POLITECNICO DI TORINO

POLITECNICO DI TORINO

Master Degree Course in Mathematical Engineering

Master Degree Thesis

# Graph Data Science and machine learning applications

**Supervisor**
Prof. Paolo Garza

**Candidate**
Antonella Cardillo

Academic Year 2023-2024

# Contents

# Chapter 1

# Introduction

Connectivity is the most widespread feature of modern networks. From social networks as Facebook or LinkedIn to communication systems, and from economic grids as marketing or user bank systems to networks of neurons with even a moderate degree of complexity are not casual, which means it is not possible to assume any statistical distribution about connections of the networks mentioned above also because these are not static. Classical statistical analysis would be able neither to describe nor to predict behaviors within connected systems.

As data becomes more interconnected and systems grow more advanced and intricate, harnessing the diverse and evolving relationships within our data is crucial, also using technologies built to leverage relationships and their dynamic nature. Graphs are powerful structures that not only excel at representing interconnected information but also support various types of analysis: each element within a graph serves as a entry point to explore the entire network, providing a variety of pathways for exploration and numerous opportunities for analysis. Capturing complex relationships between entities are often challenging to model using traditional relational databases in all interconnected domains such as social networks, recommendation systems, biological networks, fraud detection, and knowledge graphs. Graphs leverages the connectivity of data to uncover meaningful insights that may not be apparent in isolated data points or tabular data structures. These meaningful insights could be translated into graph-based features of a machine learning model, enhancing the predictive accuracy and model performance.

The main aim of this thesis will be to deepen this area, exploring its possible applications in real use cases, highlighting how graphs can offer an original and efficient solution in terms of graph feature extraction, data modeling, and computational efficiency of the algorithms used compared to classical machine learning techniques.

Chapter 2 introduces the basic graph data science concepts, covering the idea of a graph, the main areas of interest in graph data science, potential real-world examples of graph applications, and how graphs enhance the lifecycle of a machine learning project.

Chapter 3 discusses the technologies for storing connected data to leverage relationships between data. It explores the difference between native and non-native graph databases, and examines the various graph models used by graph databases, highlighting their key differences.

Chapter 4 explores different analytical tools to understand the relationships between data, either through a more localized approach with graph queries or with graph algorithms. It examines key graph algorithms for answering questions about pathways, flow dynamics, influencers and group interactions.
Chapter 5 presents three distinct real use cases approached through a graph-based methodology. The first case involves link prediction problem by analyzing historical relationships through graphs, leading to accurate forecasts of future network behaviors. The second case focuses on solving a classic recommendation problem using graphs, emphasizing their computational benefits for real-time analysis and data modeling. The third case deals with a fraud detection problem, utilizing graph-based features to enhance model performance over traditional methods.

# Chapter 2

# Introduction to Graphs and Graph Data Science

## 2.1 Networks and Graphs

A network is a set of relations between entities, which could include people, web documents, organizations, neurons, or electrical elements. A graph is the mathematical concept used to model a network, so it is, very simply, the mathematical representation of a network. Graphs are valuable to represent how objects are either physically or logically connected in networks. They are the unique models to catch not only the single data points but also relationships among data. This structure consists of a set of vertices, also called nodes, connected by edges or links. Nodes represent objects of the real world, while links represent relationships between these entities. A graph in which we attribute names and meanings to the nodes and links becomes what is known as a network.

Suppose that we have the graph shown in Figure 2.1. As pure mathematical diagram, this same graph can be used to model several types of networks in very heterogeneous domains by assigning different semantics to nodes and links, as shown in Figure 2.2:

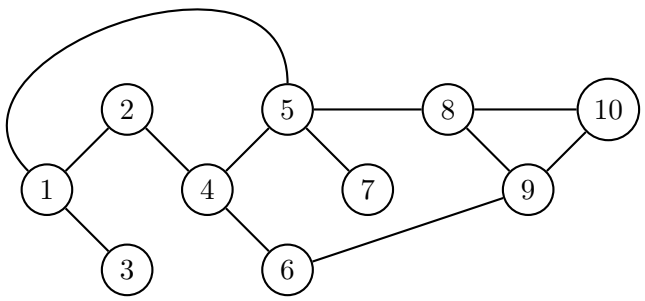


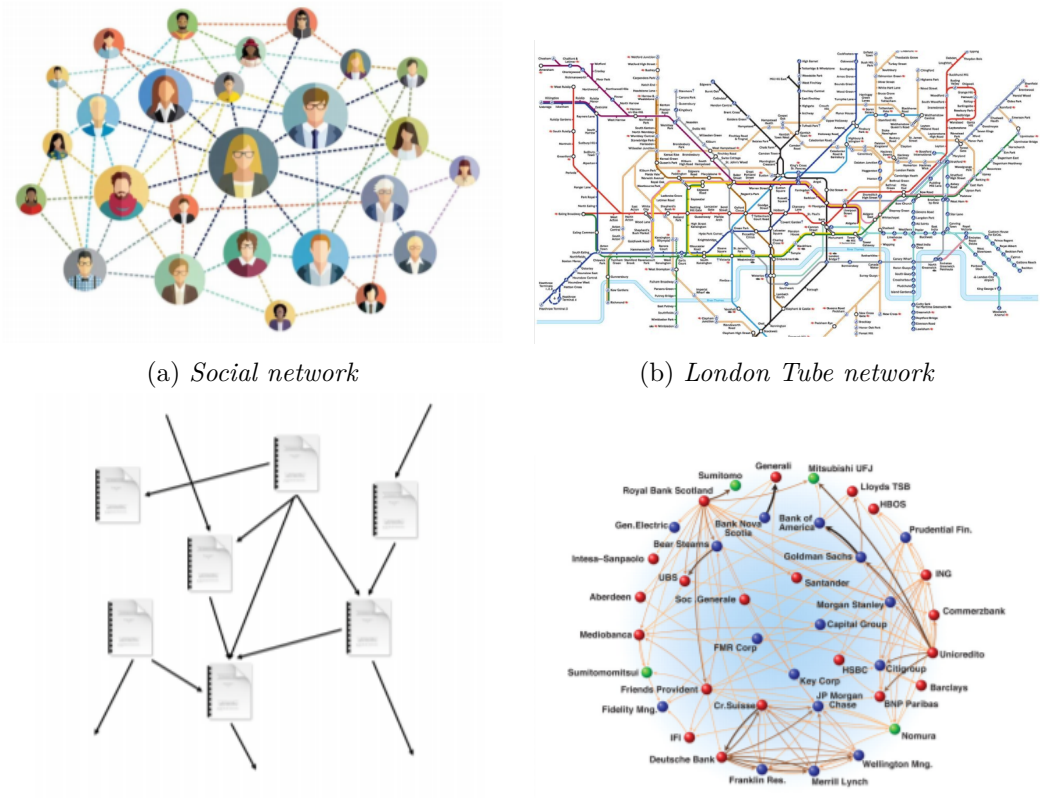

Figure 2.1: Generic graph.

- A *social network*, if the nodes are people and each link represents any sort of relationship between them (friends, family members, coworkers).
- An *informational network*, in which the nodes are information structures such as Web pages or documents, and links represent logical connections such as hyperlinks or citations.
- A *communication network*, in which nodes are electronic devices that can broadcast messages, and the edges represent direct links along which messages can be transmitted.
- A *transportation network*,if the nodes are destinations and edges represent direct connections using flights or roads or trains.

If a graph is an abstract mathematical concept, networks, as representation of some real systems, are subjected to forces that act on them and change their structure. These forces are elements that exist outside the network but influence how the network's structure evolves over time. The nature of these forces are specific to the type of network. In social networks, for example, each person has unique characteristics, and similarities between characteristics of two people influence link creation or deletion. The knowledge of these specific contexts enables the prediction of how the network will evolve over time [1].
Graphs are dynamic models: they do not have a fixed schema, so they can be continuously enriched as the network evolves and their flexible nature allows to answer different and always changing questions. Graphs are also real information maps, with a highly communicative and visual power, able to display multiple kind of information at the same time in a way that the human brain can easily understand.

## 2.2 Graph concepts

Which are the key elements of the graph world? Even though a graph is a simple structure, it is fundamental to understand how to represent it and how to use the main concepts around it. More formally than mentioned above, a graph is a pair $G = (V, E)$, where $V$ is a collection of vertices and $E$ is a collection of edges over $V$.
Graphs can be directed or undirected, depending on whether a direction of traversal is defined on the edges. In *directed* graphs, an edge $(i, j) \in E$ can be traversed from $i$ to $j$ but not in the opposite direction: the starting node $i$ is called the *tail* of the link, while the ending node $j$ is called *head* of the link. In *undirected* graphs relationships are considered bidirectional, so they can be traversed in both directions without defining a star or end node. Simply if we have arrows as links of the graph, the graph is directed and these arrows indicate the direction of the relationship.
Graphs can be weighted or unweighted: when a weight, i.e. a numerical value with some meaning is assigned to a link or a node, the graph is said to be *weighted*, otherwise it is said to be *unweighted*. In weighted graphs, the values assigned to links can denote various measures such as cost, time, distance, capacity, and other similar metrics.
Graphs can be connected or disconnected. Before we understand this distinction, let us define what a path is in a graph. A sequence of vertices in which each consecutive pair




(a) *Social network* (b) *London Tube network*

(c) *Citation network* (d) *Marketing network*

Figure 2.2: Examples of networks. [2–4]

in the sequence is connected by a link is called a *path*. A graph is considered *connected* if there exists a path between any pair of nodes within the graph, regardless of the path's length. If the graph contains isolated parts, it is considered *disconnected*. These isolated clusters, where nodes are interconnected, are known as *connected components*.

Graphs can be classified as cyclic or acyclic. A *cycle* refers to a path that begins and ends at the same node. Both directed and undirected graphs can contain cycles, but in directed graphs, paths must follow the direction of relationships.

Graphs can exhibit varying degrees of sparsity or density. The *sparsity* of a graph is determined by the number of actual relationships it contains relative to the total possible number of relationships, which would exist if every pair of nodes were linked. A graph in which each node is connected to all the other vertices is called *complete*. When the ratio between the number of relationships in the graph and the number of links in a complete graph is close to zero, the graph is said *sparse*, while if this ratio is close to one, it is defined *dense*.

Graphs can be monopartite, bipartite or k-partite. Many networks include data with various types of nodes and relationships. A graph with only one type of node and relationship is called *monopartite*. A *bipartite* graph consists of two sets of nodes, where relationships only connect nodes from different sets. For *k-partite* graphs, k denotes the number of

distinct node types in the data [5].

## 2.3 Graph Data Science Journey

Graphs are powerful structures useful not only for modeling connected information, but also for supporting multiple types of analysis. In a graph each node and each relationship is an access point for analysis, from which it is possible to traverse the rest of the graph, providing several analysis patterns. [6]
Graph Data Science is a graph-data driven approach to gain knowledge from the relationships in data. It can be broken in three main areas:

- Graph statistics provides basic connection-related metrics about a graph, such as the number of nodes and the distribution of the number of relationships. These measures usually represent the starting point in a graph analysis because they are simple to collect and a good test of early hypotheses for further more complex analysis.
- Graph analytics involves a sophisticated range of queries and algorithms specifically tailored to uncover meaningful insights within graph data, with a primary focus on the relationships among entities rather than on each individual entity. When we know precisely what we are looking for, graph queries allow to answer these specific questions; otherwise when we only know the general structure we are looking for but not the exact pattern we can use graph algorithms. Queries and algorithms are typically applied together during graph analytics phase: queries reveal local patterns in the graph because they are based on parts of the graph surrounding a node, instead algorithms concern the whole graph being extremely capable of finding structures and revealing patterns in it.
- Graph powered machine learning is the application of graph data and analytics results to train machine learning models. Graph statistics and analytics are often used in conjunction to answer certain types of questions about networks and the following insights, applied to improve machine learning.

## 2.4 Use cases for GDS

Graph Data Science helps to answer big questions involving four different areas: movement, influence, groups and interactions, patterns. These areas answer the following questions [7]:

- How do things move through a network? Uncovering how things flow through a network and the pathways they might take involves deep path analysis in order to find out the best routes across our connected data.
- What are the most influential points of the network? Determining these influencers involves detecting the control points in a network based on their position, including their connections. These nodes can accelerate or slow the flow of things through

networks from finances to opinions, performing as fast propagation nodes, bridges between less connected groups, or bottlenecks.

- What are the groups and the interactions between them? Detecting communities requires grouping and partitioning nodes based on the number and strength of interactions for uncovering unusual patterns, predicting similar behavior, finding duplicate entities, or simply preparing data for other analyses.

- What patterns are noteworthy in the network? Uncovering network patterns consists in finding similar and related information in large datasets. The goal may be to look for a known relationship pattern or compare nodes to find similarities or still evaluate the whole structure of a network to correlate patterns to certain social behavior to examine.

The power of GDS to uncover and leverage network structure drives a wide range of real use cases from financial and marketing world to manufacturing and IT networks. We report here only a few but significant examples of GDS applicability.

- *Fraud Detection.* Payment services apps try to deliver money as quickly as possible to valid users while also ensuring money is not sent for illegal purposes or hiding the true recipient through circuitous paths. Banks and credit card companies lose billions of dollars every year due to fraud and GDS can increase the amount of fraud detected reinforcing existing ML pipelines used as traditional methods of fraud detection. Identifying fraudulent activities typically begins with detecting deviations from typical transactional patterns or interactions with known fraudulent entities, analyzing relationships and behaviors within the transactional graph.

- *Recommendation engines.* Recommendation system is a classic problem when a wealth of content is provided to users and one needs to know what content users have not yet seen but are most likely to enjoy. Recommender systems became famous through Netflix or Youtube and e-commerce platforms but they drive some of the most important parts of a business from product development to human resources for preserving employees through upskilling training.

- *Supply Chain Management.* A supply chain network is the representation of supply chain elements and their interactions as a graph. They include suppliers, manufacturers, distributors, customers and so on. All these elements are independent entities, perhaps providing the same services to multiple companies but also interconnected, because they work together through informational and financial flows. Supply chains face a variety of risks, from natural disasters to contamination of raw products, delivery delays, and labor shortages. Furthermore, the efficient operation of the entire chain as a whole is ensured if the individual components work well. Detecting elements in the chain that can disrupt a large part of the chain significantly affecting normal behavior, or finding the best path, balancing cost and efficiency with customer satisfaction and sustainability could be interesting use cases in the field of supply chain management.

- *Customer 360.* Companies have increasingly more information about customers including master data (name, age, gender, address), transactions (purchase orders, types of items bought, phone calls, purchase times), relationships and more. With graphs, marketers can gain a more comprehensive view of their customers as the relationships the customers hold with each other, the relationships between all the purchased products, and more. Performing customer 360-analysis allows for optimized marketing programs and offers.

## 2.5 Graph powered machine learning

A machine learning project is a complex task that requires more then selecting the right algorithms to apply data. If we want to define a clear workflow for it, we could refer to the CRISP-DM model [8] acronym for Cross Industry Standard Process for Data Mining, a schema commonly used for data mining tasks but it can be also applied to generic machine learning projects. CRISP-DM model has six interconnected main phases, with data at the core of each single phase.

- Business understanding: understanding of the domain and the business perspective allows us to define goals and a raw project plan.
- Data understanding: the knowledge of data sources available, the different types of data and their content enables us to define an architecture design to get or extract data for the next steps of the ML workflow.
- Data preparation: collecting data from multiple sources and organizing it in the form required by the specific algorithm of the following phase involves a set of methods for merging, cleaning and enriching data. Another outcome of this phase is the identification of the database management system for storing data.
- Modeling: selecting and applying different algorithms and tuning their parameters to optimal values permit us to build a range of predictive models.
- Evaluation: evaluating models using testing data and defining performance measures is an important phase before a model can be deployed, as also reviewing the steps executed to construct the model and checking whether the business objectives are satisfied.
- Deployment: this phase consist in deploying the project in a production environment monitoring its performance constantly or simply generating reports with the results of prediction models. [9]

In brief this machine learning cycle describes how data flows from the data sources through the learning process to end users in the form of visualizations or predictions. More precisely, the goal of a ML process will be to deduce a model capable of mapping the input data known as features with the potential output, that in the case of recommendation example could consist of predicting what users could be interested in, given a list of items rated by users as features.

Defined a generic machine learning workflow through the CRISP-DM model, it can be decomposed into three macro phases each of which merges one or more of the mentioned above steps:

1. Data source management phase concerns all the tasks of gathering, merging, cleaning and preparing the training data for the learning phase. It matches to the stages of data preparation and understanding of the CRISP-DM model.

2. Learning phase is the step in which ML algorithms are applied to the training dataset and corresponds to the modeling phase.

3. Predictive phase consists in storing and then accessing the predictive models in order to provide predictions during the evaluation and deployment steps.

Graphs can empower machine learning workflow to represent these features using graph data models and to improve or simplify this mapping phase. In this sense we talk about *Graph powered machine learning*. The main aim of this thesis will be to deepen this area of Graph Data Science and explore its possible applications in real use cases.
Graphs enables to merge multiple data sources into a single connected dataset in order to represent all the knowledge as a consistent and machine-ready data structure. Providing multiple performant access patterns for data sources and allowing the process of data enrichment using external sources are also valuable graph-powered data management features during the data understanding and data preparation phases of the CRISP-DM model.
The knowledge of the specific forces acting on the networks permits to understand related network dynamics and use them to deliver better machine learning features. Several types of graph algorithms are valuable to analyze and explore data, also speeding the identification of salient features and improving the quality of learning phase. The schema flexibility of graph data technology permits various models to coexist in the same dataset and to access them as fast as possible to real time predictions. These graph-powered data analytics features are involved principally in the modeling and the deployment phases.
Graphs have also a high communication power for sharing results, analyzing them or helping people navigate data by highlighting connections between elements in a way so easily understandable to the human brain. These graph-powered data visualization features are particularly important during business and data understanding and evaluation phases.

# Chapter 3

# Storing connected data

If we want to empower our machine learning workflow through graphs, we have to be able to store, access and handle these structures efficiently using technologies built to leverage relationships between data when real systems become increasingly complex and hence data increasingly interconnected. To achieve this task, we need a general-purpose data management technology called graph database. The key to realize when to use a graph database is understanding the value of connections between data. It does not matter the particular use case domain: from finance to healthcare or logistics, the relevant aspect for which graph databases are designed is to represent relationships in order to give links the same importance as data itself and navigate them in an efficient way.

## 3.1 Native vs. non-native graph databases

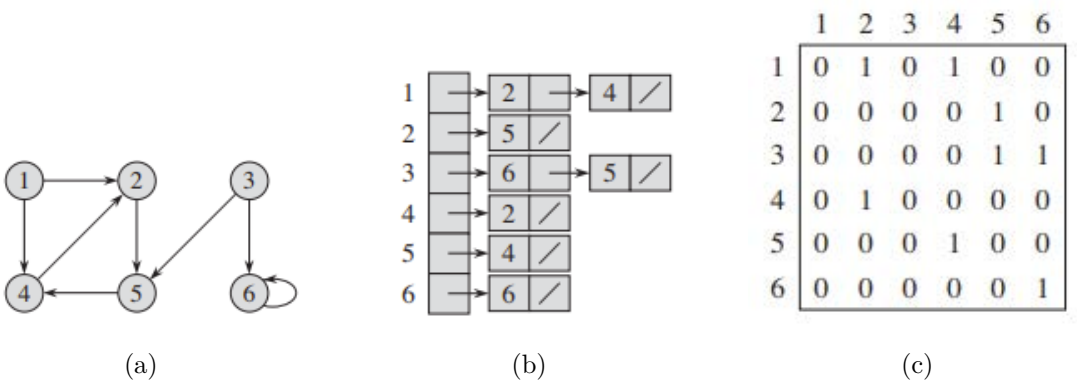
A *graph databases* is a collection of *relationships* that uses graph structures to store and manage entities or *nodes* and the connections between these entities. A DBMS built to handle graph workloads across the entire computing stack, from the query language to the database management engine and filesystem and from clustering to backup and monitoring, is called a *native graph database* [10].
Not all graph databases are native, that is, designed to understand and support graph workloads: other graph databases called *nonnative graph databases* offer a graph view on top of a nongraph storage model. The alternative non-native can be divided into two categories:

- Those that layer a graph API on top of an existing nongraph storage model, such as relational store or NoSQL data structures such as key/value, document or column-based stores

- Those that promote multimodel semantics where one system theoretically can support several data models.

There are two main elements that distinguish native graph technology from the non-native option: processing and storage. **Graph processing** refers to how a graph database manages database operations such as queries or algorithms for the analysis of graphs,

while **graph storage** refers to the underlying structure of the database that contains the graph data model. [11]
Our goal will be to understand how the two database alternatives- native and non-native one- will handle these two aspects.
A native graph database exhibits a property called *index-free adjacency*, which means that each node maintains direct references to its adjacent nodes. There are two standard ways of representing a generic graph $G = (V, E)$, directed or undirected, weighted or unweighted, in order to be processed: as a collection of adjacency lists or as an adjacency matrix.
The *adjacency list* representation of a graph $G = (V, E)$ consists of an array $Adj$ of lists, one for each vertex in $V$. For each vertex $u$ in $V$, the adjacency list $Adj[u]$ contains all the vertices $v$ for which there exists an edge $(u, v) \in E$, the vertices adjacent to $u$ in $G$.
For the *adjacency matrix* representation of a graph $G = (V, E)$, we assume that the vertices are numbered 1,2,..., $|V|$ in some arbitrary manner and this representation of $G$ consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that $a_{ij} = 1$ if $(i, j) \in E$, otherwise $a_{ij} = 0$. If $G = (V, E)$ is a weighted graph, and $w$ is the weight of the edge $(u, v)$, adjacency list can be easily adapted by storing the weight $w$ of this edge in $Adj[u]$ or setting $a_{uv} = w$, instead of 1, in the adjacency matrix representation.




