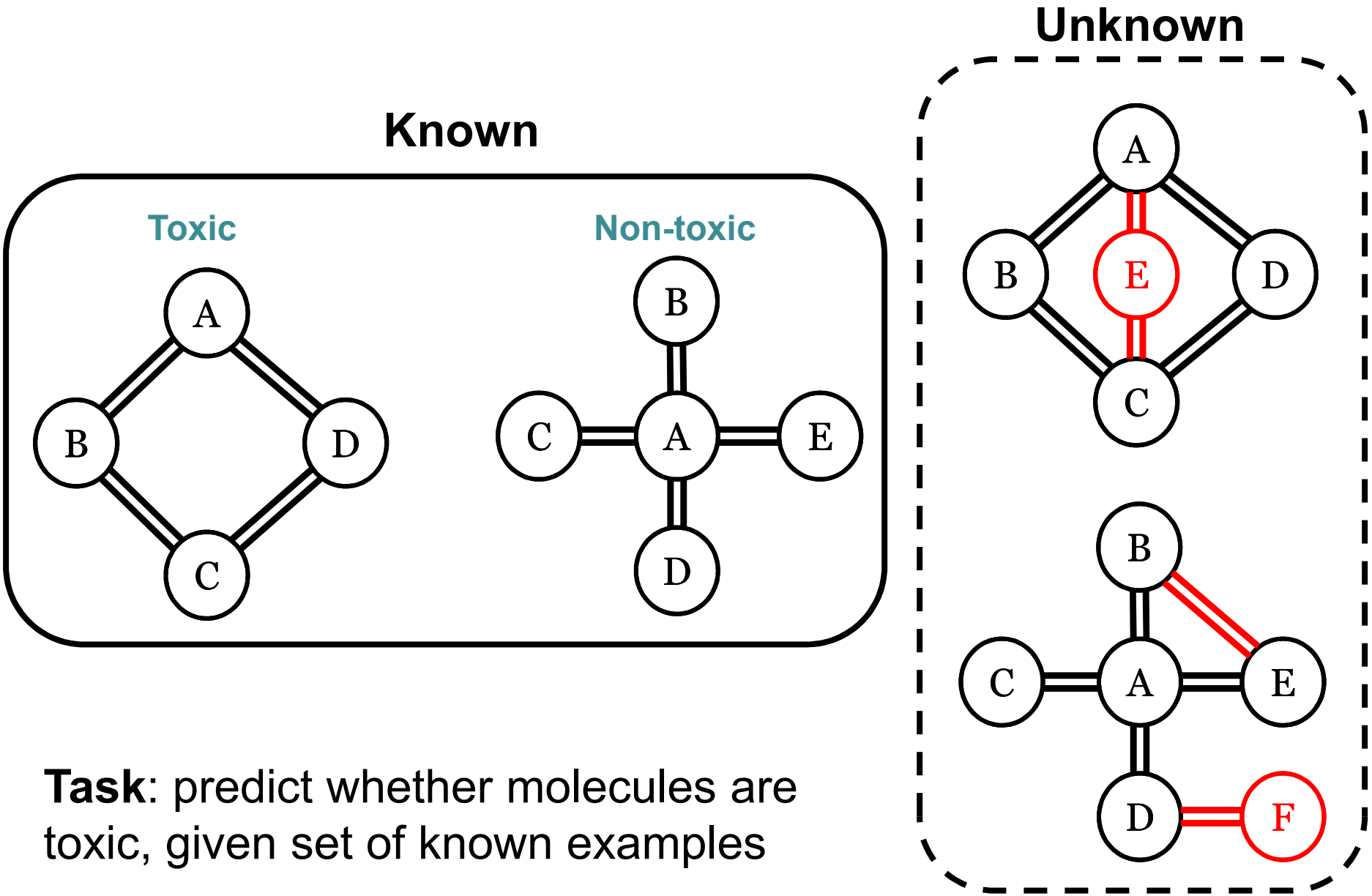
Graph Classification

Classification Outline

- **Introduction, Overview**
- **Classification using Graphs**
  - Graph classification
    - Direct Product Kernel
      - Predictive Toxicology example dataset
    - Graph embedding approach
  - Vertex classification
    - Laplacian Kernel
      - WEBKB example dataset
    - Vertex embedding approach
  - Edge/Link prediction
    - Edge embedding approach
- **gBoost – extension of "boosting" for graphs**
  - Progressively collects "informative" frequent patterns to use as features for classification / regression.
  - Also considered a frequent subgraph mining technique (similar to gSpan in Frequent Subgraph Chapter).

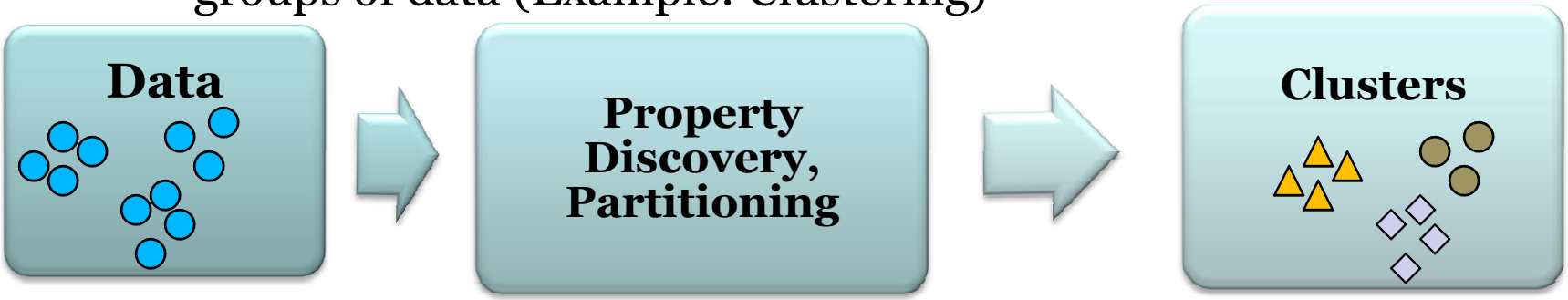
Example: Molecular Structures



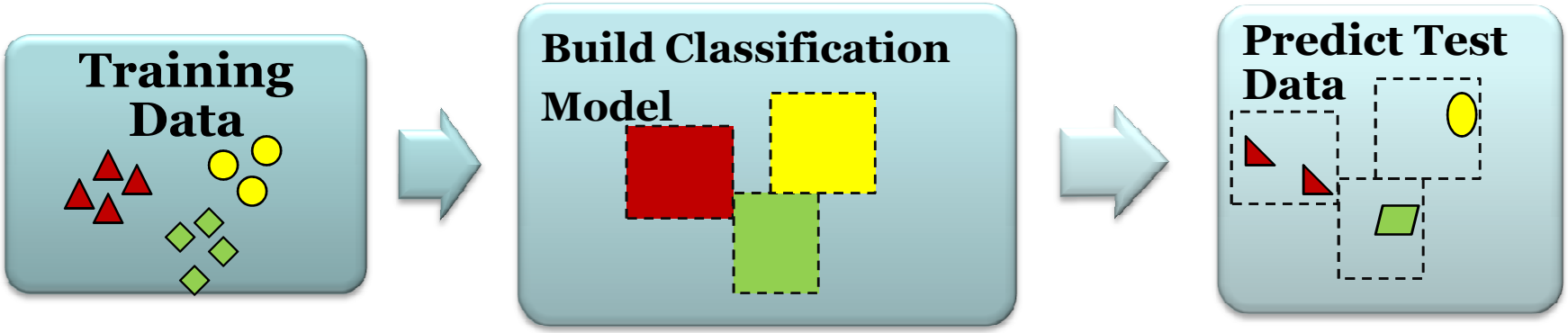
Task: predict whether molecules are toxic, given set of known examples

