Chapter 1

Introduction and motivations

Graphs are a ubiquitous data structure and a universal language for describing complex systems. In the most general view, a graph is simply a collection of objects (i.e., nodes), along with a set of interactions (i.e., edges) between pairs of these objects. For example, to encode a social network as a graph we might use nodes to represent individuals and use edges to represent that two individuals are friends (Figure 1.1). In the biological domain we could use the nodes in a graph to represent proteins, and use the edges to represent various biological interactions, such as kinetic interactions between proteins (Figure 1.2).

The power of the graph formalism lies both in its focus on *relationships between points* (rather than the properties of individual points), as well as in its generality. The same graph formalism can be used to represent social networks, interactions between drugs and proteins, the interactions between atoms in a molecule, or the connections between terminals in a telecommunications network—to name just a few examples.

Graphs do more than just provide an elegant theoretical framework, however. They offer a mathematical foundation that we can build upon to analyze, understand, and learn from real-world complex systems. In the last twenty-five years, there has been an explosion in the quantity and quality of graph-structured data that is available to researchers. With the advent of large-scale social networking platforms, massive scientific initiatives to model the interactome, food webs, databases of molecule graph structures, and billions of interconnected web-enabled devices, there is no shortage of meaningful graph data for researchers to analyze. The challenge is unlocking the potential of this data.

This book is about how we can use *machine learning* to tackle this challenge. Of course, machine learning is not the only possible way to analyze graph data.[1] But given the ever-increasing scale and complexity of the graph datasets we

[1]The field of *network analysis* independent of machine learning is the subject of entire textbooks and will not be covered in detail here Newman [2018].

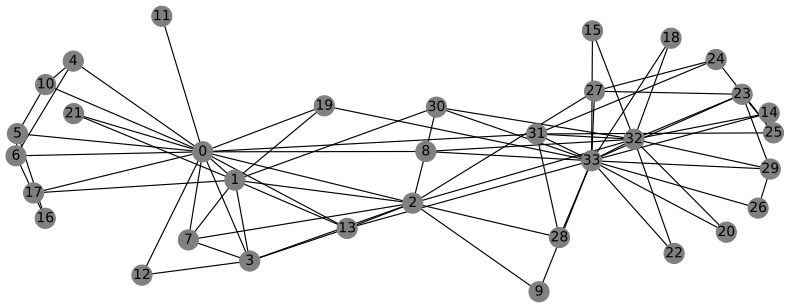

Figure 1.1: The famous *Zachary Karate Club Network* represents the friendship relationships between members of a karate club studied by Wayne W. Zachary from 1970 to 1972. An edge connects two individuals if they socialized outside of the club. During Zachary's study, the club split into two factions—centered around nodes 0 and 33—and Zachary was able to correctly predict which nodes would fall into each faction based on the graph structure [Zachary, 1977].

seek to analyze, it is clear that machine learning will play an important role in advancing our ability to model, analyze, and understand graph data.

1.1 What is a graph?

Before we discuss machine learning on graphs, it is necessary to give a bit more formal description of what exactly we mean by "graph data". Formally, a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is defined by a set of nodes $\mathcal{V}$ and a set of edges $\mathcal{E}$ between these nodes. We denote an edge going from node $u \in \mathcal{V}$ to node $v \in \mathcal{V}$ as $(u, v) \in \mathcal{E}$. In many cases we will be concerned only with *simple graphs*, where there is at most one edge between each pair of nodes, no edges between a node and itself, and where the edges are all undirected, i.e., $(u, v) \in \mathcal{E} \leftrightarrow (v, u) \in \mathcal{E}$.

A convenient way to represent graphs is through an *adjacency matrix* $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$. To represent a graph with an adjacency matrix, we order the nodes in the graph so that every node indexes a particular row/column. We can then represent the presence of edges as entries in this matrix: $\mathbf{A}[u, v] = 1$ if $(u, v) \in \mathcal{E}$ and $\mathbf{A}[u, v] = 0$ otherwise. If the graph contains only undirected edges then $\mathbf{A}$ will be a symmetric matrix, but if the graph is *directed* (i.e., edge direction matters) then $\mathbf{A}$ will not necessarily be symmetric. Some graphs can also have *weighted* edges, where $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ and the entries in the adjacency matrix are arbitary real-values rather than $\{0, 1\}$. For instance, a weighted edge in a protein-protein interaction graph might indicated the strength of the association between two proteins.

Beyond the distinction between undirected, directed and weighted edges, we will also consider graphs that have different *types* of edges. For instance, in graph representing drug-drug interactions, we might want different edges to

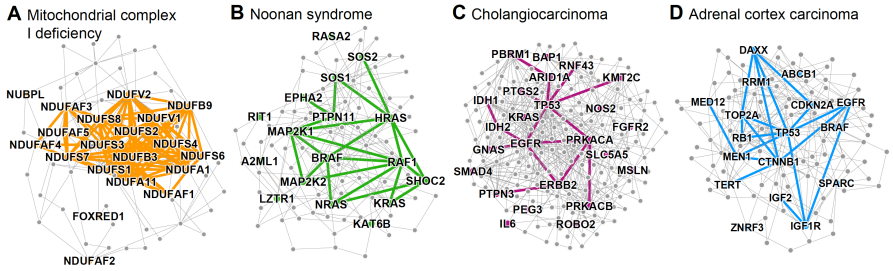

Figure 1.2: Each of the four subfigures illustrates a subset of the protein-protein interactions (PPI) known to occur in the human interactome. The bolded edges indicate interactions that are correlated with a specific disease, deficiency, or syndrome [Agrawal et al., 2018].

correspond to different side-effects that can occur when you take a pair of drugs at the same time. In these cases we can extend the edge notation to include an edge or relation type τ, e.g., $(u, \tau, v) \in \mathcal{E}$, and we can define one adjacency matrix $\mathbf{A}_\tau$ per edge type. We call such graphs *multi-relational*, and the entire graph can be summarized by an adjacency tensor $\mathcal{A} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{R}| \times |\mathcal{V}|}$, where $\mathcal{R}$ is the set of relations. Two important subsets of multi-relational graphs are often known as *heterogeneous* graphs and *multi-layer* graphs:

- In hetereogenous graphs, nodes are also imbued with *types*, meaning that we can partition the set of nodes into disjoint sets $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2 \cup ... \cup \mathcal{V}_k$ where $\mathcal{V}_i \cap \mathcal{V}_j = \emptyset, \forall i \neq j$. Edges in heterogeneous graphs generally satisfy constraints according to the node types, most commonly the constraint that certain edges only connect nodes of certain types, i.e., $(u, \tau_i, v) \in \mathcal{E} \rightarrow u \in \mathcal{V}_j, v \in \mathcal{V}_k$. For example, in a heterogeneous biomedical graph, there might be one type of node representing proteins, one type of representing drugs, and one type representing diseases. Edges representing "treatments" would only occur between drug nodes and disease nodes. Similarly, edges representing "polypharmacy side-effects " would only occur between two drug nodes. Note that in general hetergeneous graphs edges can connect two nodes that have the same type. *Multipartite* graphs are a well-known special case where edges can only connect nodes that have different types, i.e., $(u, \tau_i, v) \in \mathcal{E} \rightarrow u \in \mathcal{V}_j, v \in \mathcal{V}_k \wedge j \neq k$.

- In multi-layer graphs we assume that the graph can be decomposed in a set of k *layers*. Every node is assumed to belong to every layer, and each layer corresponds to a unique relation, representing the *intra-layer* edge type for that layer. We also assume that *inter-layer* edges types can exist, which connect the same node across layers. Multi-layer networks are best understood via examples. For instance, in a multi-layer transportation network, each node might represent a city and each layer might represent a different type of mode transportation (e.g., air travel or train travel). Intra-layer edges would then represent cities that are connected

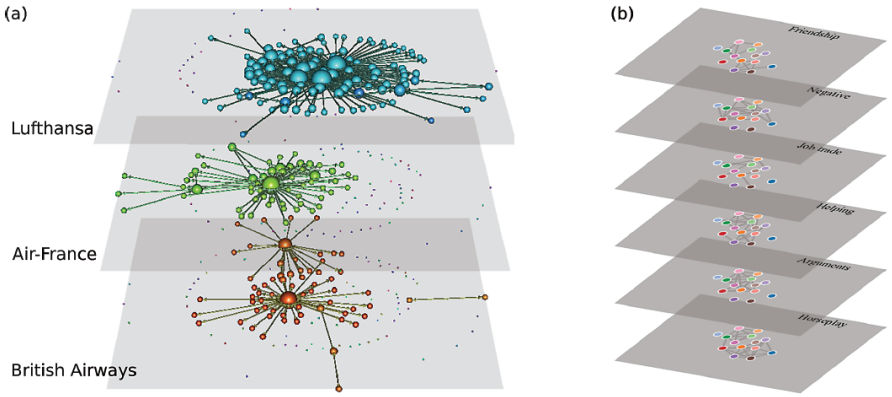

Figure 1.3: Illustration of two multi-layer graphs. The image on the left represents the network of connections between airports with the different layers representing different airlines. The image on the right represents a small social network, with the different edges representing different types of interpersonal relationships. Illustration taken from Porter [2018].

by different modes of transportation, while inter-layer edges represent the possibility of switching modes of transportation. Figure 1.3 illustrates two examples of multi-layer networks.