(a) (b) (c)

Figure 3.1: Two representations of a directed unweighted graph (a) as an adjacency list (b) and as an adjacency matrix (c) [12]

Figure 3.1b is an adjacency list representation of the directed graph in figure 3.1a. Vertex 1 has two neighbors 2 and 4, so $Adj[1]$ is the list [2,4]. Vertex 2 has only one neighbor, so $Adj[2]$ will be [5]. In this case, we consider only the outgoing links, but we do the same with the ingoing links: for directed graph we have to choose a direction and be coherent with it during adjacency list creation.
Figure 3.1c is an adjacency matrix representation of the directed graph in figure 3.1a. As for the adjacency list, it is necessary to choose one direction and use it during matrix creation. The first line, for example, is related to vertex 1. This row in the matrix has 1 in columns 2 and 4 because vertex 1 has two outgoing relationships, to vertices 2 and 4. All the other values are 0. The second row, related to vertex 2, has 1 in columns 5 because vertex 2 has one outgoing relationship to vertex 5. Note that, looking at the

columns of this matrix, it is possible to see the ingoing relationships for each vertex of the graph. The adjacency matrix of a graph requires memory directly proportional to $|V| \times |V|$, independent of the number of edges in the graph, while the memory required by an adjacency list representation of a directed or undirected graph is directly proportional to $|V| + |E|$. An adjacency list representation provides a compact way to represent sparse graphs but we may prefer an adjacency matrix representation, however, when the graph is dense or when we want to determine whether a given edge $(u, v)$ is present in the graph without searching for $v$ in the adjacency list $Adj[u]$, at cost of using more memory. [12]
Index-free adjacency implies that each node functions as a kind of micro-index of the other nearby nodes stored in the node itself, which is much cheaper than using global indexes. A traversal across a relationship in a native graph database has a constant cost, $O(1)$[1] in computational terms, therefore native graph queries perform at a constant rate simply proportional to the amount of graph searched, independent of the total size of the graph.
A nonnative graph engine is optimized for an alternative storage model, such as columnar, relational, document, or key/value data, so when dealing with graphs, the database has to perform costly translations to and from the native storage model. An approach based on data denormalization could optimize translations but this solution typically leads to high latency when querying graphs. For this reason a nonnative graph database will never be as performant as a native one because it requires a translation process.
To better understand the value of native graph database for a machine learning project and compare its performance to the alternative nonnative, we can consider a highly connected domain like the social network one. A social network exemplifies a network that is densely interconnected with a varied structure, in which the connections between data are not evenly distributed across the domain: due to its connected nature, a social network can be modeled easily as a graph, as shown in figure 3.2.
Suppose now that we would like to store this social network model by using the relational database or any other NoSQL database based on a global index. The connections between the entities in the example are represented in figure 3.3.
To find Alice's friends we have to perform an index lookup, at cost $O(log\ n)$. Since there is no stored relationship in a non-native graph database, this engine uses global indexes to simulate a connection between data and to link nodes together. Traditional indexing is crucial for improving read performance in the nonnative graph database: database index is a structure, very similar to the index of a book in its function, that organizes data records on disk in a way that speeds up its recovery, mapping record keys to the location on disk and at the same time growing when database grows, also taking up significant amounts of disk space.
Index lookups may be acceptable for occasional lookups and they can generally work for small networks, but this approach incurs significant computational costs upon reversing the traversal direction. If we want to find out who is friends with Alice, because in

[1] O notation is used in computer science to describe the performance or complexity of an algorithm. Big O specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used, in memory or on disk, by an algorithm.

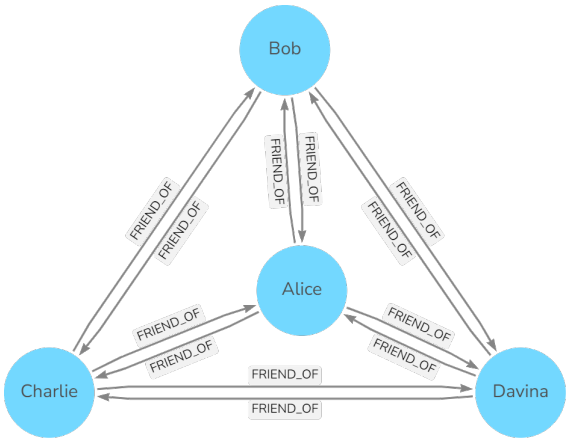



Figure 3.2: A social network example

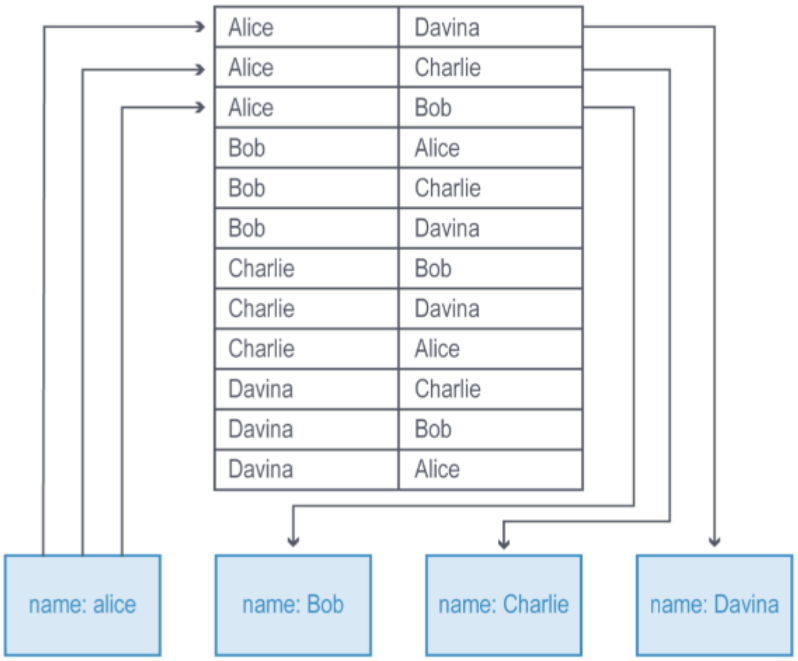



Figure 3.3: Storing the social network links in a nonnative graph processing engine [13]

general friendship is not a symmetric relationship, we would need to conduct multiple index lookups, one for each node that could potentially be friends with Alice. While it costs $O(log\ n)$ to determine Alice's friends, the reciprocal query to find who is friends with Alice costs $O(m\ log\ n)$.
In a native graph database leveraging index-free adjacency, bidirectional joins are precomputed and stored as relationships, enabling efficient traversal in both directions without requiring additional index lookups. Using such a graph database, finding Alice's friends involves simply traversing her outgoing *FRIENDOF* relationships, each at a constant

$O(1)$ cost. To discover individuals who are friends with Alice, we track each of Alice incoming *FRIENDOF* relationships back to their origins, with each traversal costing $O(1)$. To perform the same traversal required before now it costs only $O(m)$ instead $O(m\ log\ n)$: it is related only to the number of hops $m$, not to the total number of relationships $n$ [13]. Traversing the relationships in a nonnative graph processing engine is expensive, because each hop requires an index lookup, algorithmically more expensive than traversing a physical relationship as in a native graph database. This cost is amplified when we try to traverse the relationship in the opposite direction from the one for which the index was constructed or when extending traversal beyond a single step for example to compute friends-of-friends or friends-of-friends-of-friends. In their *Neo4j in Action* [14] Partner and Vukotic conducted an experiment to discover friends-of-friends within a social network, exploring up to a depth of five using both a relational database and Neo4j. The network comprises 1,000,000 individuals, each with around 50 connections and the results show how the Neo4j solution is the best one to store connected data as we see in the following table 3.4.

| Depth | RDBMS execution time(s) | Neo4j execution time(s) | Records returned |
|---|---|---|---|
| 2 | 0.016 | 0.01 | ~2500 |
| 3 | 30.267 | 0.168 | ~110,000 |
| 4 | 1543.505 | 1.359 | ~600,000 |
| 5 | Unfinished | 2.132 | ~800,000 |

Figure 3.4: Partner and Vukotic's experiment results [14]

Starting from depth three, friends-of-friends-of-friends, the difference between the two databases in terms of response time is clear: thirty seconds compared to a fraction of a second to complete this query. At a depth of five, the relational database takes too long to execute the query, whereas Neo4j returns results in about two seconds. At this stage, it becomes apparent that almost the entire network is considered part of our social circle. For many practical scenarios, it may be beneficial to refine the results to optimize performance [14].

## 3.2 Graph Modeling

Another main area for graph data management is **graph modeling**. Graph database will also adopt a specific data model besides selecting an appropriate solution to storage and processing. The model design affects the performance of all analyses executed on the graph. Therefore, modeling is a crucial aspect of data management.
In general terms, the graph model is the conceptual tool used to represent entities and the relationships among them in real-world use cases. Different models can address different problems from different perspectives, so defining the right model is not a trivial task.

Numerous graph data models are widely adopted, including property graphs, hypergraphs, and RDF triple stores. We want to analyze and compare these different solutions for graph modeling.

### 3.2.1 Property graph model: labeled vs typed property graphs

The idea behind a property graph is very simple and intuitive: in order to model complex networks, graphs seen as a list of nodes and relationships can be enriched through additional information in the form of *properties*. The goal is to attach a set of attributes to graph structures adding classes or types to nodes and relationships. There are two main categories of property graph databases: a Typed Property Graph (TPG), and a Labeled Property Graph (LPG). In general a property graph has the following characteristics:

- It consists of a set of *entities*, where an entity can represent a *node* or a *relationship*.
- Each entity has an *identifier* that identifies it in the graph.
- Each relationship has a *direction*, a *name* that determine the type of the relationship, a *start node* and *end node*.
- An entity can have a set of *properties* represented as key-value pairs.
- Nodes can be marked with one or more *labels* in LPG, which group nodes into defined categories or they can have a unique *type* as in TPG.

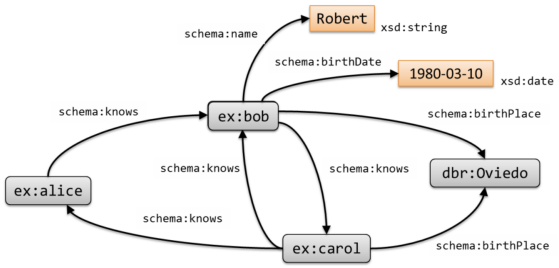
A property graph is still a graph of course, but its communication capability is greater than before [9]. Starting with clear definitions of type and label helps to explore their foundational architecture and distinguish their key differences. In graph terminology, a *type* refers to a specific category of data or object, which in the context of graphs translates to a specific node or relationship. On the other hand, a *label* is a textual identifier that categorizes data or objects: its use is optional and does not require uniqueness.
The leading native property graph database management systems are Neo4j, JanusGraph, and TigerGraph. Neo4j stands out as the sole true labeled property graph, whereas JanusGraph and TigerGraph are categorized as typed property graphs. **Neo4j** has typed required relationships and multiple optional labeled nodes, and it permits to duplicate relationship names within a graph schema, even between completely different sets of nodes. **TigerGraph** requires unique types for both nodes and relationships: each node must have a unique named type, and each relationship must have a distinct name connecting specific sets of nodes. Conversely, **JanusGraph** allows optional unique labels for nodes; if labels are not provided, the database assigns them implicitly. JanusGraph's relationships must have unique names between sets of nodes and are labeled accordingly. Although JanusGraph uses the term 'labels' for both nodes and relationships, these labels actually serve as type names rather than true labels.
JanusGraph and TigerGraph are fundamentally strongly typed graph database management systems. They require unique names for both relationships and nodes, and they ensure that a named relationship can only exist between two distinct types of nodes. They adopt an object-oriented class structure approach for the architecture of nodes and

relationships and just as in an object-oriented language, all classes and object names must be unique: if a node is named *vehicle* for example, then no other kind of node can be named *vehicle* constraining each entity so named to have the same data schema. This strongly typed approach have an advantage in programming because it ensures a one-to-one correspondence between name and type of object, preventing bugs and coding errors.
In contrast, Neo4j enables nodes to have multiple non-unique labels, offering great flexibility in graph modeling. This feature, combined with the capability to have identical relationship types between any node types, gives Neo4j the algorithmic basis and the schema adaptability needed for an iterative and experimental data science environment. Instead to spend time to shape and conform data in external tools before to import them into a graph, Neo4j imports data directly into a graph and employs graph tools to structure the data, establish the schema, and perform immediate analysis. Through the use of labels, Neo4j supports the simultaneous existence of multiple schemas within a single graph, facilitating interaction via shared relationship types. Properties lack the functionalities found in labels: labels in Neo4j are automatically indexed and they can dynamically be applied using graph algorithms. Moreover, labels are yet a powerful data analysis tool that can be used to visualize and explore data naturally [15].

### 3.2.2 The RDF vs labeled property graph models

Another way to explore and graphically depict connected data is the RDF model [16].
RDF stands for Resource Description Framework which is a W3C standard for exchanging data on the Web. At the core of RDF model is the notion of a triple, a *subject-predicate-object* data structure composed of three elements, two vertices connected by an edge, used to capture facts as "Alice likes Imagine Dragons" or "Alice loves Italy". The subject functions as a resource or node in the graph. The predicate serves as an edge, denoting a relationship, and the object can be another node or a literal value, viewed as another vertex in graph terms. Resources and relationships are distinguished by a URI, ensuring uniqueness. Consequently, nodes and edges lack internal structure and are purely identified by a unique label. This distinction highlights one of the key differences between RDF and labeled property graph models. In a labeled property graph, nodes possess a distinctive ID and a collection of key-value pairs or properties, which define them. Similarly, relationships within a labeled property graph are uniquely identified by an ID, and each relationship also includes a type and a set of key-value pairs or properties for characterizing the connections. Both nodes and relationships here have an internal structure represented by this set of key-value pairs. Using RDF model, we are doing a complete atomic decomposition of our data where nodes in the graph can be two things, resources and also literal values. Because of this atomic decomposition of the data, we typically have longer patterns when we perform queries in RDF. In contrast, labeled property graph model is much more compact with a more reduced structure where values of attributes do not represent vertices in the graph, hence it results in fast graph queries and a way to represent data very close to our logical model.
In the RDF model, it's not feasible to have connections of identical type between the same pair of nodes, as this would redundantly represent the same triple without adding

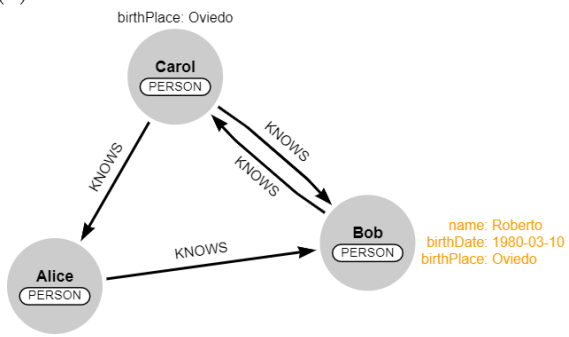
additional information. At the same time, we cannot qualify them or give connections attributes like in labeled property model: a data modeling workaround will have to be found in order to overcome these critical issues. To illustrate the differences, let is consider the same network represented through two distinct graph models shown in Figure 3.5, the property graph 3.5c and the RDF graph 3.5a.




(a)

```
prefix ex:     <http://example.org/>
prefix schema: <http://schema.org/>
prefix dbr:    <http://dbpedia.org/resource/>
prefix xsd:    <http://www.w3.org/2001/XMLSchema#>

ex:alice schema:knows      ex:bob .

ex:bob   schema:knows      ex:carol .
ex:bob   schema:name       "Robert" .
ex:bob   schema:birthDate  "1980-03-10"^^xsd:date .
ex:bob   schema:birthPlace dbr:Oviedo .

ex:carol schema:knows      ex:alice .
ex:carol schema:knows      ex:bob .
ex:carol schema:birthPlace dbr:Oviedo .
```

(b)




(c)

Figure 3.5: Two graph models of the same network, the RDF model (a) with its code in Turtle (b) [17] and the labeled property model (c).

Triple stores belong to the broader category of graph databases because they manage data that, once processed, tends to be logically interconnected. However, they are not considered native graph databases because they lack support for index-free adjacency and their storage engines are not optimized for storing property graphs. Triple stores store triples as independent facts, enabling horizontal scalability for storage but limiting their capability for deep or variable-length traversals and path queries.

### 3.2.3 Hypergraphs

A hypergraph is a graph model in which a relationship, called a hyper-edge, can connect any number of nodes at the ends of a relationship, whereas in the property graph model a relationship has only one start node and one end node. Hypergraphs are advantageous in domains characterized by numerous many-to-many relationships. Their multidimensional hyper-edges make hypergraphs a more expansive model than property graphs, therefore it is always possible to represent the information in a hypergraph as a property graph even though using more relationships and intermediary nodes. The choice will depend

on the kinds of applications we are building and the and the way we plan to shape the problem [13].

## 3.3 Nonfunctional features of a native graph database

Any data storage technology must offer assurances regarding the durability and accessibility of stored data to be considered *dependable*. What are the main nonfunctional requirements for a graph database? They need to guarantee consistency, recover from crashes, and prevent data corruption as the other traditional database systems. Further, they must scale out to ensure high availability and scale up for improved performance.
Generally native graph databases include transactional mechanisms to ensure data safety despite possible server failures or network issues or conflicts arising from concurrent transactions. Note that, not all graph databases are fully ACID as Neo4j where its ACID transactionality permits to make comparable this technology with a relational database management system in terms of dependability achieved.
We use Neo4j as a means of providing concrete examples about native graph database. In Neo4j, each transaction is managed as an in-memory object that records writes to the database. This object is facilitated by a lock manager, which applies write locks to nodes and relationships during their creation, update, and deletion. Upon rollback, the transaction object is discarded and the write locks are released. Conversely, upon successful completion, the transaction is committed to disk. [13].
To choose the right graph database we have to consider other features in addition to this transactionality:

- *Recoverability*-This feature relates to the database's ability to recover and restore operations following a malfunction. Databases, like all other software systems, are subject to some kind of failure and at some point it is inevitable that a database will crash, although the mean time between failures should be significantly long. For example, when recovering from an unclean shutdown, Neo4j examines the most recently active transaction log and applies any identified transactions to the store. After recovery, the store will be consistent with all transactions successfully committed prior to the failure.

- *Availability*-A robust database must maintain high availability to meet the growing demands of data-intensive applications. Its capability to detect and, if needed, recover from a crash ensures that data can swiftly resume availability. In a typical production scenario we cluster database instances for high availability.

- *Scalability*-It is an aggregate value that we measure across multiple axes and specifically for a graph database we have to consider capacity, latency and read/write throughput. **Capacity** is the amount of data that is possible to store in a graph database. That's a critical aspect that Neo4j solves thanks to the adoption of a dynamically sized pointers to run any size of graph workloads. Graph databases do not suffer the same **latency** problems as traditional relational databases, where more data means more join operations and consequently more index lookups. In a

native graph database, an index is used simply to find the starting node of the query and then we have simply a combination of pointer chasing and pattern matching to search the data store. Operations in a graph database require less computational effort compared to their equivalent relational operations and performance times remain nearly constant. (**read/write throughput**).

One horizontally scaling technique common in NoSQL databases is *sharding* wherein a large dataset is split in subsets distributed across several shards on different servers. These shards are tipically replicated across multiple servers to increase dependability and performance. Sharding a graph database is not straightforward at all because not all graphs have such convenient and natural boundaries, so navigating a graph could involve crossing shard boundaries multiple times. This requires many network hops, resulting in increased query times compared with the case in which everything happens on the same shard. To overcome this issue, the related nodes and, hence, the related links are stored on the same shard: the graph traversal is more efficient, but the load between shards becomes highly unbalanced. Due to the dynamic nature of graphs, this solution is not feasible in practice. It is possible to use *replication* as alternative technique for scaling a graph database: this consists in maintaining multiple copies of data, called replica, on separate computers synchronized. Neo4j uses a centralized approach with a single master, also described as *master/slave replication*. In this configuration, all writes to the database are directed exclusively to the master, the authoritative source for the data, and read operations are directed at slaves: by adding more slave nodes and routing all read requests to the slaves, we can achieve a high level of scalability [9].

## 3.4 Graph compute engines

Does it exist an alternative in terms of graph platforms to a native graph database? The answer is represented by the Graph compute engines [5]. These are read-only, nontransactional technologies that focus on the efficient execution of offline graph analytics and queries of the whole graph, performed as a series of batch steps.
Examples of such engines are Giraph, GraphLab, Graph-Engine, and Apache Spark. Just to recall, Apache Spark (henceforth just Spark) is an analytics engine for large-scale data processing. It uses a table abstraction called a DataFrame to represent and process data in rows of named and typed columns. Nodes and relationships are represented as DataFrames with a unique ID for each node and a source and destination node for each relationship. In Spark graph analysis is on this data structure not transformed into a graph format.
The choice of the production platform involves many considerations, such as the type of analysis to be run, performance needs, the existing environment, and team technical knowledge. Spark as example of graph compute engine, may be the right platform when our:

- Algorithms are fundamentally parallelizable or partitionable.

- Graph analysis can be performed offline in batch mode using data that has not been converted into a graph format.
- Team has the capability to implement custom algorithms without relying on predefined graph algorithms.