Solution: Machine Learning

- **Computationally *discover* and/or *predict* properties of interest of a set of data**
- **Two Flavors:**
  - **Unsupervised**: discover discriminating properties among groups of data (Example: Clustering)

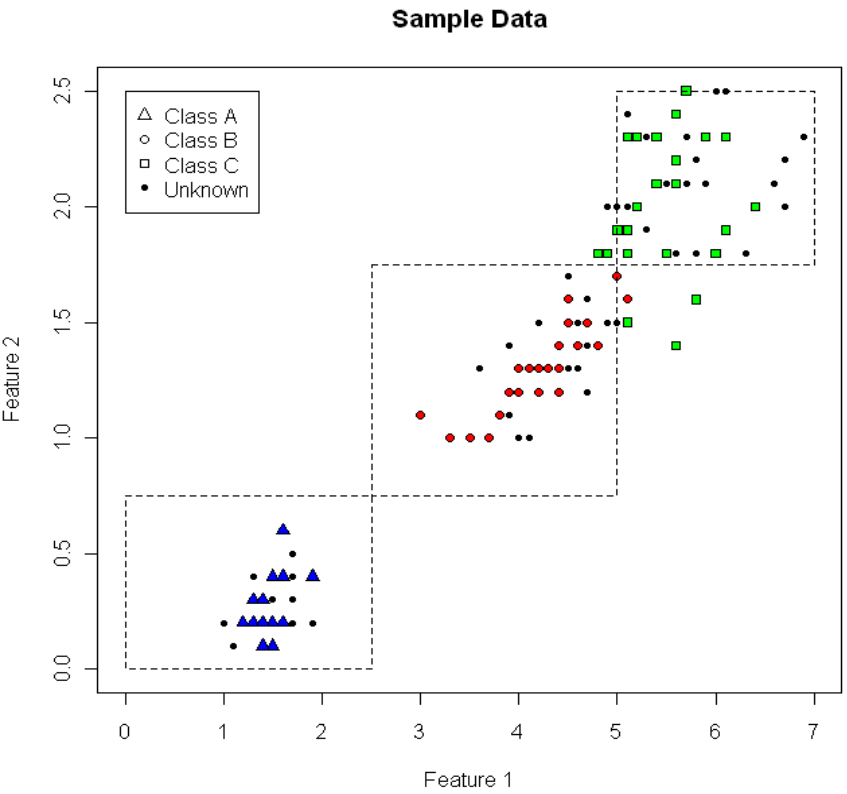


  - **Supervised**: known properties, categorize data with unknown properties (Example: Classification)

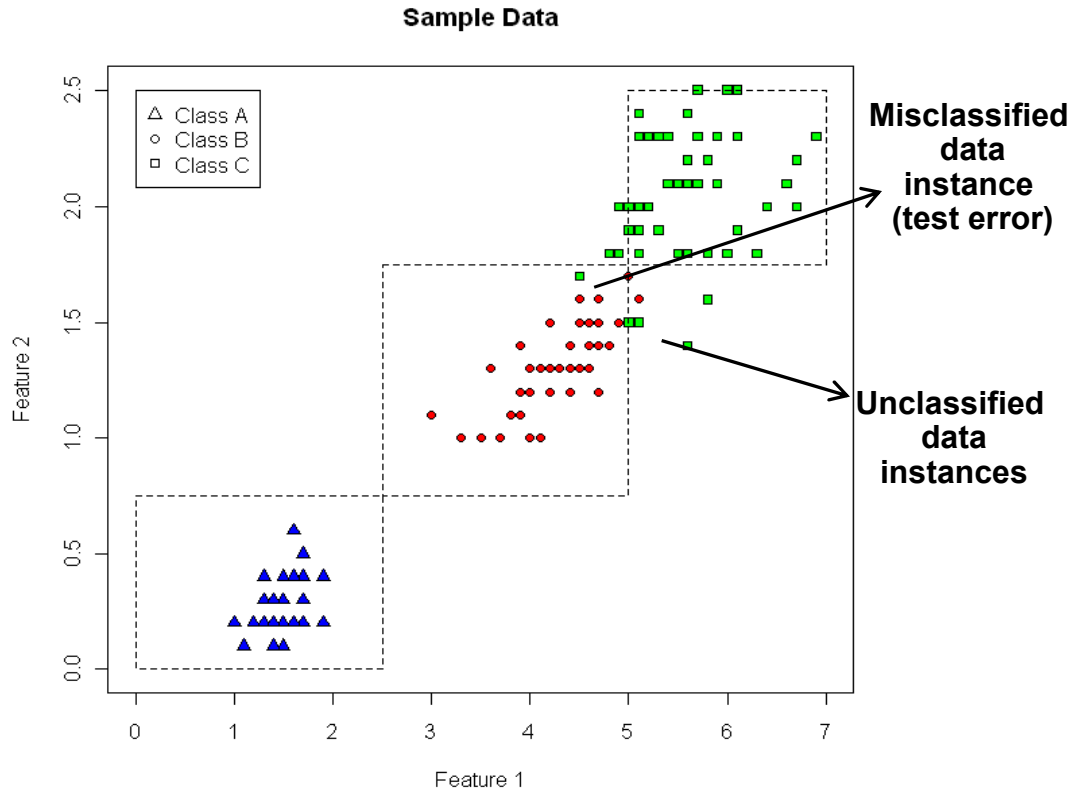


Classification

- ***Classification*: The task of assigning class labels in a discrete class label set Y to input instances in an input space X**
- **Ex: Y = { *toxic, non-toxic* }, X = {*valid molecular structures*}**



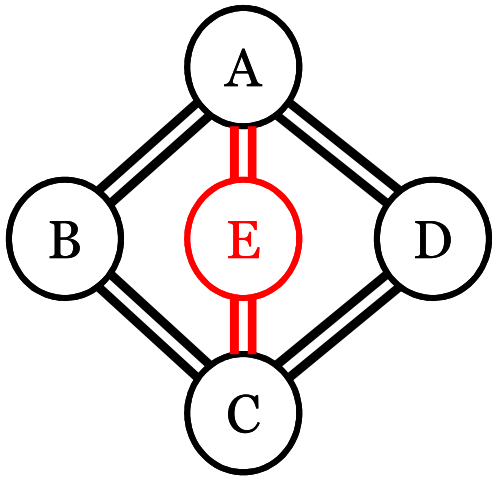
Training the classification model using the training data



