neo4j

Graph Powered Machine Learning

Essential Excerpts

Alessandro Negro

MANNING

Graph-Powered Machine Learning Essential Excerpts

Chapters chosen by Alessandro Negro

Chapter 3
Chapter 4
Chapter 7

Copyright 2020 Manning Publications
To pre-order or learn more about these books go to www.manning.com

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: Erin Twohey, corp-sales@manning.com

©2020 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Marija Tudor

ISBN: 9781617298738

contents

introduction

When Leonhard Euler first sketched out the foundational tenets of graph theory more than 200 years ago, he couldn't possibly have imagined all of the ways that his theory would be put to use in society and industry. (Then again, it's quite possible that Euler's genius *could* imagine such applications.) Today, graphs are everywhere we look: from fraud detection and social networks to recommendation engines and, of course, machine learning.

It's this versatility of the graph model that makes it so valuable to the practice of machine learning. The secret is context. By making connections more explicit, graphs enhance older-style machine learning approaches through the power of context. For example, recommendation engines benefit significantly from increased data context.

And that's exactly what I wanted to cover in the present text. This book excerpt—sponsored by Neo4j—features three key chapters from my book *Graph-Powered Machine Learning*.

Chapter 3 discusses *graphs in machine learning applications*. We'll cover graph technology in a machine-learning workflow, including data sources, graph algorithms, and data visualization. Also discussed are graphs as a processing pattern and graph-defined workflows, with concrete examples of each.

Chapter 4 covers an *introduction to recommendation engines*, with an emphasis on content-based recommendations. We'll take a closer look at modeling user data and item features. We'll also examine the clear advantages of using graphs for recommendation engines.

Chapter 7 dives deeper into *context-aware and hybrid recommendation engines*. This advanced material shows how data context increases the relevance of your recommendations—and how to use graphs to deliver that critical context. Finally, we discuss hybrid recommendation engines that combine the strengths of every recommendation paradigm discussed in the book.

I hope that you enjoy this book excerpt and find it helpful in your practice as a machine learning professional. To dive even deeper, you can purchase the entire book directly from Manning Publications, and I hope it sparks your interest in using Neo4j for your next graph-powered machine learning project.

Yours,
—Alessandro Negro

3 Graphs in machine learning applications

This chapter covers

- Learning the role of graphs in the machine learning workflow
- Understanding a system for large-scale graph processing
- Seeing how graphs are used to break down complex processing tasks

In this chapter, we explore in more detail how graphs and machine learning can fit together, helping to deliver better services to end users, data analysts, and businesspeople. The previous two chapters introduced general concepts in machine learning, such as:

- The different types of machine learning algorithms (supervised, unsupervised, and semi-supervised)
- The different phases that compose a generic machine learning project (specifically, the six phases of the CRISP-DM model: business understanding, data understanding, data preparation, modeling, evaluation, and deployment)

- The importance of data quality and quantity to create a valuable and meaningful model that can provide accurate predictions
- How to handle a large amount of data ("big data") using a graph data model

Here, we'll see how to harness the power of the graph model as a way of representing data that makes it easy to access and analyze as well as how to leverage the "intelligence" of the machine learning algorithms based on graph theory.

I'd like to start this chapter with an image (figure 3.1) that represents the path of converting raw data, available from multiple sources, into something that is more than "simple" knowledge or insight: *wisdom*.

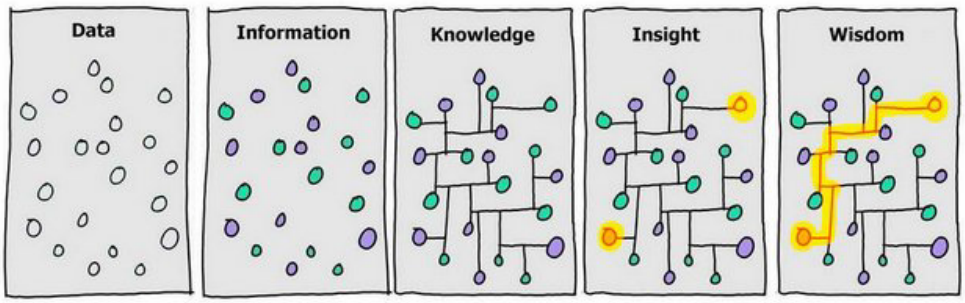


Figure 3.1 Illustration by David Somerville based on the original by Hugh McLeod.[1]

