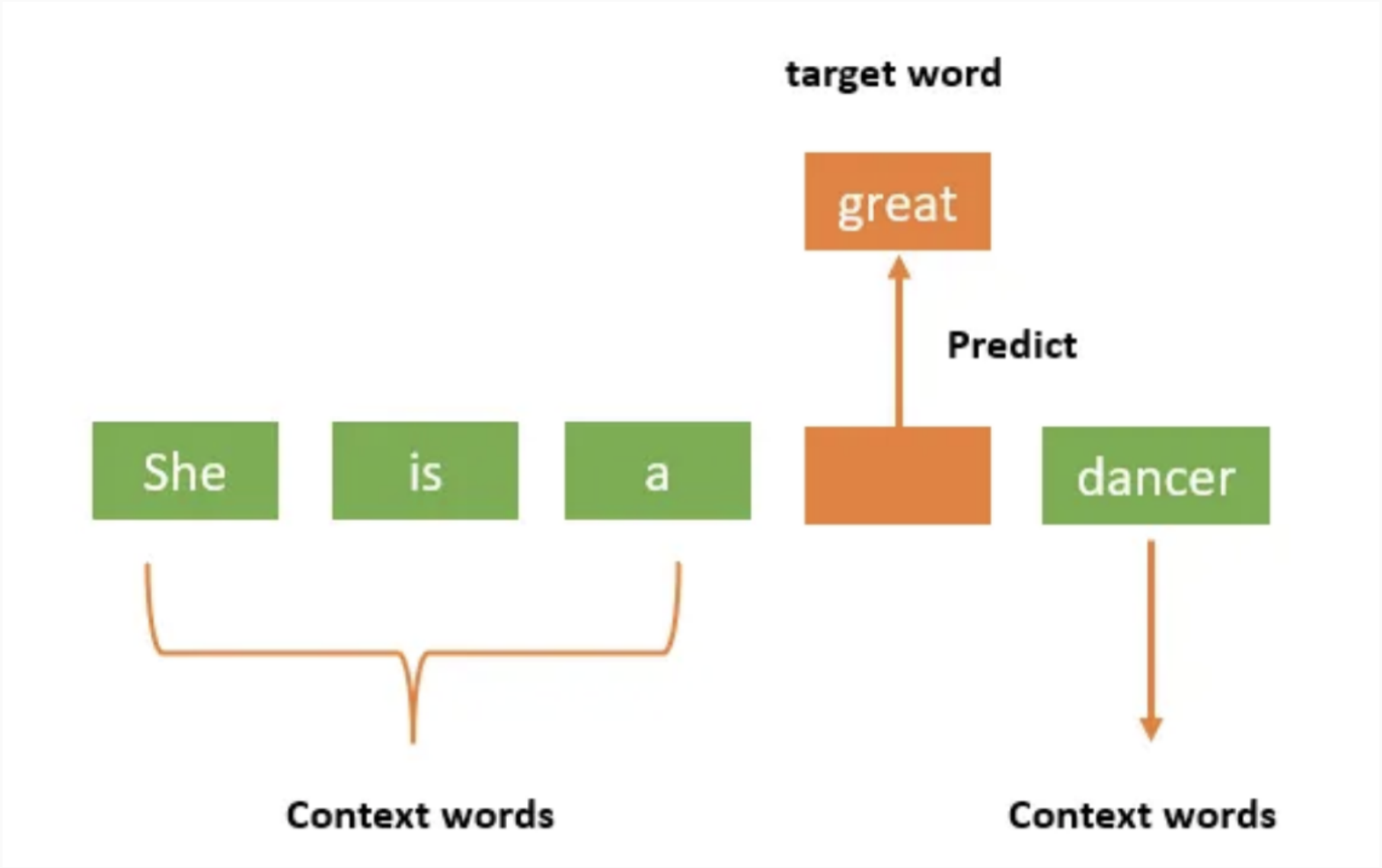
CBOW: High-Level Idea

- **Goal:** Predict a target word from its surrounding context words.
- **Example:** In the sentence "the quick brown fox jumps over", with window size 2:
  - Context: $\{\text{quick}, \text{brown}, \text{jumps}, \text{over}\}$
  - Target: fox

CBOW: High-Level Idea

- **Why "continuous"?**
  - Use dense real-valued vectors (embeddings) in a continuous space.
  - Not sparse one-hot symbols.
- **Why "bag"?**
  - Order of context words does not matter; they are summed/averaged.
  - Only the multiset (bag) of words in the window matters.

CBOW: context



CBOW: Mathematical Formulation

- Let $\mathbf{v}_w \in \mathbb{R}^d$ be the embedding of word w.
- Context words: $\{w_{-m}, w_{-m+1}, \ldots, w_{-1}, w_1, \ldots, w_m\}$ (window of size m on each side).
- **Step 1:** Average context embeddings:

$$\mathbf{h} = \frac{1}{2m} \sum_{i \in \text{context}} \mathbf{v}_i$$

CBOW: Mathematical Formulation

- **Step 2:** Predict target word w_t via softmax:

$$p(w_t \mid \text{context}) = \frac{\exp(\mathbf{v}'_{w_t} \cdot \mathbf{h})}{\sum_{w \in V} \exp(\mathbf{v}'_w \cdot \mathbf{h})}$$

  where $\mathbf{v}'_w$ is the "output embedding" of word w.

- **Loss:** Minimize cross-entropy (equivalently, maximize log-likelihood of true target word).

CBOW: Strengths and Weaknesses

Strengths:

- *Fast to train.* Fewer training examples needed per update (averages many contexts).
- Works well for *frequent* words.
- Simpler architecture.

Weaknesses:

- Less effective for *rare* words (each rare word is center less often).
- Loses information by averaging (order of context is ignored).
- Often produces slightly lower-quality embeddings than skip-gram.

Skip-Gram: High-Level Idea

- **Goal:** Predict context words from a target word.
- **Example:** In the sentence "the quick brown fox jumps over", with window size 2:
  - Target: fox
  - Create pairs:
    $(fox, quick), (fox, brown), (fox, jumps), (fox, over)$
- **Training:** For each pair, train to predict the context word from the target.
- **Intuition:** Learn embeddings where the target word's vector is good at "pointing towards" its context neighbors.