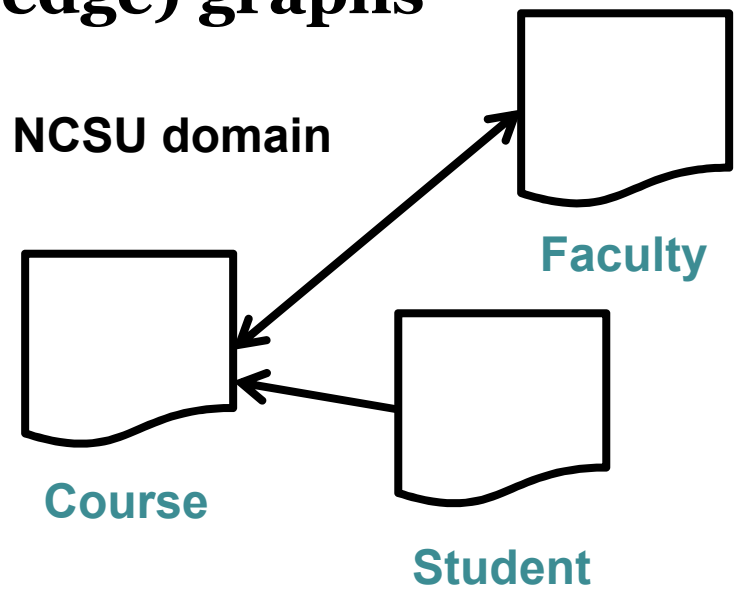
Assignment of the unknown (test) data to appropriate class labels using the model

Classification with Graph Structures

- **Graph classification (between-graph)**
  - Each full graph is assigned a class label
- **Example: Molecular graphs**



- **Vertex classification (within-graph)**
  - Within a single graph, each vertex is assigned a class label
- **Example: Webpage (vertex) / hyperlink (edge) graphs**



Relating Graph Structures to Classes?

- **Frequent Subgraph Mining**
  - Associate frequently occurring subgraphs with classes
- **Anomaly Detection**
  - Associate anomalous graph features with classes
- ***Kernel-based methods**
  - Devise kernel function capturing graph similarity, use vector-based classification via the *kernel trick*

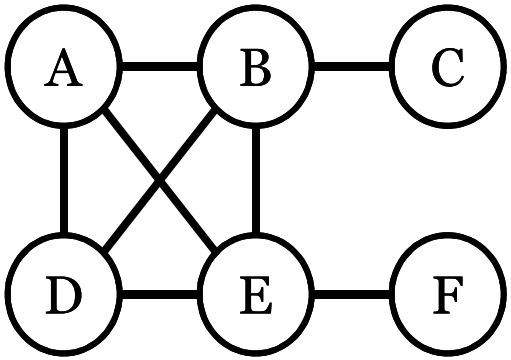
Relating Graph Structures to Classes?

- **This part focuses on kernel-based and embedding-based classification .**
- **Two step process:**
  - Devise kernel that captures property of interest
  - Apply *kernelized* classification algorithm, using the kernel function.
- **Two type of graph classification looked at**
  - Classification of Graphs
    - Direct Product Kernel
  - Classification of Vertices
    - Laplacian Kernel
- **Embedding-based Approach**
- **See Supplemental slides for *support vector machines* (SVM), one of the more well-known kernelized classification techniques.**

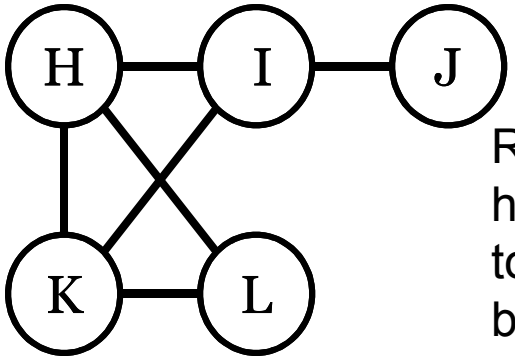
Walk-based similarity (Graph Kernels)

- **Intuition – two graphs are similar if they exhibit similar patterns when performing random walks**

Random walk vertices heavily distributed towards A,B,D,E

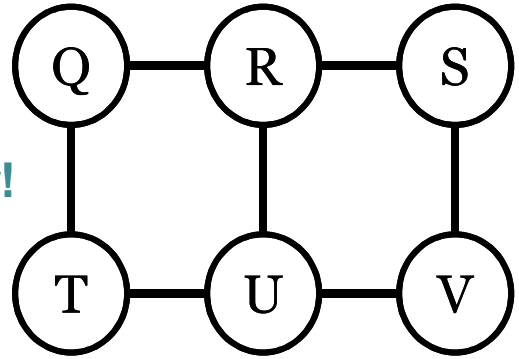


Similar!



Random walk vertices heavily distributed towards H,I,K with slight bias towards L

Not Similar!



Random walk vertices evenly distributed

Direct Product Graph – Formal Definition

Input Graphs

$$G_1 = (V_1, E_1)$$
$$G_2 = (V_2, E_2)$$

Direct Product Notation

$$G_X = G_1 \times G_2$$

Intuition

Vertex set: each vertex of V_1 paired with *every* vertex of V_2

Edge set: Edges exist only if both pairs of vertices in the respective graphs contain an edge