Neo4j, as native graph database, could be the preferred platform when:

- Algorithms are iterative and require high performance, especially in real-time scenarios.
- Graph analysis involves complex graph data and/or requires extensive path traversal.
- Results are integrated with transactional workloads to enrich the existing graph.
- Team prefers prepackaged and supported graph algorithms integrating their work with graph-based visualization tools.

Another interesting option would be to combine the potential of a graph compute engine with that of a native graph database for graph processing for example using Spark for high-level filtering and preprocessing of large datasets, as well as for data integration, while Neo4j handles specific processing and integrates with graph-based applications.

# Chapter 4

# Graph algorithms

As previously mentioned in 2.3 and 2.4, graph analytics consists in a sophisticated set of queries and algorithms intricately developed to reveal significant insights within graph data allowing the exploration of interactions between entities. While queries reveal local patterns in the graph because they are based on parts of the graph surrounding a node, graph algorithms are designed to operate across entire graphs and are highly effective in identifying structures and patterns within interconnected data. They are utilized when understanding relationships and structures is essential for answering questions about pathways, flow dynamics, influencers, and group interactions. In general we can detect four main categories in which the graph algorithms are grouped:

- **Pathfinding & search**: finds the optimal paths or evaluates route availability and quality.
- **Centrality**: determines the importance of distinct nodes in the networks in terms of influence a certain entity exert over the others.
- **Community detection**: detects groups clustering or partition options.
- **Similarity**: evaluates how alike nodes are.

Common references on the sections of this chapter are in [5], [18] and [19].

## 4.1 Community Detection Algorithms

Community formation occurs frequently across all types of networks, but what is a community? A community is a set of entities which interact with one another in a mutual relationship: they may have things in common, but that is not the key factor. We are not going to analyze their similarities but their interconnections [20].
An essential characteristic of community detection is that members typically have more connections within their own group compared to connections outside of it. This strategy is important for revealing cluster of nodes, isolated groups and network structure.
We will explore the most prominent algorithms for community detection:

- **Triangle Count** and **Clustering Coefficient** algorithms for overall relationship density.
- **Strongly Connected Components** and **Weakly Connected Components** algorithms for finding connected clusters.
- **Label Propagation** algorithm for inferring groups based on node labels.
- **Louvain Modularity** algorithm for looking at grouping quality and hierarchies.

Common use cases of community detection include:

- Fraud detection: finding fraud rings by identifying accounts that have frequent suspicious transactions and/or share identifiers between one another.
- Customer 360: disambiguating multiple records and interactions into a single customer profile so an organization has an aggregated and complete source of truth for each customer.
- Market segmentation: dividing a target market into approachable subgroups based on priorities, behaviors, interests, and other criteria.

### 4.1.1 Triangle Count and Clustering Coefficient

The Triangle Count algorithm counts the number of triangles for each node in the graph. A triangle consists of three nodes, each connected to every other node in the set. Networks with a significant number of triangles tend to display small-world structures and behaviors. Triangles are frequently utilized in calculating network metrics like the local clustering coefficient.
The Local Clustering Coefficient measures the likelihood that neighbors of a specific node are connected to each other. This coefficient assesses the degree of clustering within a group relative to its potential maximum clustering: a score of 1 signifies a clique, where every node is interconnected with every other node.
The formula to compute the local cluster coefficient is as follows:

$$C_n = \frac{2T_n}{d_n(d_n - 1)} \tag{4.1}$$

where $T_n$ is the number of triangles passing through $n$ and $d_n$ is the degree of $n$.
Moreover, the algorithm can calculate the average clustering coefficient for the entire graph. This is obtained by normalizing the sum of all local clustering coefficients.
A $C_u = 0.2$ for node $u$ in Figure 4.1 means that any two nodes connected to $u$ have a 20% chance of being connected to each other.

### 4.1.2 Weakly Connected Components and Strongly Connected Components

The Weakly Connected Components (WCC) algorithm identifies groups of linked nodes in directed and undirected graphs where two nodes are connected if there exists a

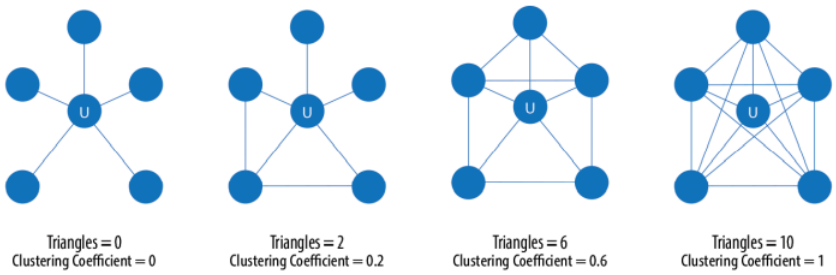



Figure 4.1: Triangle counts and clustering coefficients for node $u$ [5]

path between them. The direction of relationships on the path between two nodes is not considered here. A recommended approach is to run WCC to determine if a graph is connected before proceeding with general graph analysis. This precautionary step helps avoid running algorithms on isolated components of a graph, which could lead to inaccurate results.

The Strongly Connected Components (SCC) algorithm identifies clusters of interconnected nodes within a directed graph. A group is classified as a strongly connected component if every pair of nodes within it is linked by a directed path. It differs from the previous algorithm because the direction of relationships on the path between two nodes is relevant here. As with WCC, SCC is often used early in an analysis to understand how a graph is structured. Strongly connected components can help characterize similar behaviors or preferences within a group, with practical applications in recommendation engines.

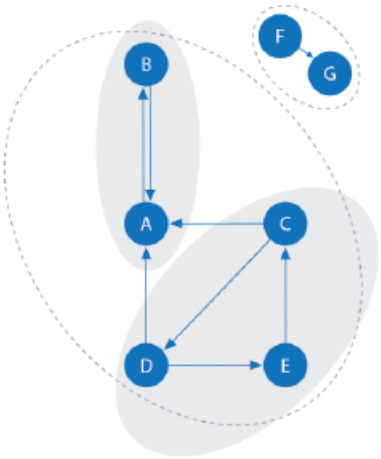



Figure 4.2: Weakly/Strongly Connected Components [5]

We can identify in the Figure 4.2 two weakly connected components shown with dashed outlines $\{A, B, C, D, E\}$ and $\{F, G\}$, and two strongly connected components shown shaded. Both community detection algorithms we've covered are deterministic, providing identical results each time they're executed. In contrast, the following two algorithms are

nondeterministic examples, potentially yielding different outcomes in multiple runs, even with the same input data.
The following two algorithms are examples of nondeterministic algorithms, where we may see different results if we run them multiple times, even on the same data.

### 4.1.3 Label Propagation

The Label Propagation Algorithm(LPA) swiftly identifies communities within a graph. In LPA, nodes choose their community based on their direct neighbors using node labels. Moreover, node and edge weights can be taken into account. The algorithm operates on the principle that a single label can quickly dominate densely connected nodes but may encounter difficulty crossing sparsely connected regions. To handle overlaps, where nodes might belong to multiple communities, LPA assigns them to the label neighborhood with the highest combined node and edge weights. Convergence in LPA is achieved when each node adopts the most prevalent label among its neighbors.
LPA is often used in large-scale networks for initial community detection, especially when weights are available. It is a nondeterministic algorithm as mentioned above, so its outcomes may vary across multiple runs on the same graph. It's important to note that the sequence in which LPA assesses nodes can impact the resultant community structures.

Figure 4.3: Label Propagation Algorithm [5]

### 4.1.4 Louvain Modularity

The Louvain Modularity algorithm detects clusters in large networks through a repeated two-step process.
The first step involves a greedy approach to assigning nodes to communities, focusing on local modularity enhancements. Modularity is calculated as the ratio of relationships within specific groups minus the expected ratio if relationships were randomly assigned among all nodes. It begins by determining the modularity change if a node joins a community with each of its adjacent neighbors. The node then becomes part of the community that offers the highest modularity increase.
The second step involves creating a new, simplified network based on the communities identified in the first step. These two steps are repeated until no additional modularity-increasing community reassignments can be made.
The Louvain algorithm is known for its efficiency among modularity-based methods. Beyond community detection, it effectively uncovers hierarchical structures across multiple scales, providing insights into network organization at varying levels of detail.

Modularity optimization algorithms typically aim to maximize a score that quantifies the quality of how nodes are assigned to communities within a network. This algorithm class, including Louvain, suffer from two issues. First, it can omit small communities within large networks. Secondly, in extensive graphs with overlapping communities, these algorithms might not accurately identify the overall maximum. In such scenarios, we will use any modularity algorithm only as a guide for a rough estimate.

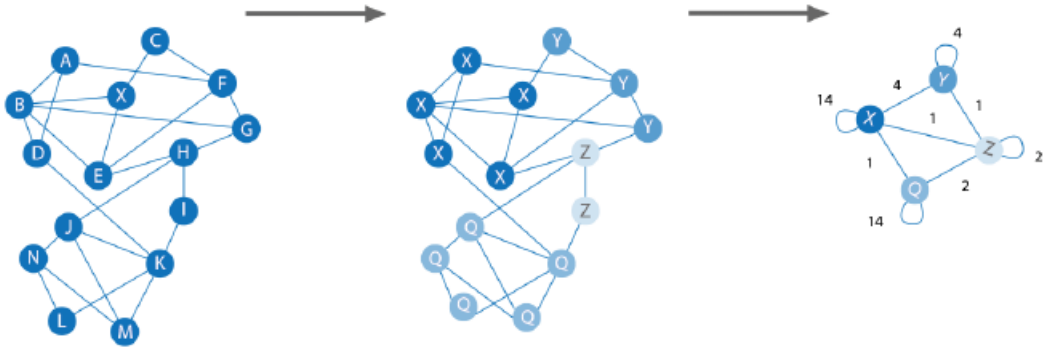



Figure 4.4: Louvain Modularity Algorithm [5]

## 4.2 Pathfinding and Graph Search Algorithms

Let us introduce this class of graph algorithms starting from a small simple graph as in Figure 4.5.

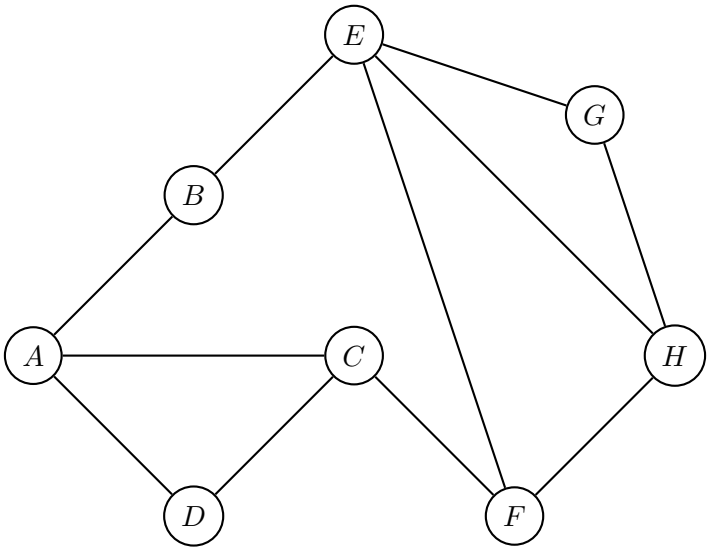



Figure 4.5: Example unweighted graph.

Looking at it, interesting questions to do could be the following ones [21]:

1. Is there a path from A to H? It is a reachability question investigating the existence of paths through the graph between a pair of nodes with no expectation that those paths are computationally optimal.

2. How many paths exist between A and H? It is a question about the robustness of the network for example to answer something about the likelihood a random traveller is gonna be able to get from A to H, when one of these paths will be interrupted.

3. What is the shortest or the cheapest path between A and H or between one node to all others? The goal here is to find paths computationally optimal, also in networks with millions nodes.

4. What is the degree of separation or the number of hops between A and H? The smaller the degree, the greater the knowledge/ influence/trust of the other will be.

We will examine the most representative pathfinding and search algorithms that will allow us to answer the questions above:

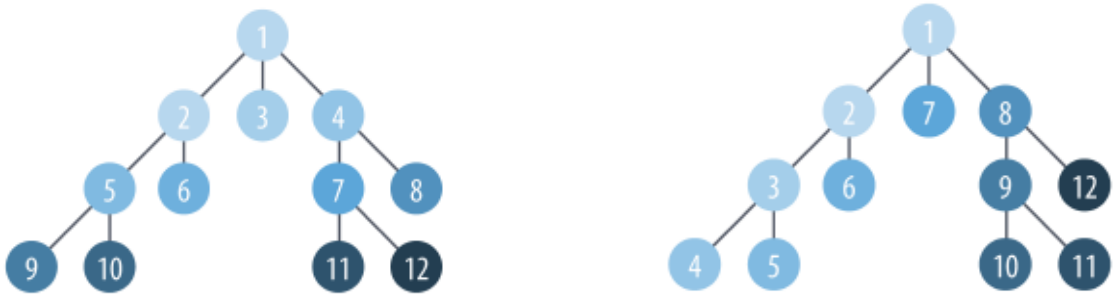
- **Breadth First Search** and **Depth First Search** algorithms for traversing a graph through its relationships.
- **Dijkstra Source-Target Shortest Path** algorithm for computing the shortest path between a source and a target node.
- **Dijkstra Single-Source Shortest Path** algorithm for computing the shortest path between one source and multiple targets through Dijkstra algorithm.
- **All Pairs Shortest Path** algorithm for finding the shortest paths between the nodes in the graph.

Common use cases of path finding include:

- Supply chain analytics: identifying the fastest path between an origin and a destination (a starting point can be a warehouse and an ending point a client or a shop) or between a raw material and a finished product.
- Customer Journey: analyzing the events that make up a customer experience. In healthcare for example, this can be the experience of an in-patient from admission to discharge.

### 4.2.1 Breadth First Search and Depth First Search

Breadth First Search algorithm (BFS) is essential for traversing graphs. It starts at a chosen node and systematically explores neighboring nodes at each successive distance level, starting from the closest and moving outward. BFS is preferred in general when the graph structure is less balanced or the target is closer to the starting point.
Depth First Search (DFS) is the other principal graph traversal algorithm. It begins at a designated node, selects a neighbor to explore, and continues exploring as far as possible down its branch before backtracking. This algorithm can be preferred over BFS




(a) Breadth First Search (b) Depth First Search

Figure 4.6: Graph Search Algorithms [5]

for example if we have deeply hierarchical data with the target node closer to an endpoint or as a practical resource for simulating potential routes in scenario modeling.
Numbers in Figure 4.6 represent the order in which these nodes have visited.
Both algorithms are often used as precursors in other algorithms for example Shortest Path or Connected Components rather than on their own.

We can observe how search algorithms set the foundation for traversing graphs. Next, we will delve into pathfinding algorithms.

### 4.2.2 Dijkstra Source-Target Shortest Path

The Dijkstra Shortest Path algorithm calculates the shortest path between a pair of nodes considering either the number of steps or the weight, depending on whether the graph is weighted or unweighted. Weights can signify different metrics such as time, distance, capacity, or cost, while 'hop' generally denotes the number of links between nodes.
Dijkstra Shortest Path algorithm begins by identifying the least-weighted connection from the starting node to its directly linked nodes. It records these weights and proceeds to the nearest node. It repeats this process by calculating a cumulative total weight from the starting node. Throughout the algorithm, it consistently selects the path with the lowest cumulative weight to advance further through the graph, continuing until it reaches the destination node.
This algorithm can be employed to find the best paths between two nodes, based on either the number of steps or the value of weighted relationships. For example, it can offer real-time solutions for determining degrees of separation, finding the shortest distance between points, or identifying the most cost-effective route.

### 4.2.3 Dijkstra Single-Source Shortest Path

The Dijkstra Single-Source Shortest Path (SSSP) algorithm computes the shortest weighted path from a starting node to all reachable nodes in the graph, following the same principles as the standard Dijkstra algorithm.

SSSP is useful for determining the most efficient route from a specified origin to each individual node. It prioritizes paths based on the cumulative path weight from the starting node, making it particularly effective for identifying optimal paths to each node, rather than aiming to visit all nodes in a single journey.

### 4.2.4 All Pairs Shortest Path

The All Pairs Shortest Path (APSP) algorithm computes a shortest path group containing all the shortest paths between all pairs of nodes, that are weighted if the graph is weighted. It is more efficient than running the SSSP algorithm for every pair of nodes in the graph.
This algorithm is typically applied to explore alternative routes in situations where the shortest path is blocked or less effective.

## 4.3 Centrality Algorithms

Centrality algorithms are employed to assess the significance of specific nodes within a graph and their influence on the entire network but what does influence mean? We have to define this concept of importance for a certain node in the network that depends by the particular use case if our goal is to answer real relevant questions as:

- Where to locate a distribution center in a logistic center in order to speed up the truck deliveries?
- Where to locate a Starbucks point or a gas station ?
- Which seller in a market has the most pricing power?

The concept of centrality is strictly linked to the abstract concept of travel: something like physical goods, info or ideas is moving across the network and the underlying graph structure represents how these things spread in the graph. Centrality algorithms also provide insights into group dynamics, encompassing aspects such as credibility, accessibility, the speed of information dissemination, and interconnections between groups. [22].
Based on the concept of influence used, we will examine the most representative centrality algorithms that will allow us to answer the questions above:

- **Degree Centrality** algorithm based on the number of connections for a certain node in a graph.
- **Closeness Centrality** algorithm for measuring the reachability of a node from all the others in a graph or subgraph.
- **Betweenness Centrality** algorithm for finding control points in a network that has the most power over flow between nodes and groups.
- **PageRank** algorithm for understanding the overall influence as we will see in detail.

Common use cases of centrality include:

- Recommendation: identify and recommend the most popular items in your content or product offering catalog, where popular is a synonymous of influential in this case.

- Supply chain analytics: find the most critical node in your supply chain network, whether it be a supplier in a network or a raw material that is part of a manufactured product.

- Fraud & Anomaly Detection: find users with many shared identifiers or who otherwise act as a bridge between many communities.

### 4.3.1 Degree Centrality

Degree Centrality is the simplest of the centrality algorithms that measures the importance of a node counting the number of incoming or outgoing relationships from a node that in mathematical terms represents its in-degree or out-degree respectively if the graph is direct or simply its degree in an undirected graph.
It examines immediate connections, which is valuable for tasks such as assessing the short-term risk of virus transmission to individuals or finding the popularity of individual nodes. However, this algorithm is also employed in global network analysis to assess metrics like the minimum degree, maximum degree, average degree, and standard deviation throughout the entire graph.

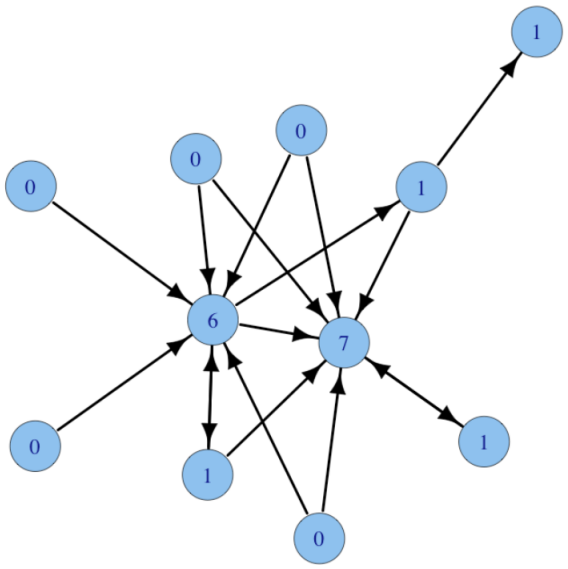



Figure 4.7: Degree Centrality Algorithm [23]

Nodes 6 and 7 in the Figure 4.7 have a high degree centrality because of the highest number of direct connections with the other nodes.

### 4.3.2 Closeness Centrality

Closeness Centrality measures how central a node is to all its neighbors within its cluster. This measure of centrality for a node $u$ is calculated using the formula:

$$C_u = \frac{1}{\sum_{v=1}^{n-1} d(u,v)} \tag{4.2}$$

where $d(u,v)$ is the shortest-path distance between another node $v$ and the node $u$ and $n$ is the number of nodes in the same group as $u$. Normalization of this score typically involves considering the average length of the shortest paths rather than their sum, as expressed in the formula for normalized closeness centrality.:

$$C_{u,norm} = \frac{n-1}{\sum_{v=1}^{n-1} d(u,v)} \tag{4.3}$$

Nodes with a high closeness score are those with the shortest distances to all other nodes, enabling them to potentially reach the entire group quickly.
Because of this algorithm is a way of detecting nodes that can most easily reached from all the others, closeness centrality has been utilized to identify ideal locations for new public services to enhance accessibility. In social analysis, it can pinpoint individuals in the most advantageous social network positions for faster information dissemination.

### 4.3.3 Betweennes Centrality

Betweennes Centrality algorithm starts by computing the shortest (weighted) paths between every pair of nodes in a connected graph using the APSP algorithm. Each node is then evaluated based on how frequently it lies along these shortest paths: nodes that appear most often on such paths receive higher betweenness centrality scores and act as crucial links between various clusters within the graph.
Betweenness Centrality quantifies how much influence a node has on the flow of information or resources within a graph. In network terms, a bridge can be either a node or a link. In smaller graphs, we can identify these influential elements by observing which node or link, if removed, would disconnect a portion of the graph.
We note that node 3 in Figure 4.8 is not involved in high number of relationships because it has only two connections but with a high betweennes centrality: if we remove this node, we disconnect two parts of the network and the flow of information or goods through it.

