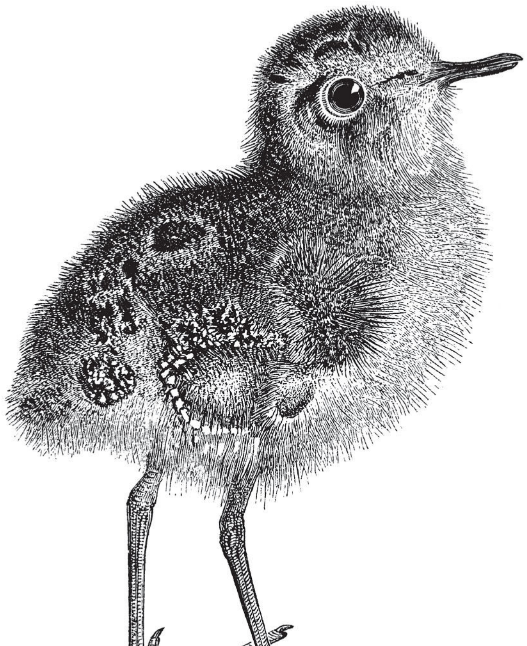
O'REILLY®

Graph-Powered Analytics and Machine Learning with TigerGraph

Driving Business Outcomes with Connected Data



Early Release
Raw & Unedited
Sponsored by
TigerGraph

Victor Lee, PhD,
Xinyu Chang &
Gaurav Deshpande

TigerGraph

Graph-Powered Analytics and Machine Learning

CONNECT all internal and external datasets and pipelines

ANALYZE connected data for deeper insights and better business outcomes

LEARN from the connected data to improve business performance over time with machine learning

Get started for free at
https://www.tigergraph.com/cloud/

Graph-Powered Analytics and Machine Learning with TigerGraph

Driving Business Outcomes with Connected Data

Victor Lee, PhD, Xinyu Chang, and Gaurav Deshpande

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Graph-Powered Analytics and Machine Learning with TigerGraph
by Victor Lee, Xinyu Chang, and Gaurav Deshpande

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Acquisitions Editor: Rebecca Novack
Development Editor: Nicole Tache
Production Editor: Daniel Elfanbaum
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

May 2022: First Edition

Revision History for the Early Release
2021-04-12: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098106652* for release details.

978-1-098-10658-4

[LSI]

Table of Contents

CHAPTER 1

Graph-Powered Machine Learning Methods

After completing this chapter, you should be able to:

- List three basic ways that graph data and analytics can improve machine learning
- Point out which graph algorithms have proved valuable for unsupervised learning
- Extract graph features to enrich your training data for supervised machine learning
- Describe how neural networks have been been extended to learn on graphs
- Provide use cases and examples to illustrate graph-powered machine learning
- Choose which types of graph-powered machine learning are right for you

We now begin the third theme of our book: Learn. That is, we're going to get serious about the core of machine learning: model training. Let's review the machine learning pipeline that we introduced in Chapter 1. For convenience, we've shown it here again, as Figure 1-1. In the first part of this book, which explored the Connect theme. That theme fits the first two stages of the pipeline: data acquisition and data preparation. Graph databases make it easy to pull data from multiple sources into one connected database and to perform entity resolution.

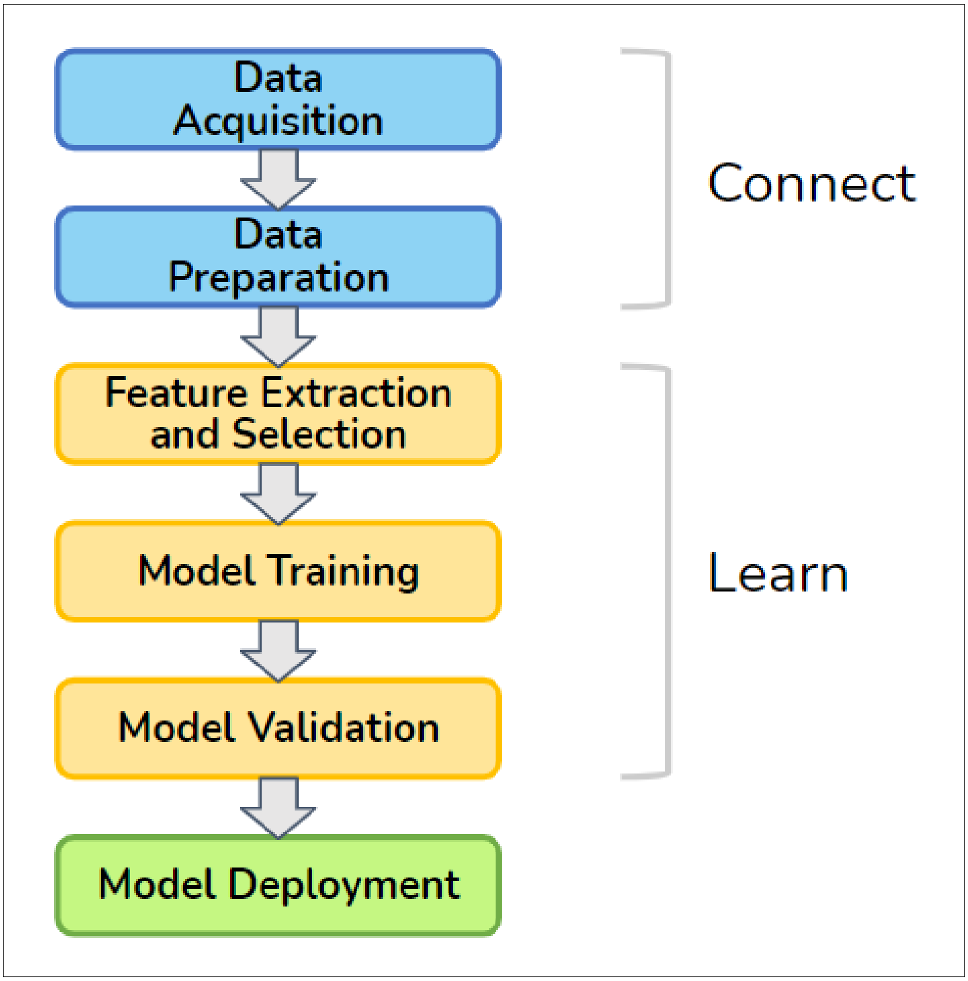


Figure 1-1. Machine learning pipeline

In this chapter, we'll show how graphs enhance the central stages in the pipeline: feature extraction and all-important model training. Features are simply the characteristics or properties of your data entities, like the age of a person or the color of a sweater. Graphs offer a whole new realm of features which are based on how an entity connects to other entities. The addition of these unique graph-oriented features provides machine learning with better raw materials with which to build its models.

This chapter has four sections. The first three sections each describe a different way that graphs enhance machine learning. First, we will start with unsupervised learning, as it is similar to the techniques we discussed in the Analyze section of the book. Second, we will turn to graph feature extraction for supervised and unsupervised machine learning. Third, we culminate with model training directly on graphs, for both supervised and unsupervised approaches. This includes techniques for clustering, embedding, and neural networks. Because they work directly on graphs, some graph databases are now available to offer in-database machine learning. The fourth section reviews the various methods in order to compare them and to help you decide which approaches will meet your needs.

Unsupervised Learning with Graph Algorithms

Unsupervised learning is the forgotten sibling of supervised learning and reinforcement learning, who together form the three major branches of machine learning. If you want your AI system to learn how to do a task, to classify things according to your categories, or to make predictions, you want to use supervised learning and/or reinforcement learning. Unsupervised learning, however, has the great advantage of being self-sufficient and ready-to-go. Unlike supervised learning, you don't need to already know the right answer for some cases. Unlike reinforcement learning, you don't have to be patient and forgiving as you stumble through learning by public trial and error. Unsupervised learning simply takes the data you have and reports what it learned

Think of each unsupervised learning technique as a specialist who is confident in their own ability and specialty. We call in the specialist, *they* tell us something that we hadn't noticed before, and we thank them for their contribution. An unsupervised learning algorithm can look at your network of customers and sales and tell you what are your actual market segments, which may not fit simple ideas of age and income. An unsupervised learning algorithm can point out customer behavior that is an outlier, far from normal, by determining "normal" from your data, not from your preconceptions. For example, an outlier can point out which customers are likely to churn (choose a competitor's product or service).

The first way we will learn from graph data is by applying graph algorithms to discover patterns or characteristics of our data. Earlier in the book, in the Analyze section, we took our first detailed look at graph algorithms. We'll now look at additional families of graph algorithms which go deeper. The line between Analyze and Learn can be fuzzy. What characterizes these new algorithms is that they look for or deal with patterns. They could all be classified as pattern discovery or data mining algorithms for graphs.

Finding Communities

When people are allowed to form their own social bonds, they will naturally divide into groups. No doubt you observed this, in school, in work, in recreation, and in online social networks. We form or join communities. However, graphs can depict any type of relationship, so it can be valuable to know about other types of communities in your data. Use cases include:

- Finding a set of highly interconnected transactions among a set of parties / institutions / locations / computer networks which are out of the ordinary: is this a possible financial crime?
- Finding pieces of software / regulation / infrastructure which are highly connected and interdependent. It is good that they are connected, as a sign of popularity, value, and synergy? Or it is a concerning sign that some components are unable to stand on their own as expected?
- In the sciences (ecology, cell biology, chemistry), finding a set of highly interacting species / proteins is evidence of (inter)dependence and can suggest explanations for such dependence.

Community detection can be used in connection with other algorithms. For example, first use a PageRank or centrality algorithm to find an entity of interest (e.g., a known or suspected criminal). Then find the community to which they belong. You could also swap the order. Find communities first, then count how many parties of note are within that community. Again, whether that is healthy or not depends on the context. Algorithms are your tools, but you are the craftsperson.

What is a Community?

In network science, a *community* is a set of vertices that have a higher density of connections within the group than to outside the group. Specifically, we look at the *edge density* of the community, which is the number of edges between members divided by the number of vertices. A significant change in edge density provides a natural boundary between who is in and who is not. Network scientists have defined several different classes of communities based on how densely they are interconnected. Higher edge density implies a more resilient community.

The following three types of communities span the range from weakest density to strongest density and everything in between.

Connected Component
: You are in if you have a direct connection to one or more members of the community. Requiring only one connection, this is the weakest possible density. In graph theory, if you simply say "component," it refers to a connected community plus the edges that connect them.

K-Core
: You are in if you have a direct connection to k or more members of the community, where *k* is a positive integer.

Clique

You are in if you have a direct connection to every other member of the community. This is the strongest possible density. Note that clique has a specific meaning in network science and graph theory.

A connected component is a k-core where $k = 1$. A clique containing c vertices is a k-core where $k = (c - 1)$, the largest possible value for k.

Figure 1-2 illustrates examples of these three definitions of community. On the left, there are three separate connected components, represented by the three different shadings. In the center, there are two separate k-cores for k = 2. The lower community, the rectangle with diagonal connections, would also satisfy $k = 3$ because each member vertex connects to three others. On the right, there are two separate cliques.

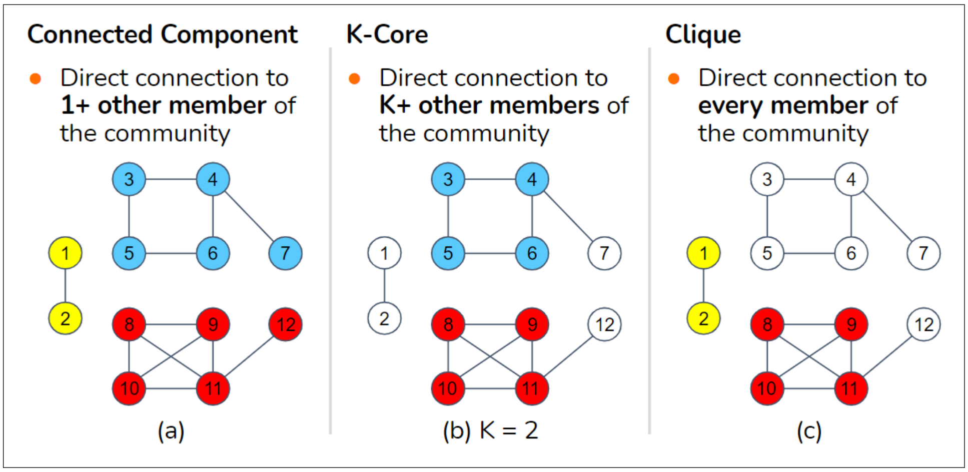

Connected Component
Direct connection to 1+ other member of the community
K-Core
Direct connection to K+ other members of the community
Clique
Direct connection to every member of the community
1
2
3
4
5
6
7
8
9
10
11
12
(a)
(b) K = 2
(c)

Figure 1-2. Three definitions of community, from weakest at left to strongest at right

Modularity-Based Communities

In many applications, however, we don't have a specific value of k in mind. In the spirit of unsupervised learning, we want the data itself to tell us what is dense enough. To address this, network scientists came up with a measure called *modularity* which looks at relative density, comparing the density within communities versus the density of connections between communities. This is like looking at the density of streets within cities versus the road density between cities. Now imagine the city boundaries are unknown; you just see the roads. If you propose a set of city boundaries, modularity will rate how good a job you did of maximizing the goal of "dense inside; not dense outside." Modularity is a scoring system, not an algorithm. A modularity-based community algorithm finds the community boundaries which produce the highest modularity score.

The Louvain algorithm, named after the University of Louvain, is perhaps the fastest algorithm for finding community boundaries with a very high modularity. It works hierarchically by forming small local communities, then substituting each community with a single meta-vertex, and repeating by forming local communities of meta-vertices.

To measure modularity (*Q*), we first partition the vertices into a set of communities so that every vertex belongs to one community. Then, considering each edge as one case, we calculate some totals and averages:

Q =[actual fraction of edges that fall within a community]
minus [expected fraction of edges if edges were distributed at random]

"Expected" is used in the statistical sense: If you flip a coin many times, you expect 50/50 odds of heads vs. tails. Modularity can handle weighted edges by using weights instead of simple counts:

Q = [average weight of edges that fall within a community]
minus [expected weight of edges if edges were distributed at random]

Note that the average is taken over the total number of edges. Edges which run from one community to another add zero to the numerator and add their weight to the denominator. It is designed this way so that edges which run between communities hurt your average and lower your modularity score.

What do we mean by "distributed at random"? Each vertex v has a certain number of edges which connect to it: the *degree* of a vertex is *d(v) = total number (or total weight)* of *v*'s edges. Imagine that each of these d(v) edges picks a random destination vertex. The bigger *d(v)* is, the more likely that one or more of those random edges will make a connection to a particular destination vertex. The expected (i.e., statistical average) number of connections between a vertex *v1* and a vertex *v2* can be computed with Equation 1-1.

Equation 1-1. Expected number of connections between two vertices based on their degrees

$$E_{rand}[wt(v1, v2)] = \frac{d(v1)d(v2)}{2m}$$

where m is the total number of edges in the graph. For the mathematically inclined, the complete formula is shown in Equation 1-2 below, where wt(i,j) is the weight of the edge between i and j, com(i) is the community ID of vertex i, and δ(a,b) is the Kronecker delta function which equals 1 if a and b are equal, 0 otherwise.

Equation 1-2. Definition of modularity

$$Q = \frac{1}{2m} \sum_{i, j \in G} \left[wt(i, j) - \frac{d(i)d(j)}{2m} \right] \delta(comm(i), comm(j))$$

Finding Similar Things

Being in the same community is not necessarily an indicator of similarity. For example, a fraud ring is a community, but it requires different persons to play different roles. Moreover, one community can be composed of different types of entities. For example, a grassroots community centered around a sports team could include persons, businesses selling fan paraphernalia, gathering events, blog posts, etc. If you want to know which entities are similar, you need a different algorithm. Fans who attend the same events and buy the same paraphernalia intuitively seem similar; to be sure, we need a well-defined way of assessing similarity.

Efficient and accurate measurement of similarity is one of the most important tools for businesses because it is what underlies personalized recommendation. We have all seen recommendations from Amazon or other online vendors suggest that we consider a product because "other persons like you bought this." If the vendor's idea of "like you" is too broad, they may make some silly recommendations which annoy you. They want to use several characteristics they know or infer about you and your interests to filter down the possibilities, and then see *what similar customers did*. Similarity is used not only for recommendations but also for following up after detecting a situation of special concern, such as a security vulnerability or a crime of some sort. We found this case. Are there similar cases out there?

Simply finding similar things does not necessarily qualify as machine learning. The first two types of similarity we will look at are important to know, but they aren't really machine learning. These types are graph-based similarity and neighborhood similarity. They lead to the third type, role similarity, which can be considered unsupervised machine learning. Role similarity looks deeply and iteratively in the graph to refine its understanding of the graph's structural patterns.

Similarity values are a prerequisite for at least three other machine learning tasks, clustering, classification, and link prediction. *Clustering* is putting things that are close together into the same group. "Close" does not necessarily mean physically close together; it means…similar. People often confuse communities with clusters. Communities are based on density of connections; clusters are based on similarity of the entities. *Classification* is deciding which of several predefined categories an item should be placed into. A common meta-approach is to say that an item should be put in the same category as that of similar items which have a known category. If things that look like and walk like it are ducks, then we conclude that it is a duck. *Link prediction* is making a calculated guess that a relationship, which does not currently exist in the dataset, either does exist or will exist in the real world. In many scenarios, entities which have similar neighborhoods (circumstances) are likely to have similar relationships. The relationship could be anything: a person-person relationship, a financial transaction, or a job change. Investigators, actuaries, gamblers, and marketers all seek to do accurate link prediction. As we can see, similarity pops up as a prerequisite for numerous machine learning tasks.

Graph-Based Similarity

What makes two things similar? We usually identify similarities by looking at observable or known properties: color, size, function, etc. A passenger car and a motorcycle are similar because they are both motorized vehicles for one or a few passengers. A motorcycle and a bicycle are similar because they are both two-wheeled vehicles. But, how do we decide whether a motorcycle is more similar to a car or to a bicycle? For that matter, how would we make such decisions for a set of persons, products, medical conditions, or financial transactions? We need to agree upon a system for measuring similarity. But, to take an unsupervised approach, is there some way that we can let the graph itself suggest how to measure similarity?

A graph can give us contextual information to help us decide how to determine similarity. Consider the axiom

> A vertex is characterized by its properties, its relationships, and its neighborhood.

Therefore, if two vertices have similar properties, similar relationships, and similar neighborhoods, then they should be similar. What do we mean by similar neighborhoods? Let's start with a simpler case: the exact same neighborhood.

> Two entities are similar if they connect to the same neighbors.

In Figure 1-3, Person A and Person B have three identical types of edges (purchased, hasAccount, and knows), connecting to the three exact same vertices (Phone model Y, Bank Z, and Person C, respectively). It is not necessary, however, to include all types of relationships or to give them equal weight. If you care about social networks, for example, you can consider only friends. If you care about financial matters, you can consider only financial relationships.

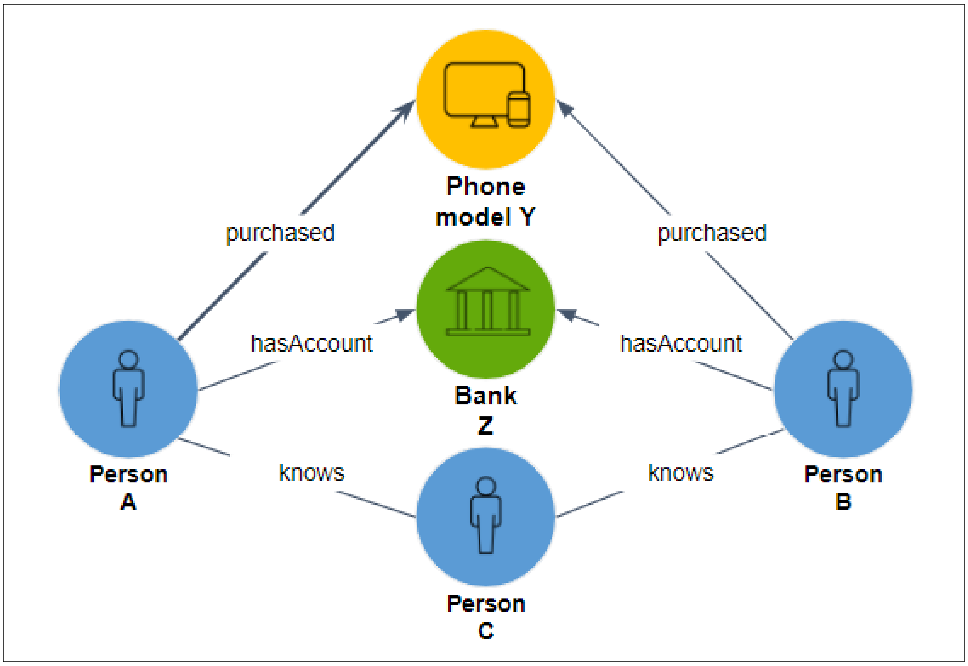


Figure 1-3. Two persons sharing the same neighborhood

To reiterate the difference between community and similarity: All five entities in Figure 1-3 are part of a connected component *community*. However, we would not say that all five are *similar* to one another. The only case of similarity suggested by these relationships is Person A being similar to Person B.

Neighborhood Similarity

It's rare to find two entities which have exactly the same neighborhoods. We'd like a way to measure and rank the degree of neighborhood similarity. The two most common measures for ranking neighborhood similarity are Jaccard similarity and cosine similarity.

Jaccard Similarity. Jaccard similarity measures the relative overlap between two general sets. Suppose you run Bucket List Travel Advisors, and you want to compare your most frequent travelers to one another based on which destinations they have visited. Jaccard similarity would be a good method for you to use; the two sets would be the destinations visited by each of two customers being compared. To formulate Jaccard similarity in general terms, suppose the two sets are N(a), the neighborhood of vertex a, and N(b), the neighborhood of vertex b. Equation 1-3 gives a precise definition of neighborhood-based Jaccard similarity:

Equation 1-3. Jaccard neighborhood similarity

$$\begin{aligned} jaccard(a,b) &= \frac{number\_of\_shared\_neighbors}{size(N(a)) + size(N(b)) - number\_of\_shared\_neighbors)} \\ &= \frac{number\_of\_shared\_neighbors}{number\_of\_unique\_neighbors} \\ &= \frac{|N(a) \cap N(b)|}{|N(a) \cup N(b)|} \end{aligned}$$

The maximum possible score is 1, which occurs if a and b have exactly the same neighbors. The minimum score is 0, if they have no neighbors in common.

Consider the following example. Three travelers, A, B, and C, have traveled to the following places as shown in Table 1-1.[1]

Table 1-1. Dataset for Jaccard similarity example

Bucket List's Top 10	A	B	C
Amazon Rainforest, Brazil	✓	✓	
Grand Canyon, USA	✓	✓	✓
Great Wall, China	✓		✓
Machu Picchu, Peru	✓	✓	
Paris, France	✓		✓
Pyramids, Egypt		✓	
Safari, Kenya		✓	
Taj Mahal, India			✓
Uluru, Australia		✓	
Venice, Italy	✓		✓

Using the table's data, the Jaccard similarities for each pair of travelers can be computed.

- A and B have three destinations in common (Amazon, Grand Canyon, Machu Picchu). Collectively they have been to nine destinations. jaccard(A, B) = 3/9 = 0.33.
- B and C have only one destination in common (Grand Canyon). Collectively they have been to ten destinations. jaccard(B, C) = 1/10 = 0.10.

1 The use of a table for our small example may suggest that a graph structure is not needed. We assume you already decided to organize your data as a graph and now are analyzing and learning from that data.

- A and C have three destinations in common (Grand Canyon, Paris, Venice). Collectively they have been to seven destinations. jaccard(A, C) = 3/7 = 0.43.

Among these three, A and C are the most similar. As the proprietor, you might suggest that C visit some of the places that A has visited, such as the Amazon and Machu Picchu. Or you might try to arrange a group tour, inviting both of them to somewhere they both have not been to, such as Uluru, Australia.

Cosine Similarity. Cosine similarity measures the alignment of two sequences of numerical characteristics. The name comes from the geometric interpretation in which the numerical sequence is the entity's coordinates in space. The data points on the grid (the other type of "graph") in Figure 1-4 illustrate this interpretation.

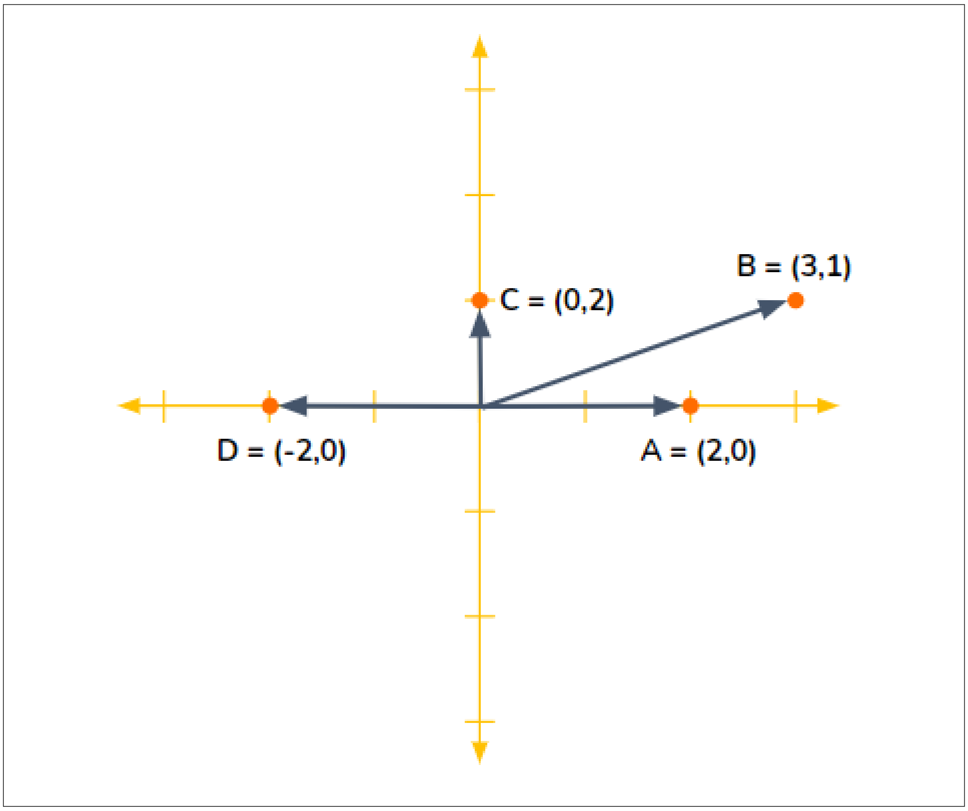


Figure 1-4. Geometric interpretation of numeric data vectors.