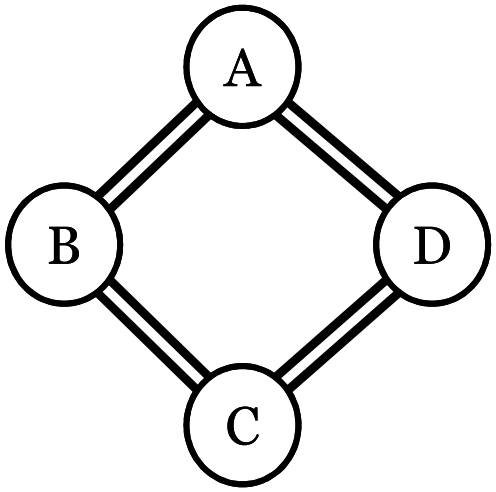
Direct Product Vertices

$$V(G_x) = \{(a, b) \in V_1 \times V_2\}$$

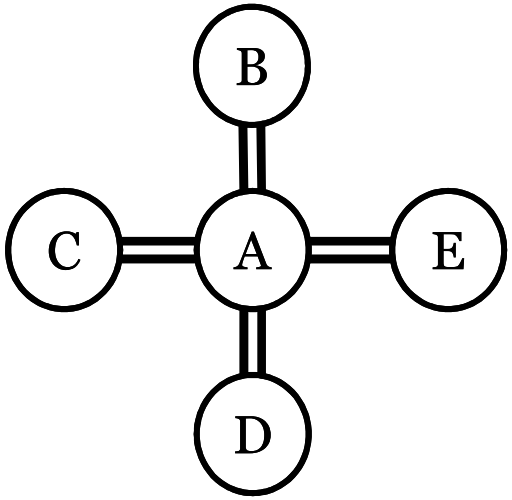
Direct Product Edges

$$E(G_x) = \{\big((a, b), (c, d)\big) | (a, c) \in E_1 and\ (b, d) \in E_2\}$$

Direct Product Graph - example



Type-A



Type-B

Type-A	A	B	C	D
A	0	1	1	0
B	1	0	0	1
C	1	0	0	1
D	0	1	1	0

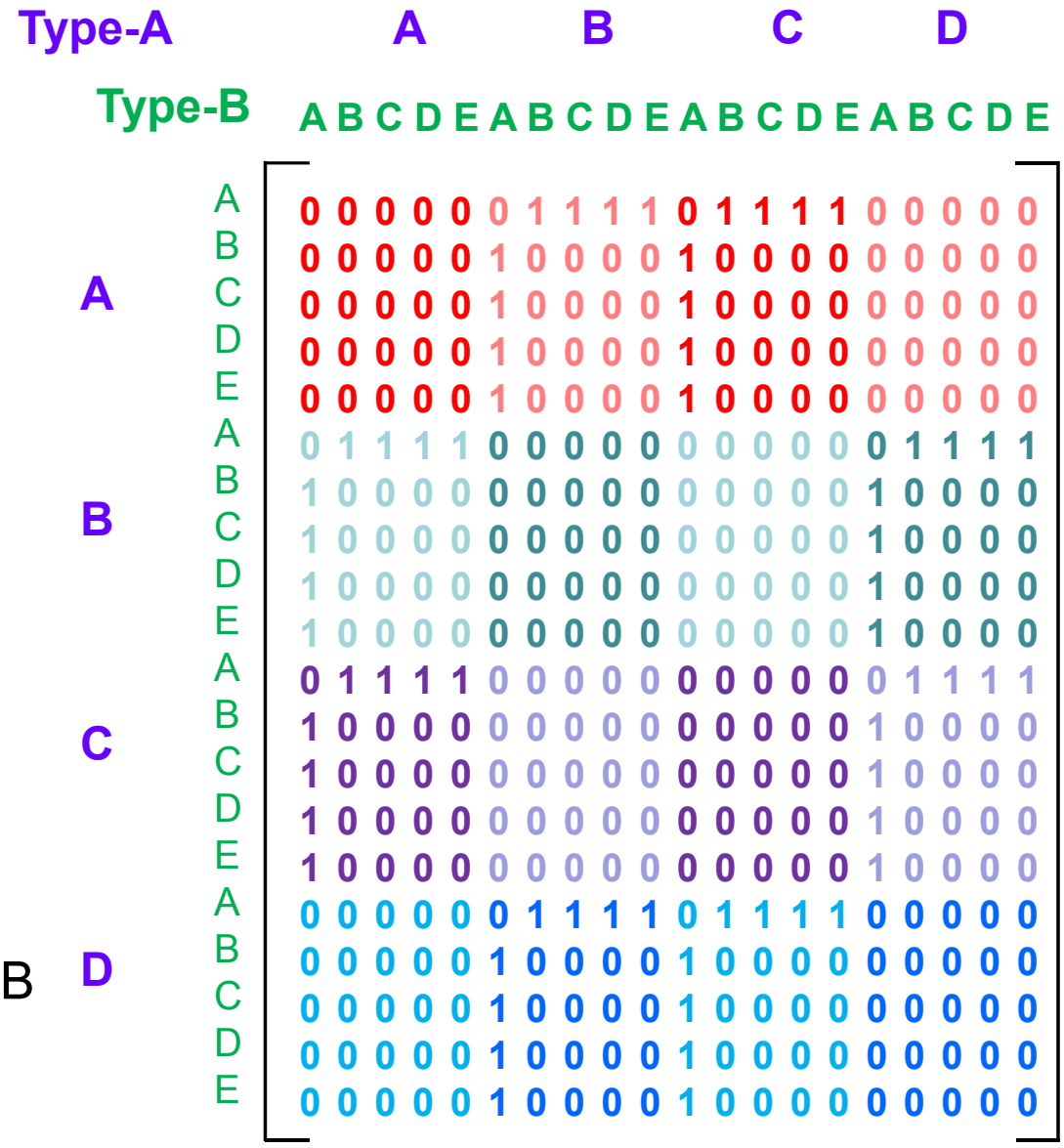
Type-B	A	B	C	D	E
A	0	1	1	1	1
B	1	0	0	0	0
C	1	0	0	0	0
D	1	0	0	0	0
E	1	0	0	0	0

Direct Product Graph Example

Type-A	A	B	C	D
A	0	1	1	0
B	1	0	0	1
C	1	0	0	1
D	0	1	1	0

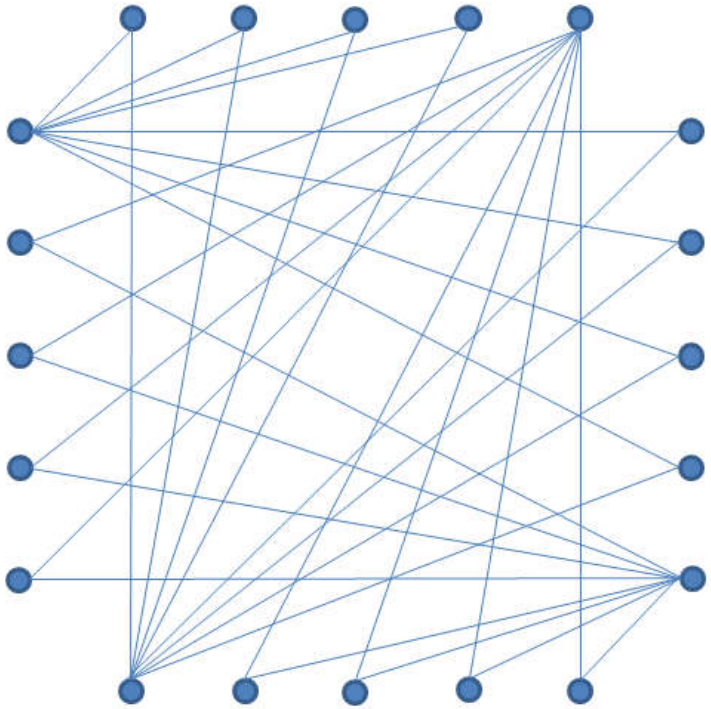
Type-B	A	B	C	D	E
A	0	1	1	1	1
B	1	0	0	0	0
C	1	0	0	0	0
D	1	0	0	0	0
E	1	0	0	0	0

Intuition: multiply each entry of Type-A by ***entire matrix*** of Type-B



Direct Product Kernel (see Kernel Chapter)

1. Compute direct product graph G_x
2. Compute the maximum in- and out-degrees of *Gx*, *di* and *do*.
3. Compute the decay constant
   $\gamma < 1 / \min(di, do)$
4. Compute the infinite weighted geometric series of walks (array *A*).
5. Sum over all vertex pairs.