Lastly, in many cases we also have *attribute* or *feature* information associated with a graph (e.g., a profile picture associated with a user in a social network). Most often these are node-level attributes that we represent using a real-valued matrix $\mathbf{X} \in \mathbb{R}^{|V| \times m}$, where we assume that the ordering of the nodes is consistent with the ordering in the adjacency matrix. In heteregenous graphs we generally assume that each different type of node has its own distinct type of attributes. In rare cases we will also consider graphs that have real-valued edge features in addition to discrete edge types, and in some cases we even associate real-valued features with entire graphs.

Graph or network? We use the term "graph" in this book, but you will see many other resources use the term "network" to describe the same kind of data. In some places, we will use both terms (e.g., for social or biological networks). So which term is correct? In many ways, this terminological difference is a historical and cultural one: the term "graph" appears to be more prevalent in machine learning community[a], but "network" has historically been popular in the data mining and (unsurprisingly) network science communities. We use both terms in this book, but we also make a distinction between the usage of these terms. We use the term *graph* to describe the abstract data structure that is the focus of this book, but we will also often use the term *network* to describe specific, real-world instan-

tiations of this data structure (e.g., social networks). This terminological distinction is fitting with their current popular usages of these terms. *Network analysis* is generally concerned with the properties of real-world data, whereas *graph theory* is concerned with the theoretical properties of the mathematical graph abstraction.

[a]Perhaps in some part due to the terminological clash with "neural networks."

1.2 Machine learning on graphs

Machine learning is inherently a problem-driven discipline. We seek to build models that can learn from data in order to solve particular tasks, and machine learning models are often categorized according to the type of task they seek to solve: Is it a *supervised* task, where the goal is to predict a target output given an input datapoint? Is it an *unsupervised* task, where the goal is to infer patterns, such as clusters of points, in the data? Or perhaps it is a *reinforcement learning* task, where the goal is for the model to learn how to act in an environment to maximize some return of rewards.

Machine learning with graphs is no different, but the usual categories of supervised, unsupervised, and reinforcement learning are not necessarily the most informative or useful when it comes to graphs. In this section we provide a brief overview of the most important and well-studied machine learning tasks on graph data. As we will see, "supervised" problems are popular with graph data, but machine learning problems on graphs often blur the boundaries between these different categories—especially supervised and unsupervised learning.

Node classification

Suppose we are given a large social network dataset with millions of users, but we know that a significant number of these users are actually bots. Identifying these bots could be important for many reasons: a company might not want to advertise to bots or bots may actually be in violation of the social network's terms of service. Manually examining every user to determine if they are a bot would be prohibitively expensive, so ideally we would like to have a model that could classify users as a bot (or not) given only a small number of manually labeled examples.

This is a classic example of *node classification*, where the goal is to predict the label y_u—which could be a type, category, or attribute—associated with all the nodes $u \in \mathcal{V}$, when we are only given the true labels on a small *training set* of nodes $\mathcal{V}_{\text{train}} \subset \mathcal{V}$. Node classification is perhaps the most popular machine learning task on graph data, especially in recent years. Examples of node classification beyond social networks include classifying the function of proteins in the interactome [Hamilton et al., 2017b] and classifying the topic of documents based on hyperlink or citation graphs [Kipf and Welling, 2016].

At first glance, node classification appears to be a straightforward variation of standard supervised classification, but there are in fact important differences. The most important difference is that the nodes in a graph are not *independent and identically distributed (i.i.d.)*. Usually when we build supervised machine learning models we assume that each datapoint is statistically *independent* from all the other datapoints—otherwise, we might need to model the dependencies between all our input points—and we also assume that the datapoints are *identically distributed*—otherwise, we have no way of guaranteeing that our model will generalize to new datapoints. Node classification completely breaks this i.i.d. assumption. Rather than modeling a set of i.i.d. datapoints, we are instead modeling a set nodes that are interconnected with each other.

In fact, the key insight behind many of the most successful node classification approaches is to explicitly leverage the connections between nodes. One particularly popular idea is to exploit *homophily*, which is the tendency for nodes to share attributes with their neighbors in the graph [McPherson et al., 2001]. For example, people tend to form friendships with others who share the same interests or demographics. Based on the notion of homophily we can build machine learning models that try to assign similar labels to neighboring nodes in a graph [Zhou et al., 2004]. Beyond homophily there are also concepts such as *structural equivalence* [Donnat et al., 2018], which is the idea that nodes with similar local neighborhood structures will have similar labels, as well as *monophily*, which presumes that nodes will be preferentially connected to nodes with different labels.[2] When we build node classification models we want to exploit these concepts and model the relationships between nodes, rather than simply treating nodes as independent datapoints.

Supervised or semi-supervised? Due to the atypical nature of node classification, researchers often refer to it as *semi-supervised* [Yang et al., 2016]. This terminology is used because when we are training node classification models, we usually have access to the full graph, including all the unlabeled (e.g., test) nodes. The only thing we are missing is the labels of test nodes, but we can still use information about these test nodes (e.g., knowledge of their neighborhood in the graph) to improve our model during traing. This is another difference from the usual supervised setting, in which unlabeled datapoints are completely unobserved during training.

The general term used for models that combine labeled and unlabeled data during traning is semi-supervised learning, so it is understandable that this term is often used in reference to node classification tasks. It is important to note, however, that standard formulations of semi-supervised learning still require the i.i.d. assumption, which does not hold for node classification. Machine learning tasks on graphs do not easily fit our standard categories!

[2]For example, gender is an attribute that exhibits monophily in many social networks.

Relation prediction

Node classification is useful for inferring information about a node based on its relationship with other nodes in the graph. But what about cases where we are missing this relationship information? What if we know only some of protein-protein interactions that are present in a given cell, but we want to make a good guess about the interactions we are missing? Can we use machine learning to infer the edges between nodes in a graph?

This task goes by many names, such as link prediction, graph completion, and relational inference, depending on the specific application domain. We will simply call it *relation prediction* here. Along with node classification, it is one of the more popular machine learning tasks with graph data and has countless real-world applications: recommending content to users in social platforms [Ying et al., 2018a], predicting drug side-effects [Zitnik et al., 2018], or inferring new facts in a relational database [Bordes et al., 2013]—all of these tasks can be viewed as special cases of relation prediction.

The standard setup for relation prediction is that we are given a set of nodes $\mathcal{V}$ and an incomplete set of edges between these nodes $\mathcal{E}_{\text{train}} \subset \mathcal{E}$. Our goal is to use this partial information to infer the missing edges $\mathcal{E} \setminus \mathcal{E}_{\text{train}}$. The complexity of this task is highly dependent on the type of graph data we are examining. For instance, in simple graphs, such as social networks that only encode "friendship" relations, there are simple heuristics based on how many neighbors two nodes share that can achieve strong performance [Lü and Zhou, 2011]. On the other hand, in more complex multi-relational graph datasets, such as biomedical knowledge graphs that encode hundreds of different biological interactions, relation prediction can require complex reasoning and inference strategies [Nickel et al., 2016]. Like node classification, relation prediction blurs the boundaries of traditional machine learning categories—often being referred to as both supervised and unsupervised—and it requires inductive biases that are specific to the graph domain.

Clustering and community detection

Both node classification and relation prediction require inferring *missing* information about graph data, and in many ways, those two tasks are the graph analogues of supervised learning. *Community detection*, on the other hand, is the graph analogue of unsupervised clustering.