Point A represents an entity whose feature vector is (2,0). B's feature vector is (3,1). Now we see why we call a list of property values a "vector." The vectors for A and B are somewhat aligned. The cosine of the angle between them is their similarity score. If two vectors are pointed in exactly the same direction, the angle between them is 0;

the cosine of their angle is 1. cos(A,C) is 0 because A and C are perpendicular; the vectors (2,0) and (0,2) have nothing in common. cos(A,D) is -1 because A and D are pointed in opposite directions. So, cos(x,y) = 1 for two perfectly similar entities, 0 for two perfectly unrelated entities, and -1 for two perfectly anticorrelated entities.

Suppose you have scores across several categories or attributes for a set of entities. You want to cluster the entities, so you need an overall similarity score between entities. The scores could be ratings of individual features of products, employees, accounts, etc. Let's continue the example of Bucket List Travel Advisors. This time, each customer has rated their enjoyment of a destination on a scale of 1 to 10, so we have numerical values, not just yes/no.

Table 1-2. Dataset for cosine similarity example

Bucket List's Top 10	A	B	C
Amazon Rainforest, Brazil	8		
Grand Canyon, USA	10	6	8
Great Wall, China	5		8
Machu Picchu, Peru	8	7	
Paris, France	9		4
Pyramids, Egypt		7	
Safari, Kenya		10	
Taj Mahal, India			10
Uluru, Australia		9	
Venice, Italy	7		10

Here are steps for using this table to compute cosine similarity between pairs of travelers.

1. List all the possible neighbors and define a standard order for the list so we can form vectors. We will use the top-down order in Table 10-1, from Amazon to Venice.
2. If the graph has D possible neighbors, this gives each vertex a vector of length D. For Table 10-2, D = 10. Each element in the vector is either the edge weight, if that vertex is a neighbor, or the null score if it isn't a neighbor.
3. Determining the right null score is important so that your similarity scores mean what you want them to mean. If a 0 means someone absolutely hated a destination, it is wrong to assign a 0 if someone has not visited a destination. A better approach is to normalize the scores. You can either normalize by entity (traveler), by neighbor/feature (destination), or by both. The idea is to replace the empty cells with a default score. You could set the default to be the average destination, or you could set it a little lower than that, because not having visited a place is a weak vote against that place. For simplicity, we won't normalize the scores; we'll

just use 6 as the default rating. Then traveler A's vector is *Wt(A)* = [8, 10, 5, 8, 9, 6, 6, 6, 6, 7].

4. Then, apply the formula in Equation 1-4 for cosine similarity:

Equation 1-4. Cosine neighborhood similarity

$$cosine(a, b) = \frac{Wt(a)\dot{W}(b)}{\| Wt(a) \| \ \| Wt(b) \|} = \frac{\Sigma_{i=1}^{D} Wt(a)_i Wt(b)_i}{\sqrt{\Sigma_{i=1}^{D} Wt(a)_i^2}\sqrt{\Sigma_{i-1}^{D} Wt(b)_i^2}}$$

Wt(a) and *Wt(b)* are the neighbor connection weight vectors for *a* and *b*, respectively. The numerator goes element by element in the vectors, multiplying the weight from a by the weight from *b*, then adding together these products. The more that the weights align, the larger the sum we get. The denominator is a scaling factor, the Euclidean length of vector *Wt(a)* multiplied by the length of vector *Wt(b)*.

Let's look at one more use case, people who rate movies, to compare how Jaccard and cosine similarity work. In Figure 1-5, we have two persons A and B who have each rated three movies. They have both rated two of the same movies, Black Panther and Frozen.

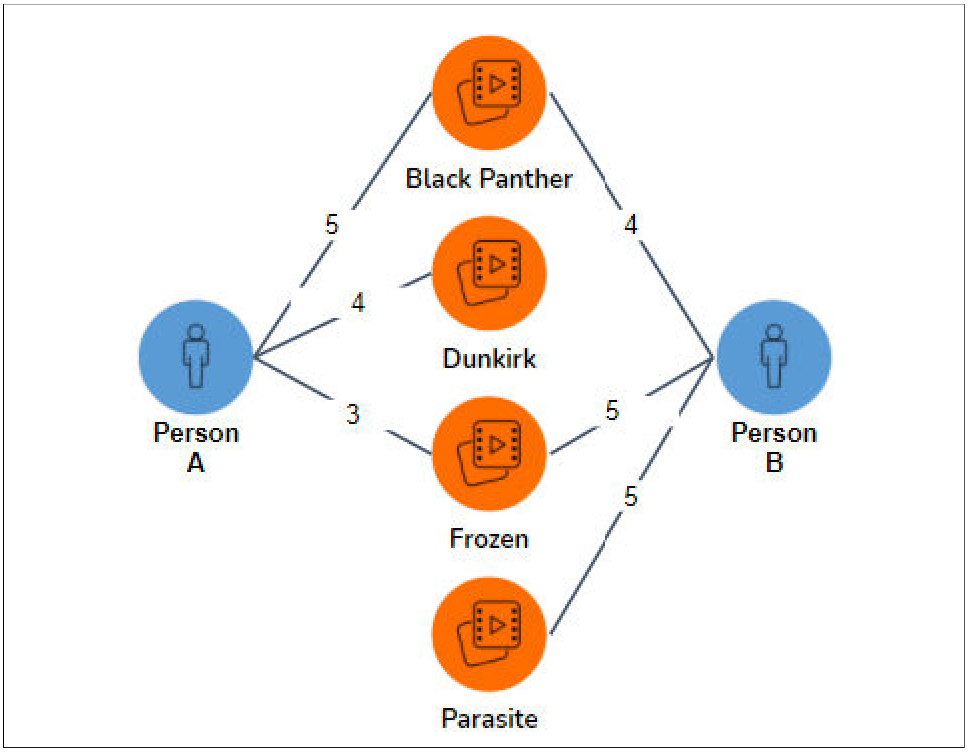


Figure 1-5. Similarity of persons who rate movies

If we only care about what movies the persons have seen and not about the scores, then Jaccard similarity is sufficient and easier to compute. This would also be your choice if scores were not available or if the relationships were not numeric. The Jaccard similarity is (size of overlap) / (size of total set) = 2 / 4 = 0.5. That seems like a middling score, but if there are hundreds or thousands of possible movies they could have seen, then it's a very high score.

If we want to take the movie ratings into account to see how similar are A's and B's taste, we should use cosine similarity. Assuming the null score is 0, then A's neighbor score vector is [5, 4, 3, 0] and B's is [4, 0, 5, 5]. For cosine similarity, the numerator is [5, 4, 3, 0] • [4, 0, 5, 5] = (5)(4)+(4)(0)+(3)(5)+(0)(5) = 35. The denominator = $\sqrt{5^2 + 4^2 + 3^2 + 0^2}\sqrt{4^2 + 0^2 + 5^2 + 5^2} = \sqrt{50}\sqrt{66} = 57.446$. The final result is 0.60927. This again seems like a reasonably good score, not strong similarity, but much better than a random pairing.

Use Jaccard similarity when the features of interest are yes/no or categorical variables.

Use cosine similarity when you have numerical variables. If you have both types, you can use cosine similarity and treat your yes/no variables as having values 1/0

Role Similarity

Earlier we said that if two vertices have similar properties, similar relationships, and similar neighborhoods, then they should be similar. We first examined the situation in which the neighborhoods contained some of the exact same members, but let's look at the more general scenario in which the individual neighbors aren't the *same*, just similar.

Consider a graph of family relationships. One person, Jordan, has two living parents, is married to someone who was born in a different country, and they have three children together. Another person, Kim, also has two living parents, is married to someone born in another country, and together they have four children. None of the neighboring entities (parents, spouses, children) are the same. The number of children is similar but not exactly the same. Nevertheless, Jordan and Kim are similar because they have similar relationships. This is called role similarity.

Moreover, if Jordan's spouse and Kim's spouse are similar, that's even better. If Jordan's children and Kim's children are similar in some way (ages, hobbies, etc.), that's even better. You get the idea. Instead of people, these could be products, components in a supply chain or power distribution network, or financial transactions.

> Two entities have similar roles if they have similar relationships to entities which themselves have similar roles.

This is a recursive definition: A and B are similar if their neighbors are similar. Where does it stop? What is the base case where we can say for certain how much two things are similar?

SimRank. In their 2002 paper, Glen Jeh and Jennifer Widom proposed SimRank which measures similarity by having equal length paths from A and B both reach the same individual; e.g., if Jordan and Kim share a grandparent, that contributes to their SimRank score. The formal definition is shown in Equation 1-5 below.

Equation 1-5. SimRank

$$simrank(a,b) = \frac{C}{|In(a)In(b)|} \sum_{u \in In(a)} \sum_{v \in In(b)} simrank(In(u), In(v))$$

$$simrank(a,a) = 1$$

In(a) and *In*(b) are the sets of in-neighbors of vertices *a* and *b*, respectively, *u* is a member of *In(a)*, *v* is a member of *In(b)*, and *C* is a constant between 0 and 1 to control the rate at which neighbors' influence decreases as distance from a source vertex increases. Lower values of *C* mean a more rapid decrease. SimRank computes a N × N array of similarity scores, one for each possible pair. This differs from cosine and Jaccard similarity which compute a single pair's score on demand. It is necessary to compute the full array of SimRank scores because the value of *SimRank(a,b)* depends of the SimRank score of pairs of their neighbors (e.g., *SimRank(u,v)*), which in turn depends on *their* neighbors, and so on. To compute SimRank, you initialize the array so that *SimRank(a,b)* = 1 if *a* = *b*; otherwise it is 0. Then calculate a revised set of scores by applying Equation 1-5 for each pair (a,b) where the SimRank scores on the right side are from the previous iteration. Note that this is like PageRank's computation except that we have N × N scores instead of just N scores. SimRank's reliance on eventually reaching a shared individual, and through equal length paths, is too rigid for some cases. SimRank does not fully embrace the idea of role similarity. Two individuals in two completely separate graphs, e.g. graphs representing two unrelated families or companies, should be able to show role similarity.

RoleSim. To address this shortcoming, Ruoming Jin et al.[2] introduced RoleSim in 2011. RoleSim starts with the (over)estimate that the similarity between any two entities is the ratio of the sizes of their neighbors. The initial estimate for RoleSim(Jordan, Kim) would be 5/6. RoleSim then uses the current estimated similarity of your neighbors to make an improved guess for the next round. Equation 1-6 has the formal definition of RoleSim.

Equation 1-6. RoleSim

$$rolesim(a,b) = (1-\beta)\max_{M(a,b)} \frac{\sum_{(u,v)\in M(a,b)} rolesim(u,v)}{max(|N(u)|,|N(v)|)} + \beta$$

$$rolesim_0(a,b) = \frac{min(|N(u)|,|N(v)|)}{max(|N(u)|,|N(v)|)}$$

The parameter β is similar to SimRank's C. The main difference is the function M. M(a,b) is a *bipartite matching* between the neighborhoods of *a* and *b*. This is like trying to pair up the three children of Jordan with the four children of Kim. *M(Jordan, Kim)* will consist of three pairs (and one child left out). Moreover, there are 24 possible matchings. For computational purposes (not actual social dynamics), assume that the oldest child of Jordan selects a child of Kim; there are four options. The next child of Jordan picks from the three remaining children of Kim, and the third child of Jor-

2 One of the authors of this book, Victor Lee, is a coauthor of RoleSim.

dan can choose from the two remaining children of Kim. This yields (4)(3)(2) = 24 possibilities. The max term in the equation means that we select the matching that yields the highest sum of RoleSim scores for the three chosen pairs. If you think of RoleSim as a compatibility score, then we are looking for the combination of pairings that gives us the highest total compatibility of partners. You can see that this is more computational work than SimRank, but the resulting scores have nicer properties:

1. There is no requirement that the neighborhoods of a and b every meet.
2. If the neighborhoods of a and b "look" exactly the same, because they have the same size, and each of the neighbors can be paired up so that their neighborhoods look the same, and so on, then their RoleSim score will be a perfect 1. Mathematically, this level of similarity is called *automorphic equivalence*.

This idea of looking at neighbors of neighbors to get a deeper sense of similarity will come up again later in the chapter.

Finding Frequent Patterns

As we've said early on in this book, graphs are wonderful because they make it easy to discover and analyze multi-connection patterns based on connections. In the Analyze section of the book, we talked about finding particular patterns, and we will return to that topic later in this chapter. In the context of unsupervised learning, the goal is to:

> Find any and all patterns which occur frequently.

Computer scientists call this the Frequent Subgraph Mining task, because a pattern of connections is just a subgraph. This task is particularly useful for understanding natural behavior and structure, such as consumer behavior, social structures, biological structures, and even software code structure. However, it also presents a much more difficult problem. "Any and all" patterns in a large graph means a vast number of possible occurrences to check. The saving grace is the threshold parameter T. To be considered frequent, a pattern must occur at least T times. If T is big enough, that could rule out a lot of options early on. The example below will show how T is used to rule out small patterns so that they don't need to be considered as building blocks for more complex patterns.

There any many advanced approaches to try to speed up frequent subgraph mining, but the basic approach is to start with 1-edge patterns, keep the patterns that occur at least T times, and then try to connect those patterns to make bigger patterns:

1. Group all the edges according to their type and the types of their endpoint vertices. For example, Shopper-(bought)-Product is a pattern.
2. Count how many times each pattern occurs.

3. Keep all the frequent patterns (having at least T members) and discard the rest. For example, we keep Shopper-(lives_in)-Florida but eliminate Shopper-(lives_in)-Guam because it is infrequent.
4. Consider every pair of groups that have compatible vertex types (e.g., groups 1 and 2 both have a Shopper vertex), and see how many individual vertices in group 1 are also in group 2. Merge these individual small patterns to make a new group for the larger pattern. For example, we merge the cases where the same person in the frequent pattern Shopper-(bought)-Blender was also in the frequent pattern Shopper-(lives_in)-Florida.
5. Repeat steps 2 and 3 (filtering for frequency) for these newly formed larger patterns.
6. Repeat step 4 using the expanded collection of patterns.
7. Stop when no new frequent patterns have been built.

There is a complication with counting (step 2). The complication is *isomorphism*, how the same set of vertices and edges can fit a template pattern in more than one way. Consider the pattern A-(friend_of)-B. If Jordan is a friend of Kim, which implies that Kim is a friend of Jordan, is that one instance of the pattern or two? Now suppose the pattern is "find pairs of friends A and B who are both friends with a third person C." This forms a triangle. Let's say Jordan, Kim, and Logan form a friendship triangle. There are six possible ways we could assign Jordan, Kim, and Logan to the variables A, B, and C. You need to decide up front whether these types of symmetrical patterns should be counted separately or merged into one instance, and then make sure your counting method is correct.

Summary

Graph algorithms can perform unsupervised machine learning on graph data. Key takeaways from this section are as follows:

- Community detection algorithms find densely interconnected sets of entities.
- Neighborhood similarity algorithms find entities who have similar relationships to others.
- Clustering is grouping by nearness, and similarity is proxy for physical nearness.
- Link prediction uses clues such as neighborhood similarity to predict whether an edge is likely to exist between two vertices.
- Frequent pattern mining algorithms find patterns of connections which occur frequently.

Extracting Graph Features

In the previous section, we showed how you can use graph algorithms to perform unsupervised machine learning. In most of those examples, we analyzed the graph as a whole to discover some characteristics, such as communities or frequent patterns.

In this section, you'll learn how graphs can provide additional and valuable features to describe and help you understand your data. A *graph feature* is a characteristic that is based on the pattern of connections in the graph. A feature can either be local – attributed to the neighborhood of an individual vertex or edge – or global – pertaining to the whole graph or a subgraph. In most cases, we are interested in vertex features – characteristics of the neighborhood around a vertex. That's because vertices usually represent the real world entities which we want to model with machine learning.

When an entity (an instance of a real-world thing) has several features and we arrange those features in a standard order, we call the ordered list a *feature vector*. Some of the methods we'll look at in this section provide individual features; others produce entire sets of features. You can concatenate a vertex's entity properties (the ones that aren't based on connections) with its graph features obtained from one or more of the methods discussed here to make a longer, richer feature vector. We'll also look at a special self-contained feature vector called an *embedding* that summarizes a vertex's entire neighborhood.

These features can provide insight as is, but one of their most powerful uses is to enrich the training data for supervised machine learning. Feature extraction is one of the key phases in a machine learning pipeline (refer back to Figure 1-1). For graphs, this is particularly important because traditional machine learning techniques are designed for vectors, not for graphs. So, in a machine learning pipeline, feature extraction is also where we transform the graph into a different representation.

In the sections that follow, we'll look at three key topics: domain-independent features, domain-dependent features, and the exciting developments in graph embedding.

Domain-Independent Features

If graph features are new to you, the best way to understand them is to look at simple examples that would work for any graph. Because these features can be used regardless of the type of data we are modeling, we say they are domain-independent. Consider the graph in Figure 1-6. We see a network of friendships, and we count occurrences of some simple domain-independent graph features.

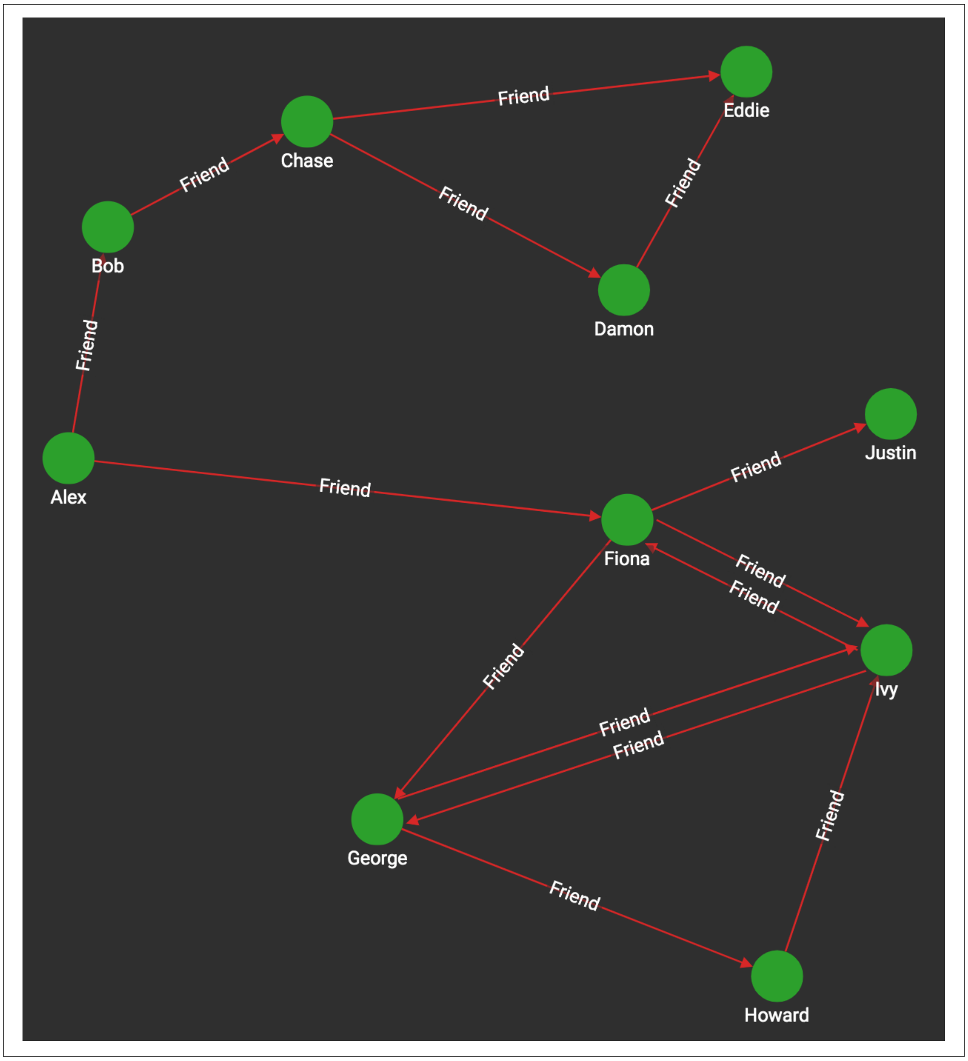


Figure 1-6. Graph with directed friendship edges

Table 1-3 shows the results for four selected vertices (Alex, Chase, Fiona, Justin) and four selected features.

Table 1-3. Examples of domain-independent features from the graph of Figure 1-6

	Number of in-neighbors	Number of out-neighbors	Number of vertices within 2 forward hops	Number of triangles (ignoring direction)
Alex	0	2 (Bob, Fiona)	6 (B, C, F, G, I, J)	0
Chase	1 (Bob)	2 (Damon, Eddie)	2 (D, E)	1 (Chase, Damon, Eddie)

Fiona	2 (Alex, Ivy)	3 (George, Ivy, Justin)	4 (G, I, J, H)	1 (Fiona, George, Ivy)
Justin	1 (Fiona)	0	0	0

You could easily come up with more features, by looking farther than one or two hops[3], by considering generic weight properties of the vertices or edges, and by calculating in more sophisticated ways: computing average, maximum, or other functions. Because these are domain-independent features, we are not thinking about the meaning of "person" or "friend". We could change the object types to "computers" and "sends data to". Domain-independent features, however, may not be the right choice for you if there are many types of edges with very different meanings.

Graphlets

Another option for extracting domain-independent features is to use graphlets[4]. *Graphlets* are small subgraph patterns, which have been systematically defined so that they include every possible configuration up to a given size limit. Figure 1-7 shows all 72 graphlets for subgraphs up to five vertices (or nodes). Note that the figure shows two types of identifiers: shape IDs (G0, G1, G2, etc.) and graphlet IDs (1, 2, 3, etc.). Shape G1 encompasses two different graphlets: graphlet 1, when the reference vertex is on the end of the 3-vertex chain, and graphlet 2, when the reference vertex is in the middle.

Counting the occurrences of every graphlet pattern around a given vertex provides a standardized feature vector which can be compared to any other vertex in any graph. This universal signature lets you cluster and classify entities based on their neighborhood structure, for applications such as predicting the world trade dynamics of a nation[5] or link prediction in dynamic social networks like Facebook.[6]

3 A hop is the generic unit of distance in a graph, from one vertex to its neighbor. We defined this in Chapter 1.

4 Graphlets were first presented by Nataša Pržulj et al. in "Modeling interactome: scale-free or geometric?".

5 "Graphlet-based Characterization of Directed Networks", Sarajlić et al., 2016.

6 "Link Prediction in Dynamic Networks using Graphlet", Rahmat et al., 2016.

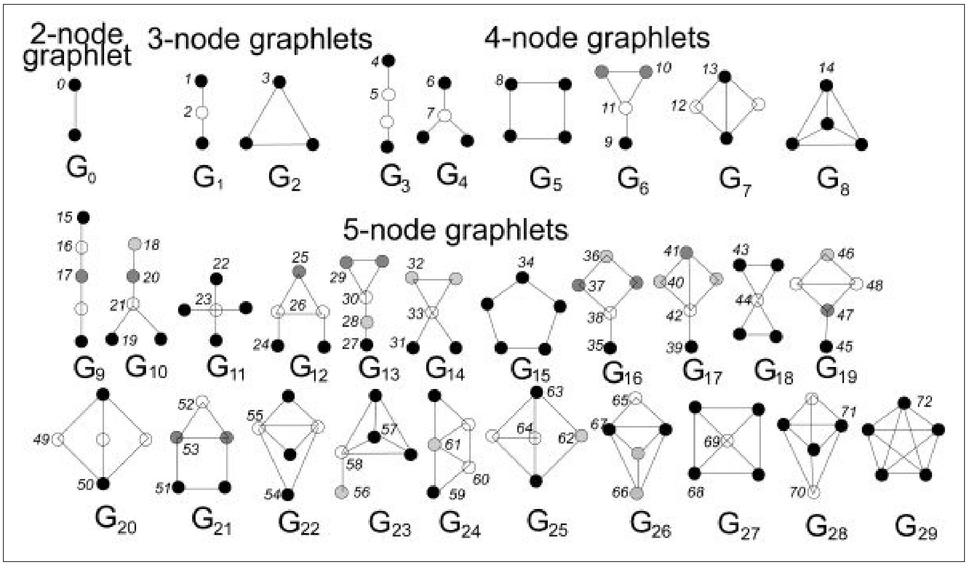


Figure 1-7. Graphlets up to five vertices (or nodes) in size[7]

A key rule for graphlets is that they are the *induced* subgraph for a set of vertices within a graph of interest. Induced means that they include ALL the edges that the base graph has among the selected set of vertices. This rule causes each particular set of vertices to match at most one graphlet pattern. For example, consider the four persons Fiona, George, Howard, and Ivy in Figure 1-6. Which shape and graphlet do they match, if any? It's shape G7, because those four persons form a rectangle with one cross connection. They do not match shape G5, the square because of that cross connection between George and Ivy. While we're talking about that cross connection, look carefully at the two graphlets for shape G7, graphlets 12 and 13. Graphlet 13's source node is located at one end of the cross connection, just as George and Ivy are. This means graphlet 13 is one of their graphlets. Fiona and Howard are at the other corners of the square, which don't have the cross connection. Therefore they have graphlet 12 in their graphlet portfolios.

There is obviously some overlap between the ad hoc features we first talked about (e.g., number of neighbors) and graphlets. Suppose a vertex A has three neighbors B, C, and D, as shown in Figure 1-8. However, we do not know about any other connections. What do we know about its graphlets?

7 From "Uncovering Biological Network Function via Graphlet Degree Signatures" by Tijana Milenković and Nataša Pržulj, licensed under CC BY 3.0

- It exhibits the graphlet 0 pattern three times. Counting the occurrences is important.
- Now consider subgraphs with three vertices.We can define three different subgraphs containing A: (A, B, C), (A, B, D), and (A, C, D). Each of those threesomes satisfies either graphlet 2 or 3. Without knowing about the connections among B, C, and D (the dotted-line edges in the figure), we can be more specific.
- Considering all four vertices, we might be tempted to say they match graphlet 7. Since there might be other connections between B, C, and D, it might actually be a different graphlet. Which one? Graphet 11 if there's one peripheral connection, graphlet 13 if it's two connections, or graphlet 14 if it's all three possible connections.