### 4.3.4 PageRank Centrality

All the previous centrality algorithms measure the direct influence of a node, whereas PageRank Centrality algorithm evaluates the importance of a node based on its linked neighbors, and iteratively considers the importance of those neighbors' neighbors. A node is authoritative if several *authoritative* nodes point to it.
The original purpose of this algorithm was to find the most "authoritative" page to rank websites in Google search results. The core idea is that a web page with many influential

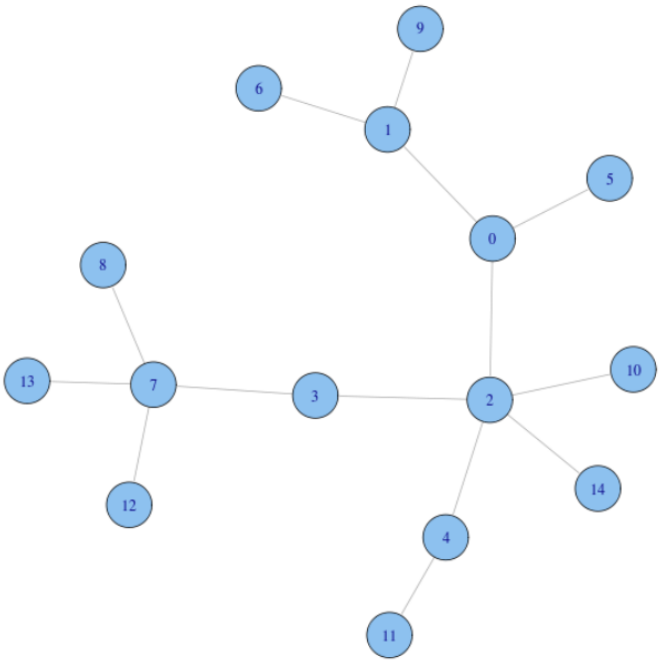



Figure 4.8: Betweennes Centrality Algorithm [24]

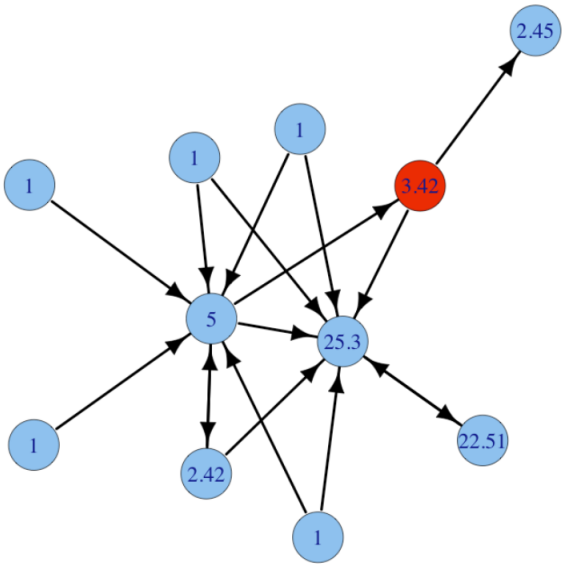



Figure 4.9: PageRank Algorithm [25]

incoming links is perceived as more credible. Therefore, connections to highly important nodes have a greater impact on the influence of the node in question compared to

connections to less significant nodes.
The red node in Figure 4.9 has the forth highest pageRank centrality value in the graph because it is connected to a very authoritative node, also it does not have so many nodes that point to it.

## 4.4 Similarity Algorithms

Similarity algorithms determine how similar pairs of nodes are based on their neighborhoods or their properties using different vector-based metrics or similarity functions: how we measure this similarity between two entities depends on the similarity function used. Similarity is in the eye of the observer that decides what factors care about to him and their relative importance.
Common use cases for similarity include:

- Fraud Detection: finding potential fraud user accounts by analyzing the similarities between the new user accounts and the flagged fraudster accounts. Possible signs could be sudden suspicious transactions or connections to parties already considered high risk.

- Recommendation System: in a e-commerce platform we want to identify items similar to the one currently being viewed by a user to increase rate of purchase.

- Entity Resolution: determine similar nodes in terms of activity or identifiers in the graph to reduce its size.

We want to investigate a graph-based structural similarity, in other terms explore how two nodes are similar based on their relationships to find for example investment opportunities similar to the successful ones or activity similar to that of a known fraudster.
We will examine the most representative similarity algorithms **K-Nearest Neighbor (KNN)**, used for determining similarity based on node properties and **Node Similarity** used for determining similarity based on the relative proportion of shared neighboring nodes in the graph.

### 4.4.1 K-Nearest Neighbors

The K-Nearest Neighbors algorithm calculates a distance metric for all pairs of nodes within a homogeneous graph, establishing new connections between each node and its $k$ closest neighbors. The distance is computed using *similarity functions*, a collection of metrics utilized to assess similarity between two arrays $p_s$, $p_t$ of numbers that represent node properties. KNN provide choices between different similarity metrics: using different metrics will of course alter the similarity score and change the interpretation slightly.
We can said that the K-Nearest Neighbors algorithm evaluates specific properties of each node by identifying the $k$ nodes with the closest similarity in these properties, which become the node's k-nearest neighbors.

### 4.4.2 Node Similarity

The input of the Node Similarity algorithm is a bipartite graph and it compares nodes belonging to one set of nodes based on their outgoing relationships to the other set of nodes. For each pair $(n, m)$ of the same node set, we collect the *outgoing neighborhood* $N(n)$ and $N(m)$ and the algorithm computes a similarity for that pair applied to $N(n)$ and $N(m)$ vectors. The output of this algorithm are new relationships between pairs of the first node set with similarity scores expressed via relationship properties.

### 4.4.3 Similarity functions

There are two types of similarity functions: categorical and numerical. Categorical functions treat arrays as sets and determine similarity based on the intersection of these sets, like the Jaccard similarity. Numerical measures assess similarity by comparing how closely the numbers at each position in the arrays align with each other and we analyze the most famous ones, Cosine similarity and Pearson similarity.
When a property is represented as a list of integers, similarity can be assessed using the Jaccard score:

$$J(p_t, p_s) = \frac{|p_t \cap p_s|}{|p_t \cup p_s|} \tag{4.4}$$

counting how many features two nodes have in common. Notice that the above formula gives a score in the range of [0,1].
When a property is a list of floating-point numbers, there are two alternatives previously mentioned for computing similarity between two nodes. The default metric used is that of Cosine similarity, and then the Pearson similarity.

$$cosine(p_s, p_t) = \frac{\sum_i p_s(i)p_t(i)}{\sqrt{\sum_i p_s(i)^2}\sqrt{\sum_i p_t(i)^2}} \tag{4.5}$$

$$pearson(p_s, p_t) = \frac{\sum_i (p_s(i) - \bar{p_s})(p_t(i) - \bar{p_t})}{\sqrt{\sum_i (p_s(i) - \bar{p_s})^2}\sqrt{\sum_i (p_t(i) - \bar{p_t}^2}} \tag{4.6}$$

Both these measures give a score in the range of $[-1, 1]$ . The score is normalized into the range $[0, 1]$ by doing $score = \frac{score + 1}{2}$.

# Chapter 5

# Graph Powered Machine Learning in Practice

We have focused on graph algorithms in the previous chapter for exploring the theoretical ideas and the working behind each algorithm and some more representative use cases. The question now is: how could we use graph algorithms to improve the learning phase of a machine learning workflow? Extracting connected features is the most practical method to begin enhancing ML predictions with graph algorithms. Just as people should use context for better decisions,to understand what is essential in a situation and determine how to apply lessons from experience to new situations, we need to include a lot of contextual information in machine learning models in order to understand related network dynamics and use them to deliver better machine learning features increasing predictions. For example, e-commerce platforms customize product recommendations by using both historical data and contextual information about customer or product similarities. Social networks influence people to vote not only by their direct relationships, also through friends of friends that could have more impact than the direct friends alone and so on.
*Connected features* are characteristics extracted from the data structure using graph-local queries, which focus on the regions around a node, or graph-global queries, which utilize graph algorithms to find salient and predictive features: putting together the right mix of connected features can increase performance because it essentially influences how our models learn.

## 5.1 Link Prediction Problem with graphs

Connected features play a role in improving machine learning in the context of link prediction.
Link prediction involves predicting the probability of a relationship forming in the future or identifying potential relationships that are absent from our graph due to incomplete data [5]. This machine learning task was popularised by a paper "The Link Prediction Problem for Social Networks" written in 2004 by Jon Kleinberg and David Liben-Nowell where they deal with this problem from the perspective of social networks, asking the question [26]:

> *Given a snapshot of a social network, can we infer which new interactions among its members are likely to occur in the near future? We formalize this question as the link prediction problem, and develop approaches to link prediction based on measures for analyzing the "proximity" of nodes in a network.*

Since the dynamic nature of networks, being able to predict future links between entities has broad applicability: we could predict future friendships in a social network making friend recommendations, new connections between molecules in a biology network, suggest potential collaborations among authors in a citation network or inferring criminal relationships in fraud detection applications.
Kleinberg and Liben-Nowell describe a set of methods specific for link prediction problem based on the idea that *the closer two nodes are, the higher the likelihood of a relationship between them.* [26]. Given a pair of nodes, these methods compute a score that could be considered a measure of closeness between them based on the graph topology, specifically on their shared neighbors.
We will report here the most representative formulas [27] where $N(x)$ is the set of nodes adjacent to node $x$ and $N(y)$ is the set of nodes adjacent to node $y$:

- Common Neighbors: the idea is that, the more common neighbors two nodes share currently, the more likely a link will form in between them in the future.

$$CN(x, y) = |N(x) \cap N(y)| \tag{5.1}$$

- Total Neighbors: the intuition is that nodes that are highly connected tend to attract more new links over time.

$$TN(x, y) = |N(x) \cup N(y)| \tag{5.2}$$

- Preferential Attachment: the "rich gets richer" idea means that the larger the current neighborhood of the two nodes, the more likely the future connection.

$$PA(x, y) = |N(x)||N(y)| \tag{5.3}$$

- Adamic Adar: this measure also considers common neighbors between two nodes but gives more weight to common neighbors with smaller degree.

$$AA(x, y) = \sum_{u \in N(x) \cap N(y)} \frac{1}{log|N(u)|} \tag{5.4}$$

To deal with this problem we will use Neo4j as native graph database and specifically, we will use the Graph Data Science (GDS) of Neo4j, the connected data analysis platform that unifies the machine learning surface and graph database into a single workspace. For queries, we will use Cypher, a standard graph query language adopted from Neo4j and other graph databases. We will also establish a connection with the Neo4j database through a Python driver in order to harness the power of Python in data science. GDS deals with the link prediction problem as a binary classifier where the target variable

to predict is a $0-1$ indicator, 0 for no link, 1 for a link. This type of link prediction works really well on an undirected graph where we are predicting one type of relationship between nodes of a single label.
Our intent now is to tackle a social network recommendation task as a link prediction problem solving it with a graph approach based on the extraction, the selection and the application of connected features in a machine learning model. The main steps [28] of our first graph analysis will be:

1. Graph model creation
2. Split the data into train and test sets
3. Graph feature engineering
4. Model selection and training
5. Model evaluation

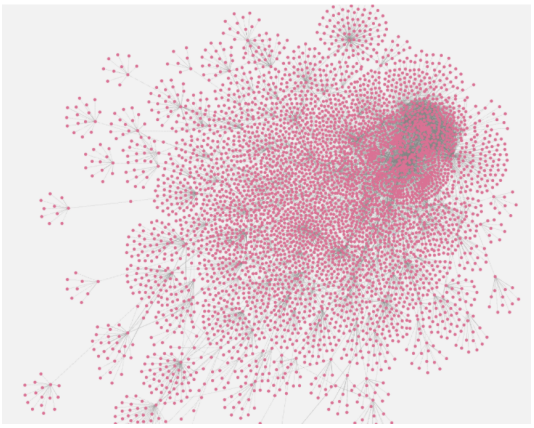
### 5.1.1 Graph Model Creation

After establishing a connection with the Neo4j database through a Python driver, we load our connected data into the graph database[1]. We will create 22,470 nodes and 171,002 relationships between them: nodes represent Facebook pages while relationships are mutual likes between sites. This is a page-page graph of verified Facebook pages as we can see in Figure 5.1.

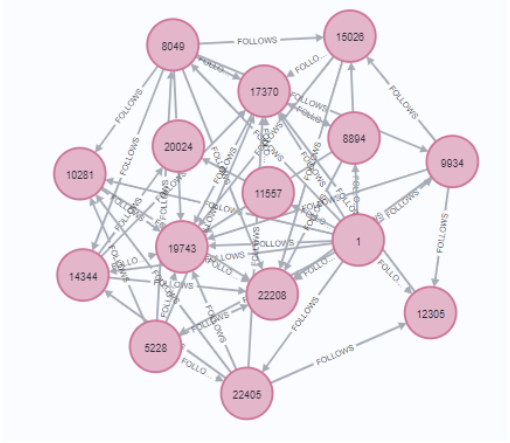
```
from neo4j import GraphDatabase


#Connection with Neo4j db
uri = "bolt://localhost:7687"
driver = GraphDatabase.driver(uri, auth=("neo4j", "password"))
with driver.session(database="neo4j") as session:
     display(session.run("CREATE CONSTRAINT ON (p:Page) ASSERT p.id IS UNIQUE;")
     .consume().counters)

#Import data into the database
query = """
LOAD CSV WITH HEADERS FROM
'file:///musae_facebook_target.csv' AS row
MERGE (p:Page {id:row.id,faceID:row.facebook_id,name:row.page_name, type:row.
    page_type})
"""
with driver.session(database="neo4j") as session:
    session.run(query)

query = """
LOAD CSV WITH HEADERS FROM
'file:///musae_facebook_edges.csv' AS row
MATCH (p1:Page {id:row.id_1})
MATCH (p2:Page {id:row.id_2})
```

[1]The data is available at `https://snap.stanford.edu/data/facebook-large-page-page-network.html`

```
MERGE (p1)-[:FOLLOWS]-(p2)
"""
with driver.session(database="neo4j") as session:
     session.run(query)
```



(a) Facebook subgraph with 4895 nodes and 10340 links




(b) Graph schema

Figure 5.1: Facebook Large Page-Page Network through Neo4j Bloom

### 5.1.2 Train and Test Datasets

Now that we have created our graph, we choose to adopt a supervised learning approach to predict future relationships between Facebook pages. To do this, it is therefore necessary to come up with train and test datasets on which we can build our binary classifier. When working with graph data, it's challenging to split the data randomly due to the risk of data leakage.

Data leakage occurs when test data influences the creation of our model inadvertently. This is particularly common in graphs because nodes in the training set may have connections to nodes in the test set. Instead, it's essential to partition the graph into training and test subgraphs while preserving their overall network structure. If the graph includes a temporal aspect, one approach is to split it at a specific time point: the training set includes data before this time, and the test set includes data after. However, this approach doesn't apply in our case.

We will use a utility algorithm SPLIT RELATIONSHIPS that divides the relationships into a holdout set and a remaining set [29]. The holdout set is divided into two classes: positive, i.e., existing relationships, and negative, i.e., non-existing relationships. We need negative examples so that our model can learn to distinguish between nodes that should be connected and those that should not be. The class is indicated by a *label* property on the relationships.

Now we can proceed to the creation of the training and test set and their respective relationships storing both into Neo4j database for our machine learning pipeline.

```
#Creation of FOLLOWS_TESTGRAPH relationships
query = """
CALL gds.alpha.ml.splitRelationships.mutate('pages', {
  relationshipTypes: ['FOLLOWS'],
  remainingRelationshipType: 'FOLLOWS_REMAINING',
  holdoutRelationshipType: 'FOLLOWS_TESTGRAPH',
  holdoutFraction: 0.2,
  randomSeed: 1337
})
YIELD createMillis, computeMillis, mutateMillis, relationshipsWritten;
"""
with driver.session(database="neo4j") as session:
    result = session.run(query)

#Creation of FOLLOWS_TRAINGRAPH relationships
query = """
CALL gds.alpha.ml.splitRelationships.mutate('pages', {
  relationshipTypes: ['FOLLOWS_REMAINING'],
  remainingRelationshipType: 'FOLLOWS_IGNORED_FOR_TRAINING',
  holdoutRelationshipType: 'FOLLOWS_TRAINGRAPH',
  holdoutFraction: 0.4,
  randomSeed: 1337
})
YIELD createMillis, computeMillis, mutateMillis, relationshipsWritten;
"""
with driver.session(database="neo4j") as session:
    session.run(query)
```

```
Examples train: 109296
Negative examples train: 54648
Positive examples train: 54648

Examples test:  68400
Negative examples test: 34200
Positive examples test: 34200
```

Note that we have a split of $62-38$ and a balanced problem in terms of positive examples and negative ones. Before we will move on, let us have a look at the contents of our train and test dataframes:

| | node1 | node2 | label |
|---|---|---|---|
| **70432** | 14498 | 12776 | 0 |
| **5638** | 1162 | 19463 | 0 |
| **47435** | 9759 | 17216 | 1 |
| **57744** | 11809 | 61 | 1 |
| **84232** | 17354 | 3798 | 1 |

(a) Training dataframe sample

| | node1 | node2 | label |
|---|---|---|---|
| **47169** | 15513 | 22068 | 0 |
| **28027** | 9223 | 22411 | 0 |
| **7119** | 2387 | 7368 | 1 |
| **9977** | 3328 | 5051 | 1 |
| **27406** | 9034 | 9572 | 0 |

(b) Test dataframe sample

Figure 5.2: Training and test dataframes

### 5.1.3 Graph feature engineering

Now it is time to engineer some features which we will use to train our model [5]. Firstly, we consider the specific link prediction measures which we mentioned about previously:

- common neighbors score ($cn$)
- preferential attachment score ($pa$)
- total neighbors score ($tn$)
- adamic adar score ($aa$)

The following function calculates each of these metrics for pairs of nodes and adds 4 columns to the train and test dataframes:

```
#Compute cn, pa, tn, aa scores
def apply_graphy_features(data, rel_type):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
           pair.node2 AS node2,
           gds.alpha.linkprediction.commonNeighbors(p1, p2, {
             relationshipQuery: $relType}) AS cn,
           gds.alpha.linkprediction.preferentialAttachment(p1, p2, {
             relationshipQuery: $relType}) AS pa,
           gds.alpha.linkprediction.totalNeighbors(p1, p2, {
             relationshipQuery: $relType}) AS tn,
           gds.alpha.linkprediction.adamicAdar(p1, p2, {
             relationshipQuery: $relType}) AS aa
    """
    pairs = [{"node1": node1, "node2": node2} for node1, node2 in data[["node1", "
    node2"]].values.tolist()]
    with driver.session(database="neo4j") as session:
        result = session.run(query, {"pairs": pairs, "relType": rel_type})
        features = pd.DataFrame([dict(record) for record in result])
    return pd.merge(data, features, on=["node1", "node2"])

training_df = apply_graphy_features(training_df, "FOLLOWS_TRAINGRAPH")
test_df = apply_graphy_features(test_df, "FOLLOWS")
```

Note that for the training dataframe, these metrics are computed solely based on the training relationships. In contrast, for the test dataframe, they are computed across the entire graph.
In social networks, predictions often rely on triangle metrics, so we calculate the number of triangles a node is involved in and its clustering coefficient to verify the impact of these two features in our model. If we look at a social network at two distinct points in time, we will generally find that a significant number of new edges have made among people with a common neighbor forming triangles, fully connected subgraphs between three nodes [1]. We could use the triangle count algorithm 4.1.1 of GDS library so that for each node we will write as node property the number of triangular structures involving it, *trianglesTrain* and *trianglesTest* properties for respectively training and test datasets. The same goes for the clustering coefficient algorithm 4.1.1 of GDS library.

```
#Counting triangles train set
query = """
CALL gds.triangleCount.write('graph1', {writeProperty: 'trianglesTrain'});
"""
with driver.session(database="neo4j") as session:
    result = session.run(query)
    df = pd.DataFrame([dict(record) for record in result])
df
#Counting triangles test set
query = """
CALL gds.triangleCount.write('graph2', {writeProperty: 'trianglesTest'});
"""
with driver.session(database="neo4j") as session:
    result = session.run(query)
    df = pd.DataFrame([dict(record) for record in result])
df
```

A value of 794953 as global triangle count for the test set indicates that the Facebook network is a interconnected network as could be guessed from the snapshot of a portion of the graph in Figure 5.1.

```
# Local clustering coefficient train set
query = """
CALL gds.localClusteringCoefficient.write('graph1', {writeProperty: '
    coefficientTrain'});
"""
with driver.session(database="neo4j") as session:
    result = session.run(query)
    df = pd.DataFrame([dict(record) for record in result])
df

# Local clustering coefficient test set
query = """
CALL gds.localClusteringCoefficient.write('graph2', {writeProperty: '
    coefficientTest'});
"""
with driver.session(database="neo4j") as session:
    result = session.run(query)
    df = pd.DataFrame([dict(record) for record in result])
df
```

The average clustering coefficient for the Facebook network is almost 0.4, which indicates that the network is quite tightly-knit.
The following function will add these features to our test and train dataframes:

```
def apply_triangles_features(data, triangles_prop, coefficient_prop):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
    pair.node2 AS node2,
    apoc.coll.min([p1[$trianglesProp], p2[$trianglesProp]]) AS minTriangles,
    apoc.coll.max([p1[$trianglesProp], p2[$trianglesProp]]) AS maxTriangles,
    apoc.coll.min([p1[$coefficientProp], p2[$coefficientProp]]) AS minCoefficient,
    apoc.coll.max([p1[$coefficientProp], p2[$coefficientProp]]) AS maxCoefficient
    """
    pairs = [{"node1": node1, "node2": node2}  for node1,node2 in data[["node1", "
    node2"]].values.tolist()]
    params = {
    "pairs": pairs,
```

```
    "trianglesProp": triangles_prop,
    "coefficientProp": coefficient_prop
    }
    with driver.session(database="neo4j") as session:
        result = session.run(query, params)
        features = pd.DataFrame([dict(record) for record in result])
    return pd.merge(data, features, on = ["node1", "node2"])

training_df = apply_triangles_features(training_df, "trianglesTrain", "
    coefficientTrain")
test_df = apply_triangles_features(test_df, "trianglesTest", "coefficientTest")
```