We're flooded by data. Data is *everywhere*. News, blog posts, emails, chats, and social media are a few examples of the multiple sources generating data that surround us. Furthermore, at the time of this writing, we're in the middle of the IoT explosion: today even my washing machine sends me data, reminding me that my pants are clean, and my car knows when I should stop driving and take a coffee break.

Data by itself is completely useless, though; on its own, it doesn't provide any value. To make sense of the data, we have to interact with it and organize it. This process produces *information*. Turning this information into *knowledge*, which reveals relationships between information items—a quality change—requires further effort. This transformation process "connects the dots," causing previously unrelated information to acquire sense, significance, and logical semantics. From knowledge come *insight* and *wisdom*, which are not only relevant but also provide guidance and can be converted into actions: producing better products, making users happier, reducing production costs, delivering better services, and more. This is where the true value of data resides, at the end of a long transformation path—and machine learning provides the necessary "intelligence" for distilling value from it.

[1] http://smrvl.com

Figure 3.1, to a certain extent, represents the learning path for the first part of this chapter:

1. Data and information are gathered from one or several sources. This is the training data on top of which any learning will happen, and it's managed in the form of a graph (section 3.1.1).
2. Once the data is organized in the form of knowledge and represented using a proper graph, machine learning algorithms can extract and build insights and wisdom on top of it (section 3.1.2).
3. The prediction models that are created, as the result of the training of a machine learning algorithm on the knowledge, are stored back in the graph (section 3.1.3), making the wisdom inferred permanent and utilizable.
4. Finally, the visualization (section 3.1.4) shows the data in a way that can easily be understood by the human brain, making the derived knowledge, insights, and wisdom accessible.

This path follows the same mental model used in the previous chapters to highlight and organize the multiple ways in which graphs can be a valuable help in your machine learning project (figure 3.2).

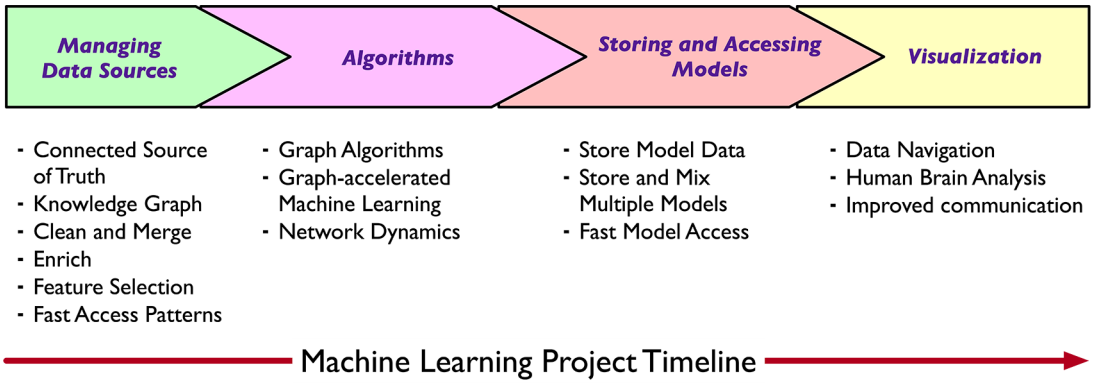


Figure 3.2 **Mental model for graph-powered machine learning.**

We'll start from the beginning of this mental model and go in deep, showing several of the many techniques and approaches that use graph features to deliver a better machine-learning project.