Skip-Gram: Mathematical Formulation

- Let $\mathbf{v}_w$ be the embedding of word w, and $\mathbf{v}'_w$ be its output embedding.
- Target word: w_t, context word: w_c (a neighbor within the window).
- **Step 1:** Embed target word:

$$\mathbf{h} = \mathbf{v}_{w_t}$$

- **Step 2:** For each context word w_c, predict it via softmax:

$$p(w_c \mid w_t) = \frac{\exp(\mathbf{v}'_{w_c} \cdot \mathbf{h})}{\sum_{w \in V} \exp(\mathbf{v}'_w \cdot \mathbf{h})}$$

Skip-Gram: Strengths and Weaknesses

Strengths:

- Works very well for *rare* words (each rare word used as target many times).
- Often produces *higher-quality* embeddings.
- Better at capturing fine-grained relationships.

Weaknesses:

- *Slower to train*. More training pairs per sentence.
- Requires careful regularization.
- Negative sampling trick is needed for practical speed.

CBOW vs Skip-Gram at a Glance

Aspect	**CBOW**	**Skip-Gram**
Direction	Context $\rightarrow$ Word	Word $\rightarrow$ Context
Training speed	Fast	Slow
Best for	Frequent words	Rare words
Quality	Good	Often better
Typical use	Speed-critical	Quality-critical

Negative Sampling Trick

- Full softmax over vocabulary is expensive (compute probabilities for all $|V| \approx 10^6$ words).
- **Negative sampling:** Replace softmax with binary classification.
- For each training pair (target, context):
  - Positive example: (target, context word)
  - Negative examples: sample k random words from vocabulary as "noise"

Negative Sampling Trick

- Train a binary classifier: "Is this word a true context neighbor, or noise?"
- Much faster: $O(k)$ instead of $O(|V|)$ per update, with $k \ll |V|$ (e.g., $k = 5$).
- Empirically: produces very similar embeddings to full softmax.

Connection to Node2Vec and DeepWalk

- Word2Vec (skip-gram) learns embeddings for words in text, where:
  - "Sentences" = text sequences.
  - "Words" = vocabulary items.
  - "Context" = neighboring words in a sliding window.
- **DeepWalk/Node2Vec:** Apply the exact same skip-gram idea to graphs:
  - "Sentences" = random walks on the graph.
  - "Words" = nodes (or graph entities).
  - "Context" = neighboring nodes along the random walk.
- Result: nodes with similar network neighborhoods get similar vectors.
- **Key insight:** The embedding technique is domain-agnostic. It works if it is possible to define a notion of "context".

Key Takeaway

- CBOW and skip-gram are **two sides of the same coin**:
  - CBOW: context $\rightarrow$ word (faster, good for frequent words)
  - Skip-gram: word $\rightarrow$ context (slower, better for rare words)

- Both learn embeddings in a **continuous vector space** using **shallow neural networks**.

- The key intuition: *words (or nodes) appearing in similar contexts should have similar embeddings*.

- The method is **general**: it applies to any sequential or contextual data, including graphs (node2vec, DeepWalk).

Word Analogies: Vector Arithmetic

- Word2Vec represents each word as a vector in a continuous space.
- An **analogy** is solved using **vector arithmetic**.
- **Example:** "king is to man as queen is to woman"
  - Compute: $\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{man}} + \mathbf{v}_{\text{woman}}$
  - Find the word whose vector is **closest** to this result (by cosine similarity).
  - Ideally: queen

Word Analogies: Vector Arithmetic

- In code (Gensim), this is:
  - **Positive words:** vectors that are **added**. Example: `["king", "woman"]`
  - **Negative words:** vectors that are **subtracted**. Example: `["man"]`
- **Intuition:**
  - Positive = capture properties you want in the result ("king-like" and "woman-like").
  - Negative = remove unwanted properties (subtract "man-ness").

Why Vector Arithmetic Works: Learned Semantic Directions

- Word2Vec is not explicitly told to create a "gender axis" or "royalty axis".
- However, during training, **systematic relationships** emerge naturally:
  - The difference vector $\mathbf{v}_{\text{woman}} - \mathbf{v}_{\text{man}}$ encodes a "gender shift".
  - This same direction applies across different word pairs.
- **Key observation:**

$$\mathbf{v}_{\text{woman}} - \mathbf{v}_{\text{man}} \approx \mathbf{v}_{\text{queen}} - \mathbf{v}_{\text{king}}$$

- Therefore:

$$\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{man}} + \mathbf{v}_{\text{woman}} \approx \mathbf{v}_{\text{queen}}$$

Embedding Methods for Link Prediction

Embedding Methods: Motivation and Context

- Embedding methods transform nodes into vectors in a continuous space, capturing structural similarity and patterns in the network.
- The goal is to map each node i to a vector $\mathbf{z}_i \in \mathbb{R}^d$ such that nodes with similar network positions are close in the embedding space.
- Embeddings can capture higher-order relationships and community structure that are missed by purely local heuristics.
- Once nodes are embedded, link prediction is reduced to estimating the likelihood of links via vector operations (e.g., dot product, cosine similarity).

Node2Vec and DeepWalk: Random Walk-Based Embeddings

- **Node2Vec** and **DeepWalk** are popular algorithms that use short random walks to generate "sentences" over the network, then apply language modeling techniques to learn embeddings.
- Random walk-based embeddings (e.g., DeepWalk) treat all neighbors equally: at each step, pick a random neighbor uniformly.
- This captures both local and global structure, but without explicit control over what type of structure is emphasized.

Motivation: Beyond Unbiased Random Walks

- Different applications may need different biases:
  - Homophily-based prediction: nodes in the same community should be similar.
  - Structural equivalence: nodes with similar roles (e.g., bridges, hubs) should be similar, even if far apart.
- **node2vec** introduces **biased random walks** via two parameters, p and q, to interpolate between these goals.

node2vec Overview

- How **node2vec** learns vector embeddings for nodes:
  1. Generate biased random walks starting from each node.
  2. Treat walks as "sentences" and train a skip-gram language model (word2vec-style).
  3. Output: a dense vector for each node.
- Nodes that frequently co-occur in walks (via the biased sampling) end up close in embedding space.
- The bias parameters p and q allow us to control whether embeddings emphasize local communities or global structural roles.