Suppose we have access to all the citation information in Google Scholar, and we make a *collaboration graph* that connects two researchers if they have co-authored a paper together. If we were to examine this network, would we expect to find a dense "hairball" where everyone is equally likely to collaborate with everyone else? It is more likely that the graph would segregate into different *clusters* of nodes, grouped together by research area, institution, or other demographic factors. In other words, we would expect this network—like many real-world networks—to exhibit a *community* structure, where nodes are much more likely to form edges with nodes that belong to the same community.

This is the general intuition underlying the task of community detection, and the challenge is to infer such latent community structures given only the input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. The many real-world applications of community detection include uncovering functional modules in genetic interaction networks [Agrawal et al., 2018] and uncovering fraudulent groups of users in financial transaction networks [Pandit et al., 2007].

Graph classification (and clustering)

The final class of popular machine learning applications on graph data involve classification (or clustering) problems over entire graphs. For instance, given a graph representing the structure of a molecule, we might want to build a model that could predict that molecule's toxicity or solubility [Gilmer et al., 2017]. Or, we might want to detect whether a computer program is malicious by analyzing a graph-based representation of its syntax and data flow [Li et al., 2019]. In these *graph classification* applications, we seek to learn over graph data, but instead of making predictions over the individual components of a single graph (i.e., the nodes and the edges), we are instead given a dataset of *multiple different graphs* and our goal is to make independent predictions specific to each graph. In the related task of *graph clustering* or similarity matching, the goal is to learn an unsupervised measure of similarity between a set of i.i.d. graphs.

Of all the machine learning tasks on graphs, graph classification is perhaps the most straightforward analogue of standard supervised classification. Each graph is an i.i.d. datapoint associated with a label, and the goal is to use a labeled set of training points to learn a mapping from datapoints (i.e., graphs) to labels. In a similar way graph clustering is the straightforward extension of unsupervised clustering for graph data. The challenge in graph classification and clustering, however, is how to define useful features that take into account the relational structure within each datapoint.

1.3 Background and Traditional Approaches

Before we introduce the concepts of graph representation learning and deep learning on graphs, it is necessary to give some methodological background and context. What kinds of methods were used for machine learning on graphs prior to the advent of modern deep learning approaches?

We will provide a very brief and focused tour of traditional learning approaches over graphs, providing pointers and references to more thorough treatments of these methodological approaches along the way. This background section will also serve to introduce key concepts from graph analysis that will form the foundation for later chapters.

Our tour will be roughly aligned with the different kinds of learning tasks on graphs. We will begin with a discussion of basic graph statistics, kernel methods, and their use for node and graph classification tasks. Following this, we will introduce and discuss various approaches for measuring the overlap between

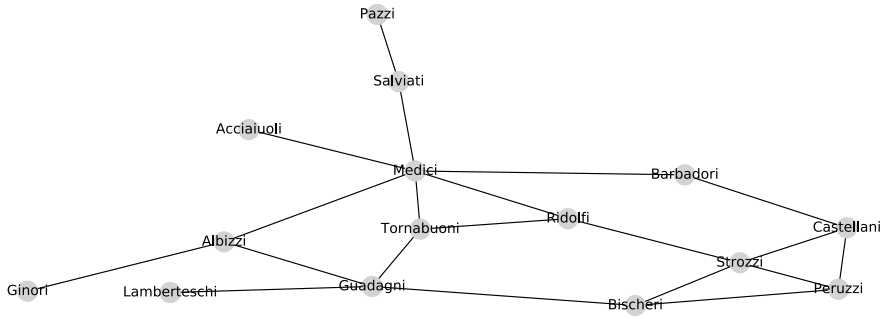

Figure 1.4: A visualization of the marriages between various different prominent families in 15th century Florence [Padgett and Ansell, 1993].

node neighborhoods, which form the basis of strong heuristics for link prediction. Finally, we will close this background section with a brief introduction of spectral clustering using graph Laplacians. Spectral clustering is one of the most well-studied algorithms for clustering or community detection on graphs, and our discussion of this technique will also introduce key mathematical concepts that will re-occur throughout this book.

1.3.1 Graph Statistics and Kernel Methods

Traditional approaches to classification using graph data follow the traditional (i.e., pre-deep-learning) machine learning paradigm. We begin by extracting some statistics or features—based on heuristic functions or domain knowledge—and then use these features as input to a standard machine learning classifier (e.g., logistic regression). In this section, we will first introduce some important node-level features and statistics, and we will follow this by a discussion of how these node-level statistics can be generalized to graph level statistics and extended to design kernel methods over graphs. Our goal will be to introduce various heuristic statistics and graph properties, which are often used as features in traditional machine learning pipelines applied to graphs.

Node-level statistics and features

Following Jackson [2010], we will motivate our discussion of node-level statistics and features with a simple (but famous) social network: the network of 15th century Florentine marriages (Figure 1.4). This network is well-known due to the work of Padgett and Ansell [1993], which used this network to illustrate the rise in power of the Medici family (depicted near the center) who came to dominate Florentine politics. Political marriages were an important way to consolidate power during the era of the Medicis, so this network of marriage connections encodes a great deal about the political structure of this time.

For our purposes, we will consider this network and the rise of the Medici from a machine learning perspective and ask the question: What features or statistics could a machine learning model use to predict the Medici's rise? In other words, what properties or statistics of the Medici family node distinguish it from the rest of the graph? And, more generally, what are useful properties and statistics that we can use to characterize the nodes in this graph.

In principle the properties and statistics we discuss below could be used as features in a node classification model (e.g., input to a logistic regression model). Obviously, we would not be able to realistically train a machine learning model on a graph as small as the Florentine marriage network, but it is still illustrative to consider the kinds of features that could be used to differentiate the nodes in such a real-world graph.

Node degree. The most obvious and straightforward node feature to examine is *degree*, which is usually denoted d_u for a node $u \in \mathcal{V}$ and simply counts the number of edges incident to a node:

$$d_u = \sum_{v \in V} \mathbf{A}[u, v]. \tag{1.1}$$

Note that in cases of directed and weighted graphs, one can differentiate between different notions of degree—e.g., corresponding to outgoing edges or incoming edges by summing over rows or columns in Equation (1.1). In general, the degree of a node is an essential statistic to consider, and it is often one of the most informative features in traditional machine learning models applied to node-level tasks.

In the case of our Florentine marriages graph, we can see that degree is indeed a good feature to distinguish the Medici family, as they have the highest degree in the graph. However, their degree only outmatches the two closest families—the Strozzi and the Guadagni—by a ratio of 3 to 2. Are there perhaps additional or more discriminative features that can help to distinguish the Medici family from the rest of the graph?

Node centrality. Node degree simply measures how many neighbors a node has, but this is not necessarily sufficient to measure the *importance* of a node in a graph. In many cases—such as our example graph of Florentine marriages—we can benefit from additional and more powerful measures of node importance. To obtain a more powerful measure of importance, we can consider various measures of what is known as node *centrality*, which can form useful features in a wide variety of node classification tasks.

One popular and important measure of centrality is the so-called *eigenvector centrality*. Whereas degree simply measures how many neighbors each node has, eigenvector centrality also takes into account how important a node's neighbors are. In particular, we define a node's eigenvector centrality e_u via a recurrence relation in which the node's centrality is proportional to the average centrality

of its neighbors:

$$e_u = \frac{1}{\lambda} \sum_{v \in V} \mathbf{A}[u, v] e_v \; \forall u \in \mathcal{V}, \tag{1.2}$$

where λ is constant. Rewriting this equation in vector notation with $\mathbf{e}$ as the vector of node centralities, we can see that this recurrence defines the standard eigenvector equation for the adjacency matrix:

$$\lambda \mathbf{e} = \mathbf{A}\mathbf{e}. \tag{1.3}$$

In other words, the centrality measure that satisfies the recurrence in Equation 1.2 corresponds to an eigenvector of the adjacency matrix. Assuming that we require positive centrality values, we can apply the Perron-Frobenius Theorem to further determine that the vector of centrality values $\mathbf{e}$ is given by the eigenvector corresponding to the largest eigenvalue of $\mathbf{A}$ [Newman, 2016].