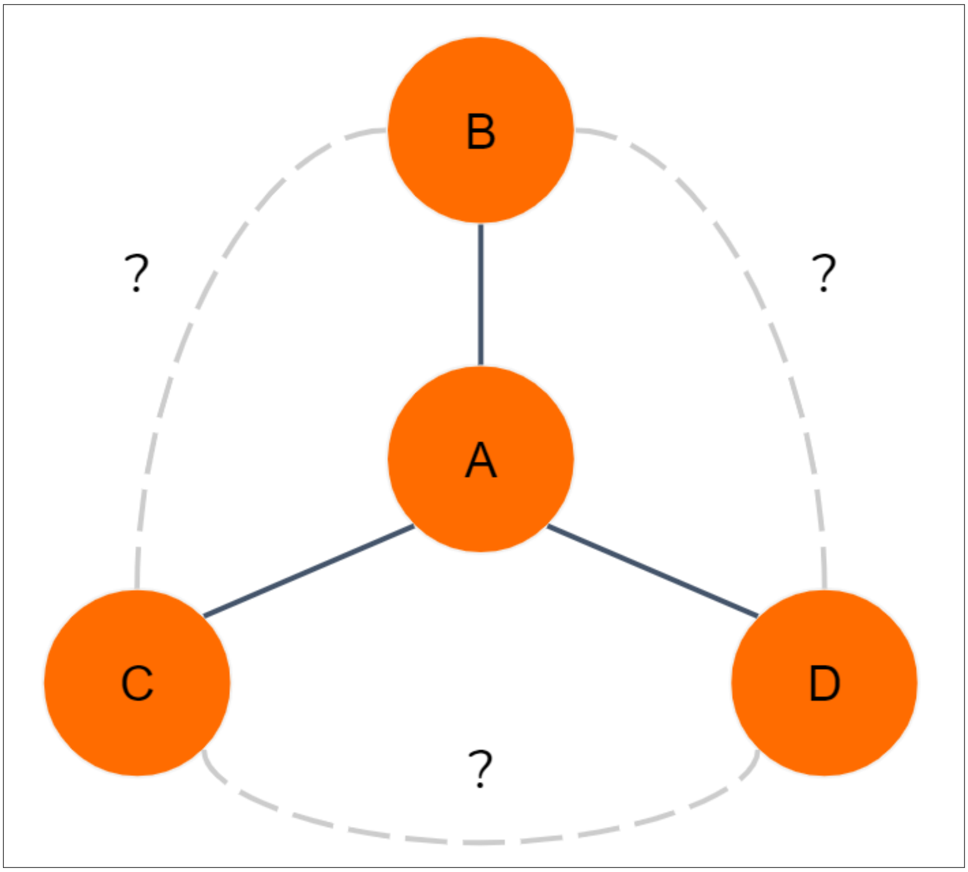


Figure 1-8. Immediate neighbors and graphlet implications

The advantage of graphlets is that they are thorough and methodical. Checking for all graphlets up to 5-node size is equal to considering all the details of the source vertex's

4-hop neighborhood. You could run an automated graphlet counter without spending time and money to design custom feature extraction. The disadvantage of graphlets is that they can require a lot of computational work, and it might be more productive to focus on a more selective set of domain-dependent features. We'll cover these types of features shortly.

Graph Algorithms

Here's a third option for extracting domain-independent graph features: graph algorithms! In particular, the centrality and ranking algorithms which we discussed in the Analyze section of the book work well because they systematically look at everything around a vertex and produce a score for each vertex. Figures 10-9 and 10-10 show the PageRank and closeness centrality[8] scores respectively for the graph presented earlier in Figure 1-7. For example, Alex has a PageRank score of 0.15 while Eddie has a PageRank score of 1. This tells us that Eddie is valued by his peers much more than Alex. Eddie's ranking is due not only to the number of connections, but also to the direction of edges. Howard, who like Eddie has two connections and is at the far end of the rough "C" shape of the graph, has a PageRank score of only 0.49983 because one edge comes in and the other goes out.

8 These algorithms were introduced in the Analyze section of the book.

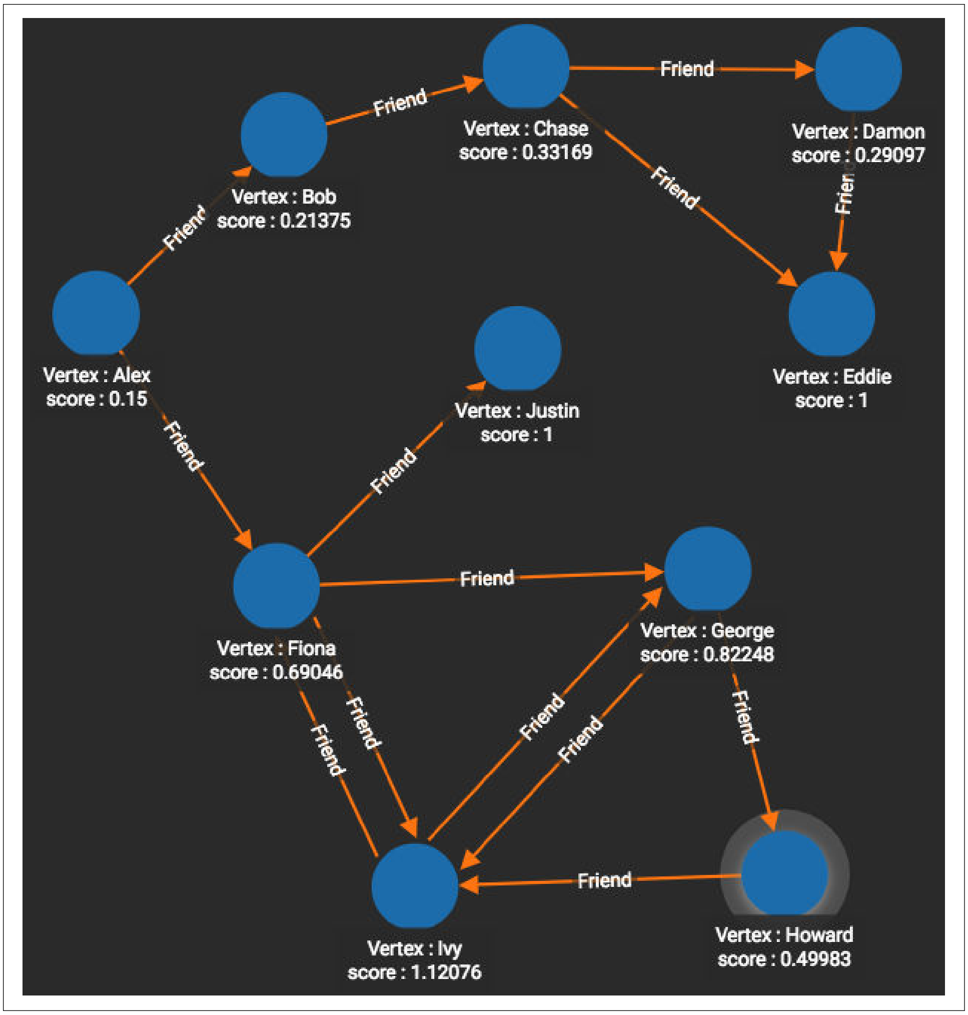


Figure 1-9. PageRank scores for the friendship graph.

The closeness centrality scores in Figure 1-10 tell a completely different story. Alex has a top score of 0.47368 because she is at the middle of the C. Eddie and Howard have scores at or near the bottom, 0.2815 and 0.29032 respectively, because they are at the ends of the C.

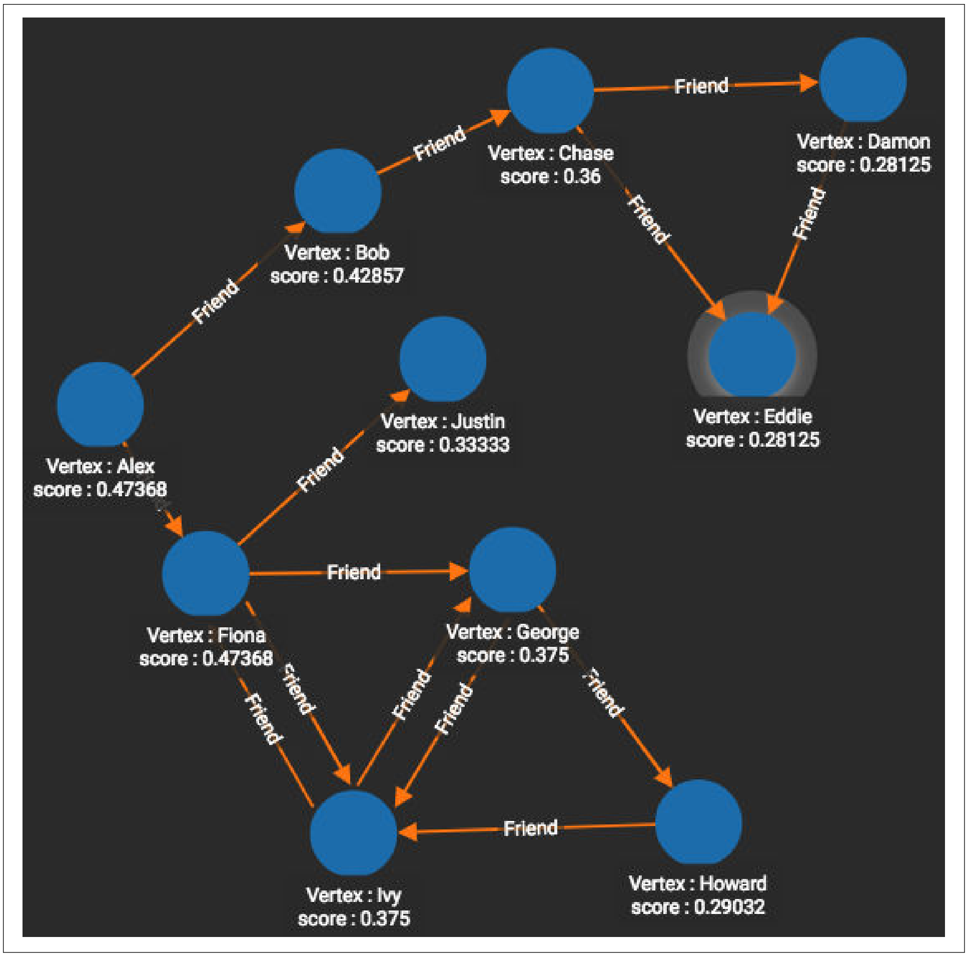


Figure 1-10. Closeness centrality scores for the friendship graph

The main advantage of domain-Independent feature extraction is its universality: generic extraction tools can be designed and optimized in advance, and the tools can be applied immediately on any data. Its unguided approach, however, can make it a blunt instrument.

Domain-independent feature extraction has two main drawbacks. Because it doesn't pay attention to what type of edges and vertices it considers, it can group together occurrences that have the same shape but have radically different meanings. The second drawback is that it can waste resources computing and cataloging features which have no real importance or no logical meaning. Depending on your use case, you may want to focus on a more selective set of domain-dependent features.

Domain-Dependent Features

A little bit of domain knowledge can go a long way towards making your feature extraction smarter and more efficient.

When extracting domain-dependent features, the first thing you want to do is pay attention to the vertex types and edge types in your graph. It's helpful to look at a display of your graph's schema. Some schemas break down information hierarchically into graph paths, such as City-(IN)-State-(IN)-Country or Day-(IN)-Month-(IN)-Year. This is a graph-oriented way of indexing and pre-grouping data according to location or date. This is the case in a graph model for South Korean COVID-19 contact tracing data, shown in Figure 1-11. While City-to-County and Day-to-Year are each 2-hop paths, those paths are simply baseline information and do not hold the significance of a 2-hop path like `Patient-(INFECTED_BY)-Patient-(INFECTED_BY)-Patient`.

You can see how the graphlet approach and other domain-independent approaches can provide confusing results when you have mixed edge types. A simple solution is to take a domain-semi-independent approach by considering only certain vertex types and edge types when looking for features. For example, if looking for graphlet patterns, you might want to ignore the month vertices and their connecting edges. You might still care about the Year of birth of Patients and the (exact) Day on which they traveled, but you don't need the graph to tell you that each Year contains twelve Months.

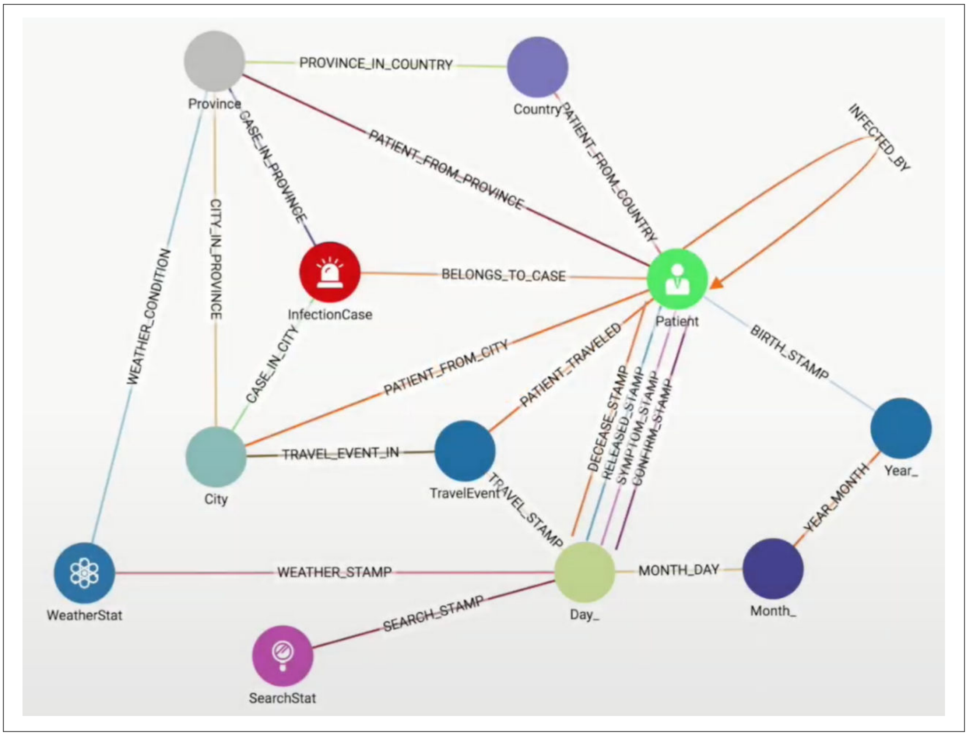


Figure 1-11. Graph schema for South Korean COVID-19 contact tracing data

With this vertex- and edge-type awareness you can refine some of the domain-independent searches. For example, while you can run PageRank on any graph, the scores will only have significance if all the edges have the same or relatively similar meanings. It would not make sense to run PageRank on the entire COVID-19 contact tracing graph. It would make sense, however, to consider only the Patient vertices and INFECTED_BY edges. PageRank would then tell you who was the most influential Patient in terms of causing infection: Patient zero, so to speak.

In this type of scenario, you also want to apply your understanding of the domain to think of small patterns with two or more edges of specific types that indicate something meaningful. For this COVID-19 contact tracing schema, the most important facts are Infection Status (InfectionCase), Who (Patient), Where (City and TravelEvent) and When (Day_). Paths that connect these are important. A possible feature is "number of travel events made by Patient P in March 2020." A more specific feature is "number of Infected Patients in the same city as Patient P in March 2020." That second feature is the type of question we posed in the Analyze section of our book. You'll find examples of vertex and edge type-specific PageRank and domain-dependent pattern queries in the TigerGraph Cloud Starter Kit for COVID-19.

Let's pause for a minute to reflect on your immediate goal for extracting these features. Do you expect these features to directly point out actionable situations, or are you building a collection of patterns to feed into a machine learning system? The machine learning system will then tell you which features matter, to what degree, and in what combination. If you're doing the latter, which is our focus for this chapter, then you don't need to build overly complex features. Instead, focus on building block features. Try to include some that provide a number (e.g., how many travel events) or a choice among several possibilities (e.g., city most visited).

To provide a little more inspiration for using graph-based features, here are some examples of domain-dependent features used in real-world systems to help detect financial fraud:

- How many shortest paths are there between a loan applicant and a blacklisted entity, up to a maximum path length (because very long paths represent negligible risk)?
- How many times has the loan applicant's mailing address, email address, or phone number been used by differently named applicants?
- How many credit card charges has this card made in the last 10 minutes?

While it is easy to see that high values on any of these measures make it more likely that a situation involves financial misbehavior, our goal is to be more precise and more accurate. We want to know what are the threshold values, and what about combinations of factors? Being precise cuts down on false negatives (missing cases of real fraud) and cuts down on false positives (labeling a situation as fraud when it really isn't). False positives are doubly damaging. They hurt the business because they are rejecting an honest business transaction, and they hurt the customer who has been unjustly labeled a crook.

Graph Embeddings: A Whole New World

Our last approach to feature extraction is graph embedding, a hot topic of research and discussion of late. Some authorities may find it unusual that I am classifying graph embedding as a type of feature extraction. Isn't graph embedding a kind of dimensionality reduction? Isn't it representation learning? Isn't it a form of machine learning itself? All of those are true. Let's first define what is a graph embedding.

An *embedding* is a representation of a topological object in a particular system such that the properties we care about are preserved (or approximated well). The last part "the properties we care about are preserved" speaks to why we use embeddings. A well-chosen embedding makes it more convenient to see what we want to see.

Here are several examples to help illustrate the meaning of embeddings:

- The earth is a sphere but we print world maps on flat paper. The representation of the earth on paper is an embedding. There are several different standard representations or embeddings of the earth as a map. Figure 1-12 shows some examples.
- Prior to the late 2010s, when someone said graph embedding, they probably meant something like the earth example. To represent all the connections in a graph without edges touching one another, you often need three or more dimensions. Whenever you see a graph on a flat surface, it's an embedding, as in Figure 1-13. Moreover, unless your data specifies the location of vertices, then even a 3-D representation is an embedding because it's a particular choice about the placement of the vertices. From a theoretical perspective, it takes up to n-1 dimensions to represent a graph with n vertices: huge!
- In natural language processing (NLP), a word embedding is a sequence of scores (i.e., a feature vector) for a given word (see Figure 1-14). There is no natural interpretation of the individual scores, but an ML program sets the scores so that words that tend to occur near each other in training documents have similar embeddings. For example, "machine" and "learning" might have similar embeddings. A word embedding is not convenient for human use, but it is very convenient for computer programs that need a computational way of understanding word similarities and groupings.

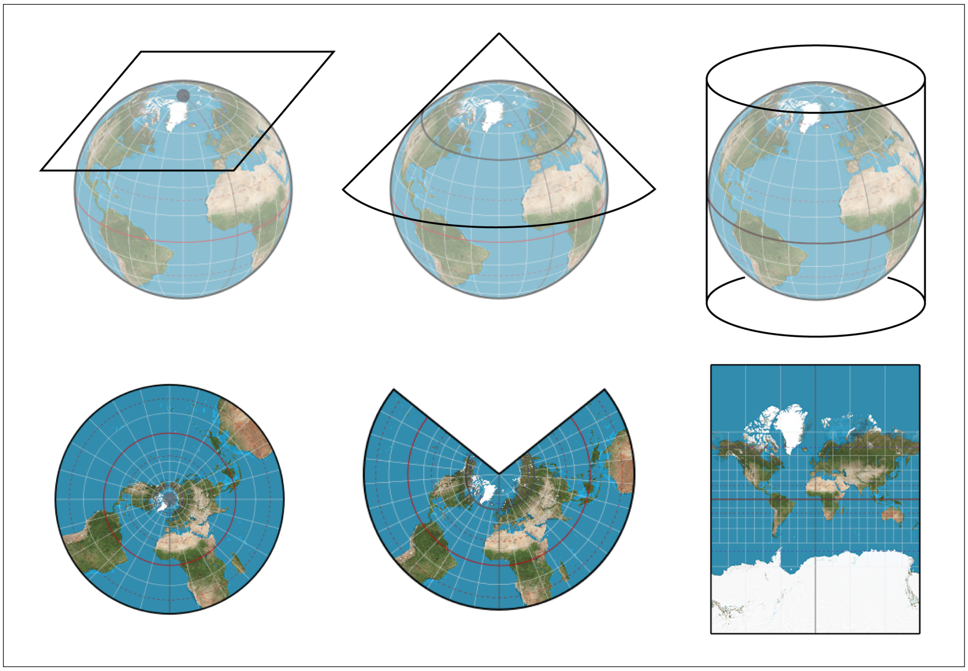


Figure 1-12. Two embeddings of the earth's surface onto 2-D space[9]

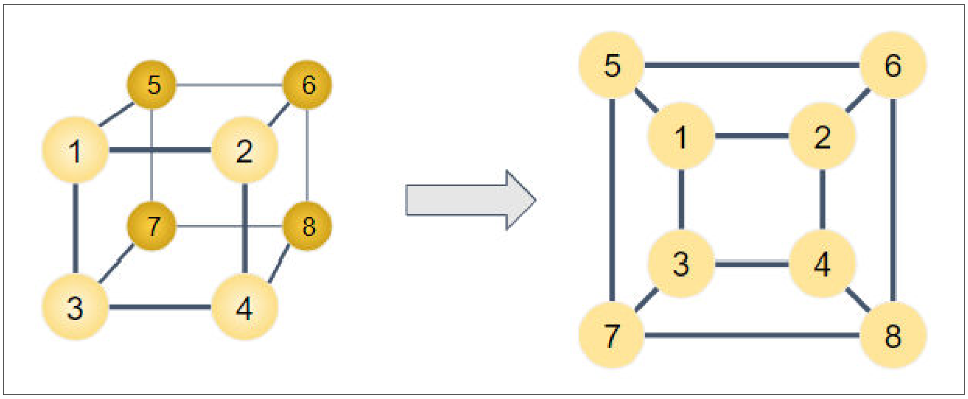
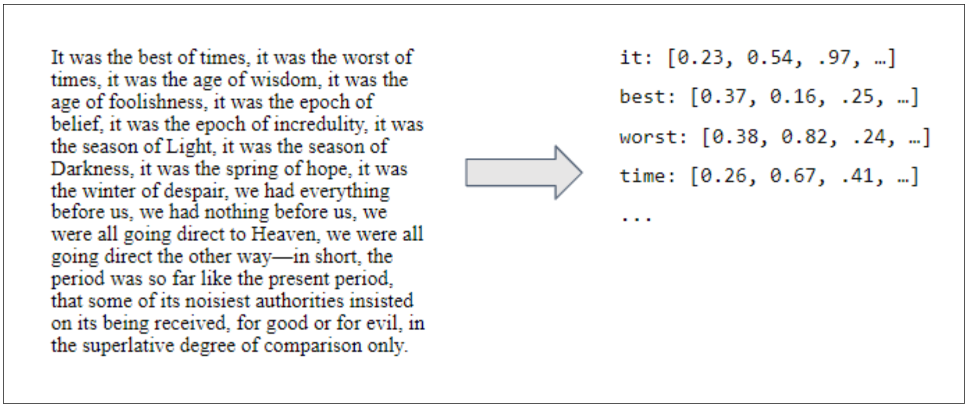


Figure 1-13. Some graphs can be embedded in 2-D space without intersecting edges

9 From Battersby, S. (2017). Map Projections. *The Geographic Information Science & Technology Body of Knowledge* (2nd Quarter 2017 Edition), John P. Wilson (ed.). DOI: 10.22224/gistbok/2017.2.7

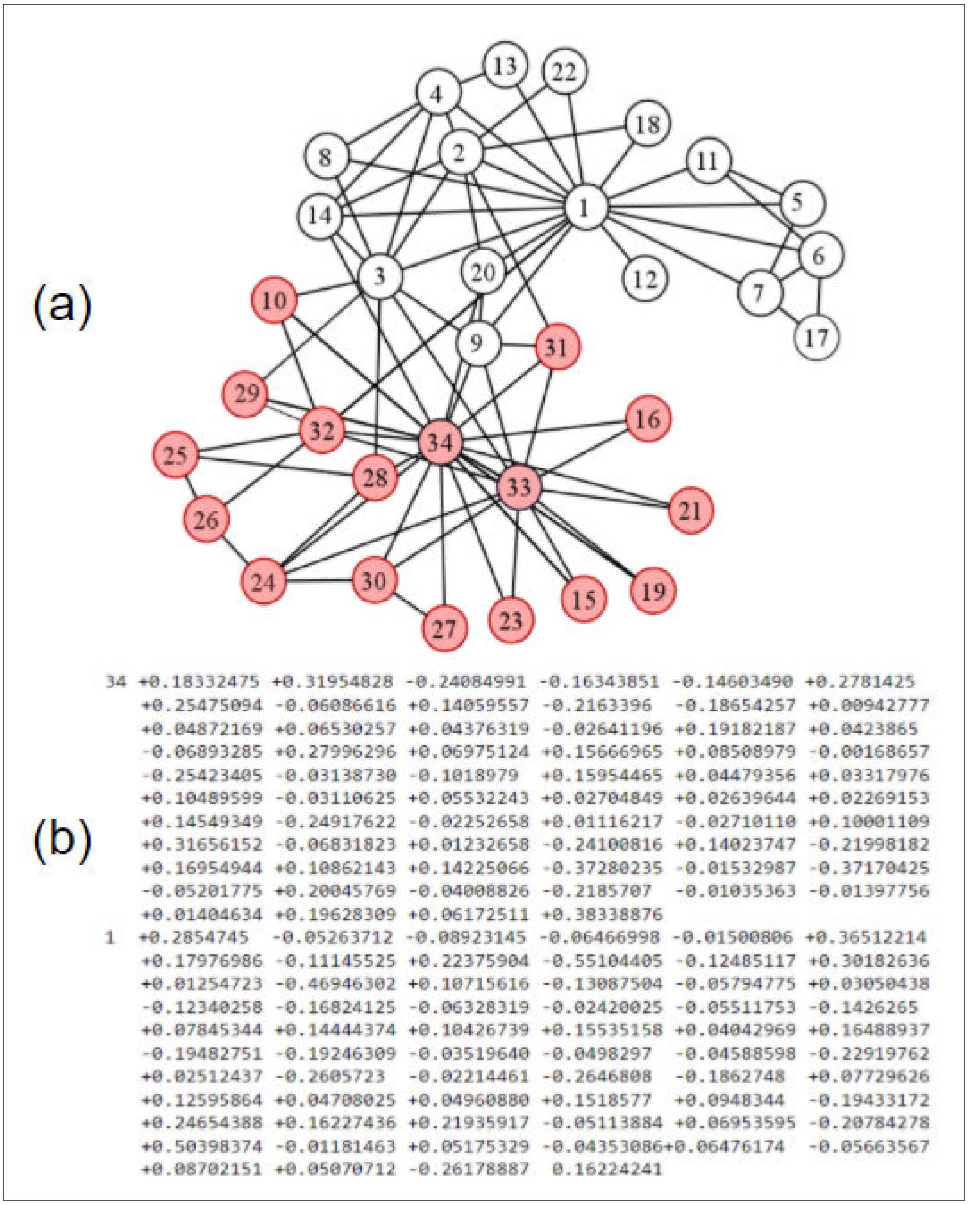

It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to Heaven, we were all going direct the other way—in short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only.

```
it: [0.23, 0.54, .97, …]
best: [0.37, 0.16, .25, …]
worst: [0.38, 0.82, .24, …]
time: [0.26, 0.67, .41, …]
...
```

Figure 1-14. Word embedding

In recent years, graph embedding has taken on a new meaning, analogous to word embedding. We compute one or more feature vectors to approximate the graph's neighborhood structure. In fact, when people say graph embedding, they often mean vertex embedding: computing a feature vector for each vertex in the graph. A vertex's embedding tells us something about how it connects to others. We can then use the collection of vertex embeddings to approximate the graph, no longer needing to consider the edges. There are also methods to summarize the whole graph as one embedding. This is useful for comparing one graph to another. In this book, we will focus on vertex embeddings.