The Optimization Objective

- For each node u, node2vec learns embeddings to maximize the log-probability of observing its walk-based neighborhood $N_S(u)$:

$$\max_{\{\mathbf{z}_u\}} \sum_{u \in V} \log P(N_S(u) \mid \mathbf{z}_u)$$

- Expanding the conditional probability using the chain rule:

$$P(N_S(u) \mid \mathbf{z}_u) = \prod_{v \in N_S(u)} P(v \mid \mathbf{z}_u)$$

- Using a softmax (or negative sampling in practice):

$$P(v \mid \mathbf{z}_u) = \frac{\exp(\mathbf{z}_v^\top \mathbf{z}_u)}{\sum_{w \in V} \exp(\mathbf{z}_w^\top \mathbf{z}_u)}$$

What is $N_S(u)$? The Walk-Based Neighborhood

- $N_S(u)$ is the **multiset of nodes that co-occur with u in random walk contexts**, sampled according to strategy S.
- Concretely:
  - From node u, generate multiple random walks: $(u = v_0, v_1, \ldots, v_L)$.
  - Use a context window of size k (e.g., $k = 5$).
  - For node v_i at position i, its context nodes are $v_{i-k}, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{i+k}$ (within walk bounds).
- $N_S(u)$ aggregates all context nodes from all walks starting at u.

Walk-Based Neighborhood: Example on a Toy Graph

- Consider a simple path graph: $0 - 1 - 2 - 3 - 4$.
- Starting from node 2, suppose we generate a walk $(2, 1, 0, 1, 2, 3, 4)$.
- With context window $k = 2$:
  - At position 0 (node 2): context $= \{1, 0\}$.
  - At position 4 (node 2): context $= \{0, 1, 3, 4\}$.
  - $N_S(2)$ collects these and all contexts from other walks started at 2.

Walk-Based Neighborhood: Example on a Toy Graph

- Notice: $N_S(2)$ includes nodes beyond immediate neighbors (e.g., node 0, node 4), reflecting the walk-induced structure.
- This is analogous to word2vec: the "context" of a word reflects which other words appear nearby in sentences, not just dictionary neighbors.

Analogy to Word2Vec

- In word2vec (language model):
  - "word" = vocabulary item (e.g., "cat", "dog").
  - "context" = words that frequently appear within a fixed window (e.g., 5 words left/right).
  - Learning objective: maximize $\log P(\text{context} \mid \text{word})$ across training sentences.
- In node2vec:
  - "word" = node ID (e.g., node 0, node 1).
  - "sentence" = random walk (sequence of nodes).
  - "context" = nodes within window k of the current node in the walk ($N_S(u)$).
  - Learning objective: maximize $\log P(N_S(u) \mid \mathbf{z}_u)$ across all nodes and sampled walks.

Second-Order Biased Random Walks: The Key Innovation

- In a plain random walk (DeepWalk), the transition from current node v to next node x depends only on v:

$$P(x \mid v) = \frac{1}{\deg(v)} \quad \text{(uniform)}$$

- In node2vec, the transition depends on **both** the current node v **and** the previous node t:

$$P(x \mid t, v) \propto \alpha_{pq}(t, x) \cdot w_{vx}$$

Second-Order Biased Random Walks: The Key Innovation

- This is called **second-order** because the transition depends on the last two nodes (previous + current).
- The bias factor $\alpha_{pq}(t, x)$ encodes preferences based on the relationship between previous node t and candidate next node x.

The Bias Factor $\alpha_{pq}(t, x)$: Definition

- Given that a walk has just moved from t to current node v, and we are deciding whether to step to neighbor x:
- Let d_{tx} = shortest-path distance between t and x in the graph.
- Define the unnormalized bias weight:

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \quad \text{(return to } t\text{)} \\ 1 & \text{if } d_{tx} = 1 \quad \text{(stay near } t\text{)} \\ \frac{1}{q} & \text{if } d_{tx} = 2 \quad \text{(move away from } t\text{)} \end{cases}$$

The Bias Factor $\alpha_{pq}(t, x)$: Definition

- The actual transition probability is then normalized:

$$P(x \mid t, v) = \frac{\alpha_{pq}(t, x) \cdot w_{vx}}{\sum_{z \in \Gamma(v)} \alpha_{pq}(t, z) \cdot w_{vz}}$$

Parameter p: Return Parameter

- p controls the probability of immediately returning to the previous node t (i.e., backtracking).
- If $x = t$, then $d_{tx} = 0$ and $\alpha_{pq}(t, x) = \frac{1}{p}$.
- **Large** p (e.g., $p = 2$):
  - $\frac{1}{p}$ is small, so backtracking is **discouraged**.
  - The walk is more likely to keep moving forward into new regions.

Parameter p: Return Parameter

- **Small** p (e.g., $p = 0.5$):
  - $\frac{1}{p}$ is large, so backtracking is **encouraged**.
  - The walk spends more time exploring the local neighborhood around the current node.
  - Default: $p = 1$ (no preference for or against backtracking).

Parameter q: In-Out Parameter

- q controls the trade-off between local exploration (BFS-like) and global exploration (DFS-like).
- Nodes at distance $d_{tx} = 2$ from t (i.e., moving away from t) have weight $\frac{1}{q}$.
- **Large** q (e.g., $q = 2$):
  - $\frac{1}{q}$ is small, so moves away from t are **down-weighted**.
  - The walk stays clustered near the current region (BFS-like).
  - Good for discovering **homophilic communities**: nodes in the same community are visited together.

Parameter q: In-Out Parameter

- **Small** q (e.g., $q = 0.5$):
  - $\frac{1}{q}$ is large, so moves away from t are **up-weighted**.
  - The walk is more exploratory (DFS-like).
  - Good for revealing **structural equivalence**: nodes with similar roles (e.g., hubs, bridges) are visited together across communities.
  - Default: $q = 1$ (no bias).