3.1 Graphs in the machine learning workflow

The CRISP-DM model described in chapter 1 [Wirth and Hipp, 2000] allows us to define a generic *machine-learning workflow* which can be decomposed, for the purposes of our discussion, into the following macro steps:

1. Select the data sources, gather data, and prepare the data.
2. Train the model (the learning phase).
3. Provide predictions.

As discussed in chapter 1, certain learning algorithms don't have a model. The instance-based algorithms use the entries in the training dataset during the prediction phase. For this class of algorithm, the second step isn't necessary. Although the graph approach can be a valid support even in these cases, we won't consider these algorithms in our analysis.

Furthermore, quite often, data needs to be visualized in multiple shapes to achieve the purpose of the analysis. Hence, visualization also plays an important role as a final step that completes the machine learning workflow, allowing further investigation.

This workflow description matches exactly the mental model in figure 3.2, which you'll see throughout this chapter (and the book) to help you figure out where we are in each step.

In such a workflow, it's important to look at the role of the graph from operational, task-based, and data flow perspectives. Figure 3.3 illustrates how data flows from the different data sources through the learning process to end users in the form of visualizations or predictions.

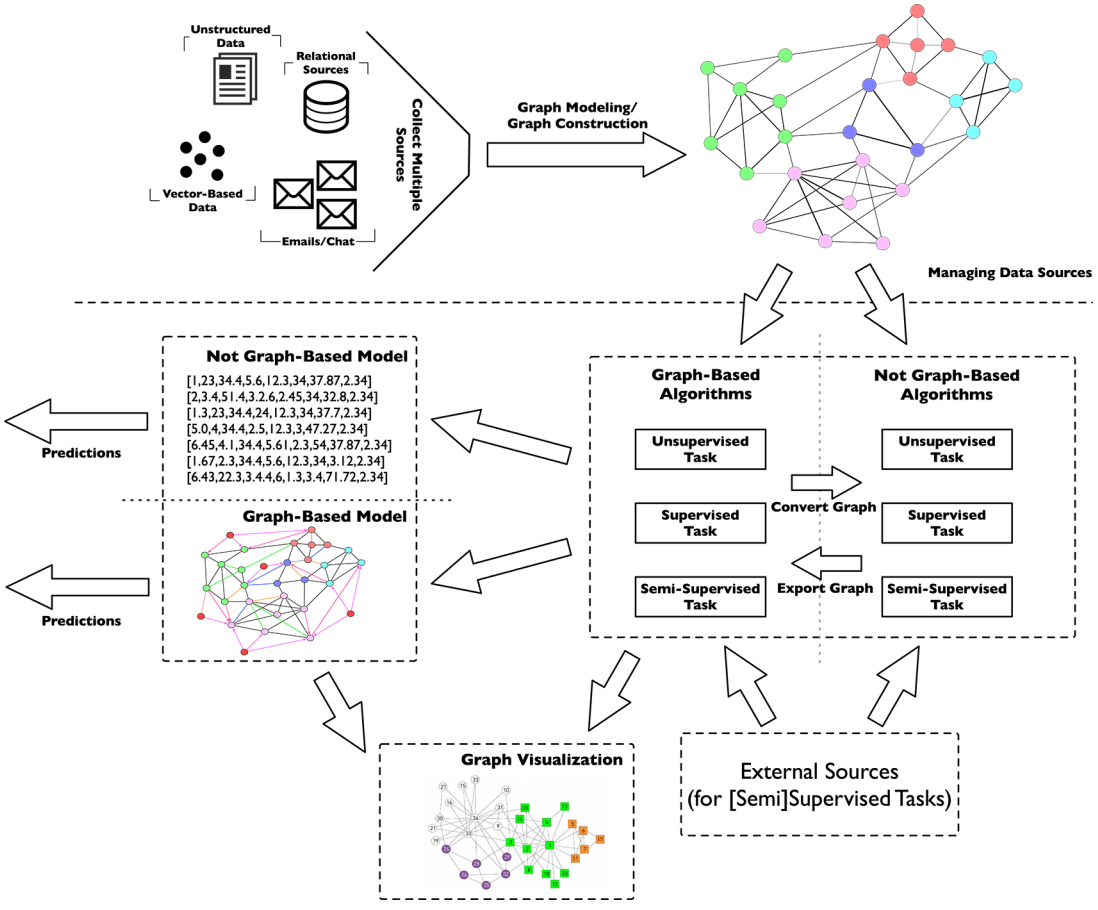


Figure 3.3 The role of graphs in the machine learning workflow.