Figure 1-15 shows an example of a graph (a) and portion of its vertex embedding (b). The embedding for each vertex (a series of 64 numbers) describes the structure of its neighborhood without directly mentioning any of its neighbors.



```
34 +0.18332475 +0.31954828 -0.24084991 -0.16343851 -0.14603490 +0.2781425
   +0.25475094 -0.06086616 +0.14059557 -0.2163396  -0.18654257 +0.00942777
   +0.04872169 +0.06530257 +0.04376319 -0.02641196 +0.19182187 +0.0423865
   -0.06893285 +0.27996296 +0.06975124 +0.15666965 +0.08508979 -0.00168657
   -0.25423405 -0.03138730 -0.1018979  +0.15954465 +0.04479356 +0.03317976
   +0.10489599 -0.03110625 +0.05532243 +0.02704849 +0.02639644 +0.02269153
   +0.14549349 -0.24917622 -0.02252658 +0.01116217 -0.02710110 +0.10001109
   +0.31656152 -0.06831823 +0.01232658 -0.24100816 +0.14023747 -0.21998182
   +0.16954944 +0.10862143 +0.14225066 -0.37280235 -0.01532987 -0.37170425
   -0.05201775 +0.20045769 -0.04008826 -0.2185707  -0.01035363 -0.01397756
   +0.01404634 +0.19628309 +0.06172511 +0.38338876
1  +0.2854745  -0.05263712 -0.08923145 -0.06466998 -0.01500806 +0.36512214
   +0.17976986 -0.11145525 +0.22375904 -0.55104405 -0.12485117 +0.30182636
   +0.01254723 -0.46946302 +0.10715616 -0.13087504 -0.05794775 +0.03050438
   -0.12340258 -0.16824125 -0.06328319 -0.02420025 -0.05511753 -0.1426265
   +0.07845344 +0.14444374 +0.10426739 +0.15535158 +0.04042969 +0.16488937
   -0.19482751 -0.19246309 -0.03519640 -0.0498297  -0.04588598 -0.22919762
   +0.02512437 -0.2605723  -0.02214461 -0.2646808  -0.1862748  +0.07729626
   +0.12595864 +0.04708025 +0.04960880 +0.1518577  +0.0948344  -0.19433172
   +0.24654388 +0.16227436 +0.21935917 -0.05113884 +0.06953595 -0.20784278
   +0.50398374 -0.01181463 +0.05175329 -0.04353086+0.06476174  -0.05663567
   +0.08702151 +0.05070712 -0.26178887  0.16224241
```

Figure 1-15. (a) Karate club graph[10] and (b) 64-element embedding for two of its vertices

10 Visualized partitioning of Zachary's Karate Club, licensed under CC BY_SA 4.0

Let's return to the question of classifying graph embeddings. As we will see, the technique of graph embedding requires unsupervised learning, so it can also be considered a form of representation learning. What graph embeddings give us is a set of feature vectors. For a graph with a million vertices, a typical embedding vector would be a few hundred elements long, a lot less than the upper limit of one million dimensions. Therefore, graph embeddings also represent a form of dimensionality reduction. And, we're using graph embeddings to get feature vectors, so they're also a form of feature extraction.

Does any feature vector qualify as an embedding? That depends on whether your selected features are telling you what you want to know. Graphlets come closest to the learned embeddings we are going to examine because of the methodical way they deconstruct neighborhood relationships.

Random Walk-based Embeddings

One of the best known approaches for graph embedding is to use random walks to get a statistical sample of the neighborhood surrounding each vertex v. A random walk is a sequence of connected hops in a graph G. The walk starts at some vertex v. It then picks a random neighbor of v and moves there. It repeats this selection of random neighbors until it is told to stop. In an unbiased walk, there is an equal probability of selecting any of the outgoing edges.

Random walks are great because they are easy to do and gather a lot of information efficiently. All feature extraction methods we have looked at before require following careful rules about how to traverse the graph; graphlets are particularly demanding due to their very precise definitions and distinctions from one another. Random walks are carefree. Just go.

For the example graph in Figure 1-16, suppose we start a random walk at vertex A. There is an equal probability of 1 in 3 that we will next go to vertex B, C, or D. If you start the walk at vertex E, there is a 100% chance that the next step will be to vertex B. There are variations of the random walk rules where there's a chance of staying in place, reversing your last step, jumping back to the start, or jumping to a random vertex.

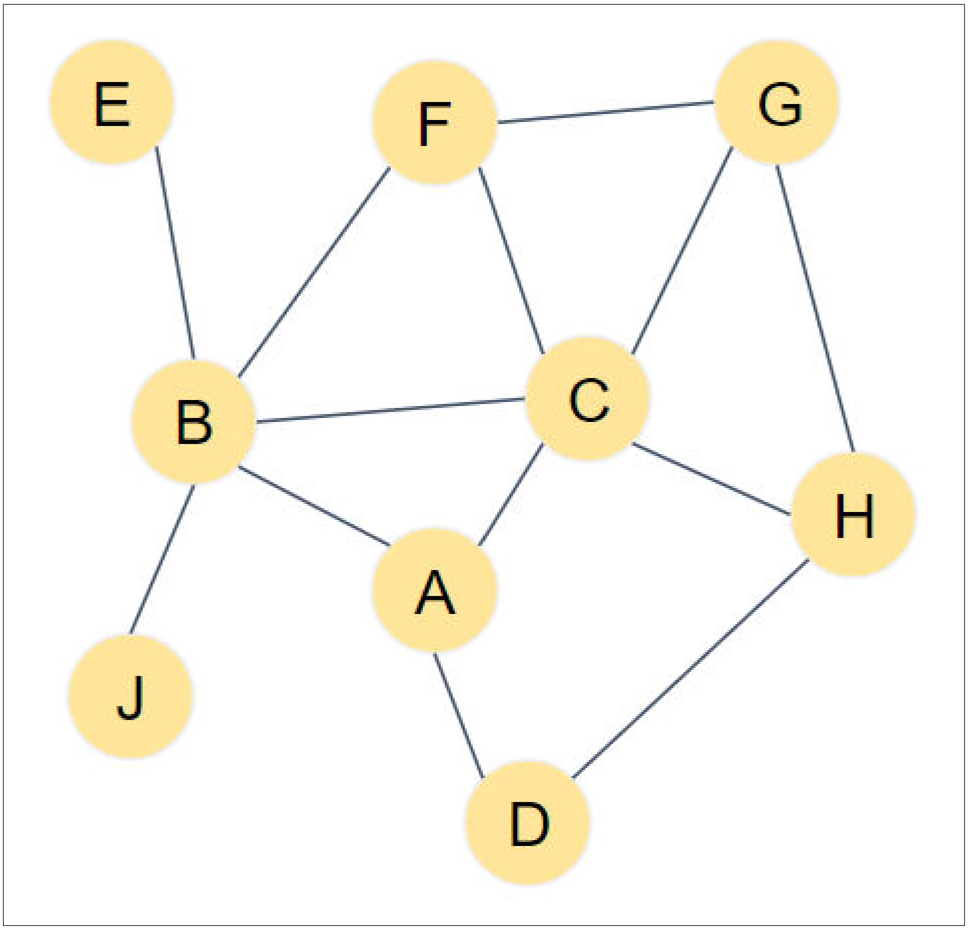


Figure 1-16. An ordinary graph for leisurely walks

Each walk can be recorded as a list of vertices, in the order in which they were visited. A-D-H-G-C is a possible walk. You can think of each walk as a signature. What does the signature tell us? Suppose that walk W1 starts at vertex 5 and then goes to 2. Walk W2 starts at vertex 9 and then goes to 2. Now they are both at 2. From here on, they have exactly the same probabilities for the remainder of their walks. Those individual walks are unlikely to be the same, but if there is a concept of a "typical walk" averaged over a sampling of several walks, then yes, the signatures of 5 and 9 would be similar. All because 5 and 9 share a neighbor 2. Moreover, the "typical walk" of vertex 2 itself would be similar, except offset by one step.

It turns out that these random walks gather neighborhood information much in the same way that SimRank and RoleSim gather theirs. The difference is that those role similarity algorithms considered *all* paths (by considering all neighbors), which is computationally expensive. SimRank and RoleSim differ from one another in how they aggregate the neighborhood information. Let's take a look at two random-walk based graph embedding algorithms that use a completely different computational method, one borrowed from neural networks.

DeepWalk. The DeepWalk algorithm (Perozzi et al. 2014) collects k random walks of length λ for every vertex in the graph. If you happen to know the word2vec algorithm (Mikolov et al. 2013), the rest is easy. Treat each vertex like a word and each walk like a sentence. Pick a window width w for the *skip-grams* and a length d for your embedding. You will end up with an embedding (latent feature vector) of length d for each vertex. The DeepWalk authors found that k=30 walks, walk length λ=40, with window width w=10 and embedding length d=64 worked well for their test graphs. Your results may vary. Figure 1-17(a) shows an example of a random walk, starting at vertex C and with a length of 16, which is 15 steps or hops from the starting point. The shadings will be explained when we explain skip-grams.

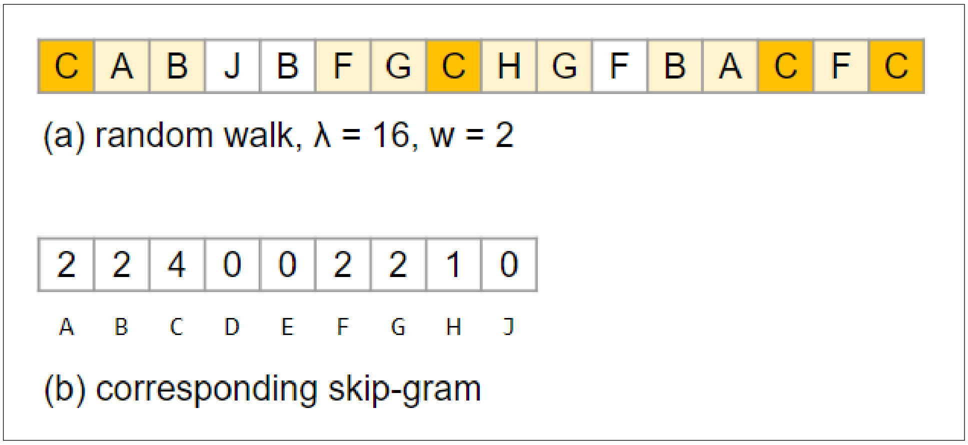

C A B J B F G C H G F B A C F C
(a) random walk, λ = 16, w = 2
2 2 4 0 0 2 2 1 0
A B C D E F G H J
(b) corresponding skip-gram

Figure 1-17. (a) Random walk vector and (b) a corresponding skip-gram

We assume you don't know word2vec, so we'll give a high-level explanation, enough so you can appreciate what is happening. This is the conceptual model. The actual algorithm plays a lot of statistical tricks to speed up the work. First, we construct a simple neural network with one hidden layer, as shown in Figure 1-18. The input layer accepts vectors of length n, where n = number of vertices. The hidden layer has length d, the embedding length because it is going to be learning the embedding vectors. The output layer also has length n.

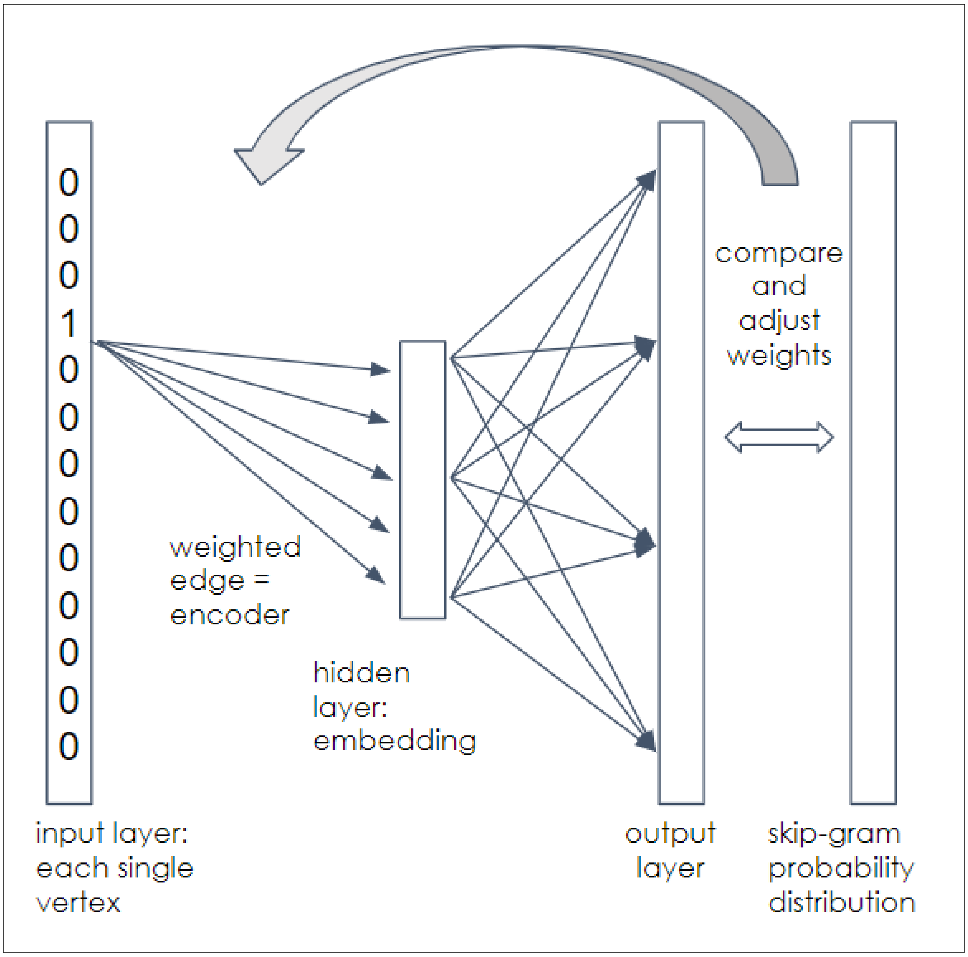

0
0
0
1
0
0
0
0
0
0
0
0
0
compare and adjust weights
weighted edge = encoder
hidden layer: embedding
input layer: each single vertex
output layer
skip-gram probability distribution

Figure 1-18. Neural network for DeepWalk

Each vertex needs to be assigned a position in the input and output layer, e.g., vertex A is at position 1, vertex B is at position 2, etc. Between the layers are two meshes of n ⨉ d connections, from each element in one layer to each element in the next layer. Each edge has a random weight initially, but we will gradually adjust the weights of the first mesh.

Start with one walk for one starting vertex. At the input, we represent the vertex using *one-hot encoding*. The vector element that corresponds to the vertex is set to 1; all other elements are set to 0. We are going to train this neural network to predict the neighborhood of the vertex given at the input.

Applying the weights in the first mesh to our one-hot input, we get a weighted vertex in the hidden layer. This is the current guess for the embedding of the input vertex.

Take the values in the hidden layers and multiply by the weights of the second mesh to get the output layer values. You now have a length-n vector with random weights.

We're going to compare this output vector with a skip-gram representation of the walk. This is where we use the window parameter w. For each vertex in the graph, count how many times it appears within w-steps before or after the input vertex v. We'll skip the normalization process, but your final skip-gram vector expresses the relative likelihood that each of the n vertices is near vertex v in this walk. Now we'll explain the result of Figure 1-17. Vertex C was the starting point for the random walk; we've used dark shading to highlight every time we stepped on vertex C. The light shading shows every step that is within w = 2 steps of vertex C. Then, we form the skip-gram in (b) by counting how many times we set foot on each vertex within the shaded zones. For example, vertex G was stepped on twice, so the skip-gram has 2 in the position for G. This is a long walk on a small graph, so most vertices were stepped on within the windows. For short walks on big graphs, most of the values will be 0.

Our output vector was supposed to be a prediction of this skip-gram. Comparing each position in the two vectors, if the output vector's value is higher than the skip-gram's value, then lower the corresponding weight in the input mesh. If the value was lower, then raise the corresponding weight.

You've processed one walk. Repeat this for one walk of each vertex. Now repeat for a second walk of each vertex, until you've adjusted your weights for k ✕ n walks. You're done! The weights of the first n ✕ d mesh are the length-d embeddings for your n vectors. What, about the second mesh? Strangely, we were never going to directly use the output vectors, so we didn't bother to adjust its weight.

Here is how to interpret and use a vertex embedding:

- For neural networks in general, you can't point to a clear real-world meaning of the individual elements in the latent feature vector.
- Based on how we trained the network, though, we can reason backwards from the skip-grams, which represent the neighbors around a vertex: vertices that have similar neighborhoods should have similar embeddings.
- If you remember the example earlier about two paths that were offset by one step, note that those two paths would have very similar skip-grams. So, vertices that are close to each other should have similar embeddings.

One critique of DeepWalk is that its uniformly random walk is too random. In particular, it may wander far from the source vertex before getting an adequate sample of the neighborhoods closer to the source. One way to address that is to include a probability of resetting the walk by magically teleporting back to the source vertex and then continuing with random steps again, as in Zhou et al. 2021. This is known as "random walk with restart".

Node2vec. Another method which has already gained popularity is node2vec [Grover and Leskovec 2016]. It uses the same skip-gram training process as DeepWalk, but it gives the user two adjustment parameters to control the direction of the walk: go farther (depth), go sideways (breadth), or go back a step. Farther and back seem obvious, but what exactly does sideways mean?

Suppose we start at vertex A of the graph in Figure 1-19. Its neighbors are vertices B, C, and D. Since we are just starting, any of the choices would be moving forward. Let's go to vertex C. For the second step, we can choose from any of vertex C's neighbors: A, B, F, G, or H.

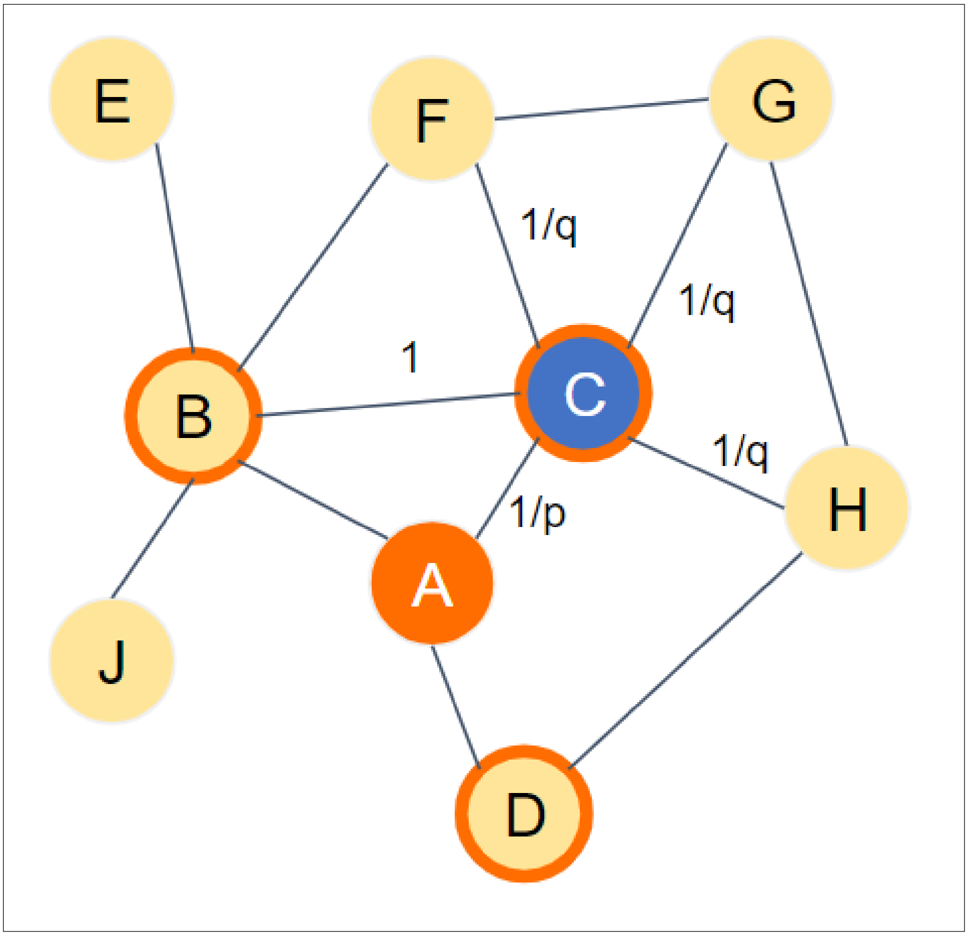


Figure 1-19. Illustrating the biased random walk with memory used in node2vec

If we remember what our choices were in our previous step, we can classify our current set of neighbors into three groups: back, sideways, and forward. We also assign a weight to each connecting edge, which represents the unnormalized probability of selecting that edge.

Back
: In our example: vertex A. Edge weight = 1/p.

Sideways
: These are the vertices that were available last step and are available this step (e.g., the union of the two sets of vertices. So, they represent a second chance to visit a different neighbor of where you were previously. In our example: vertex B. Edge weight = 1.

Forward
: There are all the vertices that don't return or go sideways. In our example: vertices F, G, and H. Edge weight = 1/q.

If we set p = q = 1, then all choices have equal probability, so we're back to an unbiased random walk. If p < 1, then returning is more likely than going sideways. If q < 1, then each of the forward (depth) options is more likely than sideways (breadth). Returning also keeps the walk closer to home (e.g., similar to breadth-first-search), because if you step back and then step forward randomly, you are trying out the different options in your previous neighborhood.

This ability to tune the walk makes node2vec more flexible than DeepWalk, which results in better models in many cases.

Besides the random walk approach, there are several other techniques for graph embedding, each with their advantages and disadvantages: matrix factorization, edge reconstruction, graph kernel, and generative models. Though already a little dated, *A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications*, by Cai et al. provides a thorough overview with several accessible tables which compare the features of different techniques.

Summary

Graphs can provide additional and valuable features to describe and understand your data. Key takeaways from this section are as follows:

- A *graph feature* is a characteristic that is based on the pattern of connections in the graph
- Graphlets and centrality algorithms provide domain-independent features for any graph.
- Applying some domain knowledge to guide your feature extraction leads to more meaningful features.

- Machine learning can produce vertex embeddings which encode vertex similarity and proximity in terms of compact feature vectors.
- Random walks are a simple way to sample a vertex's neighborhood.

Graph Neural Networks

In the popular press, it's not AI unless it uses a neural network, and it's not machine learning unless it uses deep learning. Neural networks were originally designed to emulate how the human brain works, but they evolved to address the capabilities of computers and mathematics. The predominant models assume that your input data is in a matrix or tensor; it's not clear how to present and train the neural network with interconnected vertices. Yet, are there graph-based neural networks? Yes!

Graph neural networks (GNNs) are conventional neural networks with an added twist for graphs. Just as there are several variations of neural networks, there are several variations of GNNs. The simplest way to include the graph itself into the neural network is through convolution.

Graph Convolutional Networks

In mathematics, *convolution* is how two functions affect the result if one acts on the other in a particular way. It is often used to model situations in which one function describes a primary behavior and another function describes a secondary effect. For example, in image processing, convolution takes into account neighboring pixels to improve identification of boundaries and to add artificial blur. In audio processing, convolution is used to both analyze and synthesize room reverberation effects. A convolutional neural network (CNN) is a neural network that includes convolution in the training process. For example, you could use a CNN for facial recognition. The CNN would systematically take into account neighboring pixels, an essential duty when analyzing digital images.

A graph convolutional network (GCN) is a neural network that uses graph traversal as a convolution function during the learning process. While there were some earlier related works, the first model to distill the essence of graph convolution into a simple but powerful neural network was presented in 2017 with "Semi-Supervised Classification with Graph Convolutional Networks" by Thomas Kipf and Max Welling.

For graphs, we want the embedding for each vertex to include information about the relationships to other vertices. We can use the principle of convolution to accomplish this. Figure 1-20 shows a simple convolution function, with a general model at top and a more specific example at bottom.