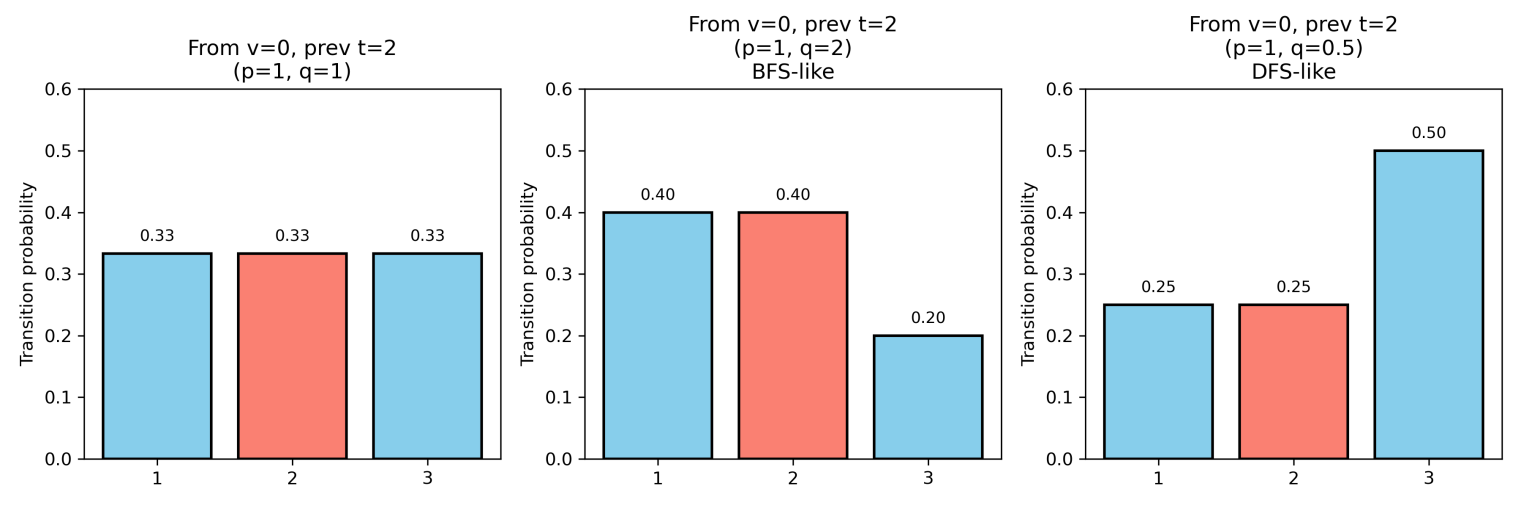
Intuition: BFS vs DFS

- **BFS-like exploration** ($q \gg 1$, e.g., $q = 2$):
- Walk explores outward layer by layer from starting node.
- Stays within tight, locally-connected regions (communities).
- Embeddings reflect community membership and local network structure.

Intuition: BFS vs DFS

- **DFS-like exploration** ($q \ll 1$, e.g., $q = 0.5$):
- Walk dives deep, exploring paths and branches far from the starting point.
- More likely to reach distant or structurally similar nodes.
- Embeddings reflect structural roles: nodes acting as hubs, bridges, or peripheries cluster together even if in different communities.

Visual Example: Transition Probabilities



The Complete node2vec Algorithm 1/2

- **Input:** Graph $G = (V, E, W)$, embedding dimension d, walk length L, walks per node r, window size k, parameters p, q.
- **Step 1:** Precompute transition probabilities $\alpha_{pq}(t, x)$ for all pairs of previous node t and candidate next node x to enable efficient sampling.
- **Step 2:** Generate walks by simulating r biased random walks of length L from each node, using the transition probabilities computed in Step 1.