The process starts, as usual, from the data available. Different data sources will have different schemas, structure, and content. Generally, the data used in machine learning applications can be classified as big data (we discussed working with such data in chapter 2). This data must be organized and managed before the learning process can begin. The graph model helps with data management by creating a connected and well-organized source of truth. The transformation from the original data shape into a graph could happen using multiple techniques that can be classified into two groups:

- *Graph modeling:* Data is converted using a modeling pattern into some graph representation. The information is the same, but in a different format, or the data is aggregated to make it more suitable to feed into the learning process.
- *Graph construction:* A new graph is created starting from the data available. The resulting graph contains more information than before.

After this preparation, the data is stored in a well-structured format ready for the next phase: the learning process. The graph representation of the data doesn't only support graph-based algorithms; it can feed multiple types of algorithm. Specifically, the graph representation is helpful for the following tasks:

- *Feature selection:* Querying a relational database or extracting a key from a value in a NoSQL database is a complex undertaking. A graph is easy to query and can merge data from multiple sources, so finding and extracting the list of variables to use for training is made simpler by the graph approach.
- *Data filtering:* The easy-to-navigate relationships between objects make it easy to filter out useless data before the training phase. This speeds up the model-building process. We'll see an example of this later when we consider the recommendation scenario.
- *Data preparation:* Graphs make it easy to clean the data, removing spurious entries, and merging data from multiple sources.
- *Data enrichment:* Extending the data with external sources of knowledge (for example, semantic networks, ontologies, taxonomies) or looping back the result of the modeling phase to build a bigger knowledge base is straightforward with a graph.
- *Data formatting:* It's easy to export the data in whichever format is necessary: vectors, documents, and so on.

In both scenarios (graph-based or non-graph-based algorithms), the result could be a model that's well suited to a graph representation; in that case it can be stored back in the graph, or it can be stored in a binary or proprietary format.

Whenever the predictive model allows it, storing the model back in the graph gives the opportunity to perform predictions as queries (more or less complex) on the graph. Moreover, the graph provides access to the same model from different perspectives and for different scopes. Recommendations, described later, are an example of the potential of this approach.

Finally, the graph model can be used to visualize the data in a graph format, which often represents a big advantage in terms of communication capabilities. Graphs are

whiteboard-friendly, so the visualizations can also improve the communication between business owners and data scientists in the early stages of a machine-learning project.

The phases and steps described here aren't all mandatory; depending on the needs of the machine-learning workflow, only part of them might be helpful or necessary. Later sections will present a range of concrete example scenarios. For each scenario presented, the role of the graph is clearly illustrated.

3.1.1 Managing data sources

As we've seen, graphs are extremely useful for encoding information, and data in graph format is increasingly plentiful. In many areas of machine learning, including natural language processing, computer vision, and recommendations, graphs are used to model local relationships between isolated data items (users, items, events, and so on) and to construct global structures from local information [Zhao and Silva, 2016]. Representing data as graphs is often a necessary step (and at other times a desirable one) when dealing with problems arising from applications in machine learning or data mining. In particular, it becomes crucial when we want to apply graph-based learning methods to the datasets (figure 3.4).