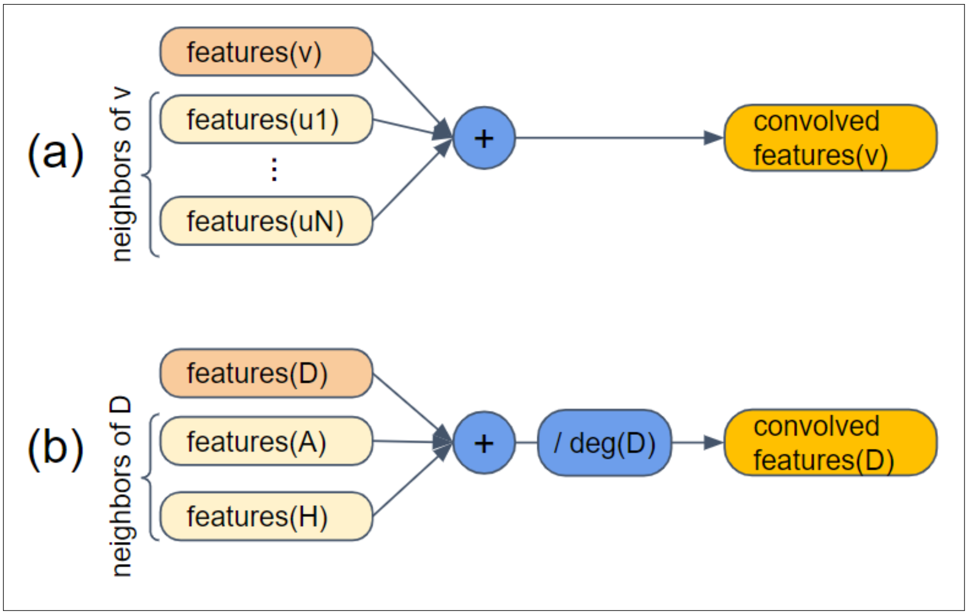


Figure 1-20. Convolution using neighbors of a vertex

In part (a) of the figure, the primary function is features(v): given a vertex v, output its feature vector. The convolution is to combine the features of v with the features of all of the neighbors of v: u1, u2,...,uN. If the features are numeric, we can just add them. The result is the newly convolved features of v. In part (b), we set v = D. Vertex D has two neighbors, A and H. We insert one more step after summing the feature vectors: divide by the degree of the primary vertex. Vertex D has 2 neighbors, so we divide by 2. This regularizes the output values so that they don't keep getting bigger and bigger. (Yes, technically we should divide by deg(v)+1, but the simpler version seems to be good enough.)

Let's do a quick example:

```
features[0](D) = [3, 1 ,4, 1]
features[0](A) = [5, 9, 2, 6]
features[0](H) = [5, 3, 5, 8]
features[1](D) = [6.5, 6.5, 5.5, 7.5]
```

By having neighbors share their feature values, this simple convolution function performs selective information sharing: it determines what is shared (features) and by whom (neighbors). A neural network which uses this convolution function will tend to evolve according to these maxims:

- Vertices which share many neighbors will tend to be similar.

- Vertices which share the same initial feature values will tend to be similar.

These properties are reminiscent of random-walk graph embeddings, aren't they? But we won't be using any random walks.

How do we take this convolution and integrate it into a neural network? Take a look at Figure 1-21.

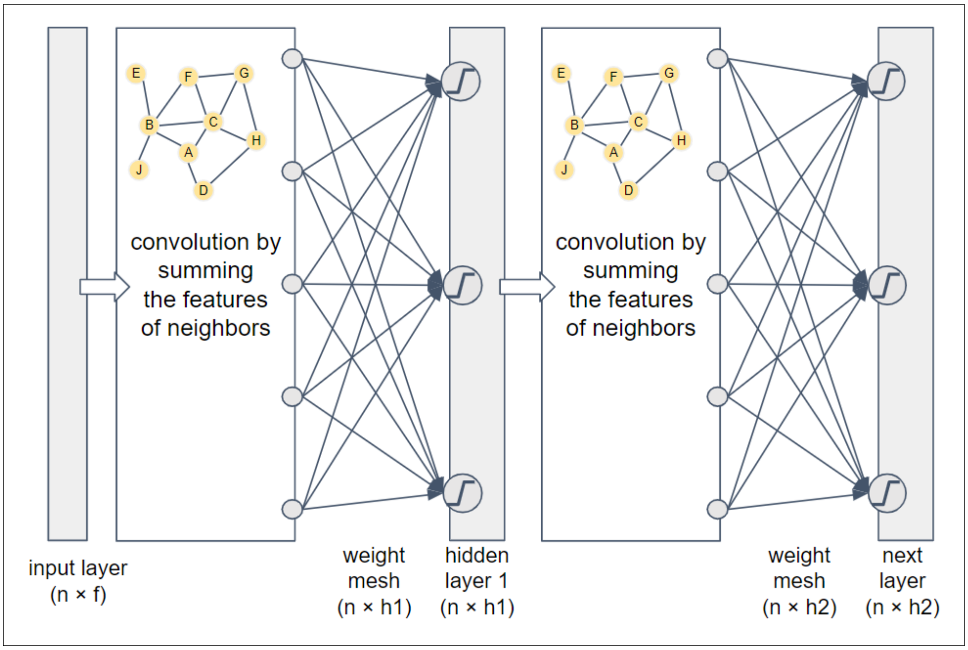


Figure 1-21. Two-layer graph convolutional network

This two-layer network flows from left to right. The input is the feature vectors for all of the graph's vertices. If the feature vectors run horizontally and we stack the vertices vertically, then we get an n ╳ f matrix, where f is the number of features. Next we apply the adjacency-based convolution. We then apply a set of randomized weights (similar to what we did with random-walk graph embedding networks) to merge and reduce the features to an embedding of size h1. Typically h1 < f. Before storing the values in the embedding matrix, we apply an activation function (indicated by the blocky "S" in a circle) which acts as a filter/amplifier. Low values are pushed lower, and high values are pushed higher. Activation functions are used in most neural networks.

Because this is a two-layer network, we repeat the same steps. The only differences are that this embedding may have a different size, where typically h2 ≤ h1, and this weight mesh has a different set of random weights. If this is the final layer, then it's considered the output layer with the output results. By having two layers, the output embedding for each vertex takes into account the neighbors within two hops. You can add more layers to consider deeper neighbors. Two or three layers often provide the best results. With too many layers, the radius of each vertex's neighborhood becomes so large that it overlaps significantly even with the neighborhoods of unrelated vertices.

Our example here is demonstrating how a GCN can be used in unsupervised learning mode. No training data or target function was provided; we just merged features of vertices with their neighbors. Surprisingly, you can get useful results from an unsupervised, untrained GCN. The authors of GCN experimented with a 3-layer untrained GCN, using the well-known Karate Club dataset.They set the output layer's embedding length of 2, so that they could interpret the two values as coordinate points. When plotted, the output data points showed community clustering which matched the known communities in Zachary's Karate Club.

The GCN architecture is general enough to be used for unsupervised, supervised, semi-supervised, or even reinforcement learning. The only difference between a GCN and a vanilla feed-forward NN is the addition of the step to aggregate the features of a vector with those of its neighbors. Figure 1-22 shows a generic model for how neural networks tune their weights. The graph convolution in GCN affects only the block labeled Forward Propagation Layers. All of the other parts (input values, target values, weight adjustment, etc.) are what determine what type of learning you are doing. That is, the type of learning is decided independently from your use of graph convolution.

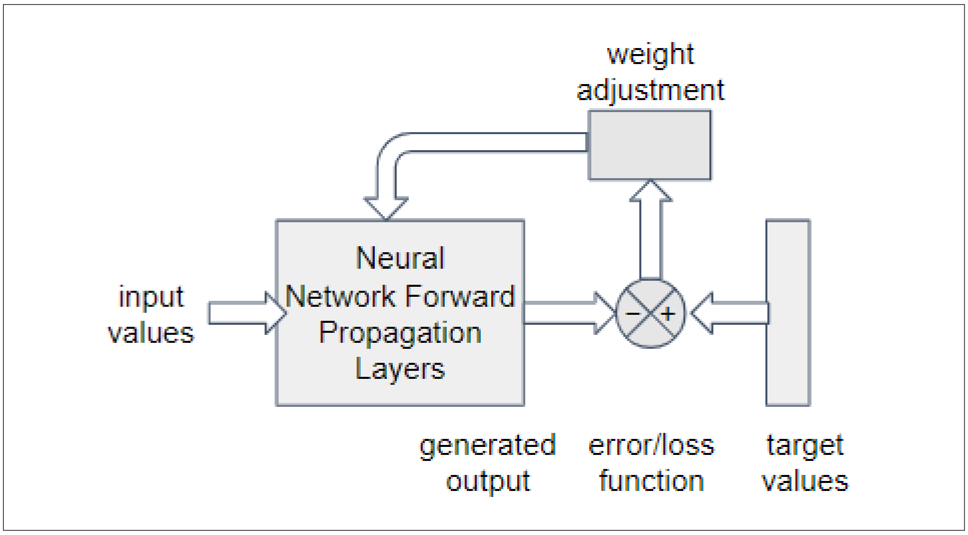


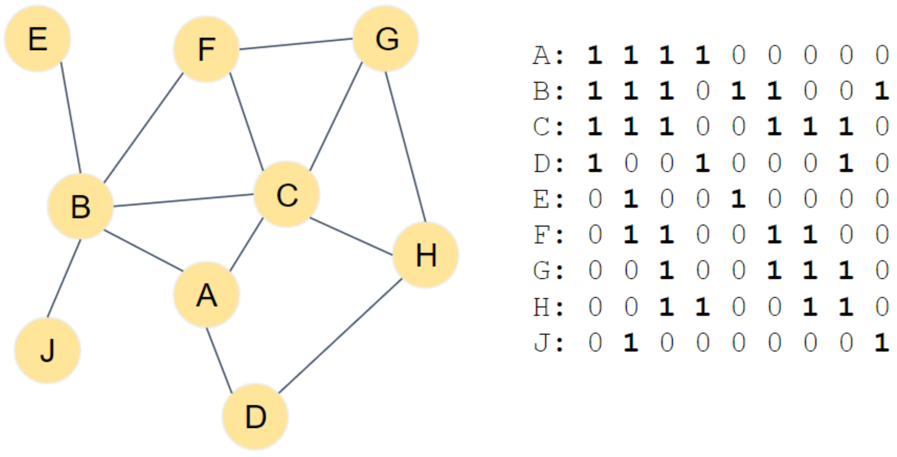
Figure 1-22. Generic model for responsive learning in a neural network

Attention neural networks use a more advanced form of feedback and adjustment. The details are beyond the scope of this book, but graph attention neural networks (GATs) can tune the weight (aka focus the attention) of each neighbor when adding them together for convolution. That is, a GAT performs a weighted sum instead of a simple sum of neighbor's features, and the GAT trains itself to learn the best weights. When applied to the same benchmark tests, GAT outperforms GCN slightly.

Matrix Algebra Formulation

If you like to think in terms of matrices, we can express the convolution process in terms of three simple, standard matrices for any graph: the adjacency matrix A, the identity matrix I, and the degree matrix D. (Feel free to skip this section is this doesn't sound like you.)

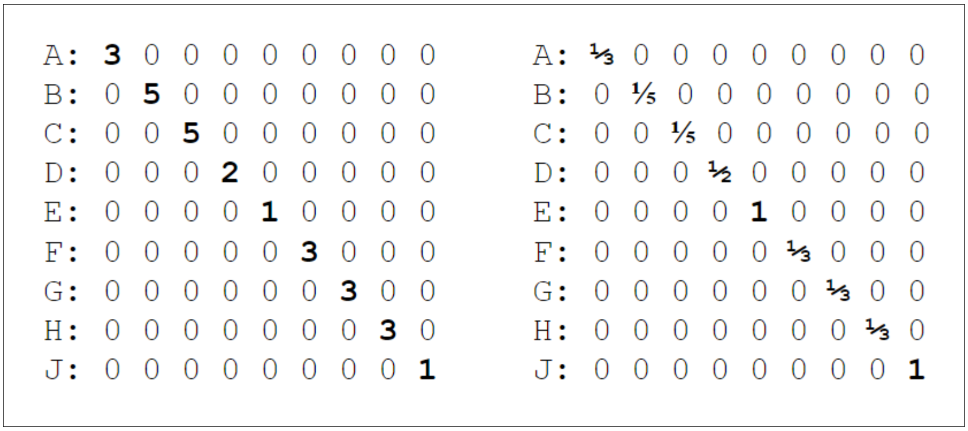
Matrix A expresses the connectivity of the graph. Number the vertices 1 to n, and number the rows and columns similarly. The value A(i, j) = 1 if there is an edge from vertex i to vertex j; otherwise A(i,j) is 0. The identity matrix says, "I am myself." I(i, j) = 1 if i = j; otherwise it is 0. It looks like a diagonal line from upper left to lower right. The matrix in Figure 1-23 is (A + I). Note that the 1's in row i tell us the convolution function for vertex i.



```
A: 1 1 1 1 0 0 0 0 0
B: 1 1 1 0 1 1 0 0 1
C: 1 1 1 0 0 1 1 1 0
D: 1 0 0 1 0 0 0 1 0
E: 0 1 0 0 1 0 0 0 0
F: 0 1 1 0 0 1 1 0 0
G: 0 0 1 0 0 1 1 1 0
H: 0 0 1 1 0 0 1 1 0
J: 0 1 0 0 0 0 0 0 1
```

Figure 1-23. A graph and matrix representing the graph's adjacency and identity: (A+I)

What about the regularization? With matrices, instead of performing division, we do multiplication by the inverse matrix. The matrix we want is the degree matrix D, where D(i, j) = degree(i) if i = j; otherwise it is 0. The degree matrix of the graph of Figure 1-16 is shown in the left half of Figure 1-24. At the right is the inverse matrix D-1.



```
A: 3 0 0 0 0 0 0 0 0      A: ⅓ 0 0 0 0 0 0 0 0
B: 0 5 0 0 0 0 0 0 0      B: 0 ⅕ 0 0 0 0 0 0 0
C: 0 0 5 0 0 0 0 0 0      C: 0 0 ⅕ 0 0 0 0 0 0
D: 0 0 0 2 0 0 0 0 0      D: 0 0 0 ½ 0 0 0 0 0
E: 0 0 0 0 1 0 0 0 0      E: 0 0 0 0 1 0 0 0 0
F: 0 0 0 0 0 3 0 0 0      F: 0 0 0 0 0 ⅓ 0 0 0
G: 0 0 0 0 0 0 3 0 0      G: 0 0 0 0 0 0 ⅓ 0 0
H: 0 0 0 0 0 0 0 3 0      H: 0 0 0 0 0 0 0 ⅓ 0
J: 0 0 0 0 0 0 0 0 1      J: 0 0 0 0 0 0 0 0 1
```

Figure 1-24. The degree matrix D and itself inverse matrix D-1

We can express the computation from layer H0 to the next layer H1 as follows:

$$H_1 = \sigma\left(D^{-1}(A+1)H_0W_1\right)$$

where σ is the activation function, and W is the weight mesh. In Kipf and Welling's paper, they made a tweak to provide a more balanced regularization:

$$H_1 = \sigma\left(D^{-1/2}(A+1)D^{-1/2}H_0W_1\right)$$

GraphSAGE

One limitation of the basic GCN model is that it does a simple averaging of vertex + neighbor features. It seems we would want some more control and tuning of this convolution. Also, large variations in the number of neighbors for different vertices may lead to training difficulties. To address this limitation, Hamilton et al. presented GraphSAGE in 2017 in their paper "Inductive Representation Learning on Large Graphs“. Like GCN, this technique also combines information from neighbors, but it does it a little differently. In order to standardize the learning from neighbors, GraphSAGE samples a fixed number of neighbors from each vertex. Figure 1-25 shows a block diagram of GraphSAGE, with sampled neighbors.

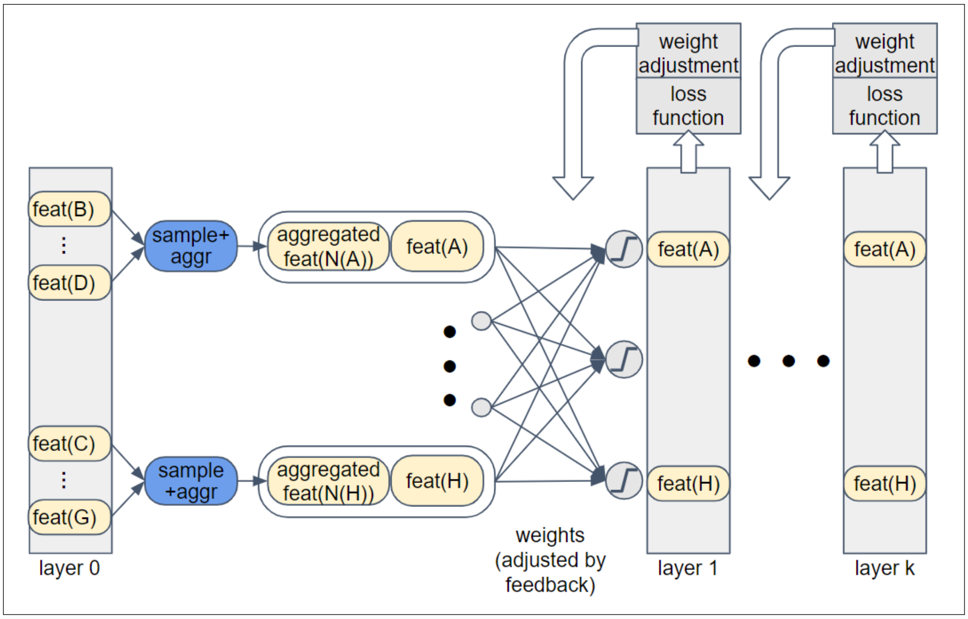


Figure 1-25. Block diagram of GraphSAGE

With GraphSAGE, the features of the neighbors are combined according to a chosen aggregation function. The function could be addition, as in GCNs. Any order-independent aggregation function could be used; LSTM with random ordering and max-pooling work well. The source vertex is not included in the aggregation as it is in GCN; instead, the aggregated feature vectors and the source vertex's feature vector are concatenated to make a double-length vector. We then apply a set of weights to mix the features together, apply an activation function, and store as the next layer's representation of the vertices. This series of sets constitutes one layer in the neural network and the gathering of information within one hop of each vertex. A GraphSAGE network has k layers, each with its own set of weights. GraphSAGE proposes a loss function that rewards nearby vertices if they have similar embeddings, while rewarding distant vertices if they have dissimilar embeddings.

Besides training on the full graph, as you would with GCN, you can train GraphSAGE with only a sample of the vertices. The fact that GraphSAGE's aggregation functions uses equally-sized samples of neighborhoods means that it doesn't matter how you arrange the inputs.That freedom of arrangement is what allows you to train with one sample and then test or deploy with a different sample. Because it can learn from a sample, GraphSAGE performs *inductive learning*. In contrast, GCN directly uses the adjacency matrix, which forces it to use the full graph with the vertices arranged in a particular order. Training with the full data and learning a model for only that data is *transductive learning*.

Whether learning from a sample will work on your particular graph depends on whether your graph's structure and features follow global trends, such that a random subgraph looks similar to another subgraph of similar size. For example, one part of a forest may look a lot like another part of a forest. For a graph example, suppose you have a Customer 360 graph including all of a customer's interactions with your sales team, website, events, their purchases, and all other profile information you have been able to obtain. Last year's customers are rated based on the total amount and frequency of their purchases. It is reasonable to expect that if you used GraphSAGE with last year's graph to predict the customer rating, it should do a decent job of predicting the ratings of this year's customers. Table 1-4 summarizes all the similarities and differences between GCN and GraphSAGE that we have presented.

Table 1-4. Comparison of GCN and GraphSAGE traits

	GCN	GraphSAGE
Neighbors for aggregation	All	sample of n neighbors
Aggregation function	mean	several options
Aggregating a vertex with neighbors?	aggregated with others	concatenated to others
Do weights need to be learned?	not for unsupervised transductive model	yes, for inductive model
Unsupervised?	yes	yes
Supervised?	with modification	with modification
Can be trained on a sample of vertices	no	yes

Summary

Graph-based neural networks put graphs into the mainstream of machine learning. Key takeaways from this section are as follows:

- The graph convolutional neural network (GCN) enhances the vanilla neural network by averaging together the feature vectors of each vertex's neighbors with its own features during the learning process.
- GraphSAGE makes two key improvements to the basic GCN: vertex and neighborhood sampling, and keeping features of vectors separate from those of its neighbors .
- GCN learns transductively (uses the full data to learn only about that data) whereas GraphSAGE learns inductively (uses a data sample to learn a model that can apply to other data samples).
- The modular nature of neural networks and graph enhancements mean that the ideas of GCN and GraphSAGE can be transferred to many other flavors of neural networks.

Comparing Graph Machine Learning Approaches

This chapter has covered many different ways to learn from graph data, but it only scratched the surface. Our goal was not to present an exhaustive survey, but to provide a framework from which to continue to grow. We've outlined the major categories of and techniques for graph-powered machine learning, we've described what characterizes and distinguishes them, and we've provided simple examples to illustrate how they operate. It's worthwhile to briefly review these techniques. Our goal here is not only to summarize but also to provide you with guidance for selecting the right techniques to help you learn from your connected data.

Use Cases for Machine Learning Tasks

Table 1-5 pulls together examples of use cases for each of the major learning tasks. These are the same basic data mining and machine learning tasks you might perform on any data, but the examples are particularly relevant for graph data.

Table 1-5. Use cases for graph data learning tasks

Task	Use case examples
Community detection	Delineating social networks
	Finding a financial crime network
	Detecting an biological ecosystem or chemical reaction network
	Discovering a network of unexpectedly interdependent components or process, such as software procedures or legal regulations
Similarity	Abstraction of physical closeness, inverse of distance
	Prerequisite for clustering, classification, and link prediction
	Entity resolution - finding two online identities that probably refer to the same real-world person
	Product recommendation or suggested action
	Identifying persons who perform the same role in different but analogous networks
Find unknown patterns	Identifying the most common "customer journeys" on your website or in your app
	Discovering that two actions - planning a summer vacation and
	Once current patterns are identified, then noticing changes
Link Prediction	Predicting someone's future purchase or willingness to purchase
	Predicting that business or personal relationship exists, even though it is not recorded in the data
Feature Extraction	Enriching your customer data with graph features, so that your ML training to categorize and model your customers will be more successful
Embedding	Transforming a large set of features to a more compact set, for more efficient computation
Classification (predicting a category)	Given some past examples of fraud, create a model for identifying new cases of fraud
	Predicting categorical outcomes for future vaccine patients, based on test results of past patients
Regression (predicting a numerical value)	Predicting weight loss for diet program participants, based on results of past participants

Once you have identified what type of task you want to perform, consider the available graph-based learning techniques, what they provide, and their key strengths and differences.

Graph-based Learning Methods for Machine Learning Tasks

Table 1-6 lists the graph algorithms and feature extraction methods which we encountered in the chapter. Table 1-7 compares graph neural network methods.

Table 1-6. Features of graph-based learning methods

Task	Graph-based Learning Methods	Comments
Community detection	connected components	one connection to the community is enough
	k-core	at least k connections to other community members
	modularity optimization, e.g. Louvain	relatively higher density of connections inside than between communities
Similarity	Jaccard neighborhood similarity	counts how many relationships in common, for nonnumeric data
	cosine neighborhood similarity	compares numeric or weighted vectors of relationships
	role similarity	defines similarity recursively as having similar neighbors
Find unknown patterns	frequent pattern mining	starts with small patterns and builds to larger patterns
Domain- independent feature extraction	graphlets	systematic list of all possible neighborhood configurations
	PageRank	rank is based on the number and rank of in-neighbors, for directed graphs among vertices of the same type
	closeness centrality	closeness = average distance to any other vertex
	betweenness centrality	how often a vertex lies on the shortest path between any two vertices; slow to compute
Domain- dependent feature extraction	searching for patterns relevant to your domain	custom effort by someone with domain knowledge
Dimensionality reduction and embedding	DeepWalk	embeddings will be similar if the vectors have similar random walks, considering nearness and role; more efficient than SimRank
	node2vec	DeepWalk with directional tuning of the random walks, for greater tuning

Graph Neural Networks: Summary and Uses

The graph neural networks presented in this chapter not only are directly useful in many cases but also templates to show more advanced data scientists how to transform any neural network technique to include graph connectivity in the training. The key is the convolution step, which takes the features of neighboring vertices into account. All of the graph neural network approaches presented can be used for either unsupervised or supervised learning.

Table 1-7. Graph neural networks

Name	Description	Uses
Graph Convolutional Network (GCN)	Convolution: average of neighbor's features	Clustering or classification on a particular graph
GraphSAGE	Convolution: average of a sample of neighbor's features	learning a representative model on a sample of a graph, in addition to clustering or classification
Graph Attention Network (GAT)	Convolution: weighted average of neighbor's features	clustering, classification, and model learning; added tuning and complexity by learning weights for the convolution

Chapter Summary

Graphs and graph-based algorithms contribute to several stages of the machine learning pipeline: data acquisition, data preparation, feature extraction, dimensionality reduction, and model training. As data scientists know, there is no golden ticket, no single technique that solves all their problems. Instead, you work to acquire tools for your toolkit, develop the skills to use your tools well, and gain an understanding about when to use them.

CHAPTER 2

Entity Resolution Revisited

A note for Early Release readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 11th chapter of the final book.

This chapter uses entity resolution for a streaming video service as an example of unsupervised machine learning with graph algorithms. After completing this chapter, you should be able to:

- Name the categories of graph algorithms which are appropriate for entity resolution as unsupervised learning.
- List three different approaches for assessing the similarity of entities.
- Understand how parameterized weights can adapt entity resolution to be a supervised learning task.
- Interpret a simple GSQL FROM clause and have a general understanding of ACCUM semantics.
- Set up and run a TigerGraph Cloud Starter Kit using GraphStudio.

Goal: Identify Real-World Users and Their Tastes

The streaming video on demand (SVoD) market is big business. Accurate estimates of the global market size are hard to come by, but the most conservative estimate may be

$50 billion in 2020[1] with annual growth rates ranging from 11%[2] to 21%[3] for the next five years or so. Movie studios, television networks, communication networks, and tech giants have been merging and reinventing themselves, in hopes of becoming a leader in the new preferred format for entertainment consumption: on-demand digital entertainment, on any video-capable device.

To succeed, SVoD providers need to have the content to attract and retain many millions of subscribers. Traditional video technology (movie theaters and broadcast television) limited the provider to offering only one program at a time per venue or per broadcast region. Viewers had very limited choice, and providers selected content that would appeal to large segments of the public. Home video on VHS tape and DVD introduced personalization; wireless digital video on demand on any personal device has put the power in the hands of the consumer.

Providers no longer need to appeal to the masses. On the contrary, the road to success is microsegmentation: to offer something for everyone. The SVoD giants are assembling sizable catalogs of existing content, as well as spending billions of dollars on new content. The volume of options creates several data management problems. With so many shows available, it is very hard for users to browse. Providers must categorize the content, categorize users, and then recommend shows to users. Good recommendations increase viewership and satisfaction.