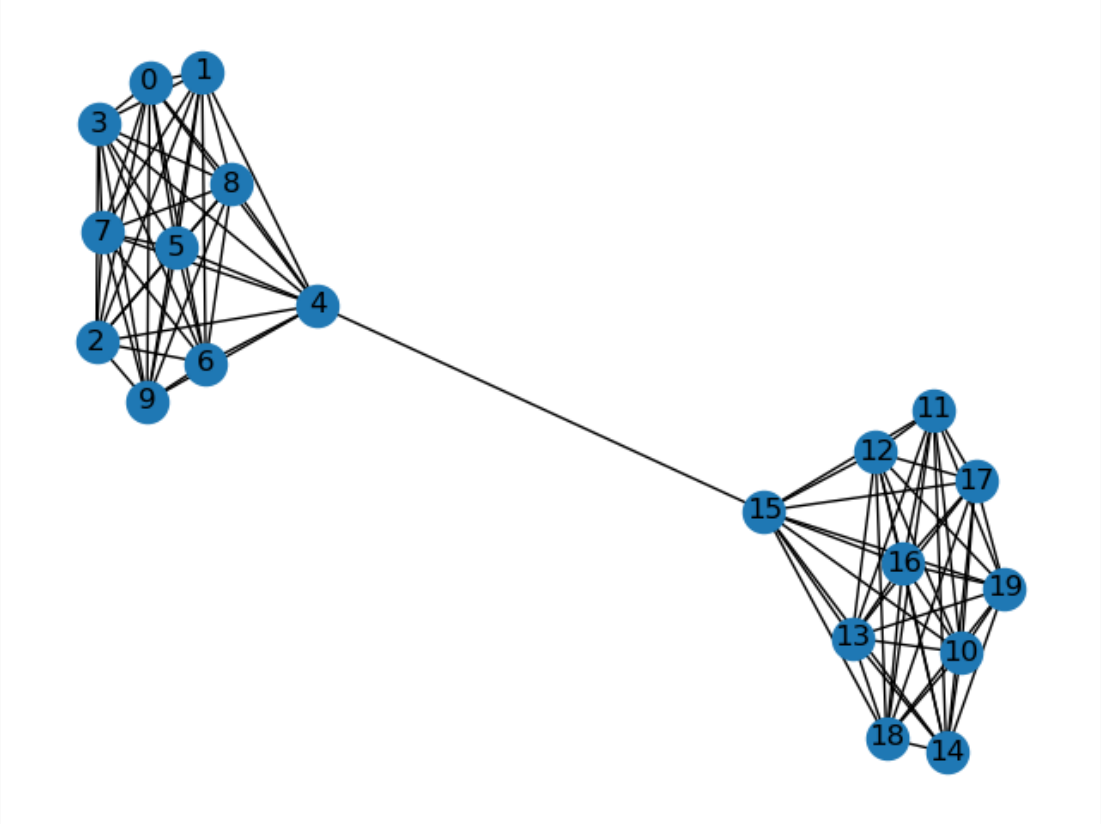
The Complete node2vec Algorithm 1/2

- **Step 3:** Train a skip-gram language model on the generated walks using word2vec, with context window k and appropriate hyperparameters (window size, negative samples, etc.).
- **Output:** Embedding matrix $Z \in \mathbb{R}^{|V| \times d}$, where row u is the d-dimensional embedding $\mathbf{z}_u$.

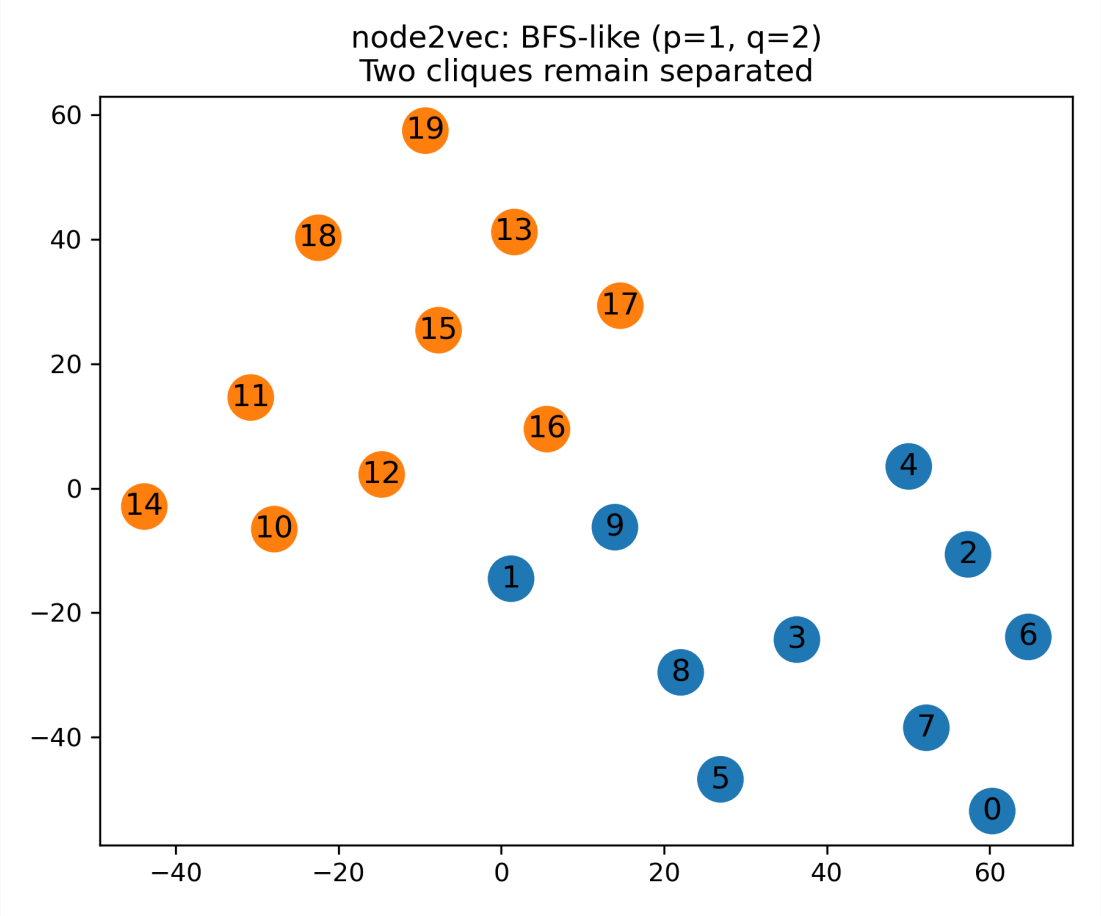
Effect of p and q on Embeddings: Two Cliques + Bridge

- Consider a graph with two dense cliques connected by a single bridge edge.
- With **BFS-like** parameters ($p = 1, q = 2$):
- Embeddings cluster tightly by clique membership.
- Bridge nodes stay close to their respective clique.
- With **DFS-like** parameters ($p = 1, q = 0.5$):
- Bridge nodes and other structurally similar nodes may cluster together.
- Community boundaries are less strict in the embedding space.