Direct Product Graph of Type-A and Type-B

$$k(G_1, G_2) = \sum_{i,j} (I - \frac{A_{ij}}{\gamma})^{-1}$$

Kernel Matrix

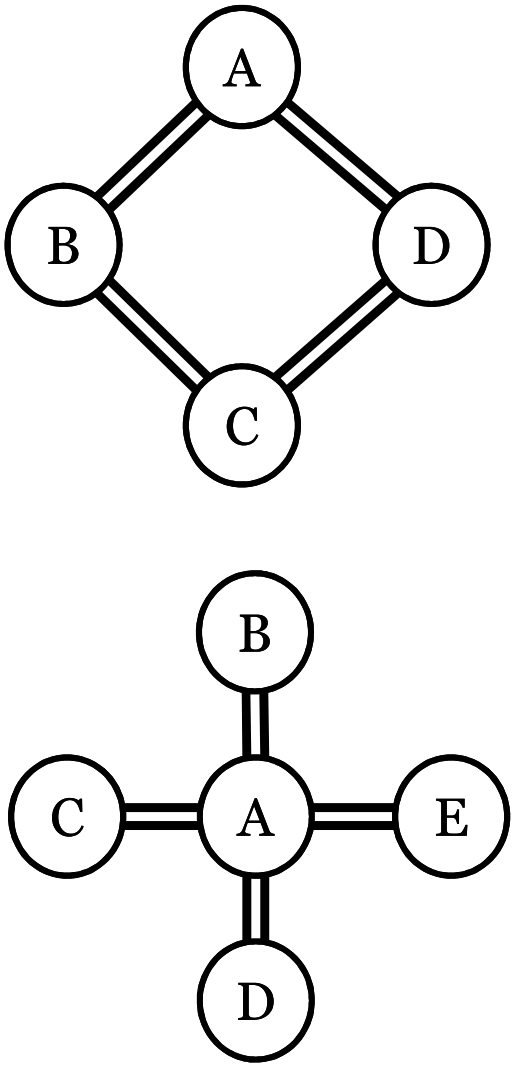
$$\begin{bmatrix} K(G_1, G_1), K(G_1, G_2), \ldots, K(G_1, G_n) \\ K(G_2, G_1), K(G_2, G_2), \ldots, K(G_2, G_n) \\ \ldots \\ K(G_n, G_1), K(G_n, G_2), \ldots, K(G_n, G_n) \end{bmatrix}$$

- **Compute direct product kernel for all pairs of graphs in the set of known examples.**
- **This matrix is used as input to SVM function to create the classification model.**
  - ***** Or any other kernelized data mining method!!!**

Predictive Toxicology (PTC) dataset

- The PTC dataset is a collection of molecules that have been tested positive or negative for toxicity.

```
1.  # R code to create the SVM model
2.  data("PTCData") # graph data
3.  data("PTCLabels") # toxicity information
4.  # select 5 molecules to build model on
5.  sTrain = sample(1:length(PTCData),5)
6.  PTCDataSmall <- PTCData[sTrain]
7.  PTCLabelsSmall <- PTCLabels[sTrain]
8.  # generate kernel matrix
9.  K = generateKernelMatrix (PTCDataSmall,
    PTCDataSmall)
10. # create SVM model
11. model =ksvm(K, PTCLabelsSmall,
    kernel='matrix')
```



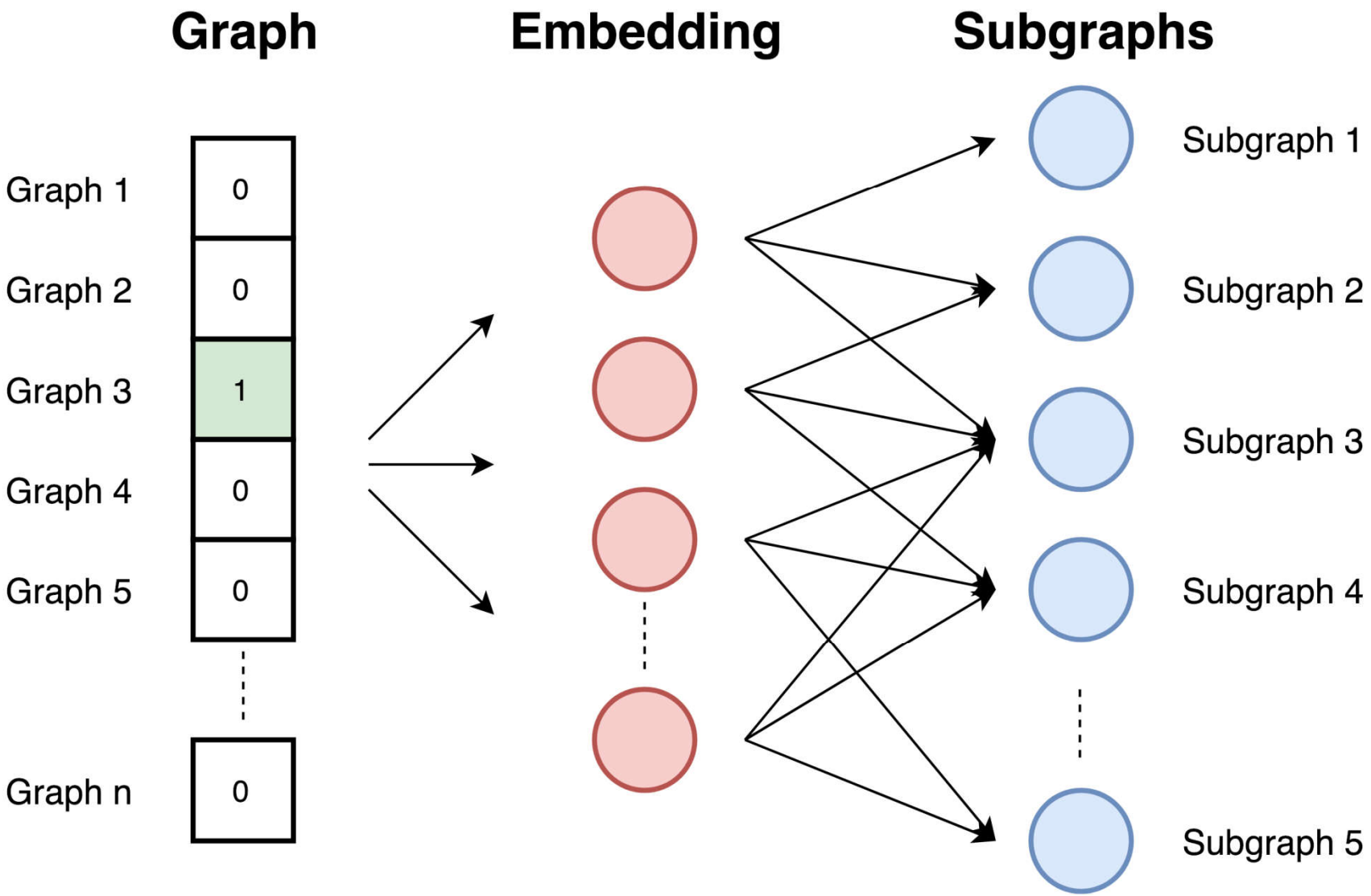
Graph embedding approach

- **The embedding-based approach embeds the whole graph. It computes one vector which describes a graph. The graph2vec approach is used since it is the best performing approach for a graph embedding.**
- **Graph2vec is based on the idea of the doc2vec approach that uses the skip-gram network. It gets an ID of the document on the input and is trained to maximize the probability of predicting random words from the document.**

Graph embedding approach

- **Graph2vec approaches consist of three steps:**
  - Sampling and relabeling all sub-graphs from the graph. Sub-graph is a set of nodes that appear around the selected node.
  - Training the skip-gram model. Graphs are similar to documents. Since documents are set of words graphs are set of sub-graphs. In this phase, the skip-gram model is trained. It is trained to maximize the probability of predicting sub-graph that exists in the graph on the input. The input graph is provided as a one-hot vector.
  - Computing embeddings with providing a graph ID as a one-hot vector at the input. Embedding is the result of the hidden layer.
- **Since the task is predicting sub-graphs, graphs with similar sub-graphs and similar structure have similar embeddings.**