One view of eigenvector centrality is that it ranks the likelihood that a node is visited on a random walk of infinite length on the graph. This view can be illustrated by considering the use of power iteration to obtain the eigenvector centrality values. That is, since λ is the leading eigenvector of $\mathbf{A}$, we can compute $\mathbf{e}$ using power iteration via[3]

$$\mathbf{e}^{(t+1)} = \mathbf{A}\mathbf{e}^{(t)}. \tag{1.4}$$

If we start off this power iteration with the vector $\mathbf{e}^{(0)} = (1, 1, ..., 1)^\top$, then we can see that after the first iteration $\mathbf{e}^{(1)}$ will contain the degrees of all the nodes. In general, at iteration $t \geq 1$, $\mathbf{e}^{(t)}$ will contain the number of length-t paths arriving at each node. Thus, iterating this process indefinitely we obtain a score that is proportional to the number of times a node is visited on paths of infinite length. This connection between node importance, random walks, and the spectrum of the graph adjacency matrix will return often throughout the ensuing sections and chapters.

Returning to our example of the Florentine marriage network, if we compute the eigenvector centrality values on this graph, we again see that the Medici family is the most influential, with a normalized value of 0.43 compared to the next-highest value of 0.36. There are, of course, other measures of centrality that we could use to characterize the nodes in this graph—some of which are even more discerning with respect to the Medici family's influence. These include *betweeness centrality*—which measures how often a node lies on the shortest path between two other nodes—as well as *closeness centrality*—which measures the average shortest path length between a node and all other nodes. These measures and many more are reviewed in detail by Newman [2018].

The clustering coefficient. Measures of importance, such as degree and centrality are clearly useful for distinguishing the prominent Medici family from the rest of the Florentine marriage network. But what about features that are

[3]Note that we have ignored the normalization in the power iteration computation for simplicity, as this does not change the main result.

useful for distinguishing between the other nodes in the graph? For example, the Peruzzi and Guadagni nodes in the graph have very similar degree (3 v.s. 4) and similar eigenvector centralities (0.28 v.s. 0.29). However, looking at the graph in Figure 1.4, there is a clear difference between these two families. Whereas the the Peruzzi family is in the midst of a relatively tight-knit cluster of families, the Guadagni family occurs in a more star-like role.

This important structural distinction can be measured using variations of the *clustering coefficient*, which measures the proportion of closed triangles in a node's local neighborhood. The popular *local variant* of the clustering coefficient is computed as [Watts and Strogatz, 1998]:

$$c_u = \frac{|(v_1, v_2) \in \mathcal{E} \ : \ v_1, v_2 \in \mathcal{N}(u)|}{\binom{d_u}{2}}. \tag{1.5}$$

The numerator simply counts the number of edges between neighbours of node u—where we use $\mathcal{N}(u) = \{v \in \mathcal{V} \ : \ (u, v) \in \mathcal{E}\}$ to denote the node neighborhood. The denominator calculates how many pairs of nodes there are in u's neighborhood.

The clustering coefficient takes its name from the fact that it measures how tightly clustered a node's neighborhood is. A clustering coefficient of 1 would imply that all of u's neighbors are also neighbors of each other. In our Florentine marriage graph, we can see that some nodes are highly clustered—e.g., the Peruzzi nodes has a clustering coefficient of 0.66—while other nodes such as the Guadagni node have clustering coefficients of 0. As with centrality, there are numerous variations of the clustering coefficient (e.g., to account for directed graphs), which are also reviewed in detail by Newman [2018]. An interesting and important property of real-world networks throughout the social and biological sciences is that they tend to have far higher clustering coefficients than one would expect if edges were sampled randomly [Watts and Strogatz, 1998].

Closed triangles, ego graphs, and motifs. An alternative way of viewing the clustering coefficient—rather than as a measure of local clustering—is that it counts the number of closed triangles within each node's local neighborhood. In more precise terms, the clustering coefficient is related to the ratio between the actual number of triangles and the total possible number of triangles within a node's *ego graph*, i.e., the subgraph containing that node, its neighbors, and all the edges between nodes in its neighborhood.

This idea can be generalized to the notion of counting arbitrary *motifs* or *graphlets* within a node's ego graph. That is, rather than just counting triangles, we could consider more complex structures, such as cycles of particular length, and we could characterize nodes by counts of how often these different motifs occur in their ego graph. By examining a node's ego graph in this way, we can essentially transform the task of computing node-level statistics and features to a graph-level task. Thus, we will now turn our attention to this graph-level problem.

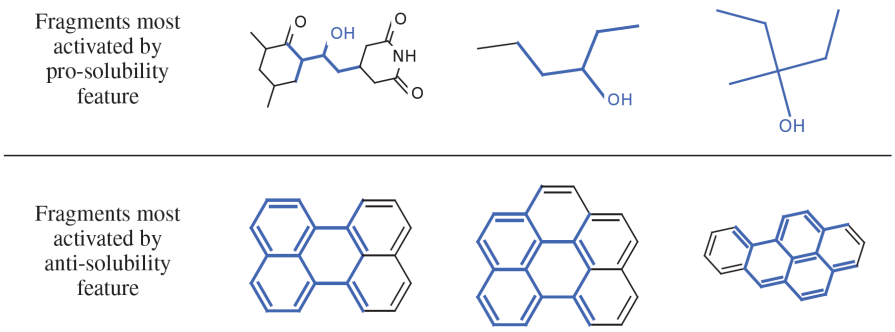

Figure 1.5: Visualization of molecular graph structures from the ESOL dataset Delaney [2004]. The graphs in the top row represent chemical compounds that are highly soluble with the blue areas denoting fragments that Duvenaud et al. [2015] detected to be highly indictive of solubility, while the bottom row are anti-soluble compounds. Visualization from Duvenaud et al. [2015].

Graph-level features and graph kernels

So far we have discussed various statistics and properties at the node level, which could be used as features for node-level classification tasks. However, what if our goal is to do graph-level classification? For example, suppose we are given a dataset of graphs representing molecules and our goal is to classify the solubility of these molecules based on their graph structure (Figure 1.5). In this section we will briefly survey approaches to extracting graph-level features for such tasks.

Many of the methods we survey here fall under the general classification of *graph kernel methods*, which are approaches to designing features for graphs or implicit kernel functions that can be used in machine learning models. We will touch upon only a small fraction of the approaches within this large area, and we will focus on methods that extract explicit feature representations, rather than approaches that define implicit kernels (i.e., similarity measures) between graphs. We point the interested reader to Kriege et al. [2019] and Vishwanathan et al. [2010] for detailed surveys of this area.

Bag of nodes. The simplest approach to defining a graph-level feature is to just aggregate node-level statistics. For example, one can compute histograms or other summary statistics based on the degrees, centralities, and clustering coefficients of the nodes in the graph, and then use this aggregated information as a graph-level representation. The downside to this approach is that it is entirely based upon local node-level information and can miss important global properties in the graph.

The Weisfieler-Lehman kernel. One way to improve the basic bag of nodes approach is using a strategy of *iterative neighborhood aggregation.* The idea with these approaches is to extract node-level features that contain more information than just their local ego graph, and then to aggregate these richer features into a graph-level representation.

Perhaps the most important and well-known of these strategies is the Weisfieler-Lehman (WL) algorithm and kernel [Shervashidze et al., 2011]. The basic idea behind the WL algorithm is the following:

1. First, we assign an initial label $l^{(0)}(v)$ to each node. In most graphs, this label is simply the degree, i.e., $l^{(0)}(v) = d_v \; \forall v \in V$.

2. Next, we iteratively assign a new label to each node by hashing the multi-set of the current labels within the node's neighborhood:

$$l^{(i)}(v) = \text{HASH}(\{\{l^{(i-1)}(u) \; \forall u \in \mathcal{N}(v)\}\}), \tag{1.6}$$

   where the double-braces are used to denote a multi-set and the HASH function maps each unique multi-set to a unique new label.

3. After running K iterations of re-labeling (i.e., Step 2), we now have a label $l^{(K)}(v)$ for each node that summarizes the structure of its K-hop neighborhood. We can then compute histograms or other summary statistics over these labels as a feature representation for the graph, i.e., the WL kernel is computed by measuring the difference between the resultant label sets for two graphs.

The WL kernel is popular, well-studied and known to have important theoretical properties. For example, one popular way to approximate graph isomorphism is to check whether or not two graphs have the same label set after T rounds of the WL algorithm, and this approach is known to solve the isomorphism problem for a broad set of graphs [Shervashidze et al., 2011].

The WL approach goes by many names—such as naive vertex refinement [Hamilton et al., 2017b] and molecular fingerprints in the biochemistry community [Duvenaud et al., 2015]—and it is a fundamentally important algorithm in graph analysis. As we will see in Part II of this book, graph neural networks—which are the modern standard for deep learning on graphs—have close theoretical connections to the WL algorithm.