These metrics are distinct from those used previously because they are specific to nodes rather than pairs of nodes. Consequently, we cannot simply add these values to our dataframe because the order of nodes in pairs is not guaranteed. So we compute the minimum and maximum value of *trianglesProp* and *coefficientProp* for each pair of nodes. The previous scores are neighborhood metrics in the sense they consider the directly connected neighboring nodes focusing in the role that each node and its set of close connections have in the network. To understand the effects of a node on the network, it is important to consider the node as part of a community because communities have a greater effect on the network than a single node and more influence in term of information exchange. We assume that nodes within the same community are more likely to establish a link if one doesn't already exist between them. Additionally, we posit that communities with stronger internal connections are more inclined to form links. To begin, we will identify larger communities using the Label Propagation algorithm 4.1.3 in Neo4j, saving the community assignments in the property *partitionTrain* for the training set and *partitionTest* for the test set.

```
#Compute communities with Label Propagation algo train set
query = """
CALL gds.labelPropagation.write('graph1',{maxIterations:20, writeProperty: '
    partitionTrain'});
"""
with driver.session(database="neo4j") as session:
    result = session.run(query)
    df = pd.DataFrame([dict(record) for record in result])
df
#Compute communities with Label Propagation algo test set
query = """
CALL gds.labelPropagation.write('graph2',{maxIterations:20, writeProperty: '
    partitionTest'});
"""
with driver.session(database="neo4j") as session:
    result = session.run(query)
    df = pd.DataFrame([dict(record) for record in result])
df
```

We will utilize the Louvain algorithm 4.1.4, which identifies intermediate clusters to uncover fine-grained communities within a graph. Each node will be assigned a property indicating its community after the initial iteration of the algorithm, the *louvainTrain* for the training set and *louvainTest* for the test set:

```
#Compute communities with Louvain algo train set
query = """
CALL gds.louvain.stream('graph1', {includeIntermediateCommunities: true})
```

```
YIELD nodeId, communityId, intermediateCommunityIds
WITH gds.util.asNode(nodeId) AS node,
     intermediateCommunityIds[0] AS smallestCommunity
SET node.louvainTrain = smallestCommunity;
"""
with driver.session(database="neo4j") as session:
    result = session.run(query)
    display(session.run(query).consume().counters)
#Compute communities with Louvain algo test set
query = """
CALL gds.louvain.stream('graph2',{includeIntermediateCommunities: true})
YIELD nodeId, communityId, intermediateCommunityIds
WITH gds.util.asNode(nodeId) AS node,
     intermediateCommunityIds[0] AS smallestCommunity
SET node.louvainTest = smallestCommunity;
"""
with driver.session(database="neo4j") as session:
    result = session.run(query)
    display(session.run(query).consume().counters)
```

The following function will add the extracted community features to our test and train dataframes:

```
def apply_community_features(data, partition_prop, louvain_prop):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
    pair.node2 AS node2,
    gds.alpha.linkprediction.sameCommunity(p1, p2, $partitionProp) AS sp,
    gds.alpha.linkprediction.sameCommunity(p1, p2, $louvainProp) AS sl
    """
    pairs = [{"node1": node1, "node2": node2}  for node1,node2 in data[["node1", "
    node2"]].values.tolist()]
    params = {
    "pairs": pairs,
    "partitionProp": partition_prop,
    "louvainProp": louvain_prop
    }
    with driver.session(database="neo4j") as session:
        result = session.run(query, params)
        features = pd.DataFrame([dict(record) for record in result])
    return pd.merge(data, features, on = ["node1", "node2"])

training_df = apply_community_features(training_df, "partitionTrain", "
    louvainTrain")
test_df = apply_community_features(test_df, "partitionTest", "louvainTest")
```

We will obtain the following samples for training and test data complete with all the extracted graph features for the different node pairs.

### 5.1.4 Model selection and training

We choose to use a supervised learning approach where the node proximity scores become the features to train a binary classifier, so we have to decide which machine learning model we are going to use. Many of the link prediction measures are computed using similar data and this means there is a feature interaction problem when it comes to training a machine learning model. We could not choose a model which assume an assumption of independence between features: if we select one of these models, we'll need to exclude

| | node1 | node2 | label | cn | pa | tn | aa | minTriangles | maxTriangles | minCoefficient | maxCoefficient | sp | sl |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **70432** | 14498 | 12776 | 0 | 0.0 | 84.0 | 19.0 | 0.000000 | 0 | 0 | 0.000000 | 0.000000 | 1.0 | 0.0 |
| **5638** | 1162 | 19463 | 0 | 0.0 | 40.0 | 13.0 | 0.000000 | 0 | 0 | 0.000000 | 0.000000 | 1.0 | 1.0 |
| **47435** | 9759 | 17216 | 1 | 2.0 | 510.0 | 59.0 | 0.597076 | 3 | 81 | 0.063529 | 0.066667 | 0.0 | 1.0 |
| **57744** | 11809 | 61 | 1 | 0.0 | 544.0 | 50.0 | 0.000000 | 0 | 8 | 0.000000 | 0.014260 | 1.0 | 0.0 |
| **84232** | 17354 | 3798 | 1 | 6.0 | 480.0 | 38.0 | 1.984908 | 22 | 23 | 0.079710 | 0.121053 | 1.0 | 1.0 |

Figure 5.3: Training dataframe with graph features

| | node1 | node2 | label | cn | pa | tn | aa | minTriangles | maxTriangles | minCoefficient | maxCoefficient | sp | sl |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **47169** | 15513 | 22068 | 0 | 0.0 | 5.0 | 6.0 | 0.000000 | 0 | 0 | 0.000000 | 0.000000 | 0.0 | 0.0 |
| **28027** | 9223 | 22411 | 0 | 0.0 | 1.0 | 2.0 | 0.000000 | 0 | 0 | 0.000000 | 0.000000 | 0.0 | 0.0 |
| **7119** | 2387 | 7368 | 1 | 6.0 | 481.0 | 44.0 | 1.440947 | 46 | 215 | 0.322823 | 0.589744 | 1.0 | 1.0 |
| **9977** | 3328 | 5051 | 1 | 20.0 | 1288.0 | 54.0 | 4.541356 | 259 | 490 | 0.473430 | 0.685185 | 1.0 | 1.0 |
| **27406** | 9034 | 9572 | 0 | 0.0 | 864.0 | 59.0 | 0.000000 | 272 | 419 | 0.774929 | 0.844758 | 0.0 | 0.0 |

Figure 5.4: Test dataframe with graph features

features with significant interactions to ensure accurate predictions. Alternately, we could select a model where feature interaction is less problematic, the ensemble methods like a random forest classifier.
Let us build a model based on all graph features in our random forest classifier.

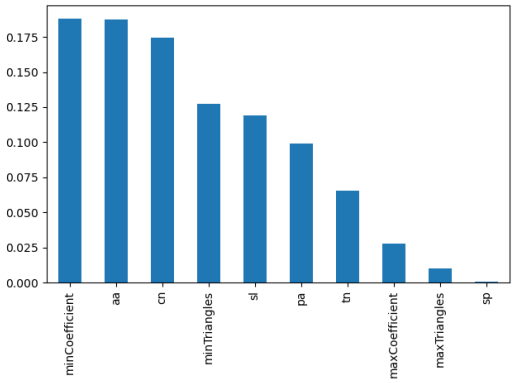
```
classifier = RandomForestClassifier(n_estimators=30, max_depth=6, random_state=0)

columns = [
    "cn", "pa", "tn","aa", #proximity features
    "minTriangles", "maxTriangles", "minCoefficient","maxCoefficient", #triangle
    features
    "sp", "sl" #community features
]
X = train[columns]
y = train["label"]
classifier.fit(X,y)
```

### 5.1.5 Model evaluation

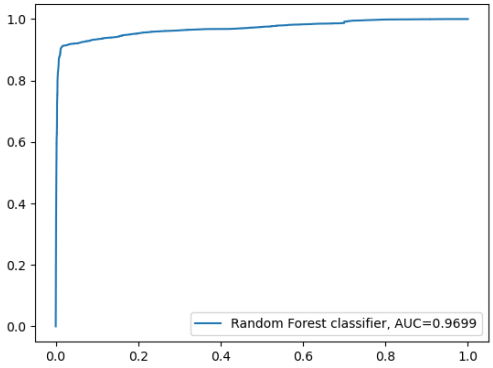
Let us evaluate our model against the test set using accuracy, precision, recall, and ROC curves as predictive metrics.
Firstly, considering the above classifier with all graph features we obtain the following results:

```
     Measure      Score
0   Accuracy   0.947602
1  Precision   0.982233
2     Recall   0.901696
```

This model has a precision of 0.982233, which means it is very good at predicting that links exist and a recall measure of 0.901696, which means it is also good at predicting when links do not exist, but to a lesser extent.




(a) Feature importance

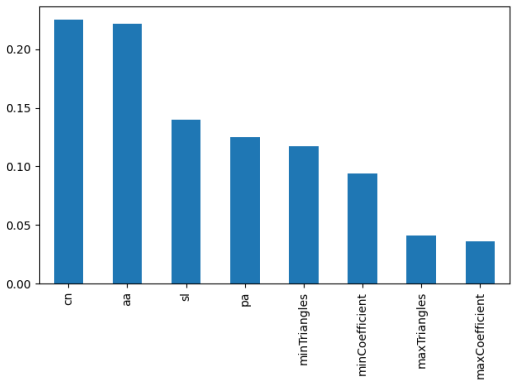



(b) ROC curve with all graph features

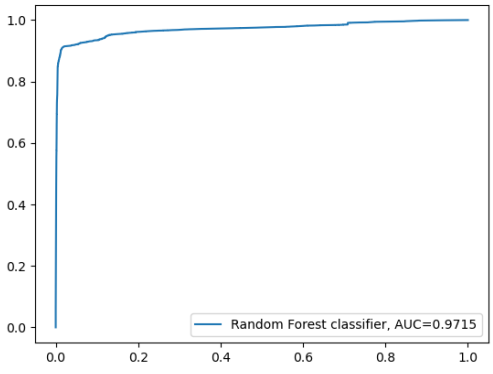
Figure 5.5: Random Forest classifier with all graph features

Could we improve the recall score in some way? Looking at Figure 5.5a, let us try to remove *sp* and *tn* features because *sp* is the least important of the community features while *tn* is the least important of the proximity measures, actually improving our recall metric:

| | Measure | Score |
|---|---|---|
| 0 | Accuracy | 0.947178 |
| 1 | Precision | 0.981397 |
| 2 | Recall | 0.911637 |




(a) Feature importance




(b) ROC curve without two graph features

Figure 5.6: Random Forest classifier without *sp* and *tn* features

## 5.2 Graph Recommender System

Another area where a graph representation of the data and a graph-based analysis can play an important role simplifying data management, mining and communication is *recommendations*.

The term *recommender system* refers to all software tools and techniques that, starting from the knowledge about users and items in question, suggest items that are likely to be of interest to a particular user [30]. In this context *item* is the general term used to identify what the system recommends to users: people on social platforms, products on any e-commerce platform, the next video on Youtube or the next movie on Netflix.

We are talking about personalized recommendations when every user receives a different list of recommendations depending on their tastes, inferred based on previous interactions with items or information collected by different techniques [31].

Although the main purpose of a recommender systems is to help companies to sell more items, they also have a lot of advantages from the user's perspective. Recommendation engines help people find what they are looking for in a short amount of time increasing user's satisfaction. Consequently, the more often a user interacts with the system, the more refined the user's knowledge becomes improving the efficiency of the recommendation system. They also suggest less popular items helping them to be discovered diversifying the items sold to users.

The way in which the information about users and items is modeled and exploited depends on the particular recommendation technique. According to this aspect and the learning algorithm used to forecast user interests and provide predictions, different types of recommender systems can be implemented.

The two main types of recommender systems are *content-based* and *collaborative filtering*. Content-based recommendation engine uses item and users's profiles to find similar items in terms of content to the ones that the user liked in the past. Collaborative filtering recommendation engine instead uses user-item interaction history to provide users recommendations: the simple idea is that if two users had the same interests in the past, they will have the same behaviour in the future [9].

Creating a successful recommender system requires understanding the complex relationships between the entities involved. Graph databases can be instrumental in this process, providing a powerful tool for capturing and analyzing the relationships between users and items dynamically, also uncovering the underlying relationships that might not be immediately apparent, such as pathways between nodes [32]. We want to create a movie real-time recommender system using Neo4j and its Graph Data Science ecosystem exploring the two primary recommendation techniques explained before and subsequently a their hybrid recommendation version.

### 5.2.1 Content-based recommendations

As mentioned above, the basic process of produced content-based recommendations relies on item and users representations to suggest items similar to those a target user liked in the past.

We want to demonstrate how a graph model can be used to represent the item and user profiles and how a graph analysis can simplify the recommendation phase.
To build our recommendation system we use the MovieLens dataset [2], a standard dataset for recommendation engines containing user ratings of the movies they watched, used in combination with data available from the Internet Movie Database (IMDb) [3] such as genres, actors, writers and directors related to films.
We summarize the characteristics and network properties of the MovieLens dataset in Figure 5.7.

| Property | Value |
|---|---|
| Number of Movies | 9742 |
| Number of Users | 610 |
| Number of Reviews (Edges) | 100836 |
| Total Nodes | 10352 |
| Size of Largest Connected Component (percentage of total nodes) | 100% |
| Edge Set Size of Largest CC (percentage of total edges) | 100% |
| Average Degree (number of ratings) of User | 149,0372578 |
| Average Degree (number of ratings) of Movie | 11,03066402 |
| Graph BiPartite density | 2% |

Figure 5.7: Network information of the MovieLens dataset

We have only two entities, Movie and User in our dataset that we model as nodes labeled respectively *Movie* and *User*. The movie features extracted from IMDb can be translated into node properties that describe and characterize each movie. In this case each node *Movie* has as properties *movieId*, *title*, *actor* list, *writer* list, *director* and *genre* list while each node *User* has *userId* as node property. To model the ratings users assigned to movies in order to represent users preferences explicitly, we can connect user nodes to the movies through the RATED relationship where the assigned *rating* is stored as a relationship property. The resulting graph model will look like Figure 5.8.
This simple model for the *Movie* node has the advantage of a one-to-one mapping between the node and the item it represents with all the properties, but it has multiple drawbacks as data duplication because for example the director name is duplicated in all movies with the same director and the same for genres, actors and so on. Moreover this model limits the efficient navigation of relationships and nodes typical of graphs because the search is based on value or string comparison.
A more advanced model for representing movies models recurring properties as nodes: new nodes appear in this model labeled as *Actor*, *Director*, *Writer* and *Genre* nodes, where each node has some properties specific to the node such as *name* for directors and

[2]The data is available at `https://grouplens.org/datasets/movielens/`

[3]The data is available at `https://www.kaggle.com/datasets/tmdb/tmdb-movie-metadata`

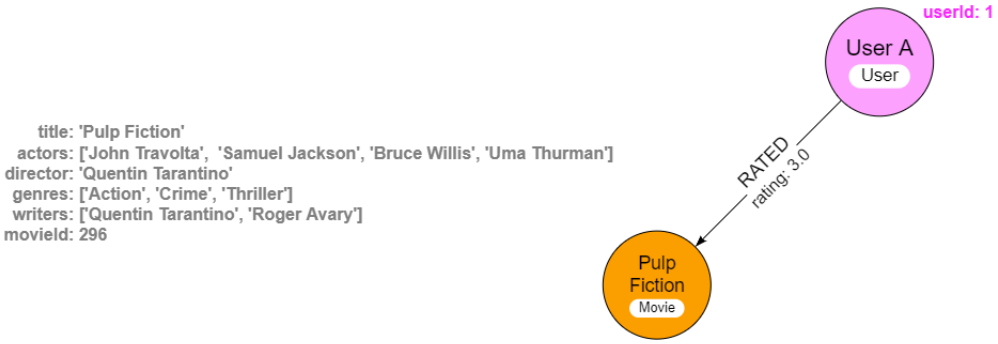



Figure 5.8: Basic graph model

writers and *genre* for movie genres. Now each node *Movie* has only the *movieId* and *title* as properties. The relationships among the node representing the movie's feature with the movie itself are expressed by edges in the graph: HAS relationship to link movies with the respective genres, DIRECTED to link the director with his movie, WROTE to link the writers of a movie to the movie itself and ACTS-IN to connect actors to the movies they starred in. The same node can play different roles and this can be modeled with different labels, different relationships or both. This approach prevents data duplication avoiding to represent the same concept in multiple nodes and it allows multiple and more efficient access patterns to the data. The resulting graph model will look like Figure 5.9. Different approaches can be used to provide content-based recommendations, depending on the information available and the models defined for both users and items. One of the most common and powerful techniques to CBRSs known as the *similarity-based retrieval* can be described as "Recommend items that are similar to those the user liked in the past" [31]. This approach requires the following steps:

1. User preferences data (RATED relationships in Figure 5.9)
2. Item features data (HAS, DIRECTED and ACTS-IN relationships in Figure 5.9)
3. A common representation for items so that the similarity among them is measurable (NODE SIMILARITY algorithm as described in 4.4.2)
4. Similarity function that, given two item representations computes the similarity between them (SIMILARITY FUNCTIONS as described in 4.4.3)
5. Computed similarities are stored in the graph as relationships between items, storing only the k topmost similar items or defining a minimum similarity threshold
6. Making the recommendations in order to predict those not-yet-rated items that could be of interest to a user.

After creating the graph property model like in Figure 5.9 implementing the previous 1 and 2 points, we want to use the Node similarity algorithm described in 4.4.2: this graph algorithm requires a bipartite graph projection as input parameter in order to select the relationships of interest on which to calculate the similarity among movies.

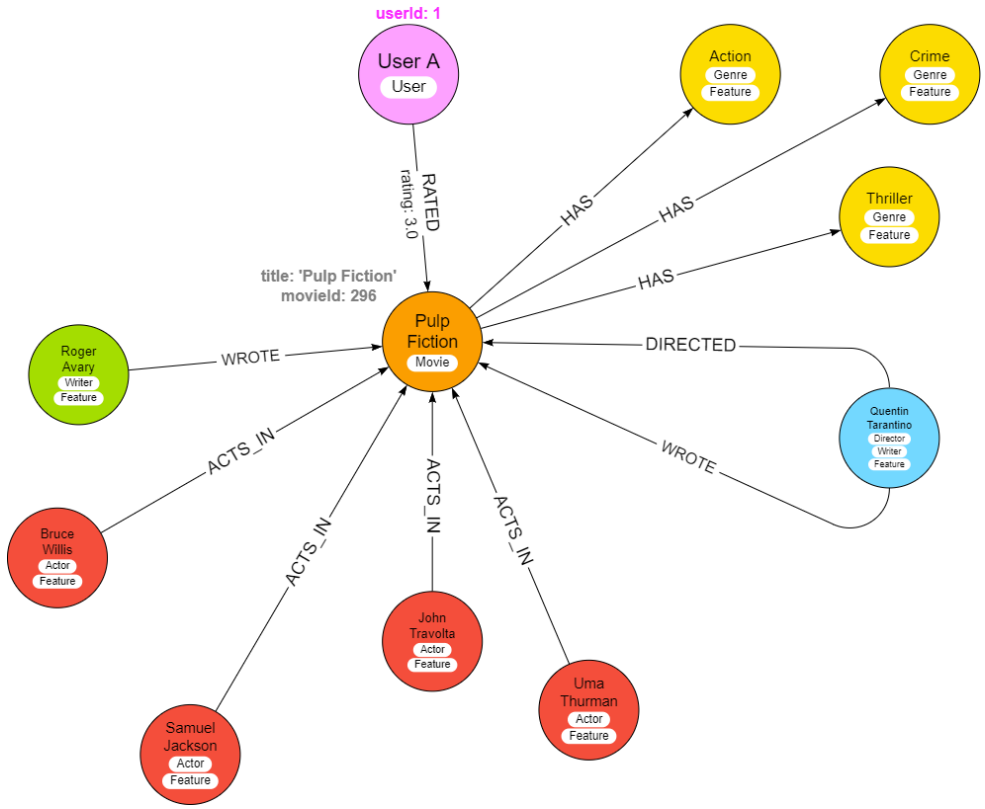



Figure 5.9: Advanced graph model

We want to compare movies based on their content i.e. genre, director and actor information.

```
#Create Graph Projection
CALL gds.graph.project(
    'contentGraph',
    ['Feature', 'Movie'],
    {
        HAS:
            {orientation: 'NATURAL'},
        DIRECTED:
            {orientation: 'REVERSE'},
        ACTS_IN:
            {orientation: 'REVERSE'}
    }
);
```

Because of Node similarity is applied to a bipartite graph, we have to add a generic label *Feature* to *Genre*, *Actor* and *Director* nodes so that *contentGraph* contains only two node sets, *Feature* and *Movie* nodes, with all the HAS, ACTS-IN and DIRECTED relationships that involve them: we want to compare movies based on their outgoing links using the Jaccard similarity function for unweighted relationships.

```
#Get Movie Similarity
```

```
CALL gds.nodeSimilarity.stream('contentGraph',{similarityMetric: 'JACCARD', topK:
    20})
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).title AS Movie1, gds.util.asNode(node2).title AS
    Movie2, similarity
ORDER BY similarity, Movie1, Movie2
```