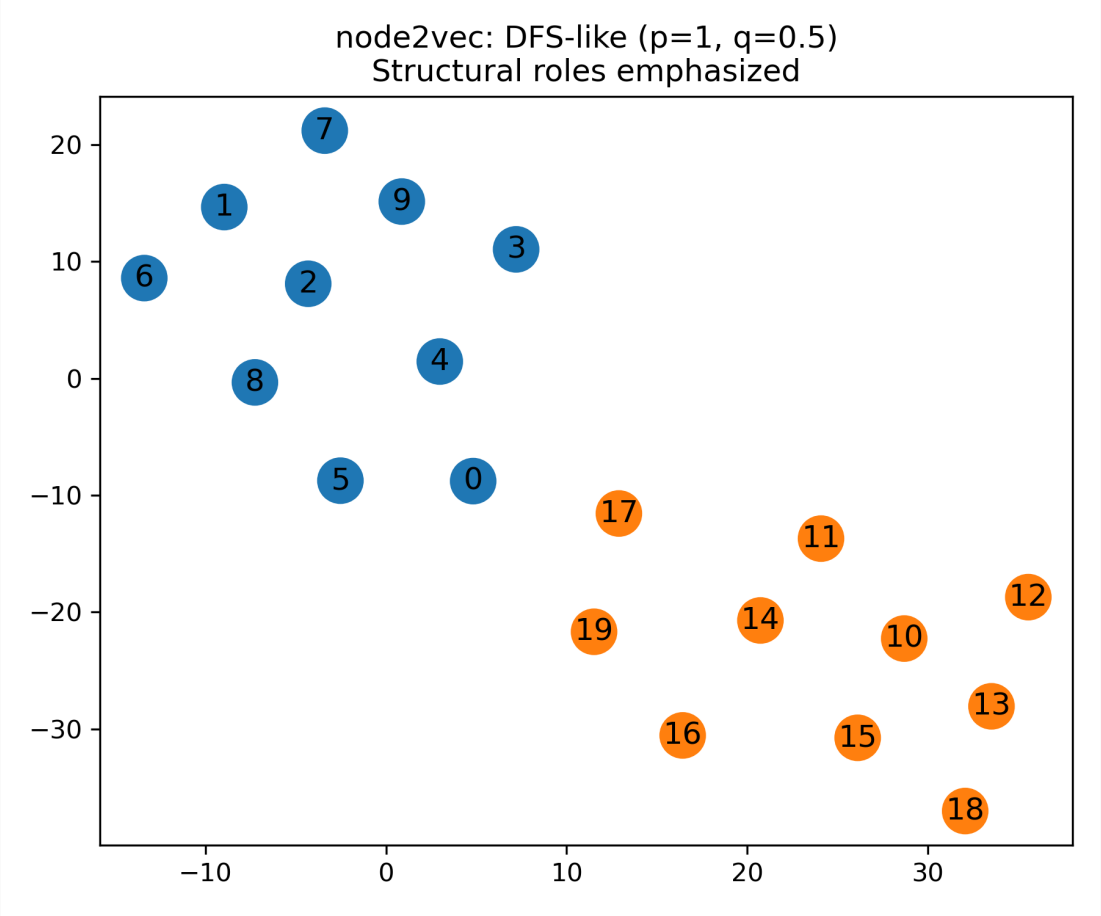
Visualizing the Effect: BFS-Like ($p = 1, q = 2$)



Visualizing the Effect: BFS-Like $(p = 1, q = 2)$



Visualizing the Effect: DFS-Like ($p = 1, q = 0.5$)



Advantages of node2vec

- **Flexibility:** Parameters p and q allow tuning toward community detection or structural roles.
- **Efficiency:** Precomputation of transition probabilities allows fast walk generation.
- **Scalability:** Works on large networks; parallelizable walk generation.
- **Interpretability:** Resulting embeddings can be used with any ML model; clear connection to graph structure.
- **Empirical performance:** Outperforms simpler methods on node classification, link prediction, and clustering tasks.

Practical Use: Link Prediction with node2vec Embeddings

- Compute node embeddings $\mathbf{z}_u, \mathbf{z}_v$ for all nodes u, v.
- For each candidate pair, construct a feature vector, e.g.:

$$\mathbf{f}_{(u,v)} = [\mathbf{z}_u \circ \mathbf{z}_v, |\mathbf{z}_u - \mathbf{z}_v|, \mathbf{z}_u^\top \mathbf{z}_v]$$

- Train a classifier (e.g., logistic regression, random forest) on pairs of known edges (positive) and random non-edges (negative).
- Use learned classifier to score unknown or held-out pairs.
- This approach combines the expressive power of node2vec embeddings with classical ML.

DeepWalk: Learning Social Representations

What is DeepWalk?

- **DeepWalk** is a simple but powerful algorithm for learning vector embeddings for nodes in a network.
- It was introduced by Perozzi, Al-Rfou, and Skiena in 2014 as one of the first random walk-based embedding methods.
- Core idea: treat nodes in a network like words in a document, and random walks through the network like sentences in text.
- Apply a language model (skip-gram, from word2vec) to learn embeddings where nodes that frequently co-occur in walks are nearby in embedding space.

The Key Analogy: Network as Text

- In natural language processing (NLP), word2vec learns embeddings by treating words in sentences as context for each other.
- DeepWalk applies the same principle to networks:
  - "Words" = nodes in the network
  - "Sentences" = random walks through the network
  - "Context" = nodes that appear nearby (within a window) in random walks
- Result: nodes that frequently appear together in random walks learn similar embeddings.

Why Random Walks?

- Random walks naturally explore the local and global structure of a network.
- Starting from any node, a random walk moves to a random neighbor, then to a random neighbor of that node, and so on.
- This process discovers:
  - **Local structure:** Nodes that are close neighbors get visited together early in walks.
  - **Global structure:** Long walks eventually reach distant nodes, capturing community membership and overall graph topology.
- By sampling many random walks of various lengths, DeepWalk captures neighborhoods at multiple scales.

Unbiased Random Walks in DeepWalk

- A random walk in DeepWalk is **unbiased**: at each step, pick a random neighbor with uniform probability.
- Formally, if current node is v with neighbors $\Gamma(v)$, the probability of moving to neighbor x is:

$$P(x \mid v) = \frac{1}{|\Gamma(v)|}$$

- This is simple and computationally fast: no need to precompute or bias transitions.
- Contrast with node2vec: node2vec introduces parameters p and q to bias walks toward BFS-like or DFS-like exploration.
- DeepWalk does not have such control; it simply explores uniformly.

DeepWalk Algorithm: Step by Step

- **Input:** Graph $G = (V, E)$, embedding dimension d, walk length L, number of walks per node r, window size k.
- **Step 1:** Generate random walks.
  - For each node $u \in V$, simulate r unbiased random walks of length L starting from u.
  - Each walk is a sequence $(u = v_0, v_1, \ldots, v_L)$ where each v_{i+1} is a random neighbor of v_i.

DeepWalk Algorithm: Step by Step

- **Step 2:** Collect walk data.
    - All walks together form a large corpus of sequences (like a corpus of sentences in text).
- **Step 3:** Train a skip-gram model on the walk corpus using word2vec.
- **Output:** Embedding matrix $Z \in \mathbb{R}^{|V| \times d}$, where row u is $\mathbf{z}_u$.