Graphlets and path-based methods Finally, just as in our discussion of node-level features, one valid and powerful strategy for defining features over graphs is to simply count the occurrence of different small subgraph structures, usually called *graphlets* in this context. Formally, the graphlet kernel involves enumerating all possible graph structures of a particular size and counting how many times they occur in the full graph (see Figure 1.6 for an example with graphlets of size 3). The challenge with this approach is that computing graphlets of size k is at least as hard as solving the graph isomorphism problem for graphs of size k. Counting these graphlets is a combinatorially difficult

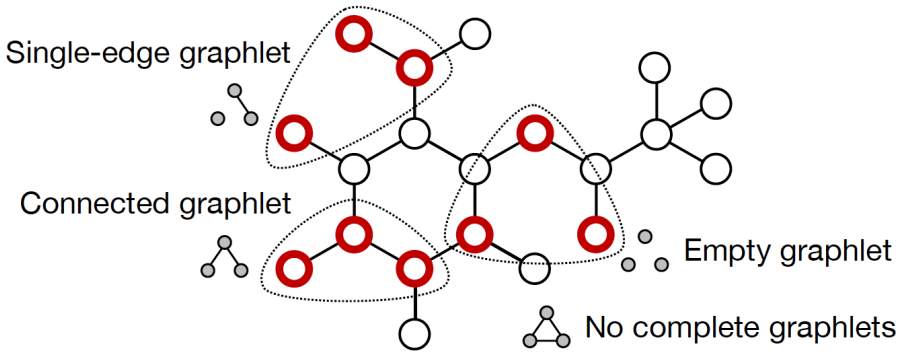

Figure 1.6: Illustration of size-3 graphlets in a simple graph. Figure taken from Kriege et al. [2019].

problem, though numerous approximations have been proposed [Shervashidze and Borgwardt, 2009].

An alternative to enumerating all possible graphlets is to use *path-based methods*. In these approaches, rather than enumerating graphlets, one simply examines the different kinds of *paths* that occur in the graph. For example, the random walk kernel proposed by Kashima et al. [2003] involves running random walks over the graph and then counting the occurrence of different degree sequences,[4] while the shortest-path kernel of Borgwardt and Kriegel [2005] involves a similar idea but uses only the shortest-paths between nodes (rather than random walks). As we will see in Part I of this book, this idea of characterizing graphs based on walks and paths is a powerful one, as it can extract rich structural information while avoiding many of the combinatorial pitfalls of graph data.

1.3.2 Neighborhood Overlap Detection

In the last section we covered various approaches to extract features or statistics about individual nodes or entire graphs. These node and graph-level statistics are useful for many classification tasks, but they are limited in that they do not quantify the *relationships* between nodes. For instance, the statistics discussed in the last section are not very useful for the task of relation prediction, where our goal is to predict the existence of an edge between two nodes (Figure 1.7).

In this section we will consider various statistical measures of neighborhood overlap between pairs of nodes, which quantify the extent to which a pair of nodes are related. For example, the simplest neighborhood overlap measure just counts the number of neighbors that two nodes share:

$$\mathbf{S}[u, v] = |\mathcal{N}(u) \cap \mathcal{N}(v)|, \tag{1.7}$$

[4] Other node labels can also be used.

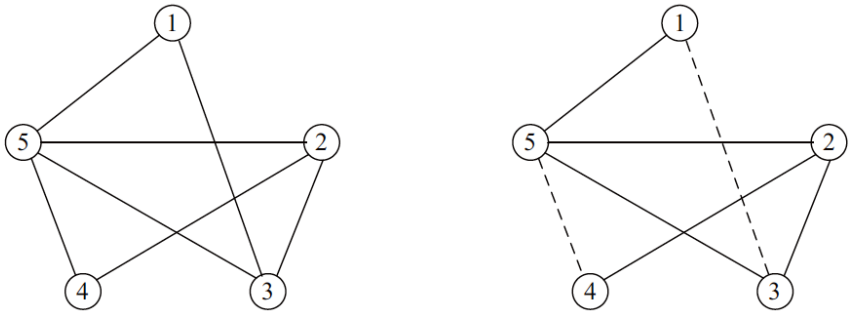

Figure 1.7: An illustration of the full graph and a subsampled graph used for training. The dotted edges in the training graph are removed when training a model or computing the overlap statistics.

where we use $\mathbf{S}[u, v]$ to denote the value quantifying the relationship between nodes u and v and let $\mathbf{S} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ denote the *similarity matrix* summarizing all the pairwise node statistics.

While there is no learning involved in any of the statistical measures discussed in this section, they are still very useful and powerful baselines for relation prediction. Given a neighborhood overlap statistic $\mathbf{S}[u, v]$, a common strategy is to assume that the likelihood of an edge is simply proportional to this statistic:

$$P(\mathbf{A}[u, v] = 1) \propto \mathbf{S}[u, v]. \tag{1.8}$$

Thus, in order to approach relation prediction using a neighborhood overlap measure, one simply needs to choose a particular overlap statistic and then set a threshold to determine when to predict the existence of an edge. Note that in the relation prediction setting we generally assume that we only know a subset of the true edges $\mathcal{E}_{\text{train}} \subset \mathcal{E}$. Our hope is that node-node similarity measures computed on the training edges will lead to accurate predictions about the existence of test (i.e., unseen) edges.

Local overlap measures

Local overlap statistics are simply functions of the number of common neighbors two nodes share, i.e. $|\mathcal{N}(u) \cap \mathcal{N}(v)|$. For instance, the Sorensen index defines a matrix $\mathbf{S}_{\text{Sorenson}} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ of node-node neighborhood overlaps with entries given by

$$\mathbf{S}_{\text{Sorenson}}[u, v] = \frac{2|\mathcal{N}(u) \cap \mathcal{N}(v)|}{d_u + d_v}, \tag{1.9}$$

which normalizes the count of common neighbors by the sum of the node degrees. Normalization of some kind is usually very important; otherwise, the overlap measure would be highly biased towards predicting edges for nodes with large degrees. Other similar approaches include the the Salton index normalizes by the product of the node degrees

$$\mathbf{S}_{\text{Salton}}[u, v] = \frac{2|\mathcal{N}(u) \cap \mathcal{N}(v)|}{\sqrt{d_u d_v}}, \tag{1.10}$$

and the Jaccard overlap:

$$\mathbf{S}_{\text{Jaccard}}(u, v) = \frac{|\mathcal{N}(u) \cap \mathcal{N}(v)|}{|\mathcal{N}(u) \cup \mathcal{N}(v)|}. \tag{1.11}$$

In general these measures seek to measure the overlap between neighborhoods while minimizing the bias towards high vs. low-degree nodes, and there are many variations of normalizing constants in the literature [Lü and Zhou, 2011].

There are also measures that go beyond simply counting the number of common neighbors and that seek to consider the *importance* of common neighbors in some way. The Resource Allocation (RA) index counts the inverse degrees of the common neighbors,

$$\mathbf{S}_{\text{RA}}[v_1, v_2] = \sum_{u \in \mathcal{N}(v_1) \cap \mathcal{N}(v_2)} \frac{1}{d_u}, \tag{1.12}$$

while the Adamic-Adar (AA) index performs a similar computation using the inverse logarithm of the degrees:

$$\mathbf{S}_{\text{AA}}[v_1, v_2] = \sum_{u \in \mathcal{N}(v_1) \cap \mathcal{N}(v_2)} \frac{1}{\log(d_u)}. \tag{1.13}$$

Both these measures give more weight to common neighbors that have low degree, with intuition that a shared low-degree neighbor is more informative than a shared high-degree one.

Global overlap measures

Local overlap measures are extremely effective heuristics for link prediction and often achieve competitive performance even compared to advanced deep learning approaches [Perozzi et al., 2014]. However, the local approaches are limited in that they only consider local node neighborhoods. For example, two nodes could have no local overlap in their neighborhoods but still be members of the same community in the graph. *Global overlap* statistics attempt to take such relationships into account.

Katz index The Katz index is the most basic global overlap statistic. To compute the Katz index we simply count the number of paths *of all length*

between a pair of nodes:

$$\mathbf{S}_{\text{Katz}}[u, v] = \sum_{i=1}^{\infty} \beta^i \mathbf{A}^i[u, v], \tag{1.14}$$

where β is a weighting term that controls how much weight is given to short v.s. long paths. In particular, a small value of $\beta < 1$ would down-weight the importance of long paths.