Node Similarity computes pair-wise similarities based on the Jaccard similarity score having as $topK = 20$ as configuration parameter to limit on the number of scores calculated per *Movie* node: only the 20 largest similarity scores are returned for each movie.

```
#Store SIMILAR relationships
CALL gds.nodeSimilarity.write('contentGraph', {
    writeRelationshipType: 'SIMILAR',
    writeProperty: 'score',
    topK: 20,
    similarityCutoff: 0.2,
    similarityMetric: 'JACCARD'
})
YIELD nodesCompared, relationshipsWritten
```

| Movie1 | Movie2 | Similarity |
|---|---|---|
| Toy Story 2 (1999) | Toy Story (1995) | 0.375 |
| Toy Story 3 (2010) | Toy Story (1995) | 0.2 |
| Tigger Movie, The (2000) | Pocahontas (1995) | 0.211 |
| Pocahontas II: Journey to a New World (1998) | Pocahontas (1995) | 0.276 |
| Return of Jafar, The (1994) | Pocahontas (1995) | 0.227 |
| Indie Game: The Movie (2012) | Heidi Fleiss: Hollywood Madam (1995) | 1.0 |
| Manufactured Landscapes (2006) | Heidi Fleiss: Hollywood Madam (1995) | 0.333 |
| Muppet Christmas Carol, The (1992) | Muppet Treasure Island (1996) | 0.363 |
| My Voyage to Italy (Il mio viaggio in Italia) (1999) | Catwalk (1996) | 0.5 |
| She's the One (1996) | Brothers McMullen, The (1995) | 0.226 |

Figure 5.10: Movie pair-wise similarities based on the Jaccard score

Next, we want to store these movie pair-wise similarities as new relationships between movies with the similarity score adding as propriety of SIMILAR relationships: we also use a minimum similarity threshold in Node Similarity algorithm set to 0.2 to limit the similarity scores stored in the graph.
Storing SIMILAR relationships in the graph model is a fundamental step if our last goal is querying the resulting model made by the RATED and SIMILAR relationships to perform personalized recommendations.
One of the most accurate approaches to predict the interest of a user in a specific item consists of considering the sum of all the similarities of the target item $p$ to the other

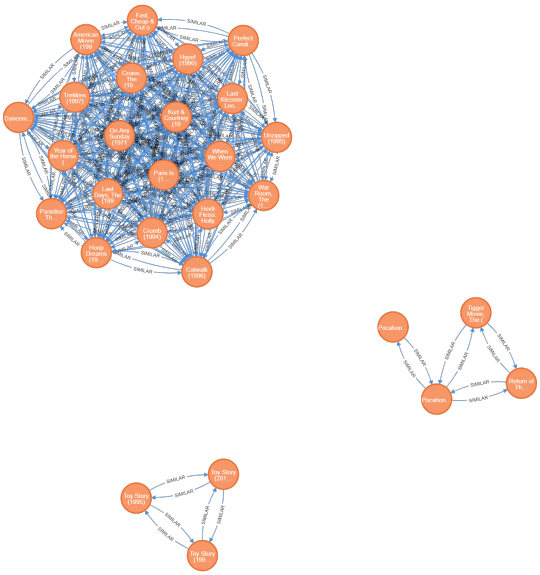


Figure 5.11: A subgraph of the similarity network between movies.

items the user interacted with before [9]:

$$interest(u, p) = \sum_{i \in Items(u)} sim(i, p) \tag{5.5}$$

where $Items(u)$ returns all the items the users has interacted with.

```
#Make recommandations for a user
MATCH (user:User)
WHERE user.userId = $userId$
WITH user
MATCH (targetMovie:Movie)
WHERE NOT EXISTS((user)-[]->(targetMovie))
WITH targetMovie, user
MATCH (user:User)-[]->(movie:Movie)-[r:SIMILAR]->(targetMovie)
RETURN targetMovie.title as Movie, sum(r.score)/count(r) as Relevance
order by Relevance desc
LIMIT 10
```

where the parameter $userId$ in this case is equal to 1. The computed relevance score can be used to rank all the not-yet-seen movies and return the top 10 movies to the specified user as recommendations like in Figure 5.12.

### 5.2.2 Collaborative filtering recommendations

Content approach is applicable only when content description related to each item is somehow available but this information might not be readily accessible, easy to collect or relevant.
An alternative to content methods is the well-known collaborative filtering recommendation engine, relies only on past user history, item ratings for example without requiring the creation of items and users profiles. It can be applied to a vast variety of scenarios

| Movie | Relevance |
|---|---|
| 10th Kingdom, The (2000) | 1.0 |
| The Man Who Killed Don Quixote (2018) | 1.0 |
| Dinotopia (2002) | 0.666 |
| 12 Chairs (1976) | 0.666 |
| Ivan Vasilievich: Back to the Future (1973) | 0.666 |
| 12 Chairs (1971) | 0.666 |
| Ant-Man and the Wasp (2018) | 0.6 |
| Gulliver's Travels (1996) | 0.5 |
| Earthsea (Legend of Earthsea) (2004) | 0.5 |
| Long Live Ghosts! (1977) | 0.5 |

Figure 5.12: Top 10 recommended movies for user 1

because analyzes relationships between users and items to predict new user-item associations. Generally, collaborative filtering is more accurate than content-based techniques but it suffers from what it is called the *cold-start problem* when the ratings matrix is sparse: this means it fails to produce reasonable recommendations in terms of accuracy for new items and new users when relatively little information is available.
Collaborative filtering techniques are generally classified into two main areas:

- *Memory-based*. These techniques, also referred to as *neighborhood methods*, attempt to find a set of users or items that have historically been similar to each other in the past (either a group of users that rated the same products the same way or a group of items that were all rated the same way by the same users) in order to predict the specific user rating for a particular item.
- *Model-based*. These methods create models for users and items that describe their behaviour via a set of factors and the weight these factors have for each item and each user. Model-based approaches are particularly effective in understanding and representing user preferences recommending exactly what the user wants.

Model-based approach delivers the best result in terms of prediction accuracy but this measure alone does not guarantee users a satisfying experience [33] [34] [35].
Memory-based methods instead capture local associations in the data allowing to recommend for example a movie quite different from the user's typical preferences or a lesser-known movie, based on strong ratings from one of their closest neighbors. We focus on this class of collaborative filtering recommendation system because it is really a graph-based task with regard to local navigation of data during predictions, the explainability of the recommendation process increasing the user's trust in the system and its stability when new information is available: in this case only a small portion of the graph is recomputed. In the end, not least, another strong point of memory-based methods is their time efficiency to provide near-instantaneous recommendations.

There are two possible approaches to memory-based recommendation for collaborative filtering:

- *Item-based* where the similarities are computed between items based on the users who interact with them (rating, clicking and so on).
- *User-based* where the similarities are computed between users based on the list of items they interact with.

Neighborhood methods have three main components whether they are item-based or user-based:

- Some definition of similarity
- Some way of using similarity between homogeneous elements to construct neighborhoods
- Some way of using a neighborhood of a user or item to make predictions

Now we will explore the main steps to realize a graph-powered collaborative filtering RS in both cases, item-based and user-based. We will always adopt the same MovieLens dataset for our purpose, without any movie feature data because the input of the recommendation process in collaborative filtering is the user-item dataset in Figure 5.13. It is a weighted bipartite graph with *Movie* and *User* nodes linked by RATED relationships with *rating* as relationship weight.
The procedure for the similarity computation is the same as that for the content-based approach but it is important to note that, with the neighborhood methods, the similarities are computed using only the interactions between movies and users.
The simple idea in the user-based approach is to find similar users to our user i.e. users that rated the same movies similarly and find other movies that the target user has not seen yet.
Firstly we create a native graph projection in Neo4j of the User-Item dataset: if we want to compute similarities between users, we must consider the RATED relationships in their *NATURAL* orientation with their respective property, *rating* to measure similarity among users. Note that we select here a different similarity function, the Cosine similarity score because our graph projection *userGraph* is weighted.

```
#Create Graph Projection for Used-based CF
CALL gds.graph.project(
    'userGraph',
    ['User', 'Movie'],
    {
       RATED:
            {
             orientation: 'NATURAL',
             properties:  'rating'
            }

    }
);
```

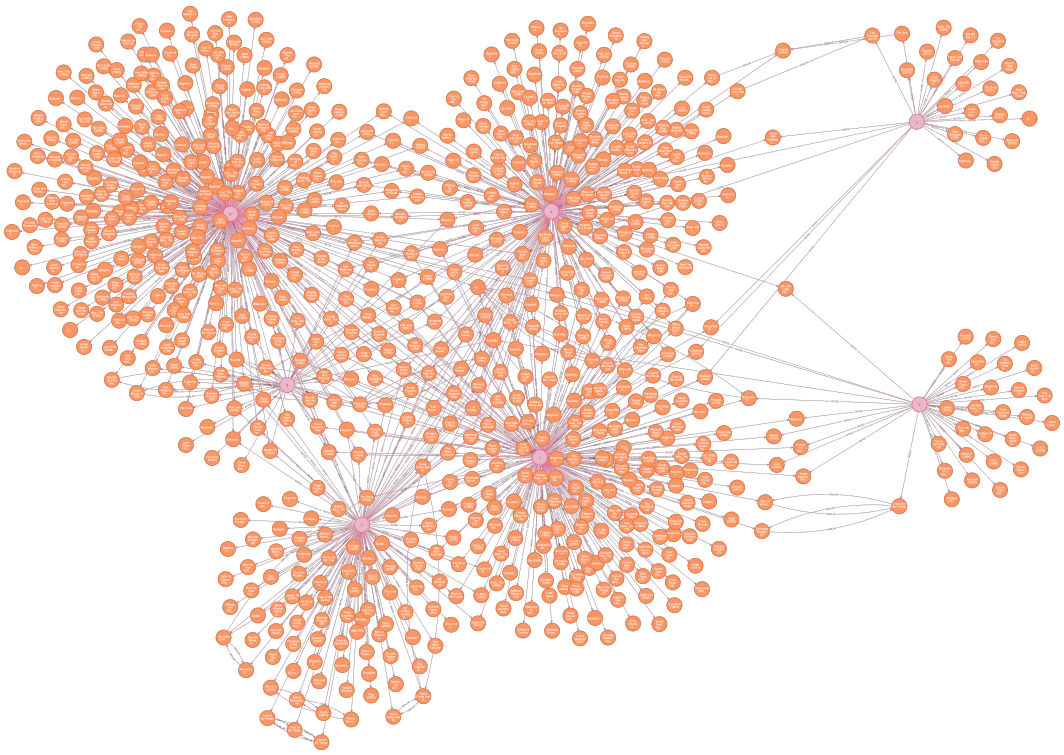


Figure 5.13: A subgraph of the bipartite network consisting of users (pink) and movies (orange)

Then, we calculate the Cosine similarity of each user with all the other users, keeping only the $topK = 20$ similarity scores for each user and storing in the original graph as new relationships between users, the SIMILAR-U relationships, only those with a similarity score greater than 0.2.

```
#Get User Similarity
CALL gds.nodeSimilarity.stream('userGraph',{similarityMetric: 'COSINE', topK: 20})
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).userId AS User1, gds.util.asNode(node2).userId AS
    User2, similarity
ORDER BY similarity, User1, User2
```

```
#Store SIMILAR_U relationships
CALL gds.nodeSimilarity.write('userGraph', {
    writeRelationshipType: 'SIMILAR_U',
    writeProperty: 'score',
    topK: 20,
    similarityCutoff: 0.2,
    similarityMetric: 'COSINE'
})
YIELD nodesCompared, relationshipsWritten
```

The number of *nodesCompared* is 610 with 9471 *relationshipsWritten* after 367 ms.

| User1 | User2 | Similarity |
|---|---|---|
| 130 | 145 | 0.828 |
| 130 | 574 | 0.828 |
| 130 | 468 | 0.822 |
| 126 | 379 | 0.811 |
| 242 | 468 | 0.794 |
| 150 | 270 | 0.775 |
| 126 | 130 | 0.766 |
| 126 | 81 | 0.764 |
| 347 | 94 | 0.757 |
| 46 | 468 | 0.752 |

Figure 5.14: User pair-wise similarities based on the Cosine score.

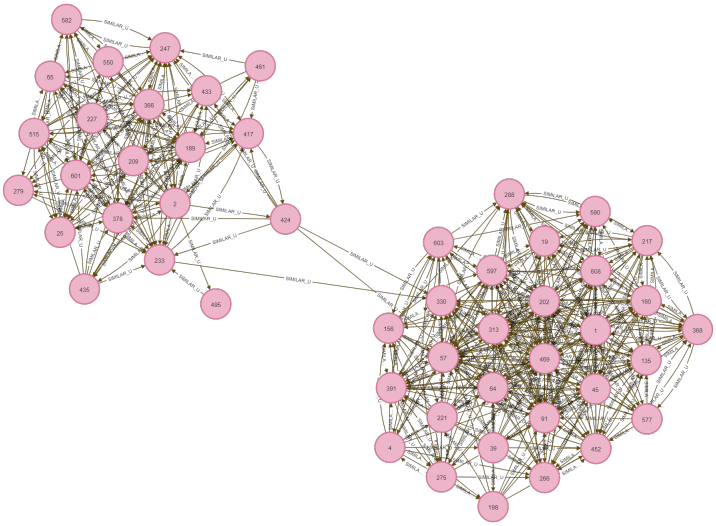


Figure 5.15: A subgraph of the similarity network between users.

To recommend movies for a user $a$, we compute a rank for movies the target user has not rated using a weighted average rating of movies other similar users have seen. To compute this weighted average rating we can adopt the following formula [9]:

$$pred(a,p) = \frac{\sum_{b \in KNN(a)} sim(a,b) \times r_{b,p}}{\sum_{b \in KNN(a)} sim(a,b)} \tag{5.6}$$

where $KNN(a)$ represents the $k$-nearest neighbors of the user $a$.

```
#Make recommandations for a user
MATCH (u1:User)-[s:SIMILAR_U]-(u2)-[r:RATED]-(m:Movie)
WHERE u1.userId = $userId$
```

```
AND NOT ( (u1)-[]-(m))
RETURN m.Title,sum(s.score*r.rating)/sum(s.score) as relevance
ORDER BY relevance DESC
LIMIT 10
```

| Movie | Relevance |
|---|---|
| Hoop Dreams (1994) | 5.0 |
| Miracle on 34th Street (1947) | 5.0 |
| Twin Peaks: Fire Walk with Me (1992) | 5.0 |
| Ghost Dog: The Way of the Samurai (1999) | 5.0 |
| Hard Core Logo (1996) | 5.0 |
| Drugstore Cowboy (1989) | 5.0 |
| Man Bites Dog (C'est arriv pr s de chez vous) (1992) | 5.0 |
| Glengarry Glen Ross (1992) | 5.0 |
| Way of the Dragon, The (a.k.a. Return of the Dragon) (Meng long guo jiang) (1972) | 5.0 |
| Trees Lounge (1996) | 5.0 |

Figure 5.16: Top 10 recommended movies for user 1 with *user-based* collaborative filtering approach

Although user-based approach have been applied successfully in different domains, in large e-commerce sites where it is necessary to handle more users than items, user-based filtering is not so faster to scan a vast number of potential neighbors. Large e-commerce sites for example Amazon use alternative techniques as item-based recommendations. It is actually a more stable approach because the average rating received by an item does not change as quickly as the average rating given by a user to different items. It is also known to perform better than the user-based approach when the ratings matrix is sparse. The main idea of the item-based approach for collaborative filtering is to compute predictions by using the similarity between items, not users, using only the interactions between movies and users.
Let us start with the native graph projection as in the user-based recommendations.

```
#Create Graph Projection for Item-based CF
CALL gds.graph.project(
    'itemGraph',
    ['User', 'Movie'],
    {
       RATED:
            {
             orientation: 'REVERSE',
             properties:  'rating'
            }

    }
);
```

We continue with the similarity computations among movies using the Cosine similarity score as before and storing them as SIMILAR-I relationships between movies in the original graph if the similarity score is greater than 0.2.

```
#Get Item Similarity
CALL gds.nodeSimilarity.stream('itemGraph',{similarityMetric: 'COSINE', topK: 20})
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).movieId AS Movie1, gds.util.asNode(node2).movieId AS
     Movie2, similarity
ORDER BY similarity, Movie1, Movie2
```

```
#Store SIMILAR_I relationships
CALL gds.nodeSimilarity.write('itemGraph', {
    writeRelationshipType: 'SIMILAR_I',
    writeProperty: 'score',
    topK: 20,
    similarityCutoff: 0.2,
    similarityMetric: 'COSINE'
})
YIELD nodesCompared, relationshipsWritten
```

The number of *nodesCompared* is 9724 with 193563 *relationshipsWritten* after 4509 ms.

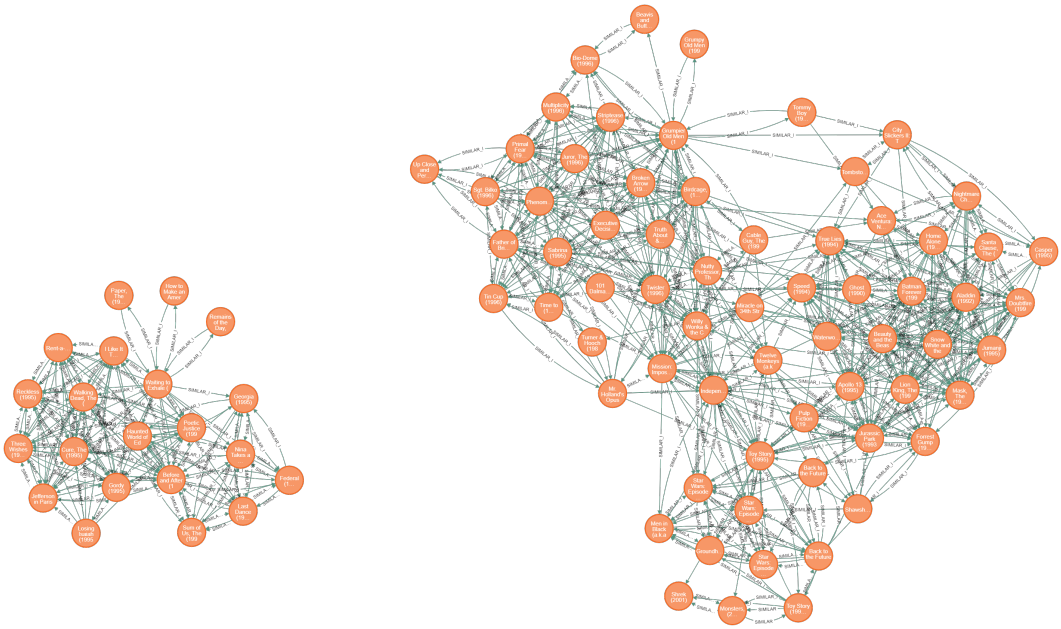


Figure 5.17: A subgraph of the similarity network between movies.