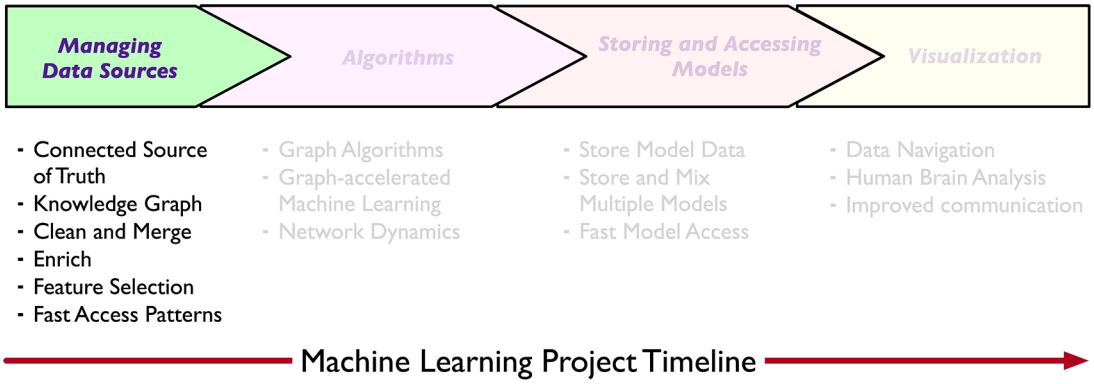


Figure 3.4 Managing data sources in the mental model.

The transformation from structured or unstructured data to a graph representation can be performed in a *lossless* manner, but this isn't always necessary (or desirable) for the purpose of the learning algorithm. Sometimes, a better model is an "aggregated view" of the data. For instance, if you're modeling a phone call between two people you can decide to have a relationship between the two entities (the caller and the receiver) for each call, or you can have a single relationship that aggregates all the calls.

A graph can be constructed from the input dataset in two ways:

- By designing a graph model that represents the data
- By using convenient graph formation criteria

In the first case, the *graph model* is an alternative representation of the same information available in the dataset itself, or in multiple datasets. The nodes and the relation-

ships in the graph are a mere representation (aggregated or not) of the data available in the original sources. Furthermore, in this case the graph acts as a connected data source that merges data coming from multiple heterogeneous sources, operating, at the end of the process, as the single trusted source of truth. There are multiple methodologies, techniques, and best practices to apply this model shift and represent data in a graph format, and we'll discuss several of them here, considering multiple scenarios. A detailed list of the most common data model patterns for this process in the second part of the book.

In the second scenario (using convenient *graph formation criteria*), the data items are stored in the graph (generally as nodes) and a graph is created using an edge construction mechanism. As an example, suppose that in your graph each node represents some text, such as a sentence or an entire document. They're *isolated entries*. There's no relationship between them unless they're connected explicitly (via a citation in a paper, for instance). In machine learning, text is generally represented as a vector where each entry contains the weight of a word or a "feature" in the text. Edges can be created (constructed) using the similarity or dissimilarity value between the vectors. A new graph is created starting from unrelated information. In such a case, the resulting graph embeds *more information* than the original datasets. This additional information is made up of several ingredients, the most important of which is the structural or topological information of the data relationships.

The result of both processes is a graph that represents the input data and that becomes the training dataset for the relevant machine learning algorithms. In some cases, these algorithms are themselves graph algorithms, so they require a graph representation of the data, and in other cases the graph is a better way of accessing the same data. The examples and scenarios described here represent both cases.

The required steps in the process of converting data into its graph representation (or creating it as a graph) are shown in figure 3.5.
Let's take a closer look at each step:

1. *Identify the data sources.* Identify the data available for algorithm training purposes as well as the sources from which such data can be extracted. This corresponds to the second phase in a machine learning project, after defining the goals (the data preparation phase of the CRISP-DM data model).
2. *Analyze the data available.* Analyze each data source available and evaluate the content, in terms of quality and quantity. To achieve good results from the training phase, it's imperative to have a large amount of good-quality data.
3. *Design the graph data model.* This step is twofold. According to the specific analytics requirements, you must:
    a. Identify the meaningful information to be extracted from the data sources.
    b. Design a specific graph model, considering the data available, access patterns, and extensibility.