Skip-Gram Model: The Learning Objective

- The skip-gram model (from word2vec) learns embeddings by maximizing the probability of context nodes given a target node.

- For a walk $(v_0, v_1, \ldots, v_L)$, define the optimization objective:

$$\max_{\mathbf{z}_u} \sum_{u \in \text{walk}} \log P(\text{context}(u) \mid \mathbf{z}_u)$$

Skip-Gram Model: The Learning Objective

- For each node u at position i in the walk, its context consists of nodes within window k: $\{v_{i-k}, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{i+k}\}$.
- The conditional probability is:

$$P(v \mid \mathbf{z}_u) = \frac{\exp(\mathbf{z}_v^\top \mathbf{z}_u)}{\sum_{w \in V} \exp(\mathbf{z}_w^\top \mathbf{z}_u)}$$

- In practice, negative sampling is used to avoid computing the expensive full softmax.

Context Window: A Simple Example

- Suppose a random walk is: $(0, 1, 2, 3, 4, 5)$ with context window $k = 2$.
- For node 2 at position 2 (center), the context is $\{0, 1, 3, 4\}$ (nodes within distance 2).
- For node 2, the skip-gram model maximizes:

$$\log P(\{0, 1, 3, 4\} \mid \mathbf{z}_2)$$

- This means: given node 2's embedding $\mathbf{z}_2$, we want to predict that nodes 0, 1, 3, 4 are its neighbors in the walk.
- By maximizing this across all walks and all nodes, embeddings are learned so that nodes that co-occur in walks have similar (nearby) vectors.

Negative Sampling: Efficient Training

- Computing the full softmax normalization $\sum_{w \in V} \exp(\mathbf{z}_w^\top \mathbf{z}_u)$ is expensive for large graphs.
- **Negative sampling** approximates this by contrasting positive pairs (actual context nodes) with negative samples (random nodes).
- For each positive context node v and target u, sample a few random "negative" nodes $w_1, \ldots, w_m$ from the node distribution.

Negative Sampling: Efficient Training

- The objective becomes:

$$\log \sigma(\mathbf{z}_u^\top \mathbf{z}_v) + \sum_{i=1}^{m} \mathbb{E}_{w_i} \log \sigma(-\mathbf{z}_u^\top \mathbf{z}_{w_i})$$

- Here σ is the sigmoid function; we maximize similarity with positive pairs and minimize with negative pairs.

- This is much faster and allows training on large networks.

DeepWalk vs node2vec

- **Similarity:** Both use random walks to sample neighborhoods, then apply skip-gram to learn embeddings.
- **Differences:**
  - DeepWalk uses **unbiased random walks**: all neighbors equally likely.
  - node2vec uses **biased (second-order) walks** via parameters p and q to control BFS/DFS trade-off.

DeepWalk vs node2vec

- **Flexibility:**
  - DeepWalk is simpler and faster (no precomputation of transition probabilities).
  - node2vec offers more control via p, q to tune toward community detection or structural roles.
- **Empirical performance:** Often similar; node2vec slightly better on some tasks, but the difference is often marginal.

Why DeepWalk Works: Theoretical Connection

- DeepWalk exploits a deep connection between random walk probabilities and graph structure.
- The probability that node v appears in a random walk starting from u is related to commute times and graph partitions.
- By learning embeddings that preserve co-occurrence in random walks, DeepWalk implicitly preserves **proximity and community structure**.
- Random walks naturally concentrate on densely-connected regions (communities), so nodes in the same community are more likely to co-occur.
- This explains why DeepWalk embeddings are good for node classification and link prediction tasks.