To recommend movies for a user $a$, we use the same query adopted in the content-based approach but with a different formula to compute the relevance score for each movie on which the ranking is formulated. This formula to predict the rating for a not-yet-seen movie in the dataset is [36]:

$$pred(a,p) = \frac{\sum_{q \in ratedItem(a)} sim(p,q) \times r_{a,q} \times |KNN(q) \cap \{p\}|}{\sum_{q \in ratedItem(a)} sim(p,q) \times |KNN(q) \cap \{p\}|} \tag{5.7}$$

where $ratedItem(a)$ considers all the movies rated by user $a$ and the term $|KNN(q) \cap \{p\}|$ is 1 if $p$ belongs to the set of nearest neighbors of $q$ and 0 otherwise. The denominator normalizes the value to not exceed the max value of the rating.

| Movie1 | Movie2 | Similarity |
|---|---|---|
| My Blueberry Nights (2007) | 127 Hours (2010) | 0.236 |
| MacArthur (1977) | Dirty Rotten Scoundrels (1988) | 0.25 |
| No Direction Home: Bob Dylan (2005) | Rudolph, the Red-Nosed Reindeer (1964) | 0.289 |
| 21 Up (1977) | Seven Up! (1964) | 1.0 |
| Affliction (1997) | Days of Heaven (1978) | 1.0 |
| Amityville 1992: It's About Time (1992) | Amityville: Dollhouse (1996) | 1.0 |
| 2048: Nowhere to Run (2017) | Sleepwalkers (1992) | 1.0 |
| 3 Ninjas Knuckle Up (1995) | Baby, The (1973) | 1.0 |
| 31 (2016) | Dead Heat (1988) | 1.0 |
| 6 Days to Air: The Making of South Park (2011) | Beowulf & Grendel (2005) | 1.0 |

Figure 5.18: Movie pair-wise similarities based on the Cosine score.

```
#Make recommandations for a user
MATCH (user:User)
WHERE user.userId = $userId$
WITH user
MATCH (targetMovie:Movie)
WHERE NOT EXISTS((user)-[]->(targetMovie))
WITH targetMovie, user
MATCH (user:User)-[r:RATED]->(movie:Movie)-[s:SIMILAR_I]->(targetMovie)
RETURN targetMovie.title as Movie, sum(s.score*r.rating)/sum(s.score) as Relevance
ORDER BY Relevance desc
LIMIT 10
```

| Movie | Relevance |
|---|---|
| Breaking Away (1979) | 5.0 |
| Atlantic City (1980) | 5.0 |
| 101 Dalmatians (One Hundred and One Dalmatians) (1961) | 5.0 |
| Born on the Fourth of July (1989) | 5.0 |
| Chariots of Fire (1981) | 5.0 |
| Five Easy Pieces (1970) | 5.0 |
| Sound of Thunder, A (2005) | 5.0 |
| True Romance (1993) | 5.0 |
| Drop Dead Fred (1991) | 5.0 |
| This Is Spinal Tap (1984) | 5.0 |

Figure 5.19: Top 10 recommended movies for user 1 with *item-based* collaborative filtering approach

### 5.2.3 A hybrid recommender system

The various recommendation approaches exploit different input and paradigms to make recommendations, each one with its pros and cons, as we previously analyzed. Building a *hybrid* recommendation engine means to combine the strengths of different models to overcome some of the problems previously mentioned as the cold-start problem for a collaborative filtering or the great amount of information needed for a content recommendation system. Among the hybridization strategies, we will focus on the *parallelized* approach: it requires at least two separate recommendation systems operate independently of one another and produce distinct recommendation lists as outputs, subsequently joined into a final set of recommendations according to the hybridization strategy [37].
We want to hybridize in a parallelized mode our two recommender systems, the content-based and the collaborative filtering: the content approach reduces the cold-start problem that occurs in the case of a new user or a new item, whereas the collaborative filtering approach work more accurately without metadata about items. We adopt the same graph model mixing multiple recommendation models in the same graph where we have user similarities based on a collaborative filtering approach SIMILAR-U, item similarities based on content approach SIMILAR and finally item similarities based on content-based approach SIMILAR-I. Now that we have stored the models in the graph, we can

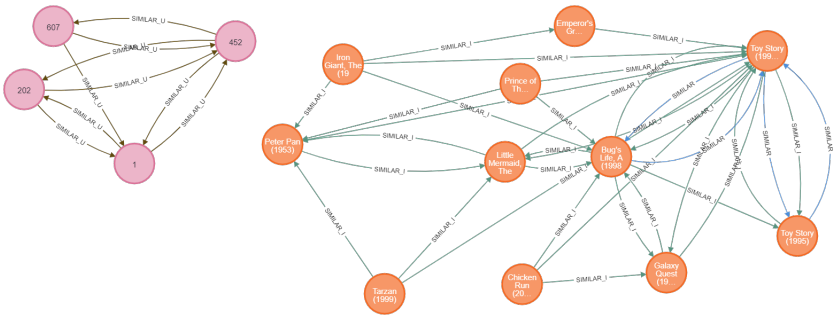



Figure 5.20: Mixing multiple recommendation models in the same graph

adopt a *weighted* hybridization strategy to provide recommendations to the target user. A *weighted* hybridization strategy combines the outputs of two or more recommender systems by computing weighted sums of their scores using the following formula:

$$score_{weighted}(u,i) = \sum_{k=1}^{n} \beta_k \times score_k(u,i) \tag{5.8}$$

where $n$ is the number of recommenders and the sum of all $\beta_k$ must be 1. It is worth noting that the value of $\beta_k$ can be dynamic, changing over the life of the recommendation system assigning for example a higher value of $\beta_k$ to the content-based system until the system has acquired enough data for the collaborative filtering approach.
In our case we want to realize a simple hybrid recommender system where $n = 2$ and $\beta_1$ is the weight for the content-based approach, whereas $\beta_2$ is the weight for the collaborative filtering approach. For example we could hypothesize to have enough data for the collaborative filtering approach so we set $\beta_1 = 0.2$ and $\beta_2 = 0.8$.

```
#Make recommandations for a user
MATCH (user:User)
WHERE user.userId = $userId$
WITH user
MATCH (targetMovie:Movie)
WHERE NOT EXISTS((user)-[]-(targetMovie))
WITH targetMovie, user
MATCH (user)-[]->(movie:Movie)-[r:SIMILAR]->(targetMovie)
WITH targetMovie, sum(r.score)/count(r) AS Score1
MATCH (user)-[r:RATED]->(movie:Movie)-[t:SIMILAR_I]->(targetMovie)
WITH targetMovie,Score1, sum(t.score*r.rating)/sum(t.score) as Score2
RETURN targetMovie.title, 0.2*Score1+0.8*Score2 as Relevance
ORDER  BY Relevance DESC
LIMIT 10
```

| Movie | Relevance |
|---|---|
| Throne of Blood (Kumonosu j ) (1957) | 3.430 |
| Crumb (1994) | 3.289 |
| Hidden Fortress, The (Kakushi-toride no san-akunin) (1958) | 3.274 |
| Good, the Bad and the Ugly, The (Buono, il brutto, il cattivo, Il) (1966) | 3.227 |
| Sanjuro (Tsubaki Sanj r ) (1962) | 3.185 |
| Solaris (Solyaris) (1972) | 3.183 |
| Monty Python's And Now for Something Completely Different (1971) | 3.176 |
| Animatrix, The (2003) | 3.175 |
| Trekkies (1997) | 2.941 |
| Saragossa Manuscript, The (Rekopis znaleziony w Saragossie) (1965) | 2.724 |

Figure 5.21: Top 10 recommended movies for user 1 with a hybrid approach, completed after 15 ms.

## 5.3 Antifraud Models For Credit Card Transaction Dataset

Fighting fraud and more generally detecting anomalies in data, is a crucial task in multiple areas such as finance, security and healthcare. While in a recommendation machine learning task the targets are the end users, in the fraud-fighting use case, the real stakeholders are the company's analysts, not users: they must adopt all the necessary measures to prevent and detect frauds.
We want to adopt a simulated credit card transaction dataset to train a classical machine learning model and then import all data in a graph model for exploring the benefits of a graph approach.
Common references on the sections of this chapter are in [38], [39], [40], [41] and [42].

### 5.3.1 Classic Antifraud Model

The dataset[4] includes transactions from the duration 1st Jan 2019 - 31st Dec 2020, comprising both genuine and fraudulent transactions. It covers transactions made by 1000 customers using credit cards across 800 merchants with 1296675 rows and 23 columns. Each row represents a certain transaction between a customer and a merchant with the following features:

- **trans-date-trans-time**: transaction time stamp
- **cc-num**: credit card number of customer
- **merchant**: merchant name
- **category**: category of merchant
- **amt**: transaction amount
- **first**: first name of credit card holder
- **last**: last name of credit card holder
- **gender**: gender of credit card holder
- **street**: street address of credit card holder
- **city**: city of credit card holder
- **state**: state of credit card holder
- **zip**: zip of credit card holder
- **lat**: latitude location of credit card holder
- **long**: longitude location of credit card holder

[4]The data is available at `https://www.kaggle.com/datasets/kartik2112/fraud-detection`

- **city-pop**: city population of credit card holder
- **job**: job of credit card holder
- **dob**: date of birth of credit card holder
- **trans-num**: transaction number
- **unix-time**: UNIX time of transaction
- **merch-lat**: latitude Location of merchant
- **merch-long**: longitude location of merchant
- **is-fraud**: nature of transaction (fraudulent or not fraudulent)

We have 1289169 genuine transactions and 7506 fraudulent transactions. The dataset has the *trans-date-trans-time* feature, so we can split it in a train and a test set using a temporal window.

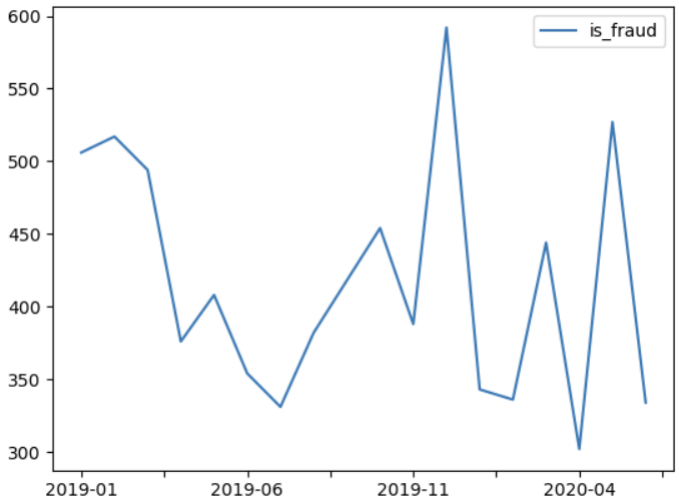



Figure 5.22: Number of frauds according to the month.

We check if the proportions between genuine and fraudulent transactions are respected in the training and test sets.

```
#% fraud vs % genuine transactions in df_train
0    0.994294
1    0.005706
```

```
#% fraud vs % genuine transactions in df_test
0    0.993482
1    0.006518
```

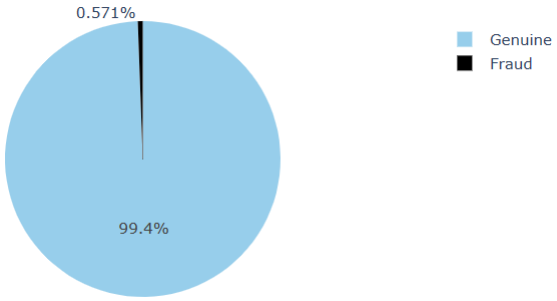



Figure 5.23: Fraudulent vs Genuine transactions in the training data

In particular we have 1157940 genuine transactions and 6645 fraudulent transactions in the training set, therefore this is a highly imbalanced dataset to be balanced in order to not produce any biases in the analysis.
Now we want to explore amount data, *amt* feature, because it is an important feature in credit card fraud analysis.
From the above plot, we note that the *amt* feature has a lot of outliers: this means that the variance in the feature is huge. Also, the fraud amount distribution is very dissimilar from the non fraud amount distribution and we can observe that fraudulent transactions often happen in small amount, therefore the transactions in which small amount is involved can be monitored more closely in order to detect and prevent credit card fraud. So we need to rescale the *amt* feature with a robust scaler.
The *job* feature has 494 unique values which make it difficult for us to visualize and analyse the feature. So we want to select the 10 most frequent values i.e. jobs where high transaction frequencies have been noted. We can also observe that all of them have completely fraudulent transactions, so we are not able to detect some jobs "more fraudulent" than others.
We also analyze the *category* feature with only 14 distinct values and we can notice that about 3 categories have more involved in fraudulent transactions. These are *grocery-pos*, *misc-net* and *shopping-net* as we can notice in Figure 5.26.
Now that we are done with the exploratory data analysis, we will proceed with rescaling the *amt* feature and encoding the categorical features, *job* and *category* taking, for each of them, the 10 most frequent values and one-hot encode them. Some features like *cc-num*, *first*, *last* and *trans-num* are not significant in context of our analysis and hence can be removed, other features like *merchant*, *state*, *city* and *street* can be dropped since it has lot of unique values and it is hard to encode all of them.

```
#Feature extraction
rob_scaler = RobustScaler()
df_train['scaled_amt'] = rob_scaler.fit_transform(df_train['amt'].values.reshape
    (-1,1))
df_test['scaled_amt'] = rob_scaler.transform(df_test['amt'].values.reshape(-1,1))
features_not_to_encode = ["is_fraud", "scaled_amt", "lat", "long", "hour", "
    merch_lat", "merch_long"] #Numeric features
features_to_encode =  ["category", "job"] #Categorical features
features = features_not_to_encode + features_to_encode
df_train_to_encode = df_train[features].copy()
```

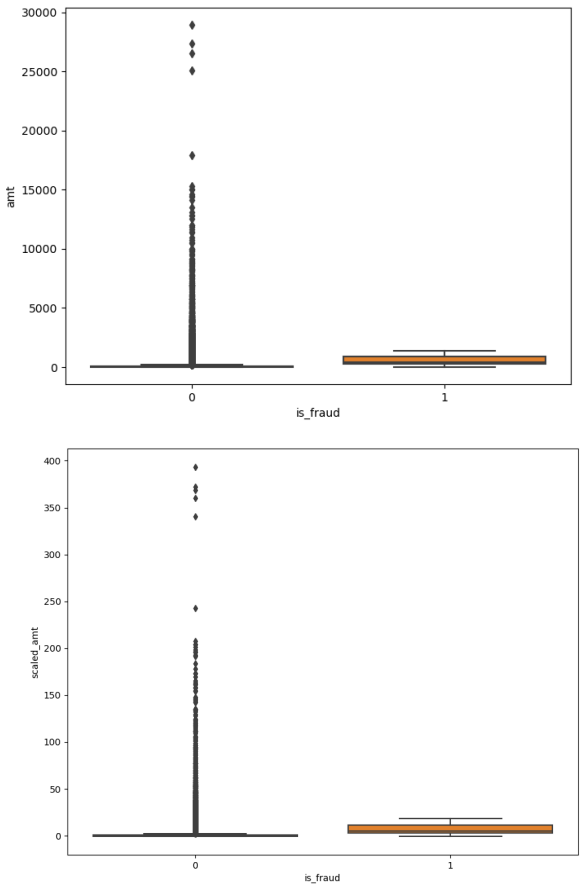



Figure 5.24: Transaction Amount vs Frauds: boxplots for *amt* and *scaled-amt* features

```
df_test_to_encode = df_test[features].copy()
```

```
#One-hot encoding for categorical features
def one_hot_encode_predict(data, categories_to_explode):
    encoded_dfs = []
    for column_name in categories_to_explode.keys():
        #Get the top most frequent values for each column

        #Create a new DataFrame with one-hot encoded columns
        encoded_df = pd.DataFrame()
        top_values = categories_to_explode[column_name]
        for value in top_values:
            encoded_df[column_name + '_' + str(value)] = (data[column_name] ==
            value).astype(int)

        encoded_dfs.append(encoded_df)
        data.drop(column_name, axis=1, inplace=True)

    encoded_data = pd.concat([data] + encoded_dfs, axis=1)
    return encoded_data
```

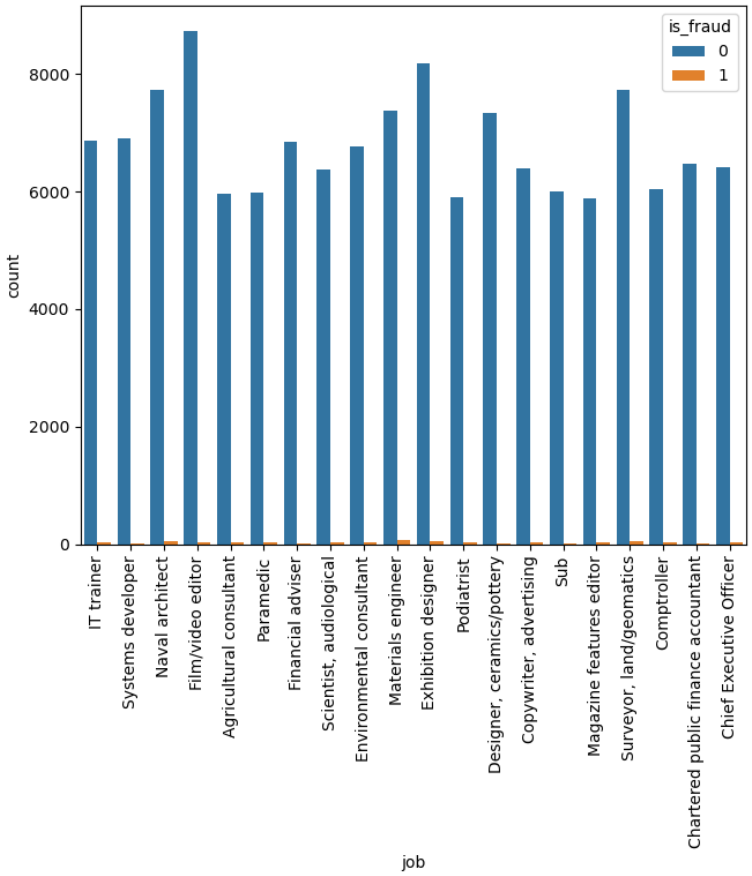



Figure 5.25: *Job* feature analysis.

As we have demonstrate, this dataset is highly imbalanced because the number of fraudulent transactions is only about 0.5% of the total dataset. This imbalance might create bias in our model building process predicting almost all data points as 'not fraud': we need to balance the classes in the target variable in order to build a fair model. For this training we will do a *stratified cross validation* with an *oversampler* as sampling technique implementing a *Random Forest Algorithm* with a hyper-parameter tuning.

After feature encoding step we proceed with a stratified cross validation. K-fold cross-validation is a resampling procedure used to evaluate machine learning models on a limited data sample based on a single parameter $k$ that refers to the number of groups that a given data sample is to be split into. *Stratification* is used in imbalanced problems to preserve the class frequencies in the individual folds to ensure that we are able to get a realistic picture of the model performance.

How to solve the problem of imbalance between classes? Using a resampling technique. Two main strategies for random resampling in imbalanced classification include Random Oversampling, which duplicates examples randomly in the minority class, and Random Undersampling, which deletes examples randomly from the majority class. We will choose the Random Oversampling technique because there is a problem of losing valuable data

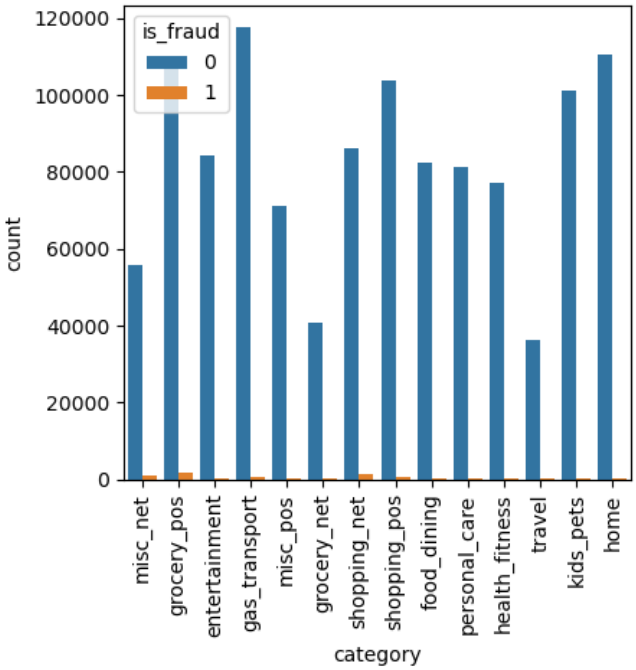



Figure 5.26: *Category* feature analysis

with the Random Under sampling method. Change to the class distribution should be only applied to the training dataset, because the intent is to influence the fit of the models, not to the test set used to evaluate their performances.
We will adopt an imbalanced-learn pipeline using the IMBLEARN package. The main purpose is to assemble several steps that can be cross-validated together while setting different parameters. During the cross-validation process we should split into training and validation segments. Then, on each segment, we should:

- Oversample the minority class.
- Train the Random Forest classifier on the training segment.
- Validate the classifier on the remaining segment.