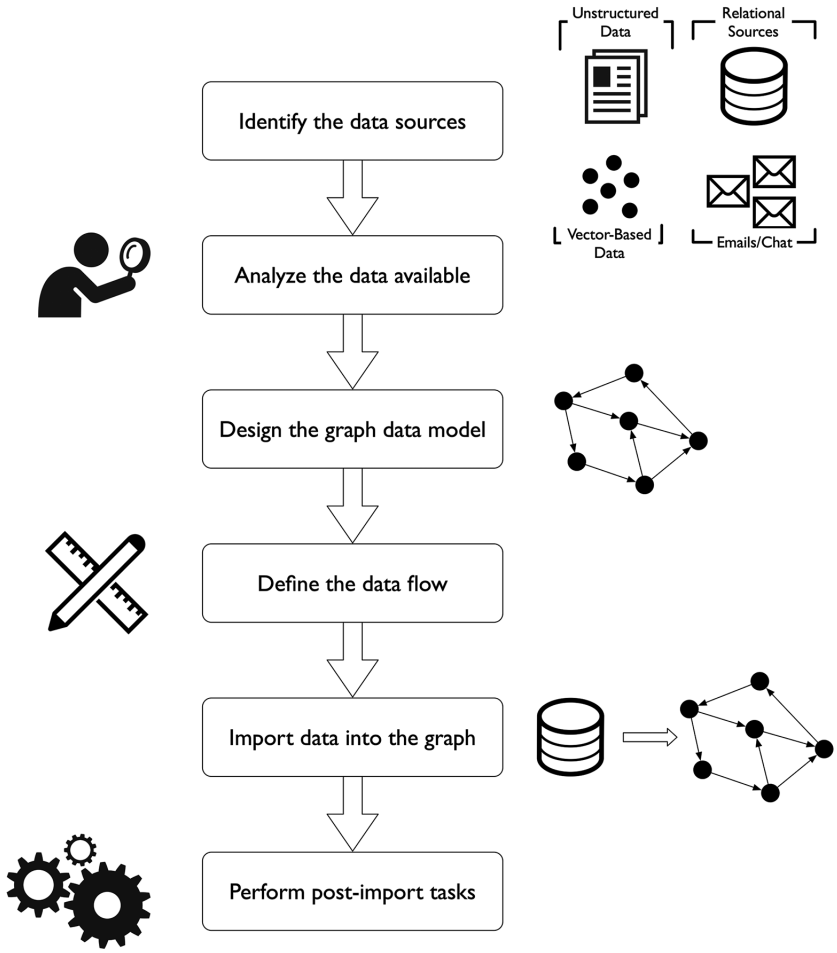


Figure 3.5 **Process of converting data into a graph representation.**

4. *Define the data flow.* Design the architecture of the ingestion process (known as the *ETL process*) that extracts, transforms, and loads the data from the multiple sources in the graph database using the schema designed.
5. *Import the data into the graph.* Start the ETL process defined in step 4. Generally, steps 3, 4, and 5 are iterative until you arrive at the right model and the right ETL process.
6. *Perform post-import tasks.* Before starting the analysis, the data in the graph might require preprocessing. These tasks include:
   a. *Data cleaning:* Remove or correct incomplete or incorrect data.
   b. *Data enrichment:* Extend the existing data sources with external sources of knowledge or with knowledge extracted from the data itself. The latter case falls under graph creation.

- *Data merging:* Because the data comes from multiple sources, related elements in the dataset can be merged in a single element or they can be connected through new relationships.

Steps 5 and 6 can be inverted or mixed; in certain cases, the data may pass through a process of cleaning before the ingestion happens. In any case, at the end of those six steps the data is ready for the next phase, which involves the learning process.

The new representation of the data provides several advantages that we investigate here through the lens of multiple scenarios and multiple models. Several of these scenarios you'll be seeing for the first time, but others were introduced in the previous two chapters and will be extended further throughout the book. For each scenario presented, the context and purpose are described. These are key aspects to define the right model and to understand the value of the graph approach for storing and managing data and of graphs as input for the next steps in the analysis.