Geometric series of matrices The Katz index is one example of a geometric series of matrices, variants of which occur frequently in graph analysis and graph representation learning. The solution to these basic form of this series is given by the following theorem.

Theorem 1. *Let* $\mathbf{X}$ *be a real-valued matrix and let* λ_1 *denote the largest eigenvalue of* $\mathbf{X}$*. Then*

$$(\mathbf{I} - \mathbf{X})^{-1} = \sum_{i=0}^{\infty} \mathbf{X}^i$$

if and only if $\lambda_1 < 1$ *and* $(\mathbf{I} - \mathbf{X})$ *is non-singular.*

Proof. Let $s_n = \sum_{i=0}^{n} \mathbf{X}^i$ then we have that

$$\begin{aligned} \mathbf{X}s_n &= \mathbf{X} \sum_{i=0}^{n} \mathbf{X}^i \\ &= \sum_{i=1}^{n+1} \mathbf{X}^i \end{aligned}$$

and

$$\begin{aligned} s_n - \mathbf{X}s_n &= \sum_{i=0}^{n} \mathbf{X}^i - \sum_{i=1}^{n+1} \mathbf{X}^i \\ s_n(\mathbf{I} - \mathbf{X}) &= \mathbf{I} - \mathbf{X}^{n+1} \\ s_n &= (\mathbf{I} - \mathbf{X}^{n+1})(\mathbf{I} - \mathbf{X})^{-1} \end{aligned}$$

And if $\lambda_1 < 1$ we have that $\lim_{n\to\infty} \mathbf{X}^n = 0$ so

$$\begin{aligned} \lim_{n\to\infty} s_n &= \lim_{n\to\infty} (\mathbf{I} - \mathbf{X}^{n+1})(\mathbf{I} - \mathbf{X})^{-1} \\ &= \mathbf{I}(\mathbf{I} - \mathbf{X})^{-1} \\ &= (\mathbf{I} - \mathbf{X})^{-1} \end{aligned}$$

□

Based on Theorem 1, we can see that the solution to the Katz index is given by

$$\mathbf{S}_{\text{Katz}} = (\mathbf{I} - \beta\mathbf{A})^{-1} - \mathbf{I}, \tag{1.15}$$

where $\mathbf{S}_{\text{Katz}} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ is the full matrix of node-node similarity values.

Leicht, Holme, and Newman (LHN) similarity One issue with the Katz index is that it is strongly biased by node degree. Equation (1.14) is generally going to give higher overall similarity scores when considering high-degree nodes, compared to low-degree ones, since high-degree nodes will generally be involved in more paths. To alleviate this, Leicht et al. [2006] propose an improved metric by considering the ratio between the actual number of observed paths and the *number of expected paths between two nodes*:

$$\frac{\mathbf{A}^i}{\mathbb{E}[\mathbf{A}^i]}, \tag{1.16}$$

i.e., the number of paths between two nodes is normalized based on our expectation of how many paths we expect under a random model.

To compute the expectation $\mathbb{E}[\mathbf{A}^i]$ rely on what is called the *configuration model*, which assumes that we draw a random network with the same set of degrees as our given network. Under this assumption, we can analytically compute that

$$\mathbb{E}[\mathbf{A}[u,v]] = \frac{d_u d_v}{2m}, \tag{1.17}$$

which states that under a random configuration model the likelihood of an edge is simply proportional to the product of the two node degrees. This can be seen by noting that there are d_u edges leaving u and each of these edges has a $\frac{d_v}{2m}$ chance of ending at v. For $\mathbb{E}[\mathbf{A}^2[u,v]]$ we can similarly compute

$$\mathbb{E}[\mathbf{A}^2[v_1,v_2]] = \frac{d_{v_1} d_{v_2}}{(2m)^2} \sum_{u\in\mathcal{V}} (d_u - 1)d_u. \tag{1.18}$$

This follows from the fact that path of length 2 could pass through any intermediate vertex u, and the likelihood of such a path is proportional to the likelihood that an edge leaving v_1 hits u—given by $\frac{d_{v_1} d_u}{2m}$—multiplied by the probability that an edge leaving u hits v_2—given by $\frac{d_{v_2}(d_u-1)}{2m}$ (where we subtract one since we have already used up one of u's edges for the incoming edge from v_1).

Unfortunately the analytical computation of expected node path counts under a random configuration model becomes intractable as we go beyond paths of length three. Thus, to obtain the expectation $\mathbb{E}[\mathbf{A}^i]$ for longer path lengths (i.e., $i > 2$), Leicht et al. [2006] rely on the fact the largest eigenvalue can be used to approximate the growth in the number of paths. In particular, if we define $\mathbf{p}_i \in \mathbb{R}^{|\mathcal{V}|}$ as the vector counting the number of length-i paths between node u and all other nodes, then we have that for large i

$$\mathbf{A}\mathbf{p}_i = \lambda_1 \mathbf{p}_{i-1}, \tag{1.19}$$

since $\mathbf{p}_i$ will eventually converge to the dominant eigenvector of the graph. This implies that the number of paths between two nodes grows by a factor of λ_1 at

each iteration, where λ_1 is the first eigenvalue of $\mathbf{A}$. Based on this approximation for large i as well as the exact solution for $i = 1$ we obtain:

$$\mathbb{E}[\mathbf{A}^i[u, v]] = \frac{d_u d_v \lambda^{i-1}}{2m}. \tag{1.20}$$

Finally, putting it all together we can obtain a normalized version of the Katz index—which we term the LNH index:

$$\mathbf{S}_{\text{LNH}}[u, v] = \mathbf{I}[u, v] + \frac{2m}{d_u d_v} \sum_{i=0}^{\infty} \beta \lambda_1^{1-i} \mathbf{A}^i[u, v], \tag{1.21}$$

where $\mathbf{I}$ is a $|\mathcal{V}| \times |\mathcal{V}|$ identity matrix indexed in a consistent manner as $\mathbf{A}$. Unlike the Katz index the LNH index accounts for the *expected* number of paths between nodes and only gives a high similarity measure if two nodes occur on more paths than we expect. Using Theorem 1 the solution to the matrix series (after ignoring diagonal terms) can be written as [Lü and Zhou, 2011]:

$$\mathbf{S}_{\text{LNH}} = 2\alpha m \lambda_1 \mathbf{D}^{-1} (\mathbf{I} - \frac{\beta}{\lambda_1} \mathbf{A})^{-1} \mathbf{D}^{-1}, \tag{1.22}$$

where $\mathbf{D}$ is a matrix with node degrees on the diagonal.

Random walk methods

Another set of global similarity measures consider random walks rather than exact counts of paths over the graph. For example, we can directly apply the PageRank approach [Page et al., 1999], where we define the stochastic matrix $\mathbf{P} = \mathbf{A}\mathbf{D}^{-1}$ and compute:

$$\mathbf{q}_u = c\mathbf{P}\mathbf{q}_u + (1 - c)\mathbf{e}_u. \tag{1.23}$$

In this equation $\mathbf{e}_u$ is a one-hot indicator vector for node u and $\mathbf{q}_u[v]$ gives the stationary probability that random walk starting at node u visits node v. Here, the c term determines the probability of the random walk restarting at node u at each timestep. Without this restart probability, the random walk probabilities would simply converge to a normalized variant of the eigenvector centrality. However, with this restart probability we instead obtain a measure of importance specific to the node u, since the random walks are continually being "teleported" back to that node. The solution to this recurrence is given by

$$\mathbf{q}_u = (1 - c)(\mathbf{I} - c\mathbf{P})^{-1}\mathbf{e}_u, \tag{1.24}$$

and we can define a node-node random walk similarity measure as

$$\mathbf{S}_{\text{RW}}[u, v] = \mathbf{q}_u[v] + \mathbf{q}_v[u], \tag{1.25}$$

i.e., the similarity between a pair of nodes is proportional to how likely we are to reach each node from a random walk starting from the other node.

1.3.3 Graph Laplacians and Spectral Methods

We now turn to the problem of learning to cluster the nodes in graph. This section will also motivate the task of learning low dimensional embeddings of nodes. We begin with the definition of some important matrices that can be used to represent graphs.

Graph Laplacians

Adjacency matrices can represent graphs without any loss of information. However, there are alternative matrix representations of graphs that have useful mathematical properties. These matrix representations are called *Laplacians* and are formed by various transformations of the adjacency matrix.