Graph embedding approach



Kernels for Vertex Classification

- **Regularized Laplacian**

$$K = \sum_{i=1}^{\infty} \gamma^i (-L)^i$$

Example

- **Example: word-webpage graph**
  - **Vertex – webpage**
  - **Edge – set of pages containing same word**

$$Adjacency\ Matrix\ A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_{ij} = \begin{cases} 1, & \text{if and vertex } v_i \text{ belongs to edge } e_j \text{ in the hypergraph} \\ 0, & \text{otherwise.} \end{cases}$$

Laplacian Matrix

- **In the mathematical field of graph theory the Laplacian matrix (L), is a matrix representation of a graph.**
- **L = D – M**
  - **M – adjacency matrix of graph (e.g., A*A^T from hypergraph flattening)**
  - **D – degree matrix (diagonal matrix where each (i,i) entry is vertex i's [weighted] degree)**
- **Laplacian used in many contexts (e.g., spectral graph theory)**

$$AA^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 2 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 2 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

$$D = \sum_j [AA^T]_{ij}$$

$$D = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

$$L = D - AA^T$$

$$L = \begin{bmatrix} 3 & -1 & -1 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & -1 & -1 & 0 & 0 \\ -1 & -1 & 3 & -1 & 0 & 0 & 0 \\ -1 & -1 & -1 & 4 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix}$$

Normalized Laplacian Matrix

- ***Normalizing*** **the matrix helps eliminate bias in matrix toward high-degree vertices**

$$L_{i,j} := \begin{cases} 1 & \text{if } i = j \text{ and } \deg(v_i) \neq 0 \\ \dfrac{-1}{\sqrt{\deg(v_i)\deg(v_j)}} & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases}$$

Original L

$$L = \begin{bmatrix} 3 & -1 & -1 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & -1 & -1 & 0 & 0 \\ -1 & -1 & 3 & -1 & 0 & 0 & 0 \\ -1 & -1 & -1 & 4 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix}$$

Regularized L

$$L = \begin{bmatrix} 1.0 & -0.20 & -0.3 & -0.2 & 0.0 & 0.0 & 0.0 \\ -0.2 & 1.0 & -0.2 & -0.2 & -0.2 & 0.0 & 0.0 \\ -0.3 & -0.2 & 1.0 & -0.2 & 0.0 & 0.0 & 0.0 \\ -0.2 & -0.2 & -0.2 & 1.0 & -0.2 & 0.0 & 0.0 \\ 0.0 & -0.2 & 0.0 & -0.2 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & -0.5 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.5 & 1.0 \end{bmatrix}$$

Laplacian Kernel

- **Uses walk-based geometric series, only applied to regularized Laplacian matrix**
- **Decay constant NOT degree-based – instead tunable parameter < 1**

$$K = \sum_{i=1}^{\infty} \gamma^i (-L)^i$$

$$K = (I + \gamma L)^{-1}$$

$$L = \begin{bmatrix} 1.0 & -0.20 & -0.3 & -0.2 & 0.0 & 0.0 & 0.0 \\ -0.2 & 1.0 & -0.2 & -0.2 & -0.2 & 0.0 & 0.0 \\ -0.3 & -0.2 & 1.0 & -0.2 & 0.0 & 0.0 & 0.0 \\ -0.2 & -0.2 & -0.2 & 1.0 & -0.2 & 0.0 & 0.0 \\ 0.0 & -0.2 & 0.0 & -0.2 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & -0.5 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.5 & 1.0 \end{bmatrix}$$

Regularized L

WEBKB dataset

- The WEBKB dataset is a collection of web pages that include samples from four universities website.
- The web pages are assigned into five distinct classes according to their contents namely course, faculty, student, project and staff.
- The web pages are searched for the most commonly used words. There are 1073 words that are encountered at least with a frequency of 10.

```
1. # R code to create the SVM model
2. data(WEBKB)
3. # generate kernel matrix
4. K = generateKernelMatrixWithinGraph(WEBKB)
5. # create sample set for testing
6. holdout <- sample (1:ncol(K), 20)
7. # create SVM model
8. model =ksvm(K[-holdout,-holdout], y,
   kernel='matrix')
```

Node Embedding

- **In 2014, DeepWalk: Online Learning of Social Representations**
- Each graph as a document
- Random walks will be the sentences in this document
- Random Walk In graph G(V,E):
  - A sequences of nodes <v1, v2, …, vk>, such that each (vi, vi+1) is an edge in E

General method for node embedding

Algorithm 1 DeepWalk(G, w, d, γ, t)

Input: graph $G(V, E)$
window size w
embedding size d
walks per vertex γ
walk length t
Output: matrix of vertex representations $\Phi \in \mathbb{R}^{|V| \times d}$

Initialization: Sample Φ from $\mathcal{U}^{|V| \times d}$
Build a binary Tree T from V
for $i = 0$ to γ **do**
  $\mathcal{O} = \text{Shuffle}(V)$
  for each $v_i \in \mathcal{O}$ **do**
    $\mathcal{W}_{v_i} = RandomWalk(G, v_i, t)$
    SkipGram(Φ, $\mathcal{W}_{v_i}$, w)
  end for
end for

Algorithm 2 SkipGram(Φ, $\mathcal{W}_{v_i}$, w)

for each $v_j \in \mathcal{W}_{v_i}$ **do**
  for each $u_k \in \mathcal{W}_{v_i}[j - w : j + w]$ **do**
    $J(\Phi) = -\log \Pr(u_k \mid \Phi(v_j))$
    $\Phi = \Phi - \alpha * \frac{\partial J}{\partial \Phi}$
  end for
end for

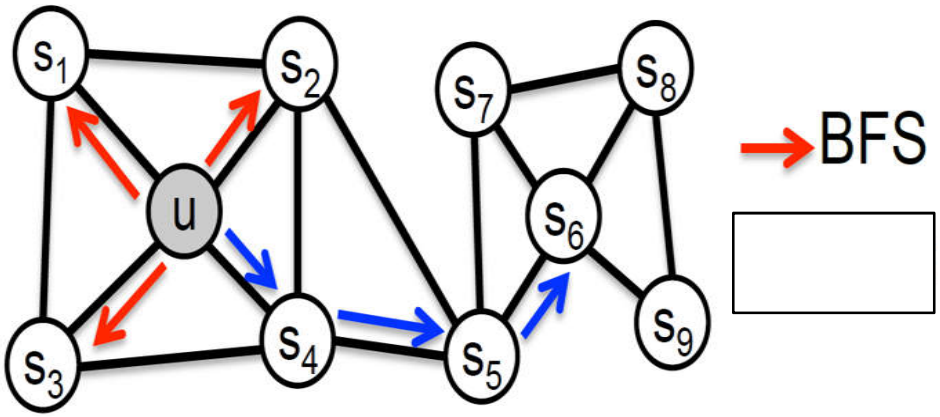
Classic Search Strategies

- **The problem of sampling neighborhoods N_S of a source node u can be viewed as a form of local search.**
- **There are two extreme sampling strategies for generating neighborhood sets:**
  - Breadth-first Sampling (BFS)
  - Depth-first Sampling (DFS)