Part II describes in detail the techniques for representing different datasets using a graph model. This chapter highlights, through the example scenarios, the primary advantages of using a graph to manage the data available for training the prediction model.

Monitor a subject

> Suppose again that you're a police officer. You want to track a suspect and predict their future movements using cellular tower data collected from the continuous monitoring signals every phone sends to (or receives from) all towers it can reach. Using graph models and graph clustering algorithms, it's possible to structure cellular tower data and represent a subject's positions and movements in a simple and clear manner. A predictive model can then be created.

The goal in this scenario is to monitor a subject and create a predictive model that identifies location clusters relevant to the subject's life and that's able to predict and anticipate subsequent movements according to the subject's current position and last movements [Eagle, Quinn, and Clauset, 2009].

The data in this scenario is cellular tower data generated by the interaction between the subject's phone and the cellular towers, as represented in figure 3.6.

For the purpose of such as analysis, data can be collected from the towers or from the phones belonging to the monitored subjects. The data from the cellular towers is easy to obtain with the necessary permissions, but it requires a lot of cleaning (removing the irrelevant numbers) and merging (data from multiple cellular towers). Gathering data from the phones requires hacking, which isn't always possible, but this data is clean and already merged. In their paper, Eagle, Quinn, and Clauset consider this second data source, and we do the same here, but the results and the considerations are the same regardless of the source of the data.

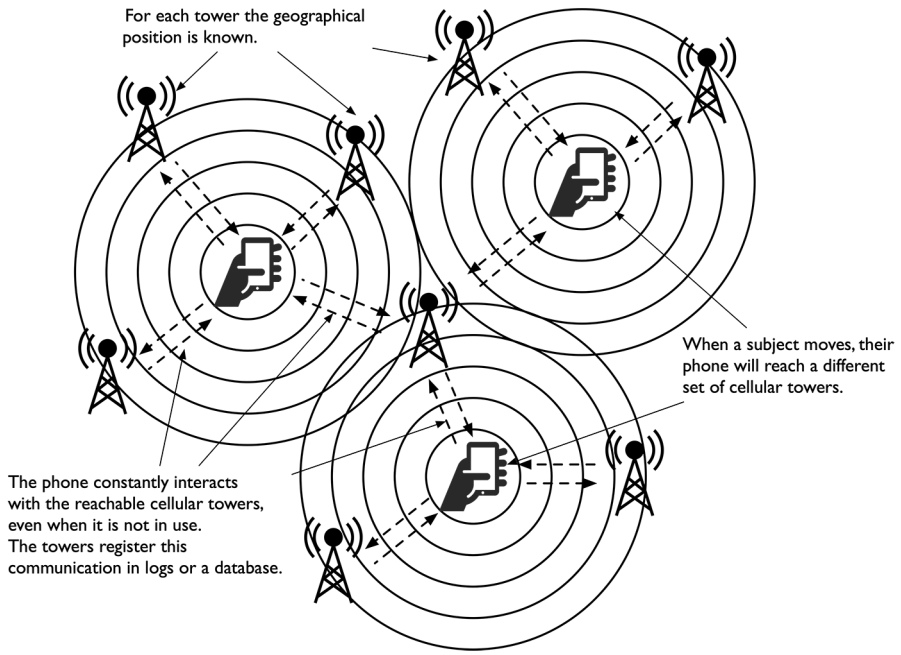


Figure 3.6 **Phones communicating with cellular towers.**

Let's suppose that the phones will provide data in the format shown in table 3.1.

Table 3.1 **Examples of the Data Provided by a Single Phone with the 4 towers (identified by their id) reached at each timestamp.**

Phone identifier	Timestamp	Cellular tower 1	Cellular tower 2	Cellular tower 3	Cellular tower 4
562d6873b0fe	1530713007	eca5b35d	f7106f86	1d00f5fb	665332d8
562d6873b0fe	1530716500	f7106f86	1d00f5fb	2a434006	eca5b35d
562d6873b0fe	1530799402	f7106f86	eca5b35d	2a434006	1d00f5fb
562d6873b0fe	1531317805	1d00f5fb	665332d8	f7106f86	eca5b35d
562d6873b0fe	1533391403	2a434006	665332d8	eca5b35d	1d00f5fb

For the sake of simplicity, this table represents the data provided by a single phone (the phone identifier is always the same). Each phone records the four towers with the strongest signals at 30-second intervals.