Unnormalized Laplacian The most basic Laplacian matrix is the unnormalized Laplacian, which is defined as:

$$\mathbf{L} = \mathbf{D} - \mathbf{A}, \tag{1.26}$$

where $\mathbf{A}$ is the adjacency matrix and $\mathbf{D}$ is the degree matrix. The Laplacian has a number of important properties:

1. It is symmetric and positive semi-definite.
2. The following vector identity holds $\forall \mathbf{x} \in \mathbb{R}^{|\mathcal{V}|}$

$$\mathbf{x}^\top \mathbf{L}\mathbf{x} = \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{V}} \mathbf{A}[u, v](\mathbf{x}[u] - \mathbf{x}[v])^2 \tag{1.27}$$

$$= 2 \sum_{(u,v) \in \mathcal{E}} (\mathbf{x}[u] - \mathbf{x}[v])^2 \tag{1.28}$$

3. $\mathbf{L}$ has $|V|$ non-negative eigenvalues: $0 = \lambda_{|\mathcal{V}|} \leq \lambda_{|\mathcal{V}|-1} \leq ... \leq \lambda_{|1|}$

The Laplacian and connected components The Laplacian summarizes many important properties of the graph. For example, we have the following theorem:

Theorem 2. *The geometric multiplicity of the* 0 *eigenvalue of the Laplacian* $\mathbf{L}$ *corresponds to the number of connected components in the graph.*

Proof. This can be seen by noting that for any eigenvector $\mathbf{e}$ of the eigenvalue 0 we have that

$$\mathbf{e}^\top \mathbf{L}\mathbf{e} = 0 \tag{1.29}$$

by the definition of the eigenvalue-eigenvector equation. And, the result in Equation (1.29) implies that

$$\sum_{(u,v) \in \mathcal{E}} (\mathbf{e}[u] - \mathbf{e}[v])^2 = 0. \tag{1.30}$$

The equality above then implies that $\mathbf{e}[u] = \mathbf{e}[v], \forall (u,v) \in \mathcal{E}$, which in turn implies that $\mathbf{e}[u]$ is the same constant for all nodes u that are in the same connected component. Thus, if the graph is fully connected then the eigenvector for the eigenvalue 0 will be a constant vector of ones for all nodes in the graph, and this will be the only eigenvector for eigenvalue 0, since in this case there is only one unique solution to Equation (1.29).

Conversely, if the graph is composed of multiple connected components then we will have that Equation 1.29 holds independently on each of the blocks of the Laplacian corresponding to each connected component. That is, if the graph is composed of K connected components, then the Laplacian matrix can be written as

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_1 & & & \\ & \mathbf{L}_2 & & \\ & & \ddots & \\ & & & \mathbf{L}_K \end{bmatrix}, \tag{1.31}$$

where each of the $\mathbf{L}_k$ blocks in this matrix is a valid graph Laplacian of a fully connected subgraph of the original graph. Since they are valid Laplacians only fully connected graphs, for each of the $\mathbf{L}_k$ blocks we will have that Equation (1.29) holds and that each of these sub-Laplacians has an eigenvalue of 0 with multiplicity 1 and an eigenvector of all ones (defined only over the nodes in that component). Moreover, since $\mathbf{L}$ is a block diagonal matrix, its spectra is given by the union of the spectra of all the $\mathbf{L}_k$ blocks, i.e., the eigenvalues of $\mathbf{L}$ are the union of the eigenvalues of the $\mathbf{L}_k$ matrices and the eigenvectors of $\mathbf{L}$ are the union of the eigenvectors of all the $\mathbf{L}_k$ matrices with 0 values filled at the positions of the other blocks. Thus, we can see that each block contributes one eigenvector for eigenvalue 0, and this eigenvector is an indicator vector for the nodes in that connected component. □

Normalized Laplacians In addition to the unnormalized Laplacian there are also two popular normalized variants of the Laplacian. The symmetric normalized Laplacian is defined as

$$\mathbf{L}_{\text{sym}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{L}\mathbf{D}^{-\frac{1}{2}}, \tag{1.32}$$

while the random walk Laplacian is defined as

$$\mathbf{L}_{\text{RW}} = \mathbf{D}^{-1}\mathbf{L} \tag{1.33}$$

Both of these matrices have similar properties as the Laplacian, but generally their algebraic properties differ by small constants due to the normalization.

For example, Theorem 2 holds exactly for $\mathbf{L}_{\text{RW}}$. For $\mathbf{L}_{\text{sym}}$, Theorem 2 holds but with the eigenvectors for the 0 eigenvalue scaled by $\mathbf{D}^{\frac{1}{2}}$. As we will see throughout this book, these different variants of the Laplacian can be useful for different analysis and learning tasks.

1.3.4 Graph Cuts and Clustering

In Theorem 2, we saw that the eigenvectors corresponding to the 0 eigenvalue of the Laplacian can be used to assign nodes to clusters based on which connected component they belong to. However, this approach only allows us to cluster nodes that are already in disconnected components, which is trivial. In this section we take this idea one step further and show that the Laplacian can be used to give an optimal clustering of nodes *within a fully connected graph.*

Graph cuts In order to motivate the Laplacian spectral clustering approach, we first must define what we mean by an *optimal* cluster. To do so, we define the notion of a *cut* on a graph. Let $\mathcal{A} \subset \mathcal{V}$ denote a subset of the nodes in the graph and let $\bar{\mathcal{A}}$ denote the complement of this set, i.e., $\mathcal{A} \cup \bar{\mathcal{A}} = \mathcal{V}, \mathcal{A} \cap \bar{\mathcal{A}} = \emptyset$. Given a partitioning of the graph into K non-overlapping subsets $\mathcal{A}_1, ..., \mathcal{A}_K$ we define the cut value of this partition as

$$\text{cut}(\mathcal{A}_1, ..., \mathcal{A}_K) = \frac{1}{2} \sum_{k=1}^{K} |(u, v) \in \mathcal{E} \ : \ u \in \mathcal{A}_k, v \in \bar{\mathcal{A}}_k|. \tag{1.34}$$

In other words, the cut is simply the count of how many edges cross the boundary between the partition of nodes. Now, one option to define an *optimal clustering* of the nodes into K clusters would be to select a partition that minimizes this cut value. There are efficient algorithms to solve this task, but a known problem with this approach is that it tends to simply make clusters that consist of a single node [Stoer and Wagner, 1997].

Thus, instead of simply minimizing the cut we generally seek to minimize the cut while also enforcing that the partitions are all reasonably large. One popular way of enforcing this is by minimizing the *Ratio Cut*:

$$\text{RatioCut}(\mathcal{A}_1, ..., \mathcal{A}_K) = \frac{1}{2} \frac{\sum_{k=1}^{K} |(u, v) \in \mathcal{E} \ : \ u \in \mathcal{A}_k, v \in \bar{\mathcal{A}}_k|}{|\mathcal{A}_k|}, \tag{1.35}$$

which penalizes the solution for choosing small cluster sizes. Another popular solution is to minimize the *Normalized Cut (NCut)*:

$$\text{NCut}(\mathcal{A}_1, ..., \mathcal{A}_K) = \frac{1}{2} \frac{\sum_{k=1}^{K} |(u, v) \in \mathcal{E} \ : \ u \in \mathcal{A}_k, v \in \bar{\mathcal{A}}_k|}{\text{vol}(\mathcal{A}_k)}, \tag{1.36}$$

where $\text{vol}(\mathcal{A}) = \sum_{u \in \mathcal{A}} d_u$. The NCut enforces that all clusters have a similar number of edges incident to their nodes.