To find the best parameters for our model from a given set of values in a grid, we also implement a *Grid Search CV* on a Random Forest classifier as hyper-parameter tuning.

```
#Implementing the imbalanced-learn pipeline
from imblearn.over_sampling import RandomOverSampler
from imblearn.pipeline import Pipeline, make_pipeline

imba_pipeline = make_pipeline(RandomOverSampler(random_state=42),
                              RandomForestClassifier(random_state=13))
params = {
        'n_estimators': [100, 200, 300],
        'max_depth': [4, 6, 8],
}
```

```
kf = StratifiedKFold(n_splits=4, random_state=None, shuffle=False)

new_params = {'randomforestclassifier__' + key: params[key] for key in params}
grid_imba = GridSearchCV(imba_pipeline, param_grid=new_params, cv=kf, scoring='f1'
    ,
                             return_train_score=True, n_jobs=-1)
grid_imba.fit(X_train, y_train)
print('Best parameters:', grid_imba.best_params_)
pipeline_over =  grid_imba.best_estimator_
```

The best parameters for our classifier are 8 as *max-depth* and 200 as *n-estimators*, so we will just use the best parameters found after the hyper-parameter tuning.

```
#Building our model with the best_params_
rc = RandomForestClassifier(random_state=42, n_estimators=200, max_depth= 8,
    n_jobs=-1)
pipeline_over = make_pipeline(RandomOverSampler(random_state=42), rc)
pipeline_over.fit(X_train, y_train)
y_predicted_val_prob = pipeline_over.predict_proba(X_test)[:, 1]
```

A classifier is only as good as the metric used to evaluate it. Standard metrics treat all classes as equally important, so using conventional metrics such as accuracy in imbalanced problems may lead to misleading conclusions because these metrics are insensitive to skewed distributions. Imbalanced classification problems typically rate classification errors with the minority class as more important than those with the majority class. As such, they require performance metrics focused on the minority class, the class where the observations needed to train an effective model are missing, making it particularly challenging.
Precision-Recall metrics can be useful for imbalanced problems, but which metric between precision and recall is most important here? In the case of credit card fraud detection, we want to avoid False Negatives as much as possible. A false negative case means that a fraudulent positive transaction is assessed to genuine transaction, which is detrimental. Recall is more important than Precision because in this use case we would like to have less False Negatives in trade off to have more False Positives.
The precision-recall curve shows the trade-off between precision and recall for different thresholds and we can clearly observe the point where we have to start trading a lot of precision for better recall score. For this scope we want to compare the classification reports of the model for two different thresholds, 0.5 and 0.57.

```
#Classification report and confusion matrix with a threshold greater or equal than
    0.5
              precision    recall  f1-score   support

           0       1.00      0.96      0.98    131229
           1       0.13      0.94      0.23       861

    accuracy                           0.96    132090
   macro avg       0.57      0.95      0.61    132090
weighted avg       0.99      0.96      0.97    132090

[[125999   5230]
 [    51    810]]
```

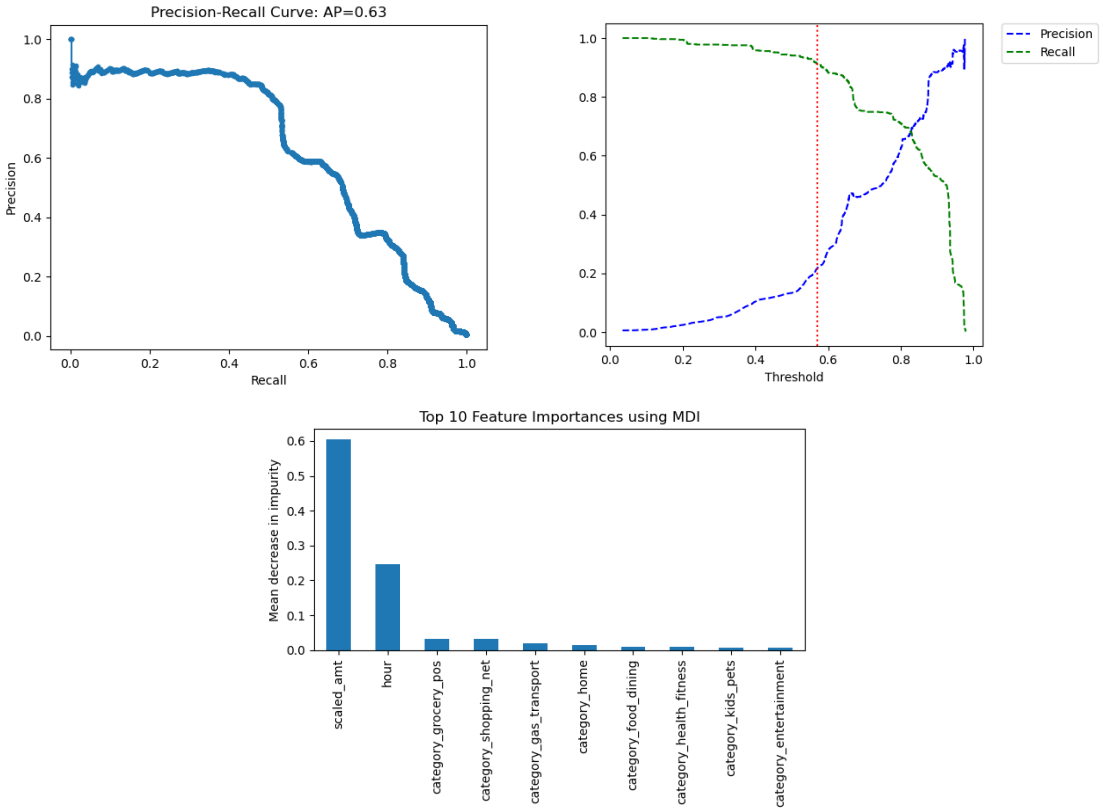



Figure 5.27: Precision-Recall curve with selecting threshold and Top 10 Feature Importance

```
#Classification report and confusion matrix with a threshold greater or equal than
    0.57
              precision    recall  f1-score   support

           0       1.00      0.98      0.99    131229
           1       0.22      0.91      0.35       861

    accuracy                           0.98    132090
   macro avg       0.61      0.95      0.67    132090
weighted avg       0.99      0.98      0.98    132090

[[128409   2820]
 [    75    786]]
```

So we select a threshold greater or equal than 0.57 with a recall score of 0.91 on the minority class and a number of false negative of 75 with an improvement in term of f1-score of 0.35. Average Precision (AP) plotted in Figure 5.27 summarizes such a plot by calculating the weighted mean of precisions achieved at each threshold, considering the increase in recall from the previous threshold.

### 5.3.2 Antifraud Model with Graph Features

Graphs might be relevant in a fraud detection use case because of the relational nature of this problem due to the data interdependence. The idea now is to import all data into a graph model, enriching it with new graph features directly extracted from the network to demonstrate how this type of approach is more efficient.
Firstly we aggregate the transactions on *cc-num* and *merchant* features, adding a new feature *num-trans* for each row so whenever we have the same pair of sender and receiver of the transaction, we aggregate the result based on the number of transactions, number of frauds and total exchanged amount. The resulting graph model has the following structure with 1667 nodes and 461785 relationships.

| | cc_num | merchant | is_fraud | amt | num_trans |
|---|---|---|---|---|---|
| 0 | 60416207185 | fraud_Abbott-Rogahn | 0 | 69.83 | 3 |
| 1 | 60416207185 | fraud_Abbott-Steuber | 0 | 159.44 | 4 |
| 2 | 60416207185 | fraud_Abernathy and Sons | 0 | 55.09 | 3 |
| 3 | 60416207185 | fraud_Abshire PLC | 0 | 156.15 | 2 |
| 4 | 60416207185 | fraud_Adams, Kovacek and Kuhlman | 0 | 155.39 | 2 |
| ... | ... | ... | ... | ... | ... |
| 461780 | 4992346398065154184 | fraud_Zemlak Group | 0 | 493.40 | 3 |
| 461781 | 4992346398065154184 | fraud_Zemlak, Tillman and Cremin | 0 | 130.46 | 4 |
| 461782 | 4992346398065154184 | fraud_Ziemann-Waters | 0 | 186.05 | 4 |
| 461783 | 4992346398065154184 | fraud_Zieme, Bode and Dooley | 0 | 373.75 | 7 |
| 461784 | 4992346398065154184 | fraud_Zulauf LLC | 0 | 265.91 | 1 |

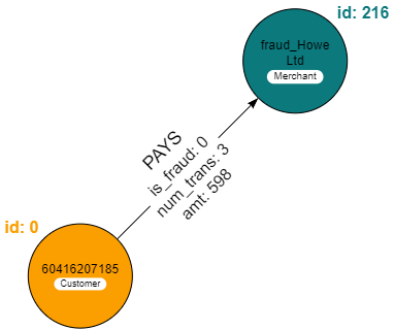


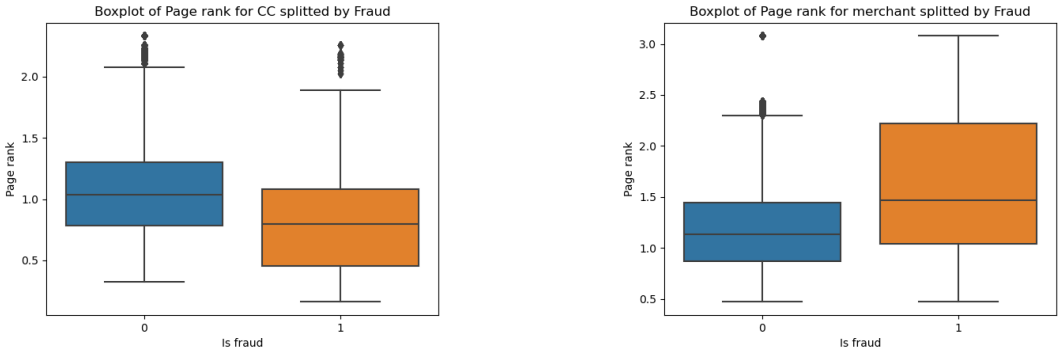

Figure 5.28: From aggregated tabular data to graph model

Because of we have people as nodes and relationships between people as edges, we could refer to this graph as a social network and adopt *social network analysis* or link analysis in general as a tool for improving the quality of fraud detection systems. The goal now is to determine which types of unstructured network information extracted from a network can be translated into meaningful characteristics of our entities. To do that we can use two types of approaches, *score-based* and *cluster-based*: while the former assign a score to each node analyzing the social network node by node, the second split the social network into communities of nodes considering the relationships.
We want to compute the pageRank centrality as scored-based metric using the pageRank algorithm mentioned in 4.3.4. It is important to consider the node not only individually but as part of a community of nodes because communities that behave the same way are likely to have a greater impact on the network than a single node. So we want to implement Louvain algorithm on our graph in order to store for each node the *communityId* property that contains the id of the community the node belongs to.

```
#Graph proiection
G, res = gds.graph.project(
  'undiGraph',
  ['Customer','Merchant'],
  {
    'PAYS': {
      'orientation': 'UNDIRECTED',
      'properties': ['total_amt']
    }
  }
```

```
)
#Pagerank computation
df_page = gds.pageRank.stream(gds.graph.get('undiGraph'), maxIterations=20,
          dampingFactor= 0.85, relationshipWeightProperty = 'total_amt')

#Louvain community detection execution
df_train_community = gds.louvain.stream(gds.graph.get('undiGraph'),
                     relationshipWeightProperty = 'total_amt')
```

The next step of our analysis is to add these graph features into our machine learning model distinguishing each connected feature according to the label. So we will insert four adding features *pr-cc*, *pr-merchant*, *community-cc* and *community-merchant*.




| communityId | size |
|---|---|
| 1331 | 1474 |
| 498 | 28 |
| 1105 | 23 |
| 627 | 22 |
| 231 | 20 |
| 1611 | 19 |
| 1553 | 16 |
| 1293 | 15 |
| 1415 | 11 |
| 1052 | 11 |
| 1233 | 10 |
| 275 | 9 |
| 1548 | 9 |

Figure 5.29: Graph features analysis

```
#Adding graph features
def add_graph_features(df, df_pr, df_communities):
    df_graph = df.merge(df_pr, left_on='cc_id', right_on='nodeId')
    df_graph = df_graph.rename(columns={"score": "pr_cc"}).drop("nodeId", axis=1)
    df_graph = df_graph.merge(df_pr, left_on='merc_id', right_on='nodeId')
    df_graph = df_graph.rename(columns={"score": "pr_merchant"}).drop("nodeId",
    axis=1)
```

```
    df_graph = df_graph.merge(df_communities, left_on='cc_id', right_on='nodeId')
    df_graph = df_graph.rename(columns={"communityId": "community_cc"}).drop(["
    nodeId","intermediateCommunityIds"], axis=1)
    df_graph = df_graph.merge(df_communities, left_on='merc_id', right_on='nodeId'
    )
    df_graph = df_graph.rename(columns={"communityId": "community_merchant"}).drop
    (["nodeId","intermediateCommunityIds"], axis=1)
    return df_graph

df_train_graph = add_graph_features(df_train_with_id_node_merch, df_page,
                 df_train_community)
df_test_graph = add_graph_features(df_test_with_id_node_merch, df_page,
                 df_train_community)
```

```
#Pre-processing training data
features_not_to_encode = ["is_fraud", "scaled_amt", "lat", "long", "hour", "
    merch_lat", "merch_long"]

graph_features = ["pr_cc", "pr_merchant", "community_cc", "community_merchant"]

features_to_encode = ["category", "job"]
features = features_not_to_encode + features_to_encode + graph_features

df_train_to_encode_graph = df_train_graph[features].copy()
df_test_to_encode_graph = df_test_graph[features].copy()

dict_cat_top_populated = one_hot_encode_top_populated_fit(df_train_to_encode_graph
    .copy(), features_to_encode, 10)

df_train_encoded_top_values = one_hot_encode_predict(df_train_to_encode_graph.copy
    (), dict_cat_top_populated)
df_test_encoded_top_values = one_hot_encode_predict(df_test_to_encode_graph.copy()
    , dict_cat_top_populated)
```

```
#Building our model with the best_params_ and graph features
rc = RandomForestClassifier(random_state=42, n_estimators=200, max_depth=8, n_jobs
    =-1)
pipeline_over = make_pipeline(RandomOverSampler(random_state=42), rc)
pipeline_over.fit(X_train, y_train)
y_predicted_val_prob = pipeline_over.predict_proba(X_test)[:, 1]
```

```
#Classification report and confusion matrix with a threshold greater or equal than
     0.57
              precision    recall  f1-score   support

           0       1.00      0.98      0.99    131229
           1       0.26      0.92      0.41       772

    accuracy                           0.98    132001
   macro avg       0.63      0.95      0.70    132001
weighted avg       1.00      0.98      0.99    132001

#Confusion matrix
[[129245   1984]
 [    63    709]]
```

We can see in 5.30 the introduction of graph features improve our results in terms of Average Precision with an improvement in terms of f1-score of 0.41.

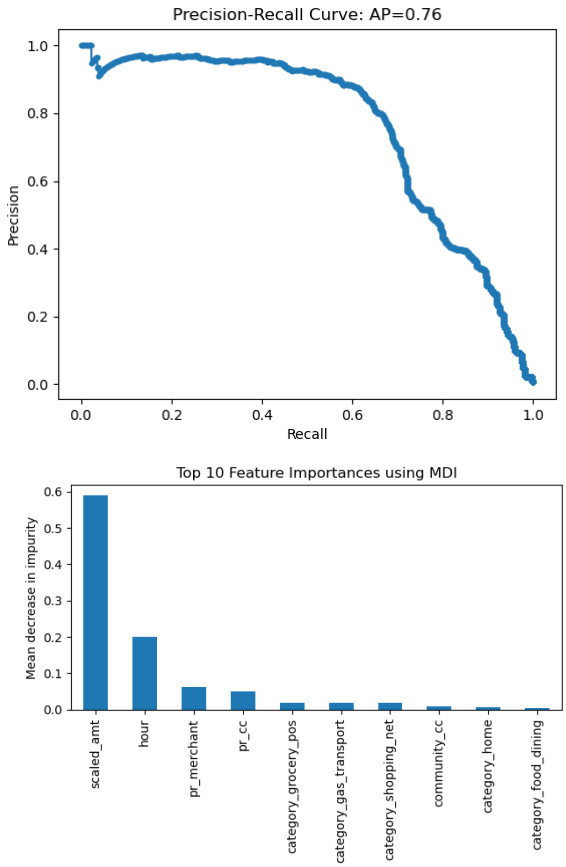



Figure 5.30: Precision-Recall curve with graph features and Top 10 Feature Importance

# Chapter 6

# Conclusion and future developments

We explored a graph-based approach for the three presented problems. For the link prediction problem, we identified specific graph proximity and clustering features based on historical social interactions to predict future friendships. The graph-powered machine learning model achieves an AUC score of around 0.97.
This model can be applied to any monopartite graphs where some relationships between nodes are known, aiming to accurately predict new ones. Examples include e-commerce product networks, citation networks, or biological networks.
For the recommendation system, we built a comprehensive graph-driven recommendation engine, employing various approaches from a content-based method to collaborative filtering, and finally, a hybrid approach. The key advantage is the flexibility offered by graph-based data representation and dynamic data management: independent models from each recommender can be stored collectively and accessed easily during the recommendation phase, facilitating real-time analysis.
For the anti-fraud detection model, we deployed two distinct approaches to address this imbalanced issue: a traditional model and a graph-based model incorporating centrality and cluster features. This enabled prediction of potentially suspicious connections among entities in a transaction network, achieving an average precision score of 0.76, compared to 0.63 obtained with the conventional model.
What are the next stages in this data science journey? Two emergent areas of Graph Data Science journey are Graph Embedding and Graph Neural Network (GNNs) [43]. Let us try to understand without going into technical details what embedded graph and graph neural networks are.
Richness of graph data could be a double-edge sword because firstly it expresses a wealth of information to process and for many ML tasks it is not so simple to identify features to be extracted for training a model. Secondly, classical ML techniques require matrices, not graphs as input format, hence the need to transform graph data to a compressed and tabular form.
Given an input graph, an embedding technique may be to find an encoding function capable of converting nodes and relationships in dimensional vectors after that classical

machine learning could be apply. Graph embedding captures the essence of a vertex's nature as a set of *latent features* in the sense that we cannot really describe them, transforming graph structure into a compact set of vertex vectors. This embedding technique allows to gather graph features without a process of feature extraction.
Generally deep learning approaches the problem of representation learning by introducing representations that are expressed in terms of other and simpler representations [44]. Graph Neural Networks, GNNs, combine the added insight from connected data with the modeling power of neural networks for prediction and classification tasks using graph structure during the training cycle not just as pre-processing step. More specifically, GNNs strenght consists in generating representations of nodes that depend on the topology of the graph as well as on any feature information about nodes and relationships we have, because we insert into the traditional neural network data flow an additional step called graph convolution, during which for each vertex we convolve/combine its features with the features of its neighbors.

# Bibliography


[1] D. Easley and J. Kleinberg, *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, 2010.

[2] "Social network analytics." https://medium.com/analytics-vidhya/social-network-analytics-f082f4e21b16.

[3] "London tube network." http://mng.bz/G6wN.

[4] "Economic networks: The new challenges." https://www.science.org/doi/10.1126/science.1173644.

[5] A. Hodler and M. Needham, *Graph Algorithms: Practical Examples in Apache Spark & Neo4j*. O'Reilly Media, Inc., 2019.

[6] *5-Graph-Data-Science-Basics-Everyone-Should-Know*.

[7] A. Hodler and M. Needham, *Graph Data Science (GDS) For Dummies, Neo4j Special Edition*. John Wiley & Sons, Inc., 2021.

[8] R. Wirth and J. Hipp, "CRISP-DM: Towards a Standard Process Model for Data Mining.," in *Proceedings of the Fourth International Conference on the Practical Application of Knowledge Discovery and Data Mining*, pp. 29–39, 2000.

[9] A. Negro and J. Webber, *Graph-Powered Machine Learning*. Manning Pubblications Co, 2021.

[10] J. Webber, "Not all graph databases are created equal: Why you need a native graph," in *Database Trends and Applications*, 2018.

[11] J. Stegeman, *Native vs. Non-Native Graph Database*, 2023.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algortihms*. The MIT Press, 2022.

[13] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*. O'Reilly Media, Inc., 2015.

[14] A. Vukotic, N. Watt, D. Fox, T. Abedrabbo, and J. Partner, *Neo4j in Action*. Manning, 2014.

[15] B. Avey, *Labeled vs Typed Property Graphs - All Graph Databases are not the same*, 2021.

[16] J. Barrasa, *RDF Triple Stores vs. Labeled Property Graphs: What is the Difference?*, 2021.

[17] "The RDF ecosystem." https://book.validatingrdf.com/bookHtml008.html.

[18] "Graph Algorithms." https://neo4j.com/docs/graph-data-science/current/ algorithms/.

[19] "Graph Algorithms." https://graphacademy.neo4j.com/courses/graph-data-science-fundamentals/1-graph-algorithms/.

[20] "Using Graph Algorithms for Advanced Analytics Part3, Community Detection." https://www.youtube.com/watch?v=s3HvMvyHTUY&t=853s.
[21] "Using Graph Algorithms for Advanced Analytics Part1, Shortest Paths." https://www.youtube.com/watch?v=Ra0qORVKsWs&t=1819s.
[22] "Using Graph Algorithms for Advanced Analytics Part2, Centrality." https://www.youtube.com/watch?v=msbR-S-R8&t=1682s.
[23] "Degree Centrality." https://www.sci.unich.it/ francesc/teaching/network/degree.html.
[24] "Betweenness Centrality." https://www.sci.unich.it/ francesc/teaching/network/-betweeness.html.
[25] "PageRank Centrality." https://www.sci.unich.it/ francesc/teaching/network/-pagerank.html.
[26] J. Kleinberg and D. Liben-Nowell, *The Link Prediction Problem for Social Networks*, 2004.
[27] "Topological link prediction." https://neo4j.com/docs/graph-data-science/current/algorithms/linkprediction.
[28] "Link prediction." https://neo4j.com/developer/graph-data-science/link-prediction.
[29] "Split relationships." https://neo4j.com/docs/graph-data-science/current/machine-learning/pre-processing/split-relationships/.
[30] Ricci, L. Rokach, and B. Shapira, *Recommender Systems Handbook*. Springer, 2015.
[31] D. Jannach, M. Zanker, A. Felfernig, and G. Friedrich, *Recommender Systems: An Introduction*. Cambridge University Press, 2010.
[32] "Building a Real-Time Product Recommender System with Graph Databases." https://medium.com/badal-io/building-a-real-time-product-recommender-system-with-graph-databases-leveraging-neo4j-and-bigquery-65b5b361d276.
[33] Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl, "Evaluating collaborative filtering recommender systems," in *ACM Transactions on Information Systems22:1*, pp. 5–53, 2004.
[34] Y. Koren, "Factorization meets the neighborhood: a multifaceted collaborative filtering model," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 426–434, 2008.
[35] G. Takacs, I. Pilaszy, B. Nemeth, and D. Tikk, "Major components of the gravity recommendation system," in *SIGKDD Explorations Newsletter 9:2*, pp. 80–83, 2007.
[36] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms.," in *Proceedings of the 10th International World Wide Web Conference*, pp. 285–295, 2001.
[37] R. Burke, "Hybrid recommender systems: Survey and experiments.," in *User Modeling and User-Adapted Interaction 12:4*, pp. 331–370, 2002.
[38] "Best techniques and metrics for Imbalanced Dataset." https://www.kaggle.com/code/marcinrutecki/best-techniques-and-metrics-for-imbalanced-dataset.
[39] "Credit Card Fraud Detection." https://www.kaggle.com/code/gopibollineni/credit-card-fraud-detection.
[40] "Antifraud Model For Payment Transactions Dataset."

https://github.com/JulienGenovese/JulienGenovese/tree/master/use-cases/antifraud-with-graph.

[41] "Tour of Evaluation Metrics for Imbalanced Classification." https://machinelearningmastery.com/tour-of-evaluation-metrics-for-imbalanced-classification.

[42] "Exploring Fraud Detection With Neo4j & Graph Data Science." https://medium.com/@zach.blumenfeld/exploring-fraud-detection-with-neo4j-graph-data-science-part-4-cff013808a45.

[43] "Graph algorithms & graph machine learning: Making sense of today's choices." https://www.youtube.com/watch?v=$d1oLRt1\_X8$.

[44] G. Ian, Y. Bengio, and A. Courville, *Deep Learning*. The MIT Press, 2016.