Breadth-first Sampling (BFS)

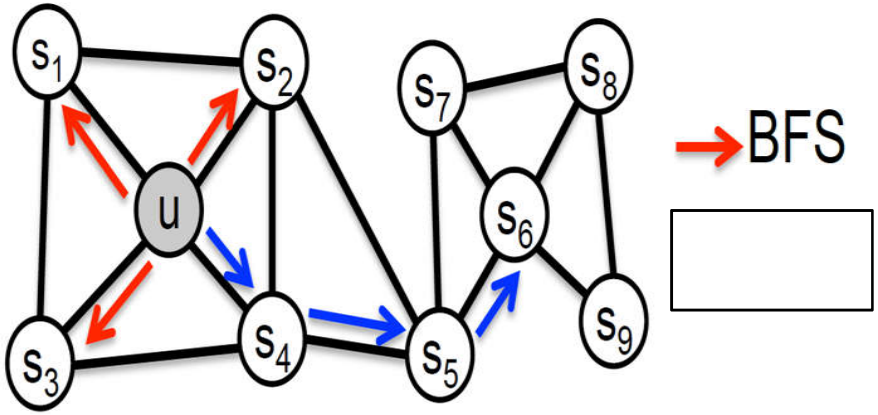
- **Neighborhood N_S is restricted to nodes which are immediate neighbors of the source u**
- **For a neighborhood of size $k = 3$ BFS samples nodes**

s_1, s_2, s_3



BFS⇨ Structural Equivalence

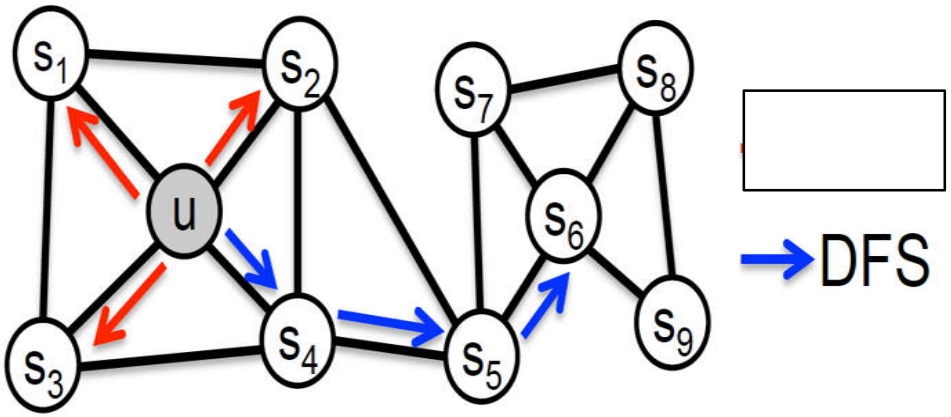
- **Nodes that have similar structural roles in networks should be embedded closely together.**
  - E.g., nodes u and s_6 in fig
- **Restricting search to nearby nodes, BFS gives microscopic view.**
- **Network roles such as bridges and hubs can be inferred using BFS.**



Depth-first Sampling (DFS)

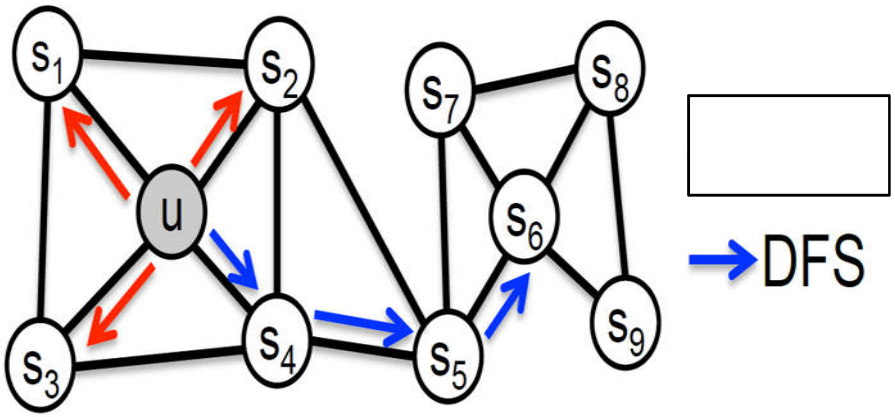
- **Neighborhood consists of nodes sequentially sampled at increasing distances from the source node** u
- **For a neighborhood of size** $k = 3$ **DFS samples nodes**

s_4, s_5, s_6



DFS ➪ Homophily

- **Nodes that are highly interconnected and belong to similar network communities should be embedded closely together.**
  - E.g., nodes u and s_1 in fig
- **DFS sampled nodes reflect a macro-view of nodes neighborhood.**



Flexible notion of neighborhood

- **Authors design a flexible neighborhood sampling strategy which allows them to smoothly interpolate between BFS and DFS**

- **The above sampling strategy is achieved by a flexible biased random walk that explores neighborhoods in a BFS as well as DFS fashion.**

Drawbacks of DeepWalk and LINE

- **DeepWalk: learns *d-dimensional* feature representations by simulating uniform random walks. Can be observed as a special case of *node2vec* with parameters *p = 1* & *q = 1*.**

- **LINE: learns *d-dimensional* feature representations in two steps:**
  - *d / 2* dimensions by **BFS-style**
  - *d / 2* dimensions by sampling nodes at **2-hop** distance

- **Networks represent a mixture of homophily and structural equivalence, which are not effectively covered by the above two methods.**

How Node2vec take random walks

Default Setup: Parameters Just the same as DeepWalk

- **Dimensionality (D) : 128**
- **Number of walks starting from each node (r) : 10**
- **Walk Length (l) : 80**
- **Context Size (k) : 10**

Each random walk:

- Step 1: initial node
- Step 2: look at the neighbors, select one as the next node
- Step 3: repeat step 2 until the length of random walk is equal l

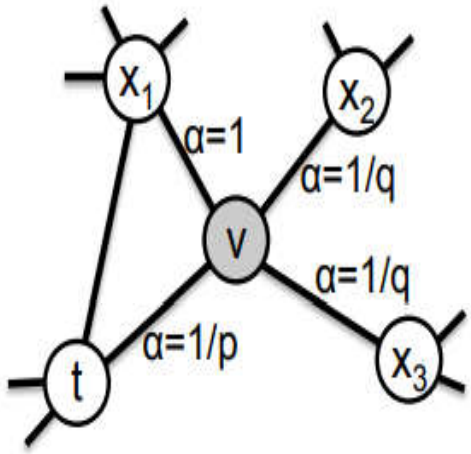
$$P(c_i = x \mid c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} & \text{if } (v, x) \in E \\ 0 & \text{otherwise} \end{cases}$$

Say that each node has degree at least one

Where node2vec really comes in

Biasing the selection of next node

- Last edge in random walk : **t** → **v**
- Currently at node **v**
- How to select next node from **v** neighbors ?