Approximating the RatioCut with the Laplacian spectrum We will now derive an approach to find a cluster assignment that minimizes the RatioCut using the Laplacian spectrum. (A similar approach can be used to minimize the NCut value as well.) For simplicity, we will consider the case where we $K = 2$ and we are separating our nodes into two clusters. Our goal is to solve the following optimization problem

$$\min_{\mathcal{A} \in \mathcal{V}} \text{RatioCut}(\mathcal{A}, \bar{\mathcal{A}}). \tag{1.37}$$

To rewrite this problem in a more convenient way, we define the following vector $\mathbf{a} \in \mathbb{R}^{|\mathcal{V}|}$:

$$\mathbf{a}[u] = \begin{cases} \sqrt{\frac{|\bar{\mathcal{A}}|}{|\mathcal{A}|}} & \text{if } u \in \mathcal{A} \\ -\sqrt{\frac{|\mathcal{A}|}{|\bar{\mathcal{A}}|}} & \text{if } u \in \bar{\mathcal{A}} \end{cases}. \tag{1.38}$$

Combining this vector with our properties of the graph Laplacian we can see that

$$\begin{aligned}
\mathbf{a}^\top \mathbf{L}\mathbf{a} &= \sum_{(u,v) \in \mathcal{E}} (\mathbf{a}[u] - \mathbf{a}[v])^2 && (1.39)\\
&= \sum_{(u,v) \in \mathcal{E} \,:\, u \in \mathcal{A}, v \in \bar{\mathcal{A}}} (\mathbf{a}[u] - \mathbf{a}[v])^2 && (1.40)\\
&= \sum_{(u,v) \in \mathcal{E} \,:\, u \in \mathcal{A}, v \in \bar{\mathcal{A}}} \left(\sqrt{\frac{|\bar{\mathcal{A}}|}{|\mathcal{A}|}} - \left(-\sqrt{\frac{|\mathcal{A}|}{|\bar{\mathcal{A}}|}} \right) \right)^2 && (1.41)\\
&= \text{cut}(\mathcal{A}, \bar{\mathcal{A}}) \left(\frac{|\mathcal{A}|}{|\bar{\mathcal{A}}|} + \frac{|\bar{\mathcal{A}}|}{|\mathcal{A}|} + 2 \right) && (1.42)\\
&= \text{cut}(\mathcal{A}, \bar{\mathcal{A}}) \left(\frac{|\mathcal{A}| + |\bar{\mathcal{A}}|}{|\bar{\mathcal{A}}|} + \frac{|\mathcal{A}| + |\bar{\mathcal{A}}|}{|\mathcal{A}|} \right) && (1.43)\\
&= |\mathcal{V}| \text{RatioCut}(\mathcal{A}, \bar{\mathcal{A}}). && (1.44)
\end{aligned}$$

Thus, we can see that $\mathbf{a}$ allows us to write the Ratio Cut in terms of the Laplacian (up to a constant factor). In addition, $\mathbf{a}$ has two other important properties:

$$\sum_{u \in \mathcal{V}} \mathbf{a}[u] = 0 \Leftrightarrow \mathbf{a} \perp \mathbb{1} \tag{1.45}$$

and

$$\|\mathbf{a}\|^2 = |\mathcal{V}|. \tag{1.46}$$

Both of these properties can be verified algebraically by the reader.

Putting this altogether we can rewrite the Ratio Cut minimization problem

in Equation (1.37) as

$$\begin{aligned} &\min_{\mathcal{A}\in\mathcal{V}} \mathbf{a}^\top \mathbf{L}\mathbf{a} \\ &s.t. \\ &\quad \mathbf{a} \perp \mathbb{1} \\ &\quad \|\mathbf{a}\|^2 = |\mathcal{V}| \\ &\quad \mathbf{a} \text{ defined as in Equation 1.38.} \end{aligned} \tag{1.47}$$

Unfortunately, however, this is an NP-hard problem since the restriction that $\mathbf{a}$ is defined as in Equation 1.38 requires that we are optimizing over a discrete set. The obvious relaxation is to remove this discreteness condition and simplify the minimization to be over real-valued vectors:

$$\begin{aligned} &\min_{\mathbf{a}\in\mathbb{R}^{|\mathcal{V}|}} \mathbf{a}^\top \mathbf{L}\mathbf{a} \\ &s.t. \\ &\quad \mathbf{a} \perp \mathbb{1} \\ &\quad \|\mathbf{a}\|^2 = |\mathcal{V}|. \end{aligned} \tag{1.48}$$

By the Rayleigh-Ritz Theorem, the solution to this optimization problem is given by the second-smallest eigenvector of $\mathbf{L}$ (since the smallest eigenvector is equal to $\mathbb{1}$).

Thus, we can approximate the minimization of the RatioCut by setting $\mathbf{a}$ to be the second-smallest eigenvector[5] of the Laplacian. To turn this real-valued vector into a set of discrete cluster assignments, we can simply assign nodes to clusters based on the sign of $\mathbf{a}[u]$, i.e.,

$$\begin{cases} u \in \mathcal{A} & \text{if } \mathbf{a}[u] \geq 0 \\ u \in \bar{\mathcal{A}} & \text{if } \mathbf{a}[u] < 0. \end{cases} \tag{1.49}$$

In summary, the second-smallest eigenvector of the Laplacian is a continuous approximation to the discrete vector that gives an optimal cluster assignment (with respect to the RatioCut). An analogous result can be shown for approximating the NCut value, but it relies on the second-smallest eigenvector of the normalized Laplacian $\mathbf{L}_{\text{RW}}$ [Von Luxburg, 2007].

1.3.5 Generalized spectral clustering

In the last section we saw that the spectrum of the Laplacian allowed us to find a meaningful partition of the graph into two clusters. In particular, we saw that the second-smallest eigenvector could be used to partition the nodes into different clusters. This general idea can be extended to an arbitrary number of K clusters by examining the K smallest eigenvectors of the Laplacian. The steps of this general approach are as follows:

[5]Note that by second-smallest eigenvector we mean the eigenvector corresponding to the second-smallest eigenvalue.

1. Find the K smallest eigenvectors of $\mathbf{L}$ (excluding the smallest): $\mathbf{e}_{|\mathcal{V}|-1}, \mathbf{e}_{|\mathcal{V}|-2}, ..., \mathbf{e}_{|\mathcal{V}|-K}$.

2. Form the matrix $\mathbf{U} \in \mathbb{R}^{|\mathcal{V}| \times K-1}$ with the eigenvectors from Step 1 as columns.

3. Represent each node by its corresponding row in the matrix $\mathbf{U}$, i.e.,

$$\mathbf{z}_u = \mathbf{U}[u] \, \forall u \in \mathcal{V}.$$

4. Run K-means clustering on the *embeddings* $\mathbf{z}_u \, \forall u \in \mathcal{V}$.

As with the discussion of the $K = 2$ case in the previous section, this approach can be adapted to use the normalized Laplacian , and the approximation result for $K = 2$ can also be generalized to this $K > 2$ case [Von Luxburg, 2007].

The general principle of spectral clustering is a powerful one. We can represent the nodes in a graph using the spectra of the graph Laplacian, and this representation can be motivated as a principled approximation to an optimal graph clustering. There are also close theoretical connections between spectral clustering and random walks on graphs as well as the field of graph signal processing. We will discuss many of these connections in future chapters.

1.4 Graph Representation Learning: An Overview

The central problem in machine learning on graphs is finding a way to incorporate information about graph-structure into a machine learning model. For example, in the case of link prediction in a social network, one might want to encode pairwise properties between nodes, such as relationship strength or the number of common friends. Or in the case of node classification, one might want to include information about the global position of a node in the graph or the structure of the node's local graph neighborhood. The challenge—from a machine learning perspective—is that there is no straightforward way to encode this high-dimensional, non-Euclidean information about graph structure into a feature vector.

In the previous section we saw a number of traditional approaches to learning over graphs. We discussed how graph statistics and kernels can extract feature information for classification tasks. We saw how neighborhood overlap statistics can provide powerful heuristics for relation prediction. And, we offered a brief introduction to the notion of spectral clustering, which allows us to cluster nodes into communities in a principled manner. However, all the approaches discussed in Section 1.3 are limited due to the fact that they require careful, hand-engineered statistics and measures. These hand-engineered features are inflexible—*i.e.*, they cannot adapt during the learning process—and designing these features can be a time-consuming and expensive process.

This book is about an alternative approach to learning over graphs: *graph representation learning*. Instead of extracting hand-engineered features, we will seek to *learn* representations that encode structural information about the graph. The idea behind these *representation learning* approaches is to learn a mapping that embeds nodes, or entire (sub)graphs, as points in a low-dimensional vector space $\mathbb{R}^d$. The goal is to optimize this mapping so that geometric relationships in the embedding space reflect the structure of the original graph. After optimizing the embedding space, the learned embeddings can be used as feature inputs for downstream machine learning tasks. The key distinction between representation learning approaches and previous work is how they treat the problem of representing graph structure. Previous work treated this problem as a pre-processing step, using hand-engineered statistics to extract structural information. In contrast, representation learning approaches treat this problem as machine learning task itself, using a data-driven approach to learn embeddings that encode graph structure.