While predicting customers' interests is hard enough, the streaming video industry also needs to overcome a multifaceted *entity resolution* problem. Entity resolution, you may recall, is the task of identifying two or more entities in a dataset which refer to the same real-world entity, and then linking or merging them together. In today's market, streaming video providers face at least three entity resolution challenges. First, each user may have multiple different authorization schemes, one for each type of device they use for viewing. Second, corporate mergers are common, which require merging the databases of the constituent companies. For example, Disney+ combines the catalogs of Disney, Pixar, Marvel, and National Geographic Studios. HBO Max brings together HBO, Warner Bros., and DC Comics. Third, SVoD providers may form a promotional, affiliate, or partnership arrangement with another company: a customer may be able to access streaming service A because they are a customer of some other service B. For example, customers of AT&T fiber internet service may qualify for free HBO Max service.

1 "Video Streaming Market Report", Grand View Research, February 2021

2 "Video Streaming (SVoD) Report", Statista.com

3 "Video Streaming Market Report", Grand View Research, February 2021

Solution Design

Before we can design a solution, let's start with a clear statement of the problem we want to solve.

Title: Problem Statement

Each real-world user may have multiple digital identities. The goal is to discover the hidden connections between these digital identities and then to link or merge them together. By doing so, we will be able to connect all of the information together, forming a more complete picture of the user. In particular, we will know all the videos that a person has watched, in order to get a better understanding of their personal taste and to make better recommendations.

Now that we've crafted a clear problem statement, let's consider a potential solution -- entity resolution. Entity resolution has two parts: deciding which entities are probably the same, and then resolving entities. Let's look at each part in turn.

Learning Which Entities Are The Same

If we are fortunate enough to have training data showing us examples of entities that are in fact the same, we can use supervised learning to train a machine learning model. In this case, we do not have training data. Instead, we will rely on the characteristics of the data itself, looking at similarities and communities to perform unsupervised learning.

To do a good job, we want to build in some domain knowledge. What are the situations for a person to have multiple online identities, and what would be the clues in the data? Here are some reasons why a person may create multiple accounts:

- A user creates a second account because they forgot about or forgot how to access the first one.
- A user has accounts with two different streaming services, and the companies enter a partnership or merge.
- A person may intentionally set up multiple distinct identities, perhaps to take advantage of multiple membership rewards or to separate their behavioral profiles (e.g., to watch different types of videos on different accounts). The personal information may be very different, but the device IDs might be the same.

Whenever the same person creates two different accounts at different moments, there can be variations in some details for trivial or innocuous reasons. The person decides to use a nickname. They choose to abbreviate a city or street name. They mistype. They have multiple phone numbers and email addresses to choose from, and they make a different choice for no particular reason. Over time, more substantial changes may occur, to address, phone number, device IDs, and even to the user's name.

While several situations can result in one person having multiple online identities, it seems we can focus our data analysis on only two patterns. In the first pattern, most of the personal information will be the same or similar, but a few attributes may differ. Even when two attributes differ, they may still be related, such as use of a nickname or a misspelling of an address. In the second pattern, much of the information is different, but one or more key pieces remain the same, such as home phone number or birthdate, and behavioral clues (such as what type of videos they like and what time of day do they watch them) may suggest that two identities belong to the same person.

To build our solution, we will need to use some similarity algorithms and also a community detection or clustering algorithm to group similar entities together.

Resolving Entities

Once we have used the appropriate algorithms to identify a group of entities that we believe to be the same, what will do about it? We want to update the database somehow to reflect this new knowledge. There are two possible ways to accomplish this: merge the group into one entity or link the entities in a special way so that whenever we look at one member of the group, we will readily see the other related identities.

Merging the entities makes sense only when some online identities are considered incorrect, so we want to eliminate them. For example, suppose a customer has two online accounts because they misspelled their name or forgot that they had an account already. Both the business owner and the customer wants to eliminate one account and to merge all of the records (purchase history, game scores, etc.) into one account. Knowing which account to eliminate takes more knowledge of each specific case than we have in our example.

We will take the safer route and simply link entities together. Specifically, we will have two types of entities in the graph: one representing digital identities and the other representing real-world entities. After resolution, the database will show one real-world entity having an edge to each of its digital identities, as illustrated in Figure 2-1.

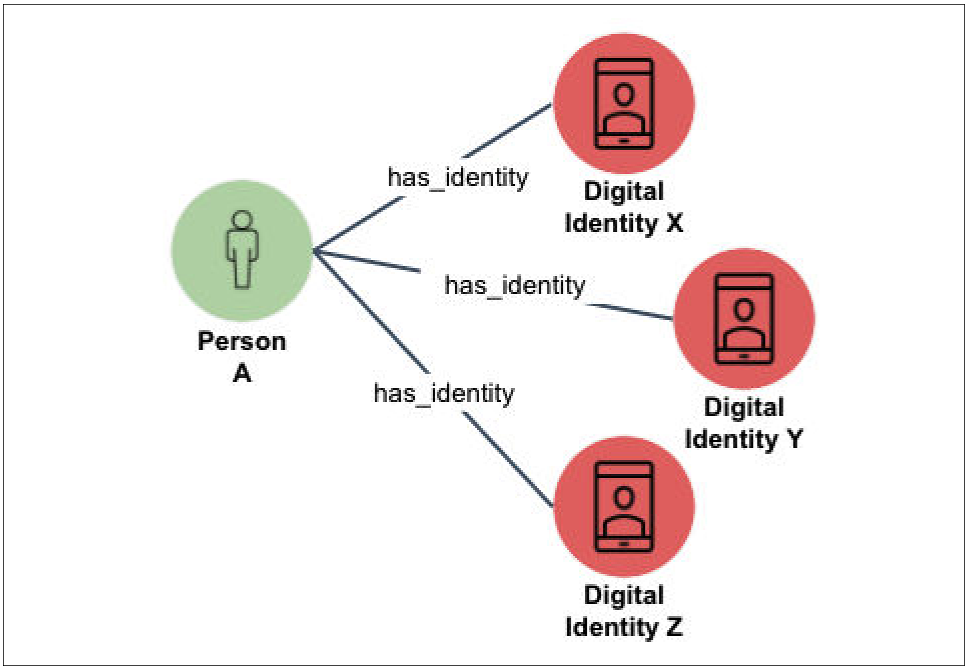


Figure 2-1. Digital entities linked to a real-world entity, after resolution

Implementation

The implementation of graph-based entity resolution we present below is available as a TigerGraph Cloud Starter Kit. As usual, we will focus on using the GraphStudio visual interface. All of the necessary operations could also be performed from a command line interface, but that would lack the visual interface.

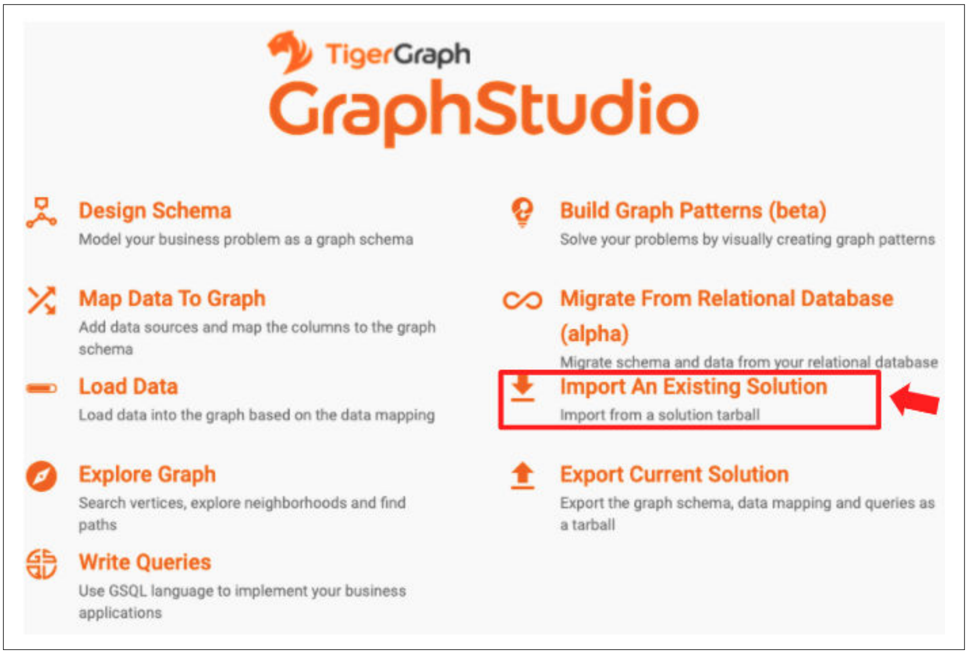
Starter Kit

To get and install your Starter Kit, do one of the following::

1. Use a TigerGraph Cloud instance preloaded with this example, go to tgcloud.io and select the Starter Kit called In-Database Machine Learning for Big Data Entity Resolution.
2. Load this Starter Kit onto your own machine which is running TigerGraph, either on-premises or on the cloud, go to www.tigergraph.com/starterkits.
    a. Find In-Database Machine Learning for Big Data Entity Resolution.
    b. Download Data Set and the solution package corresponding to your version of the TigerGraph platform.
    c. Start your TigerGraph instance. Go to the GraphStudio home page.

d. Click Import An Existing Solution, as highlighted in Figure 2-2, and select the solution package which you downloaded

Importing a GraphStudio Solution will delete your existing database. If you wish to save your current design, perform a GraphStudio Export Solution and also gsql backup.



Design Schema
Model your business problem as a graph schema

Build Graph Patterns (beta)
Solve your problems by visually creating graph patterns

Map Data To Graph
Add data sources and map the columns to the graph schema

Migrate From Relational Database (alpha)
Migrate schema and data from your relational database

Load Data
Load data into the graph based on the data mapping

Import An Existing Solution
Import from a solution tarball

Explore Graph
Search vertices, explore neighborhoods and find paths

Export Current Solution
Export the graph schema, data mapping and queries as a tarball

Write Queries
Use GSQL language to implement your business applications

Figure 2-2. Importing a GraphStudio solution

Graph Model

The Starter Kit is preloaded with a graph model based on an actual SVoD business. The name of the graph is *Entity_Resolution*. When you start GraphStudio, you are working at the global graph view. To switch to the Entity_Resolution graph, click on the circular icon in the upper left corner. A dropdown menu will appear, showing you the available graphs and letting you create a new graph. Click on Entity_Resolution.

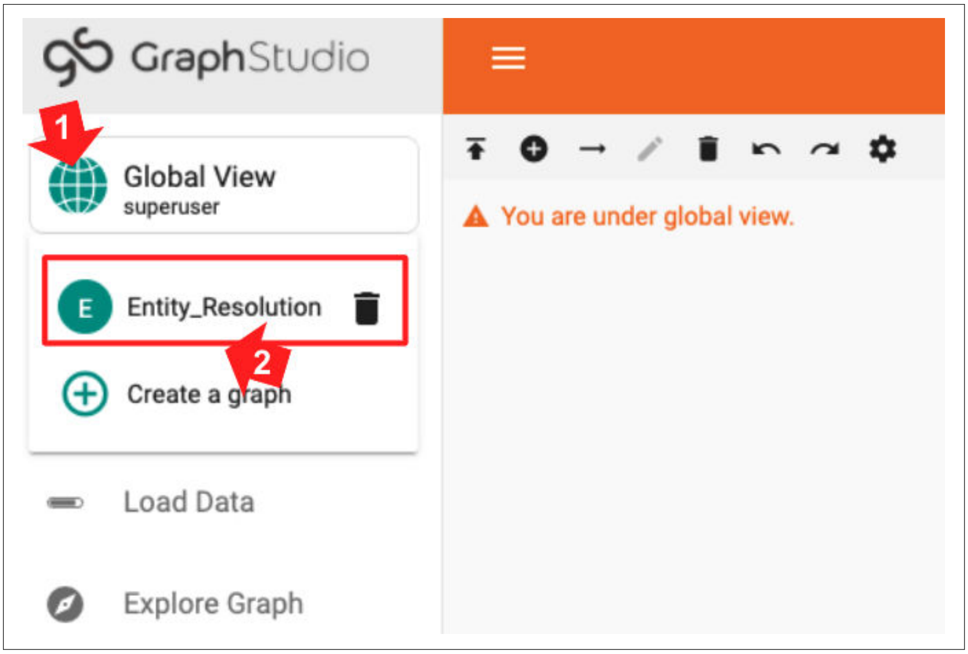


Figure 2-3. Selecting the graph to use

You should now see a graph model or schema like the one in Figure 2-4 in the main display panel.

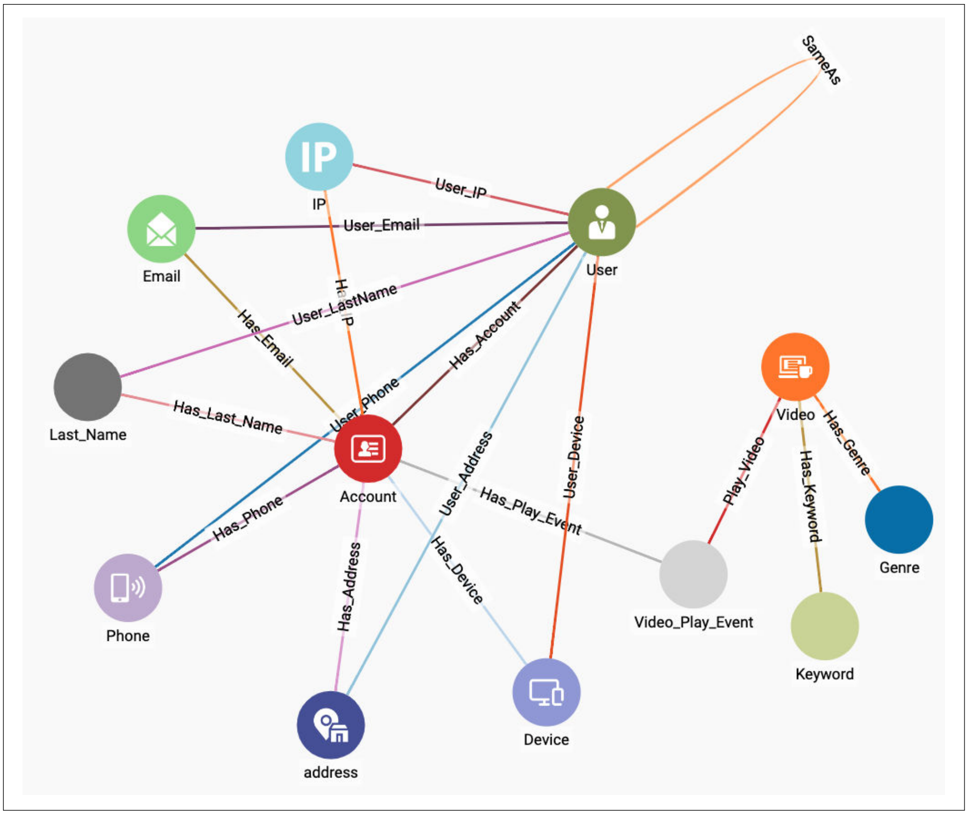


Figure 2-4. Graph schema for video customer accounts

You can see that Account, User, and Video are hub vertices with several edges radiating from them. The other vertices represent the personal information about users and the characteristics of videos. We want to compare the personal information of different users. Following good practice for graph-oriented analytics, if we want to see if two or more entities have something feature in common, e.g., email address, we model that feature as a vertex instead of as a property of a vertex. Table 2-1 gives a brief explanation of each of the vertex types in the graph model. Though the starter kit's data contains much data about videos, we will not focus on the videos themselves in this exercise. We are going to focus on entity resolution of Accounts.

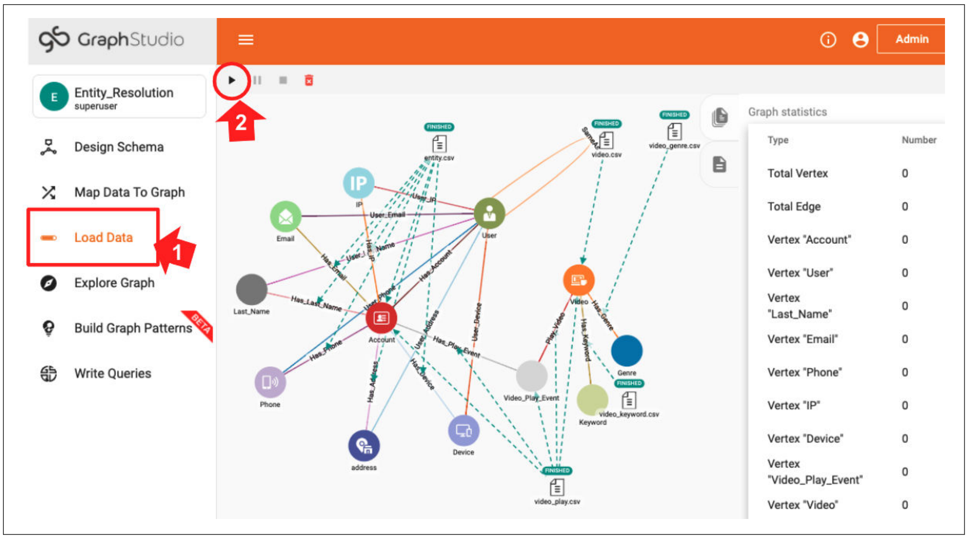
Table 2-1. Vertex types in the graph model

Account	an account for a SVoD user, a digital identity
User	a real-world person. One User can link to multiple Accounts
IP, Email, Last_User, Phone, Address, Device	attributes of an Account. They are represented as vertices instead of internal properties of Account to facilitate linking Accounts/Users that share a common attribute.
Video	a video title offered by an SVoD

Keyword, Genre	attributes of a Video
Video_Play_Event	the time and duration of a particular Account viewing a particular Video

Data Loading

In TigerGraph Starter Kits, the data is included, but it is not yet loaded into the database. To load the data, switch to the Load Data page (step 1 of Figure 2-5), wait a few seconds until the Load button in the upper left of the main panel becomes active, and then click it (step 2). You can watch the progress of the loading in the real-time chart at the right (not shown). Loading the 84K vertices and 270K edges should take only about 40 seconds on the TGCloud free instances; faster on the paid instances.

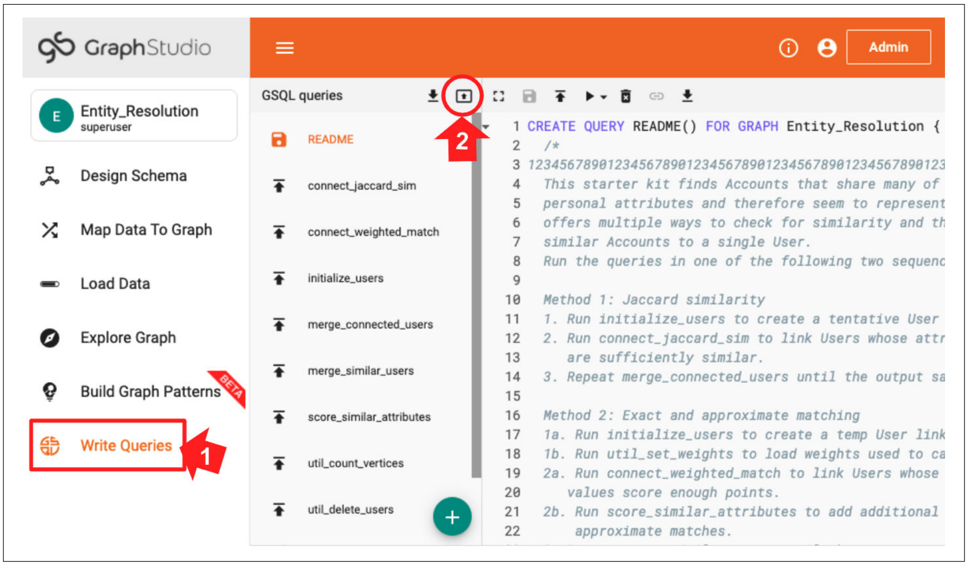


Type	Number
Total Vertex	0
Total Edge	0
Vertex "Account"	0
Vertex "User"	0
Vertex "Last_Name"	0
Vertex "Email"	0
Vertex "Phone"	0
Vertex "IP"	0
Vertex "Device"	0
Vertex "Video_Play_Event"	0
Vertex "Video"	0

Figure 2-5. Loading data in a Starter Kit

Queries and Analytics

We will analyze the graph and run graph algorithms by composing and executing queries in GSQL, TigerGraph's graph query language. When you first deploy a new Starter Kit, you need to install the queries. Switch to the Write Queries page (step 1 of Figure 2-6). Then at the top right of the list of queries, click the Install All icon (step 2).



```
CREATE QUERY README() FOR GRAPH Entity_Resolution {
  /*
12345678901234567890123456789012345678901234567890123
  This starter kit finds Accounts that share many of
  personal attributes and therefore seem to represent
  offers multiple ways to check for similarity and th
  similar Accounts to a single User.
  Run the queries in one of the following two sequenc

  Method 1: Jaccard similarity
  1. Run initialize_users to create a tentative User
  2. Run connect_jaccard_sim to link Users whose attr
     are sufficiently similar.
  3. Repeat merge_connected_users until the output sa

  Method 2: Exact and approximate matching
  1a. Run initialize_users to create a temp User link
  1b. Run util_set_weights to load weights used to ca
  2a. Run connect_weighted_match to link Users whose
      values score enough points.
  2b. Run score_similar_attributes to add additional
      approximate matches.
```

Figure 2-6. Installing queries

For our entity resolution use case, we have a three-stage plan requiring three or more queries.

Initialization

For each `Account` vertex, create its own `User` vertex and link them. Accounts are online identities, and Users represent real-world persons. We begin with the hypothesis that each `Account` is a real person.

Similarity Detection

Apply one or more similarity algorithms to measure the similarity between `User` vertices. If we consider a pair to be similar enough, then we create a link between them, using the `SameAs` edge type shown in Figure 2-5.

Merging

Find the connected components of linked User vertices. Pick one of them to be the main vertex. Transfer all of the edges of the other members of the community to the main vertex. Delete the other community vertices.

For reasons we will explain when we talk about merging, you may need to repeat steps 2 and 3 as a pair, until the Similarity Detection step is no longer creating any new connections.

We will present two different methods for implementing entity resolution in our use case. The first method uses Jaccard similarity (introduced in Chapter 10) to count exact matches of neighboring vertices. Merging will use a Connected Component algorithm. The second method is a little more advanced, suggesting a way to handle both exact and approximate matches of attribute values. Approximate matches are a good way to handle minor typos or the use of abbreviated names.

Method 1: Jaccard Similarity

For each of the three stages, we'll give a high level explanation, directions for operations to perform in TigerGraph's GraphStudio, what to expect as a result, and a closer look at some of the GSQL code in the queries.

Initialization

Recall in our model that an Account is a digital identity, and a User is a real person. The original database contains only Accounts. The initialization step creates a unique temporary User linked to each Account. And, for every edge from an Account to one of the attribute vertices (Email, Phone, etc.), we create a corresponding edge from the User to the same set of attribute vertices. Figure 2-7 shows an example. The three vertices on the left and the two edges connecting them are part of the original data. The initialization step creates the User vertex and the three dotted-line edges. As a result, each User starts out with the same attribute neighborhood as its Account.

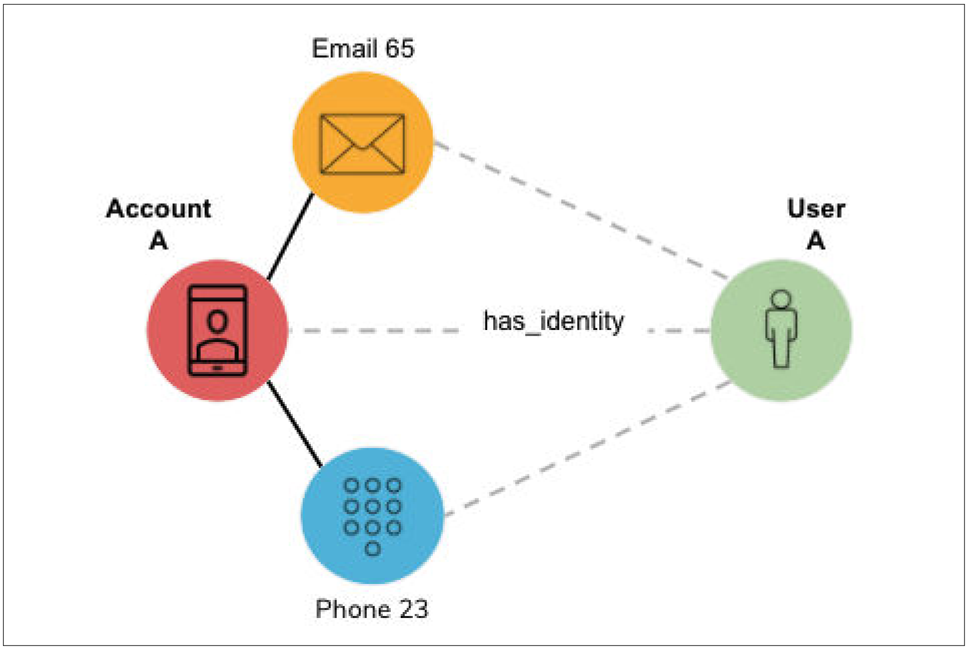
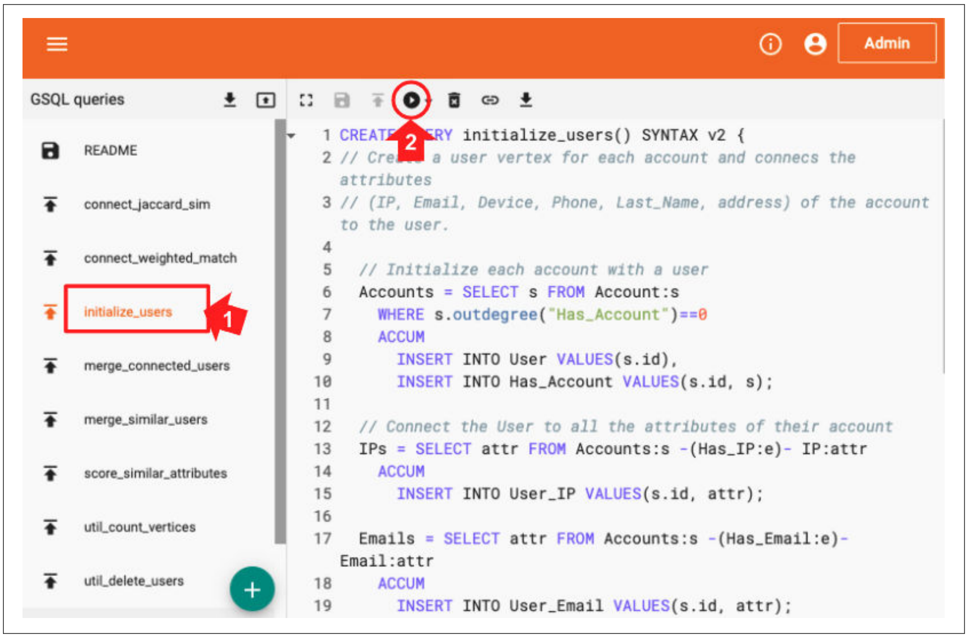


Figure 2-7. User vertex and edges created in the initialization step

Run the GSQL query `initialize_users` by selecting the query name from the list (step 1 in Figure 2-8) and then clicking the Run icon above the code panel (step 2).