How much do you like to go back to **t** ? ⟶ parameter "**p**"

How much do you like to go far from **t** ? ⟶ parameter "**q**"

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases}$$

Overview of node2vec algorithm

Algorithm 1 The *node2vec* algorithm.

LearnFeatures (Graph $G = (V, E, W)$, Dimensions d, Walks per node r, Walk length l, Context size k, Return p, In-out q)
  π = PreprocessModifiedWeights(G, p, q)
  $G' = (V, E, \pi)$
  Initialize $walks$ to Empty
  for $iter = 1$ **to** r **do**
    for all nodes $u \in V$ **do**
      $walk$ = node2vecWalk(G', u, l)
      Append $walk$ to $walks$
  f = StochasticGradientDescent(k, d, $walks$)
  return f

node2vecWalk (Graph $G' = (V, E, \pi)$, Start node u, Length l)
  Inititalize $walk$ to $[u]$
  for $walk\_iter = 1$ **to** l **do**
    $curr = walk[-1]$
    V_{curr} = GetNeighbors($curr$, G')
    s = AliasSample(V_{curr}, π)
    Append s to $walk$
  return $walk$

Edge Embedding

Link prediction deals with pairs of nodes

We need to find embeddings of edges

Embedding of edge (u,v) done by a binary operator $g(u, v) : V \times V \longrightarrow R^d$

Operator	Symbol	Definition
Average	$\boxplus$	$[f(u) \boxplus f(v)]_i = \frac{f_i(u)+f_i(v)}{2}$
Hadamard	$\boxdot$	$[f(u) \boxdot f(v)]_i = f_i(u) * f_i(v)$
Weighted-L1	$\|\cdot\|_1$	$\|f(u) \cdot f(v)\|_{1i} = \|f_i(u) - f_i(v)\|$
Weighted-L2	$\|\cdot\|_2$	$\|f(u) \cdot f(v)\|_{2i} = \|f_i(u) - f_i(v)\|^2$

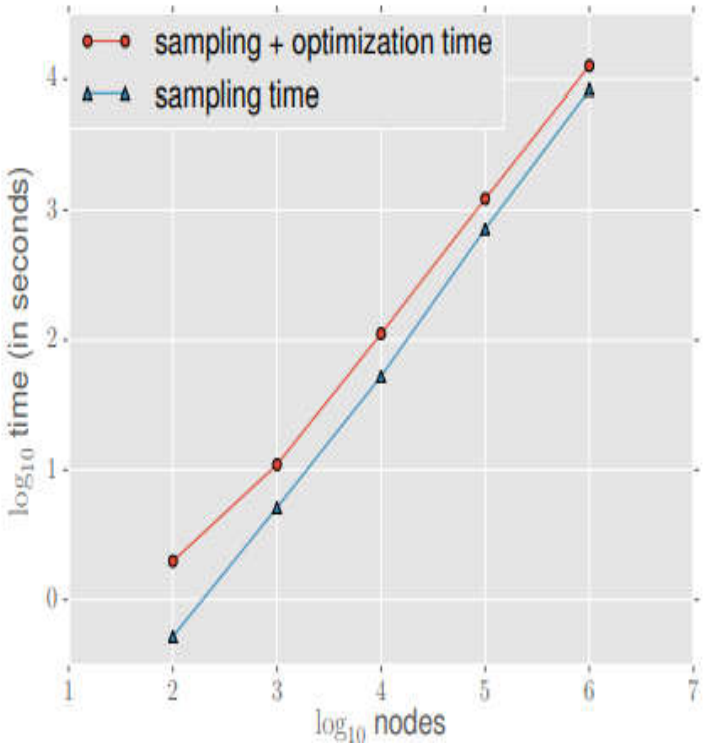
Node2vec Scalability

Process time is linear in number of nodes ⟶ O(a |V|)

- **a is constant relying on R (numWalks) and L (walkLength)**

For Optimization (SGD) use negative sampling

Example: Erdos-Renyi graphs with an average degree of 10



Traditional Classifications

- **Decision Trees**
  - Classification model → tree of conditionals on variables, where leaves represent class labels
  - Input space is typically a set of discrete variables
- **Bayesian belief networks**
  - Produces directed acyclic graph structure using Bayesian inference to generate edges.
  - Each vertex (a variable/class) associated with a probability table indicating likelihood of event or value occurring, given the value of the determined dependent variables.
- **Support Vector Machines**
  - Traditionally used in classification of real-valued vector data.
  - See Kernels chapter for kernel functions working on vectors.

Ensemble Classification

- **Ensemble learning: algorithms that build multiple models to enhance stability and reduce selection bias.**
- **Some examples:**
  - Bagging: Generate multiple models using samples of input set (with replacement), evaluate by averaging / voting with the models.
  - Boosting: Generate multiple *weak* models, weight evaluation by some measure of model accuracy.

Evaluating, Comparing Classifiers

- **Performance Metrics**
- **A very brief, "typical" classification workflow:**
  1. Partition data into *training*, *test* sets.
  2. Build classification model using only the training set.
  3. Evaluate accuracy of model using only the test set.
- **Modifications to the basic workflow:**
  - Multiple rounds of training, testing (cross-validation)
  - Multiple classification models built (bagging, boosting)
  - More sophisticated sampling (all)

Substructure-Based Graph Classification

- **Basic idea**
  - Extract graph substructures $F = \{g_1, \ldots, g_n\}$
  - Represent a graph with a feature vector $\mathbf{x} = \{x_1, \ldots, x_n\}$
    - where x_i is the frequency of g_i in that graph
  - Build a classification model

- **Different features and representative work**
  - Tree and cyclic patterns [Horvath et al.]
  - Minimal contrast subgraph [Ting and Bailey]
  - Frequent subgraphs [Deshpande et al.; Liu et al.]
  - Graph fragments [Wale and Karypis]

Direct Mining of Discriminative Patterns

- **Avoid mining the whole set of patterns**
  - Harmony [Wang and Karypis]
  - DDPMine [Cheng et al.]
  - LEAP [Yan et al.]
  - MbT [Fan et al.]
- **Find the most discriminative pattern**
  - A search problem?
  - An optimization problem?
- **Extensions**
  - Mining top-k discriminative patterns
  - Mining approximate/weighted discriminative patterns

Graph Kernels

- **Motivation:**
  - Kernel based learning methods doesn't need to access data points
    - They rely on the kernel function between the data points
  - Can be applied to any complex structure provided you can define a kernel function on them

- **Basic idea:**
  - Map each graph to some significant set of patterns
  - Define a kernel on the corresponding sets of patterns

Kernel-based Classification

- Random walk
  - Basic Idea: count the matching random walks between the two graphs
  - Marginalized Kernels
    - Gärtner '02, Kashima et al. '02, Mahé et al.'04

$$K(G_1, G_2) = \sum_{h_1} \sum_{h_2} p(h_1) p(h_2) K_L(l(h_1), l(h_2))$$

- h_1 and h_2 are paths in graphs G_1 and G_2
- $p(h_1)$ and $p(h_2)$ are probability distributions on paths
- $K_L(l(h_1), l(h_2))$ is a kernel between paths, e.g.,

$$K_L(l_1, l_2) = \begin{cases} 1 & \text{if } l_1 = l_2, \\ 0 & otherwise. \end{cases}$$