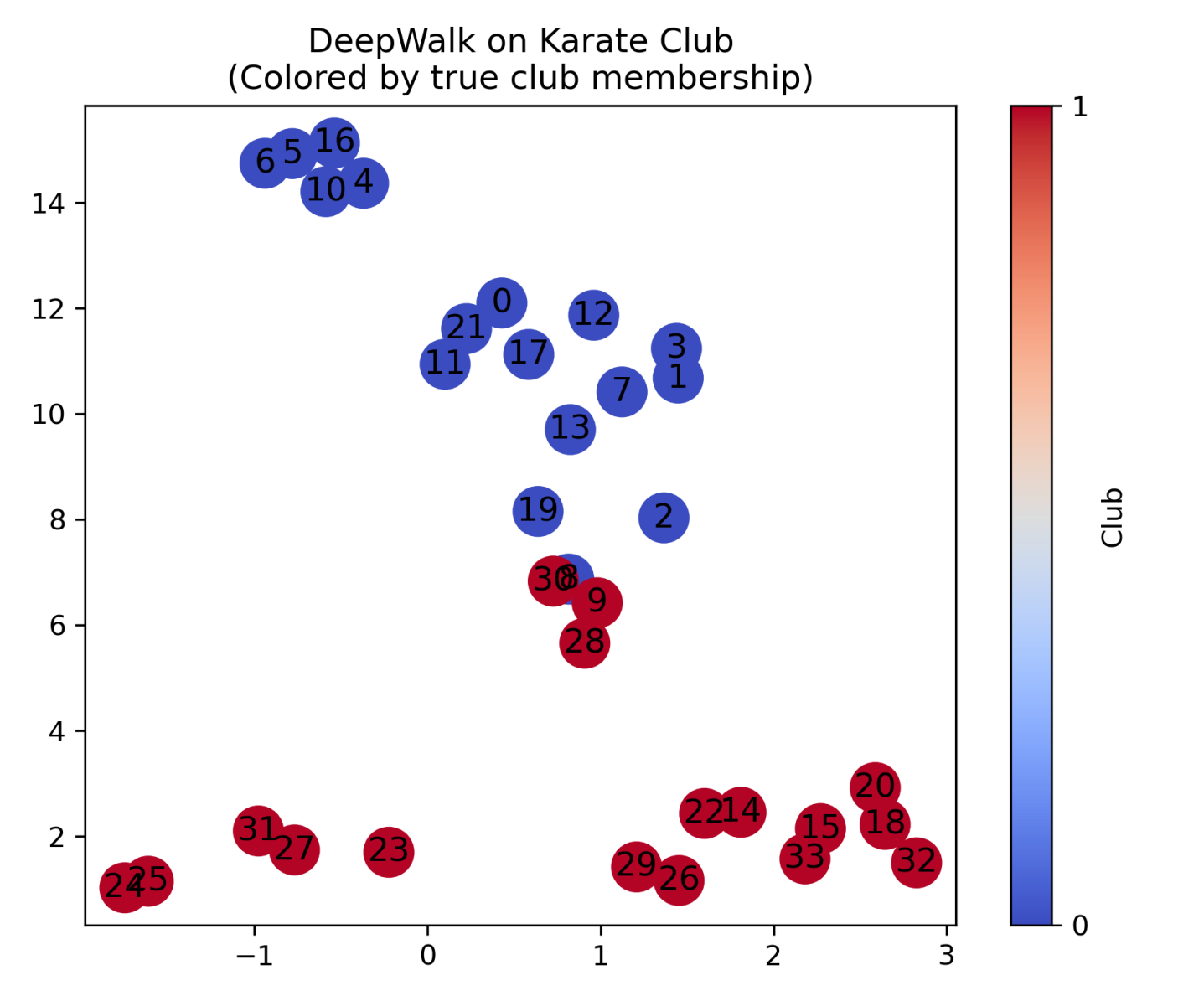
Hyperparameters of DeepWalk

- **Embedding dimension** d: Higher d captures more information but increases memory and training time. Typical values: 64–128.
- **Walk length** L: Longer walks explore further but are slower to compute. Typical: 40–80.
- **Number of walks per node** r: More walks give richer training signal but increase computation. Typical: 10–80.
- **Context window** k: Size of the neighborhood in skip-gram. Typical: 5–10. Larger k uses more context.
- **Negative samples** m: Number of random negatives per positive. Typical: 5–15.
- These hyperparameters must be tuned for specific

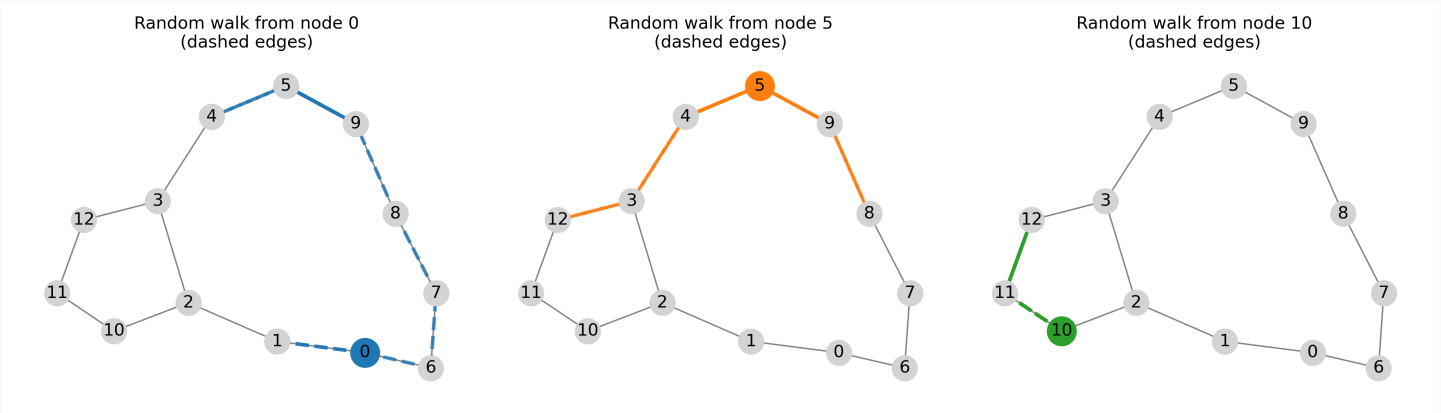
Practical Use: Node Classification with DeepWalk

- Compute embeddings $\mathbf{z}_u$ for all nodes using DeepWalk.
- For a downstream task like node classification:
  - Train a classifier (e.g., logistic regression, SVM, neural network) on labeled nodes using their embeddings as features.
  - Apply the trained classifier to unlabeled nodes to predict their labels.
- Similarly, for link prediction:
  - Construct pairwise features from embeddings (e.g., elementwise product, concatenation).
  - Train a binary classifier on known edges (positive) and random non-edges (negative).
  - Score unknown pairs to predict future links.

DeepWalk on Karate Club: Example



Comparison: Random Walks from Different Starting Nodes



Advantages of DeepWalk

- **Simplicity:** Easy to implement and understand; no complex bias parameters.
- **Speed:** Unbiased walks are fast; no need for precomputing transition probabilities.
- **Scalability:** Random walk generation is parallelizable; works on large networks.
- **Theoretical grounding:** Connection to commute times and random walk proximity well-understood.
- **Empirical effectiveness:** Strong performance on node classification, link prediction, and clustering.
- **Flexibility:** Works on weighted, directed, and undirected graphs.

Limitations of DeepWalk

- **No bias control:** Unbiased walks don't allow tuning toward specific structures (unlike node2vec's p, q).
- **Hyperparameter tuning:** Multiple hyperparameters (d, L, r, k, m) require careful tuning for best results.
- **Locality bias:** Random walks tend to stay in communities; may not capture global structure as well as other methods.
- **Computational cost:** Generating many long walks can be slow for very large graphs.
- **Stochasticity:** Results depend on random sampling; different runs may yield different (but similar) embeddings.

DeepWalk in Context: Random Walk-Based Methods

- DeepWalk opened the door to random walk-based embedding methods.
- Extensions and improvements include:
- **node2vec**: Biased walks via p, q parameters.
- **MetaPath2Vec**: Random walks on heterogeneous networks using meta-paths.
- **Graph2Vec**: Extending to entire graphs instead of individual nodes.
- Today, graph neural networks (GNNs) often outperform these methods, but DeepWalk remains a valuable baseline and foundation.