This query has no input parameters, so it will run immediately without any additional steps from the user.



```
1  CREATE    ERY initialize_users() SYNTAX v2 {
2  // Cre    a user vertex for each account and connecs the
   attributes
3  // (IP, Email, Device, Phone, Last_Name, address) of the account
   to the user.
4
5    // Initialize each account with a user
6    Accounts = SELECT s FROM Account:s
7      WHERE s.outdegree("Has_Account")==0
8      ACCUM
9        INSERT INTO User VALUES(s.id),
10       INSERT INTO Has_Account VALUES(s.id, s);
11
12   // Connect the User to all the attributes of their account
13   IPs = SELECT attr FROM Accounts:s -(Has_IP:e)- IP:attr
14     ACCUM
15       INSERT INTO User_IP VALUES(s.id, attr);
16
17   Emails = SELECT attr FROM Accounts:s -(Has_Email:e)-
   Email:attr
18     ACCUM
19       INSERT INTO User_Email VALUES(s.id, attr);
```

Figure 2-8. Running the `initialize_users` query

Let's take a look at the GSQL code. The block of code below shows the first 20 lines of *initialize_users*. If you are familiar with SQL, then GSQL may look familiar. The comment at the beginning (lines 2 and 3) lists the 6 types of attribute vertices to be included. Lines 6 to 10 create a *User* vertex (line 9) and an edge connecting the *User* to the *Accounts* (line 10) for each *Account* (line 6) which doesn't already have a neighboring *User* (line 7).

```
1   CREATE QUERY initialize_users() SYNTAX v2 {
2   // Create a User vertex for each Account, plus edges to connect attributes
3   // (IP, Email, Device, Phone, Last_Name, Address) of the Account to the User
4
5   // Initialize each account with a user
6   Accounts = SELECT s FROM Account:s
7   WHERE s.outdegree("Has_Account")==0
8   ACCUM
9     INSERT INTO User VALUES(s.id),
10    INSERT INTO Has_Account VALUES(s.id, s);
11
12  // Connect the User to all the attributes of their account
13  IPs = SELECT attr FROM Accounts:s -(Has_IP:e)- IP:attr
14  ACCUM
15    INSERT INTO User_IP VALUES(s.id, attr);
16
17  Emails = SELECT attr FROM Accounts:s -(Has_Email:e)- Email:attr
18  ACCUM
19    INSERT INTO User_Email VALUES(s.id, attr);
20  // Remaining code omitted for brevity
21  }
```

GSQL's ACCUM clause is an iterator with parallel asynchronous processing. Think of it as "FOR EACH set of vertices and edges which match the pattern in the FROM clause, do the following."

Lines 13 to 15 take care of *IP* attribute vertices: If there is a Has_IP edge from an Account to an IP vertex, then insert an edge from the corresponding User vertex to the same IP vertex. While the alias `s` defined in line 13 refers to an Account, `s.id` in line 15 can refer to a User because the source vertex of a User_IP edge may only be a User, and we recently used `s.id` to create a User (line 9). Lines 17 to 19 take care of *Email* attribute vertices in an analogous way. The code blocks for the remaining four attribute types (*Device, Phone, Last_Name*, and *Address*) have been omitted for brevity.

Similarity Detection

Jaccard similarity counts how many attributes two entities have in common, divided by the total number of attributes between them. Each comparison of attributes results in a yes/no answer; a miss is as good as a mile. Figure 2-9 shows an example where User A and User B each have three attributes; two of those match (Email 65 and Device 87). Therefore, A and B have two attributes in common.

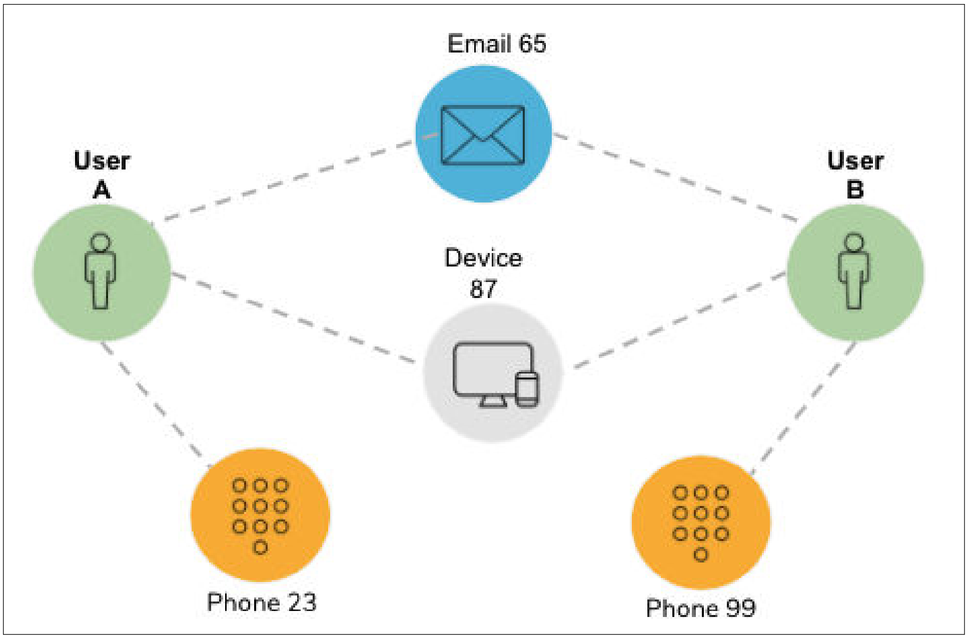


Figure 2-9. Jaccard similarity example

They have a total of four distinct attributes (Email 65, Device 87, Phone 23, and Phone 99); therefore, the Jaccard similarity is 2/4 = 0.5.

Do: Run *connect_jaccard_sim*.

The query *connect_jaccard_sim* computes this similarity score for each pair of vertices. If the score is at or above the given threshold, then create a Same_As edge to connect the two Users. The default threshold is 0.5, but you can make it higher or lower. Jaccard scores range from 0 to 1. Figure 2-10 shows the connections for User vertices 1, 2, 3, 4, and 5, using Jaccard similarity and a threshold of 0.5. For these five vertices, we need communities that range in size from one vertex alone (User 3) to two clusters of three (Users 1 and 2).

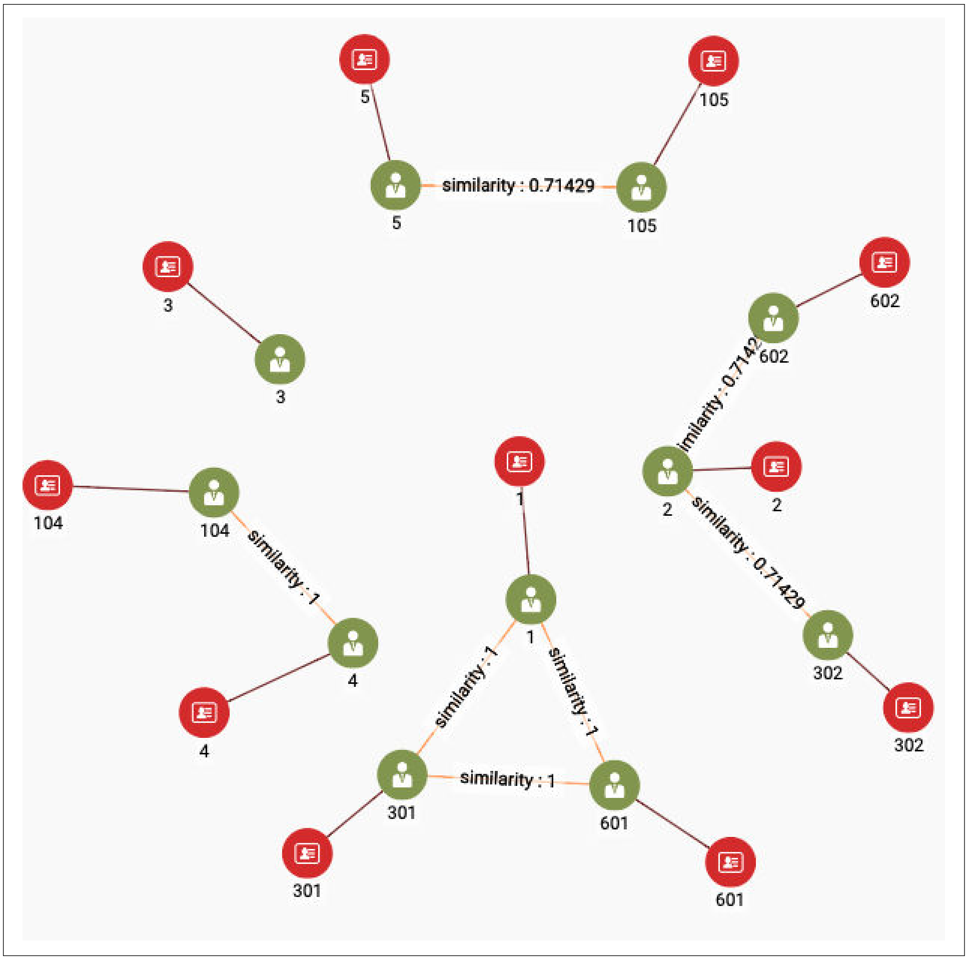


Figure 2-10. Connections for User vertices 1, 2, 3, 4, and 5, using Jaccard similarity and a threshold of 0.5

How to Visualize User Communities

You can use the Explore Graph page in TigerGraph GraphStudio to create a display like the one in Figure 2-10.

1. Click the Select Vertices icon at the top of the left-side menu (step 1 in Figure 2-11).
2. Choose the User type vertex. Enter the vertex ID 1 and click the Select icon. (step 2 in Figure 2-11). User 1 will appear in the display pane. Repeat for vertex IDs 2, 3, 4, and 5.

3. Shift-click the vertices so that all of them are selected (step 3, Figure 2-12).
4. Click the Expand From Selected Vertices icon, the next item in the left-side menu (step 4).
5. You are now presented with a checklist of all the edge types you wish to traverse, followed by a checklist of all the target vertex types. We want to include only the User and Account vertex types (step 5). This specifies a one-hop exploration.
6. We need to explore multiple hops, including the full communities. We don't know the diameter of the communities, but let's just guess that three hops is enough. Click the Add Expansion Step button at the bottom (step 6). Another set of checklists appear. Again, select only the User and Account vertex types. This is the second hop. Repeat these steps to set your third hop.
7. Click the Expand button above the checklists (step 7).

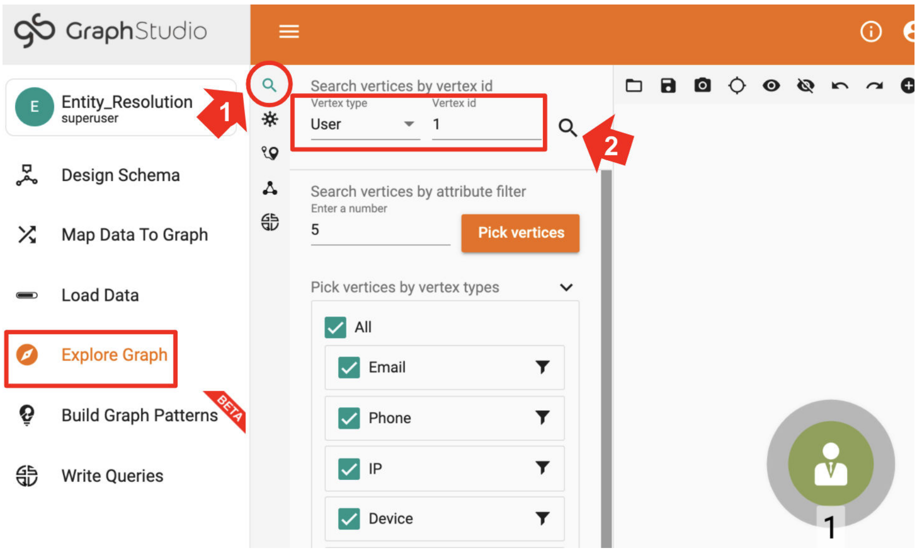
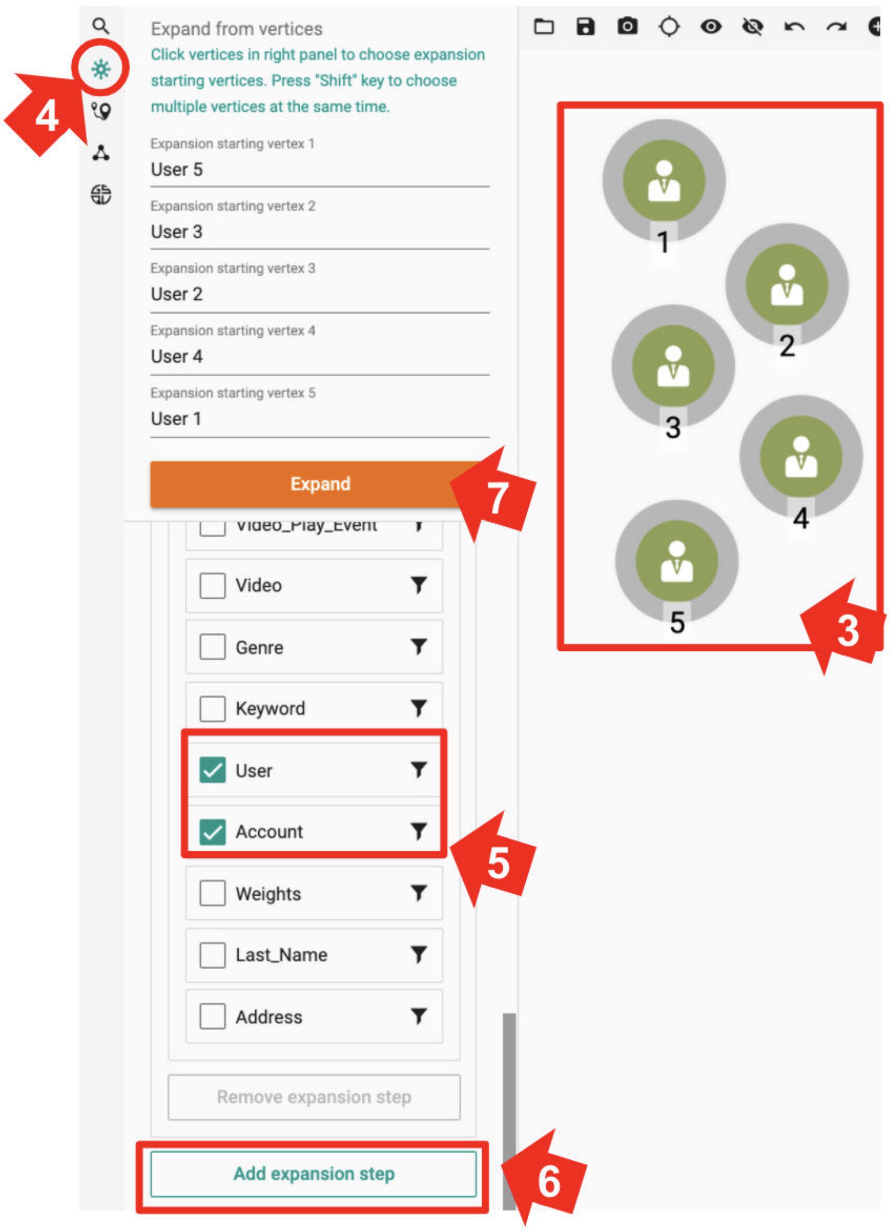


Figure 2-11. Selecting vertices on the Explore Graph page.



Expand from vertices

Click vertices in right panel to choose expansion starting vertices. Press "Shift" key to choose multiple vertices at the same time.

Expansion starting vertex 1: User 5

Expansion starting vertex 2: User 3

Expansion starting vertex 3: User 2

Expansion starting vertex 4: User 4

Expansion starting vertex 5: User 1

Expand

[✓ User]

[✓ Account]

Figure 2-12. Expanding from vertices

Rather than showing the full code, we'll just focus on a few excerpts. In the first code snippet, we'll show how to count the attributes in common between every pair of Users, using a single SELECT statement. The statement uses pattern-matching to describe how two such Users would be connected, and then it uses an accumulator to count the occurrences.

```
1   Others = SELECT B FROM
2   Start:A -()- (IP|Email|Phone|Last_Name|Address|Device):n -()- User:B
3   WHERE B != A
4   ACCUM
5     A.@intersection += (B -> 1); // tally each path A->B,
6     @@path_count += 1            // count the total number of paths
```

GSQL's FROM clause describes a left-to-right path moving from vertex to vertex via edges. Each sequence of vertices and edges which fit the requirements form one "row" in the temporary "table" of results, which is passed on to the ACCUM and POST-ACCUM clauses for further processing.

The FROM clause in lines 1 to 2 presents a graph path pattern to search for. `FROM User:A -()- (IP|Email|Phone|Last_Name|Address|Device):n -()- User:B` defines a two-hop path pattern:

- `User:A` means start from a User vertex, aliased to `A`,
- `-()-` means pass through any edge type
- `(IP|Email|Phone|Last_Name|Address|Device):n` means arrive at one of these six vertex types, aliased to `n`
- `-()-` means pass through another edge of any type, and finally
- `User:B` means arrive at a User vertex, aliased to `B`

In line 3, (`WHERE B != A`) ensures that we skip the situation of a loop where A = B. Line 4 announces the start of an ACCUM clause. Line 5 (`A.@intersection += (B -> 1); // tally each path A->B`) is a good example of GSQL's support for parallel processing and aggregation: for each path from A to B, append a key → value record attached to A. The record is (B, +=1). That is, if this is the first record associating B with A, then set the value to 1. For each additional record where B is A's target, then increment the value by 1. Hence, we're counting how many times there is a connection from A to B, via one of the six specified edge types. Line 6 (`ACCUM @@path_count += 1`) is just for bookkeeping purposes, to count now many of these paths we find.

Let's look at one more code block, the final computation of Jaccard similarity and creation of connections between Users.

```
Result = SELECT A FROM User:A
ACCUM FOREACH (B, overlap) IN A.@intersection DO
FLOAT score = overlap*1.0/(@@deg.get(A) + @@deg.get(B) - overlap),
IF score > threshold THEN
  INSERT INTO EDGE SameAs VALUES (A, B, score), // FOR Entity Res
  @@insert_count += 1,
  IF score != 1 THEN
    @@jaccard_heap += SimilarityTuple(A,B,score)
  END
END
END;
```

This SELECT block does the following:

1. For each User A, iterate over its record of similar Users B and the number of common neighbors, aliased to `overlap`.
2. For each such pair (A, B), compute the Jaccard score, using `overlap` as well as the number of qualified neighbors of A and B (`@@deg.get(A)` and `@@deg.get(B)`), computed earlier.
3. If the score is greater than the threshold, insert a SameAs edge between A and B.
4. `@@insert_count` and `@@jaccard_heap` are for reporting statistics and are not essential.

Merging

In our third and last stage, we merge together the connected communities of User vertices which we created in the previous step. For each community, we will select one vertex to be the survivor or lead. The remaining members will be deleted; all of the edges from an Account to a non-lead will be redirected to point to the lead User.

Do: run query *merge_connected_users*. Look at the JSON output. Note whether t says `converged = TRUE or FALSE.`

Figure 2-13 displays the User communities for Accounts 1, 2, 3, 4, and 5. The user communities have been reduced to a single User (real person). Each of those Users links to one or more Accounts (digital identities). We've achieved our entity resolution.

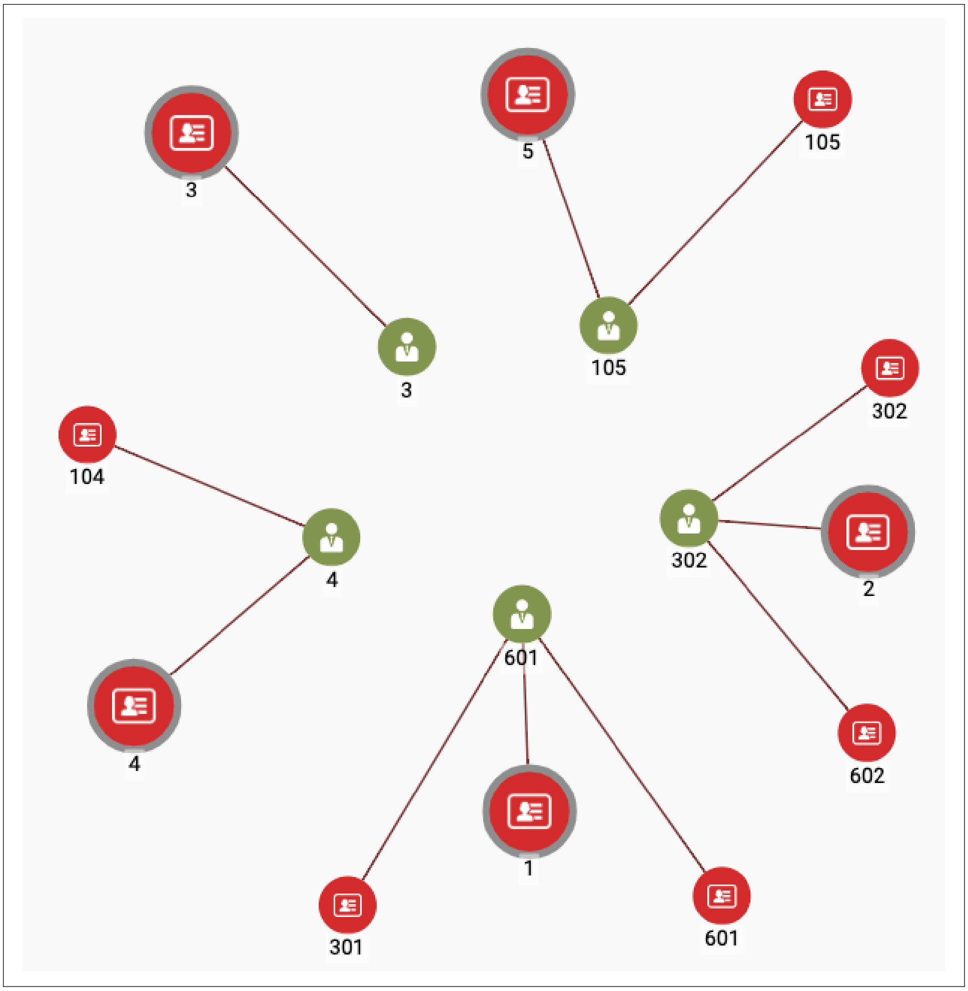


Figure 2-13. Entity resolution achieved, using Jaccard similarity

The *merge_connected_users* algorithm has four stages:

1. Find connected users using the connected component algorithm.
2. In each component, select a lead user.
3. In each component, redirect the attribute connections from other users to the lead user.
4. Delete the users that are not the lead user and all of the Same_As edges.

We'll take a closer look at the GSQL code for the Connected Components algorithm.

```
1   Users = {User.*};
2   Updated_users = SELECT s FROM Users:s
3     POST-ACCUM
4       s.@min_user_id = s;
5
6   WHILE (Updated_users.size() > 0) DO
7     IF verbose THEN PRINT iteration, Updated_users.size(); END;
8     Updated_users = SELECT t
9      FROM Updated_users:s -(SameAs:e)- User:t
10      // Propagate the internal IDs from source to target vertex
11     ACCUM t.@min_user_id += s.@min_user_id   // t gets the lesser of t & s ids
12     HAVING t.@min_user_id != t.@min_user_id' // accum is accum's previous val
13      ;
14    iteration = iteration + 1;
15  END;
```

Line 1 initializes the vertex set called `Users` to be all User vertices. Lines 2 to 4 initializes an accumulator variable `@min_user_id` for each User. The initial value is the vertex's internal ID (different from the externally visible ID). This variable will store the algorithm's current best guess at the community ID for this vertex: all vertices having the same value of @min_user_id belong to the same community. It's important to note that this accumulator is a MinAccum. Whenever you input a new value to a MinAccum, it retains the lesser of its current value and the new input value.

Lines 8 to 11 says that for each connected User pair from `s` to `t`, set `t.@min_user_id` to the lesser of `s` and `t`'s community IDs. Line 12 says that the output set (`Updated_users` in line 8) should contain only those User vertices who updated their community ID in this round. Note the tick mark ' at the end of the line; this is actually a modifier for the accumulator t.@min_user_id. It means "the previous value of the accumulator"; this lets us easily compare the previous to current values. When no vertices have changed their ID, then we can exit the WHILE loop (line 6).

Are We There Yet?

It might seem that one pass through the three steps – initialize, connect similar entities, and merge connected entities – should be enough. The merging, however, can create a situation in which new similarities arise. Take a look at Figure 2-14, which depicts the attribute connections of User 302 after Users 2 and 602 have been merged into it. Accounts 2, 302, and 602 remain separate, so you can see how each of them contributed some attributes. Because User 302 has more attributes than before, it is now possible that it is more similar than before to some other (possibly newly merged) User. Therefore, we should run another round of similarity connection and merge. Repeat these steps until no new similarities arise.

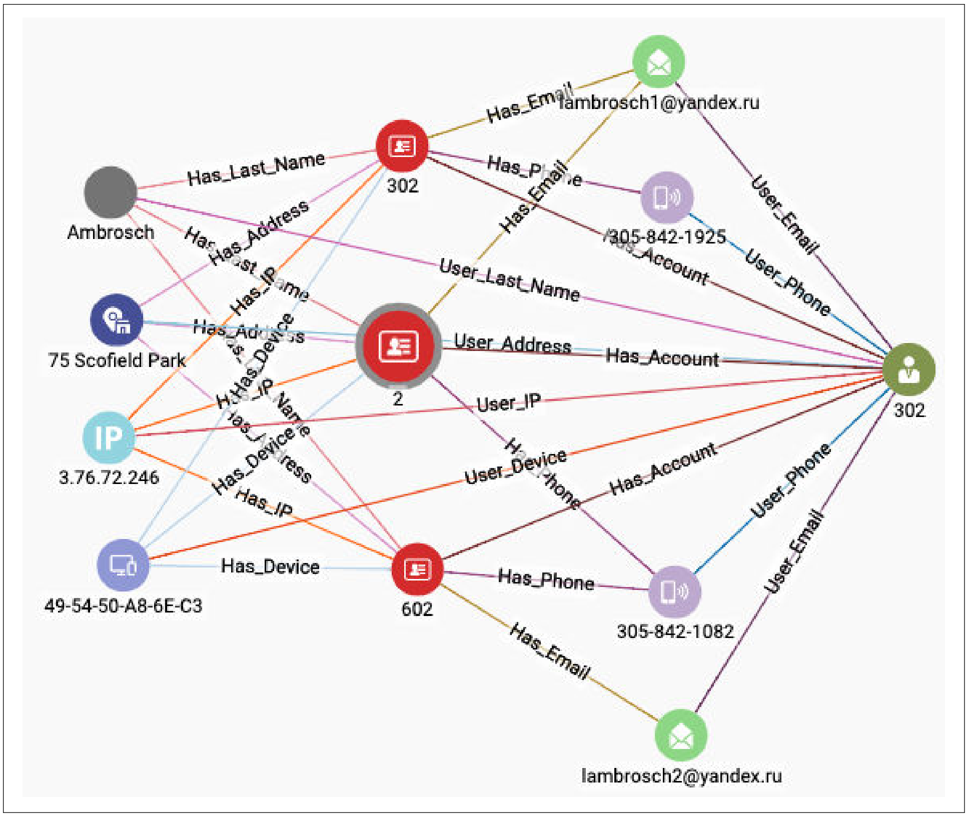


Figure 2-14. User with more attributes after entity resolution

For convenient reference, here is the sequence of queries we ran for simple entity resolution using Jaccard similarity:

Run *initialize_users*

Run *connect_jaccard_sim*

Run *merge_connected_users*

Repeat steps 2 & 3 until the output of *merge_connected_users* says conversed = TRUE

Reset

After you've finished, or at any time, you might want to restore the database to its original state. You need to do this if you want to run the entity resolution process from the start again. The query *util_delete_users* will delete all User vertices and all edges connecting to them. Note that you need to change the input parameter are_you_sure from FALSE to TRUE. This manual effort is put in as a safety precaution.

Deleting bulk vertices (*util_delete_users*) or creating bulk vertices (*initialize_users*) can take tens of seconds in the modestly sized free TigerGraph Cloud instances. Go to the Load Data page to check the live statistics for User vertices and User-related edges to see if the creation or deletion has finished.

Method 2: Scoring Exact and Approximate Matches

The previous section demonstrated a nice and easy graph-based entity resolution technique, but it is too basic for real-world use. It relies on exact matching of attribute values, whereas we need to allow for almost-the-same values, which arise from unintentional and intentional spelling variations. We also would like to make some attributes more important than others. For example, if you happen to have date-of-birth information, you might be strict about this attribute matching exactly. While persons can move, have multiple phone numbers and email addresses, they can only have one birthdate. In this section, we will introduce weights to adjust the relative importance of different attributes. We will also provide a technique for approximate matches of string values.

If you already used your starter kit to run Method 1, be sure to reset it. (See the Reset section at the end of Method 1.)

Initialization

We are still using the same graph model with User vertices representing real persons and Account vertices representing digital accounts. So, we are still using the initialize_users query to set up an initial set of User vertices.

We are adding another initialization step. We are going to assign weights to each of the six attributes: IP, Email, Phone, Address, Last_Name, and Device. If this were a relational database, we would store those weights in a table. Since this is a graph, we are going to store them in a vertex. We only need one vertex, because one vertex can have multiple attributes. However, we are going to be even more fancy. We are going to use a map type attribute, which will have six key → value entries. This allows us to use the map like a lookup table: tell me the name of the key (attribute name), and I'll tell you the value (weight).

Do:

1. Run *initialize_users*. Check the graph statistics on the Load Data page to make sure that all 900 Users and related edges have been created.

2. Run *util_set_weights*. The weights for the six attributes are input parameters for this query. Default weights are included, but you may change them if you wish. If you want to see the results, run *util_print_vertices*.

Scoring Weighted Exact Matches

We are going to do our similarity comparison and linking in two phases. In phase one, we are still checking for exact matches because exact matches are more valuable than approximate matches; however, those connections will be weighted. In phase two, we will then check for approximate matches for our two attributes which have alphabetic values: Last_Name and Address.

In weighted exact matching, we create weighted connections between Users, where higher weights indicate stronger similarity. The net weight of a connection is the sum of the contributions from each attribute that is shared by the two Users. Figure 2-15 illustrates the weighted match computation. Earlier, during the initialization phase, you established weights for each of the attributes of interest. In the figure, we use the names `wt_email` and `wt_phone` for the weights associated with matching Email and Phone attributes, respectively.

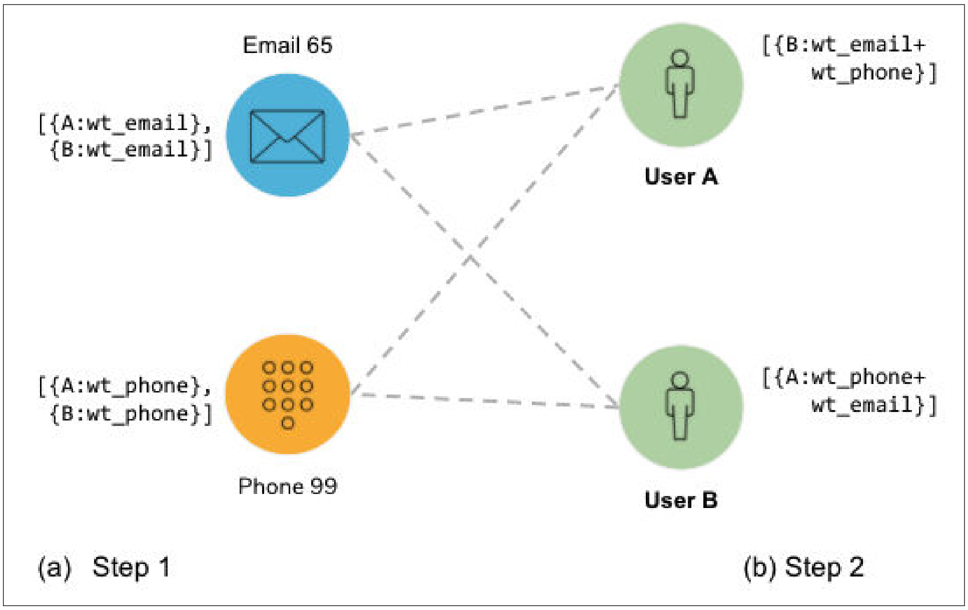


Figure 2-15. Two-phase calculation of weighted matches

The weighted match computation has two steps. In step 1, we look for connections from Users to Attributes and record a weight on each attribute for a connection to each User. Both User A and User B connect to Email 65, so Email 65 records

`A:wt_email` and `B:wt_email`. Each User's weight needs to be recorded separately. Phone 99 also connects to Users A and B so it records analogous information.

In step 2, we look for the same connections but in the other direction, with Users as the destinations. Both Email 65 and Phone 99 have connections to User A. User A aggregates their records from step 1. Note that some of those records refer to User A. User A ignores those, because it is not interested in connections to itself! In this example, it ends up recording `B:(wt_email + wt_phone)`. We use this value to create a weighted Same_As edge between Users A and B. You can see that User B has equivalent information about User A.

Do: run *connect_weighted_match*.

Figure 2-16 shows one of the communities generated by *connect_weighted_match*. This particular community is the one containing User/Account 5. The figure also shows connections to two attributes, Address and Last_Name. The other attributes such as Email were used in the scoring but are not shown, to avoid clutter.

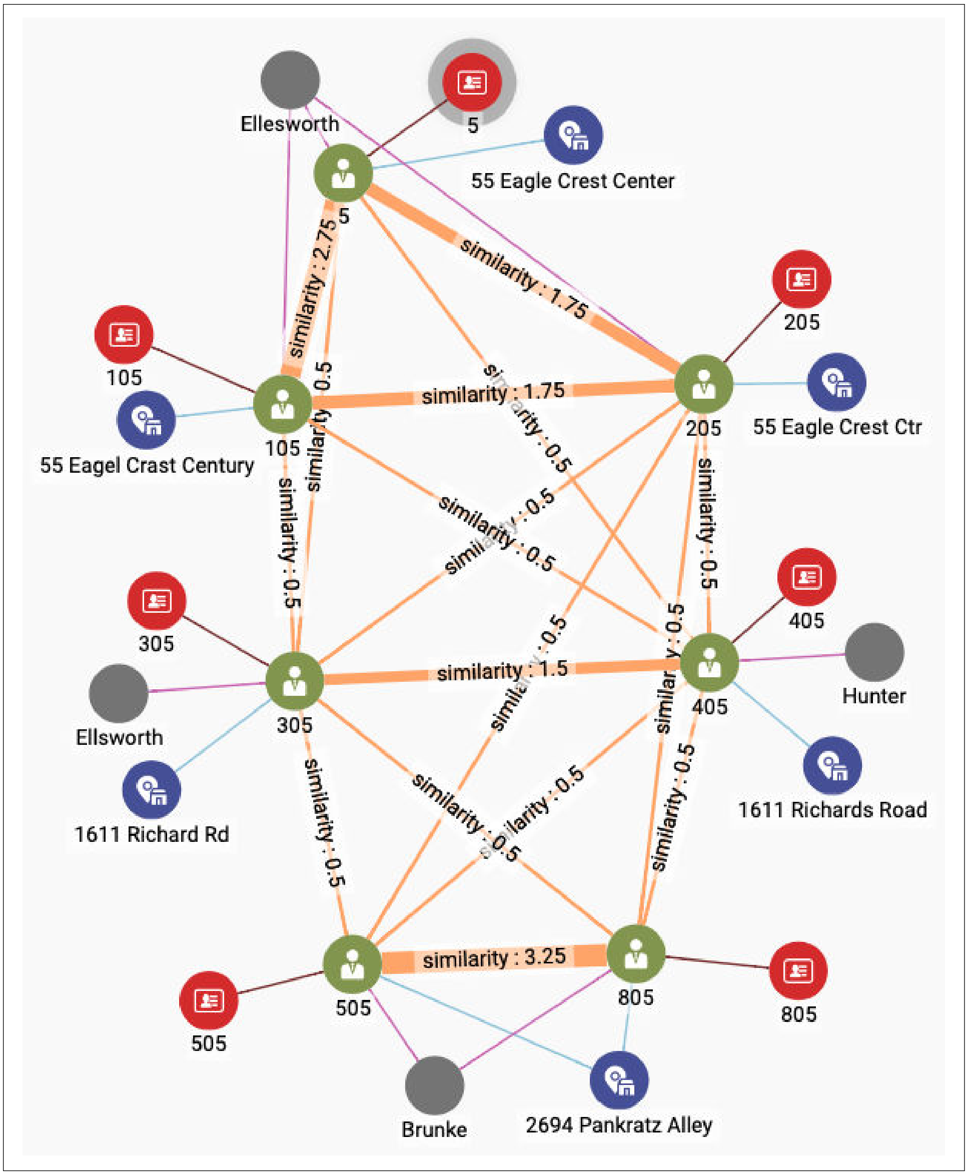


Figure 2-16. User community including Account 5 after exact weighted matching

The thickness of the SameAs edges indicates the strength of the connection. The strongest connection is between Users 505 and 805 at the bottom of the screen. In fact, we can see three subcommunities of Users among the largest community of seven members:

- Users 5, 105, and 205 at the top. The bond between Users 5 and 105 is a little stronger, for reasons not shown. All three share the same last name. They have similar addresses.
- Users 305 and 405 in the middle. Their last names and address are different, so some of the attributes not shown must be the cause of their similarity.
- Users 505 and 805 at the bottom. They share the same last name and address, as well as other attributes.

Scoring Approximate Matches

We can see (in Figure 2-16) that some Users have similar names (Ellsworth vs. Ellesworth) and similar addresses (Eagle Creek Center vs. Eagle Crest Ctr). A scoring system that looks only for exact matchings gives us no credit for these near misses. An entity resolution system would like to be able to assess the similarity of two text strings and to assign a score to the situation. Do they differ by a single letter, like Ellsworth and Ellsworth? Are letters transposed, like Center and Cneter? Computer scientists like to think of the *edit distance* between two text strings: how many single-letter changes of value or position are needed to transform string X into string Y?

We are going to use the Jaro-Winkler similarity to measure the similarity between two strings, an enhancement of Jaro similarity. Given two strings s1 and s2, who have m matching characters and t transformation steps between them, then their Jaro similarity is defined as shown in Equation 2-1.

Equation 2-1. Definition of Jaro similarity

$$Jaro(s1, s2) = \frac{1}{3}\left(\frac{m}{|s1|} + \frac{m}{|s2|} + \frac{m-t}{m}\right)$$

If the strings are identical, then m = |s1| = |s2|, and t =0, so the equation simplifies to (1 + 1 + 1)/3 = 1. On the other hand, if there are no letters in common, then the score = 0. We then multiply this score by the weight for the attribute type. For example, if the attribute's weight is 0.5, and if the JaroWinkler similarity score is 0.9, then the net score is 0.5 * 0.9 = 0.45.

Jaro-Winkler similarity takes Jaro as a starting point and adds an additional reward if the beginnings of each string, reading from the left end, match exactly.

Do: Run *score_similar_attributes.*

The *score_similar_attributes* query considers the User pairs which already are linked by a Same_As edge. It computes the weighted Jaro-Winkler similarity for the Last_Name and the Address attributes, and adds those scores to the existing similarity score. We chose Last_Name and Address because they are alphabetic instead of numeric. This is an application decision rather than a technical one. Figure 2-17 shows the results after adding in the scores for the approximate matches.

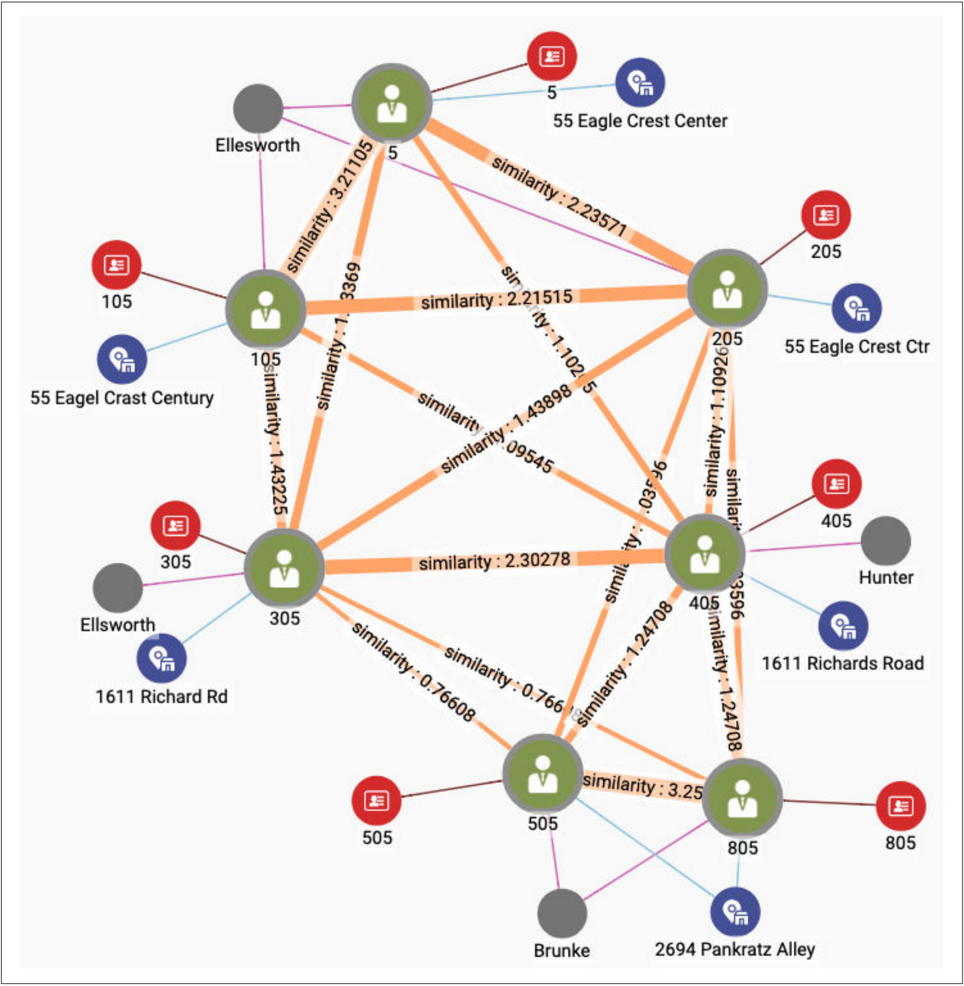


Figure 2-17. User community including Account 5 after exact and approximate weighted matching

Comparing Figures 11-16 and 11-17, we notice the following changes:

- The connections among Users 1, 105, and 205 have strengthened due to their having similar addresses.
- User 305 is more strongly connected to the trio above due to a similar last name.
- The connection between 305 and 405 has strengthened due to their having similar addresses.

- User 405 is more strongly connected to Users 505 and 805 due to the name Hunter having some letters in common with Brunke. This last effect might be considered an unintended consequence of the Jaro-Winkler similarity measure not being as judicious as a human evaluator would be.

Comparing two strings is a general purpose function which does not require graph traversal, so we have implemented it as a simple string function in GSQL. Because it is not yet a built-in feature of the GSQL language, we took advantage of GSQL's ability to accept a user-supplied C++ function as a user-defined function (UDF). The UDFs for `jaroDistance(s1, s2)` and `jaroWinklerDistance(s1, s2)` are included in this starter kit. You can invoke them from within a GSQL query anywhere that you would be able to call a built-in string function.

The code snippet below shows how we performed the approximate matching and scoring for the Address feature:

```
1   Connected_users = SELECT A
2     // Find all linked users, plus each user's address
3     FROM Connected_users:A -(SameAs:e)- User:B,
4        User:A -()- Address:A_addr,
5        User:B -()- Address:B_addr
6     WHERE A.id < B.id    // filter so we don't count (A,B) & (B,A)
7     ACCUM @@addr_match += 1,
8     // If addresses aren't identical compute JaroWinkler * weight
9     IF do_address AND A_addr.val != B_addr.val THEN
10      FLOAT sim = jaroWinklerDistance(A_addr.id,B_addr.id) * addr_wt,
11      @@sim_score += (A -> (B -> sim)),
12      @@string_pairs += String_pair(A_addr.id, B_addr.id),
13      IF sim != 0 THEN @@addr_update += 1 END
14    END
15  ;
```

Lines 3 to 5 are an example of a *conjunctive path pattern*, that is, a compound pattern comprising several individual patterns, separated by commas. The commas act like boolean AND. This conjunctive pattern means "find a User A linked to a User B, and find the Address connected to A, and find the Address connected to B." Line 6 filters out the case where A = B and prevents a pair (A, B) from being processed twice.

Line 9 filters out the case where A and B are different but have identical addresses. If their addresses are the same, then we already gave them full credit when we ran *connect_weighted_match*. Line 10 computes the weighted scoring, using the jaroWinklerDistance function and the weight for Address. Line 11 stores this value in a lookup table temporarily. Lines 12 and 13 are just to record our activity, for informative output at the end.

Merging Similar Entities

In Method 1, we had a simpler scheme for deciding whether to merge two entities: if their Jaccard score was greater than some threshold, then we created a SameAs edge. The decision was made. Merge everything that has a SameAs edge. We want a more nuanced approach now. Our scoring has adjustable weights, and the SameAs edges record our scores. We can use another threshold score to decide which Users to merge.

Take another look at Figure 2-17. We can see the effect of threshold score for merg-Ing. If we set the threshold at 3.0, there would be only two merges in this community: (5, 105) and (505, 805). If we set it at 2.0, we will get the three communities that we spoke about earlier: (5, 105, 205), (305, 405), and (505, 805). If we set the threshold at 1.0, all seven Users will be merged into 1.

We only need to make two small changes to merge_connected_users to let the user set a threshold.

- Add a threshold parameter to the query header:

  ```
  CREATE QUERY merge_connected_users(FLOAT threshold=1.0, BOOL ver
  bose=FALSE)
  ```

- In the connected component algorithm, add a WHERE clause to check the SameAs edge's similarity value:

```
WHILE (Updated_users.size() > 0) DO
  IF verbose THEN PRINT iteration, Updated_users.size(); END;
  Updated_users = SELECT t
    FROM Updated_users:s -(SameAs:e)- User:t
    WHERE e.1  similarity > threshold
      // Propagate the internal IDs from source to target vertex
        // t gets the lesser of t & s ids
    ACCUM t.@min_user_id += s.@min_user_id
        // accum' is accum's previous val
    HAVING t.@min_user_id != t.@min_user_id'
  ;
  iteration = iteration + 1;
END;
```

Run *merge_connected_users*. Pick a threshold value and see if you get the result that you expect.

Figure 2-18 shows the three different merging results for threshold values of 1.0, 2.0, and 3.0, to the community we have been following.

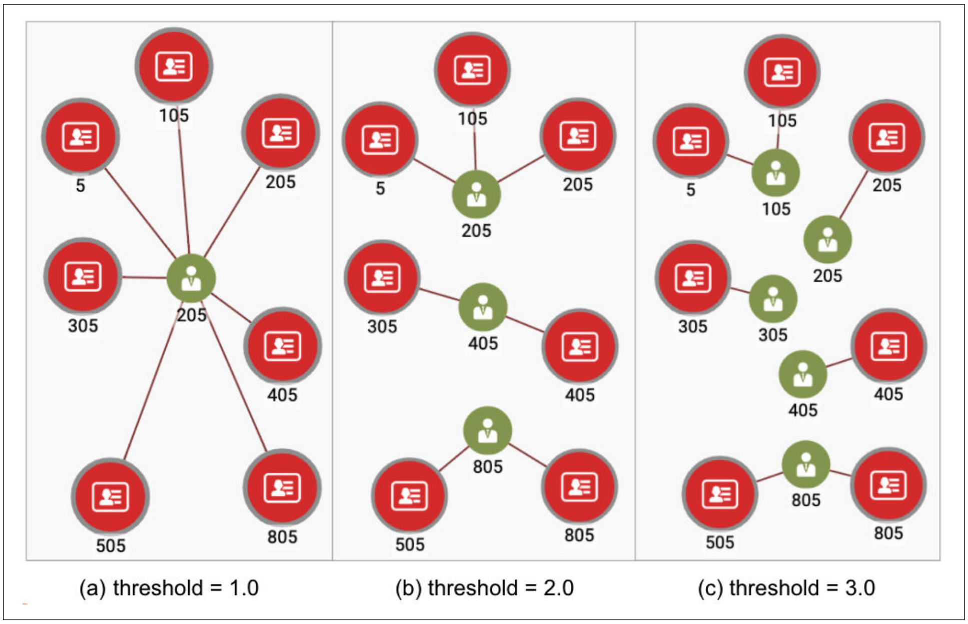


Figure 2-18. Entity resolution with different threshold levels

That concludes our second and more nuanced method of entity resolution.

For convenient reference, here is the sequence of queries we ran for entity resolution using weighted exact and approximate matching:

1. Run initialize_users
2. Run util_set_weights
3. Run connect_weighed_match
4. Run score_similar_attributes
5. Run merge_connected_users
6. Repeat steps 3, 4 and 5 until the output of merge_connected_users says conversed = TRUE

Chapter Summary

In this chapter we saw how graph algorithms and other graph techniques can be used for more sophisticated entity resolution than the simple entity resolution presented earlier in this book. Similarity algorithms and the connected component algorithm play key roles. We considered several schemes for assessing the similarity of two entities: Jaccard similarity, weighted sum of exact matches, and Jaro-Winkler similarity for comparing text strings.

These approaches can readily be extended to supervised learning if training data becomes available. There are a number of model parameters that can be learned, to improve the accuracy of the entity resolution: the scoring weights of each Attribute for exact matching, tuning the scoring of approximate matches, and thresholds for merging similar Users.

We saw how we use the FROM clause in GSQL queries to select data in a graph, by expressing a path or pattern. We also saw examples of the ACCUM clause and accumulators being used to compute and store information such as the common neighbors between vertices, a tally, an accumulating score, or even an evolving ID value, marking a vertex as a member of a particular community.

This chapter showed us how graph-based machine learning can improve the ability of enterprises to see the truth behind the data. In the next chapter, we'll apply graph machine learning to one of the most popular and important use cases: product recommendation.

About the Authors

Victor Lee is Head of Product Strategy and Developer Relations at TigerGraph. His Ph.D. dissertation was on graph-based similarity and ranking. Dr. Lee has co-authored book chapters on decision trees and dense subgraph discovery. Teaching and training have also been central to his career journey, with activities ranging from developing training materials for chip design to writing the first version of TigerGraph's technical documentation, from teaching 12 years as a full-time or part-time classroom instructor to presenting numerous webinars and in-person workshops.

Xinyu Chang is responsible for designing and supporting sophisticated graph database and analytics deployments around the globe. He excels at meeting customer needs with solutions that enable them to achieve better outcomes. Xinyu co-authored the GSQL query language with Dr. Alin Deutsch and others at TigerGraph. He has a Master's degree in Computer Science from Kent State University and a Bachelor's degree in Computer Science and Japanese from Dalian University of Foreign Languages.

Gaurav Deshpande is the VP of Marketing at TigerGraph. Gaurav brings TigerGraph a proven track record in leading teams in creating new products, establishing new markets, and dominating industries. At TigerGraph, Gaurav's team has grown the lead base, pipeline, and revenue 10x in three years and racked up over 50 awards including Most Disruptive Startup (Strata 2018), Gartner Cool Vendor 2020, and Leader in Forrester Wave Report for Graph Data Platforms Q4 2020. Previously, Gaurav spent 15 years overseeing marketing for IBM's Artificial Intelligence, Blockchain, and Cloud portfolios for the Banking and Financial markets, Telecommunications, and Retail. He also built out and positioned IBM's Big Data and Analytics portfolio, driving 45 percent year-over-year growth. Before IBM Gaurav led two startups through explosive growth, i2 Technologies (IPO), and Trigo Technologies (acquisition by IBM).