The analysis requires us to identify locations in which the monitored subject spends time. The cellular tower data available on the phone cannot provide this information by itself because it only contains the identifiers of the four towers with the highest signal strengths, but starting from such data it's possible to identify key locations by passing through a graph representation of the data and a graph algorithm.

This data can therefore be modeled as a graph that represents a cellular tower network (CTN). As you'll recall from the previous chapter, each node in this graph is a unique cellular tower; edges exist between any two nodes that co-occur in the same record, and each edge is assigned a weight according to the total amount of time that pair of nodes co-occurred in a record, across all records. A CTN is generated for each subject that shows every tower logged by that individual's phone during the monitoring period. The result looks like figure 3.7.

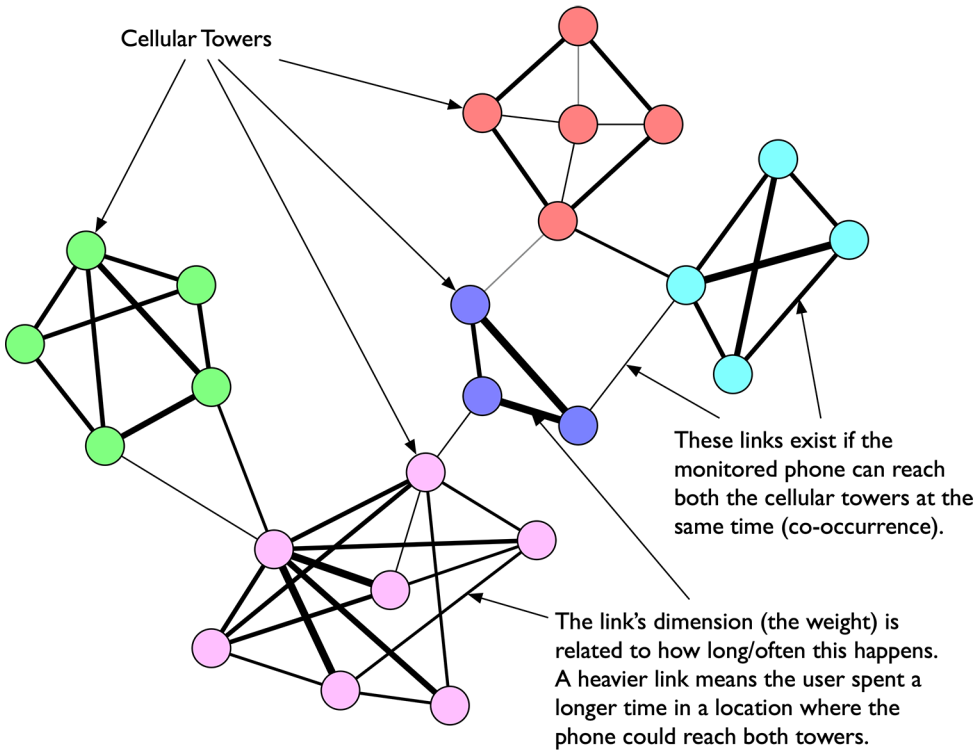


Figure 3.7 **A graph representation of the CTN for a single subject.**

A graph clustering algorithm is then applied to this graph to identify the main locations where the subject spent a significant amount of time (for example, at the office, at home, at the supermarket, at church, and so on). The result of the analysis looks like figure 3.8, in which multiple clusters are identified and isolated.

This scenario shows how well-adapted a graph model is to represent the data for the specific purposes of this analysis. By performing a graph-based analysis using the community detection algorithm, we can easily identify areas where the subject spends a lot of time—a task that would be difficult, if not impossible, with other representations or analysis methods.

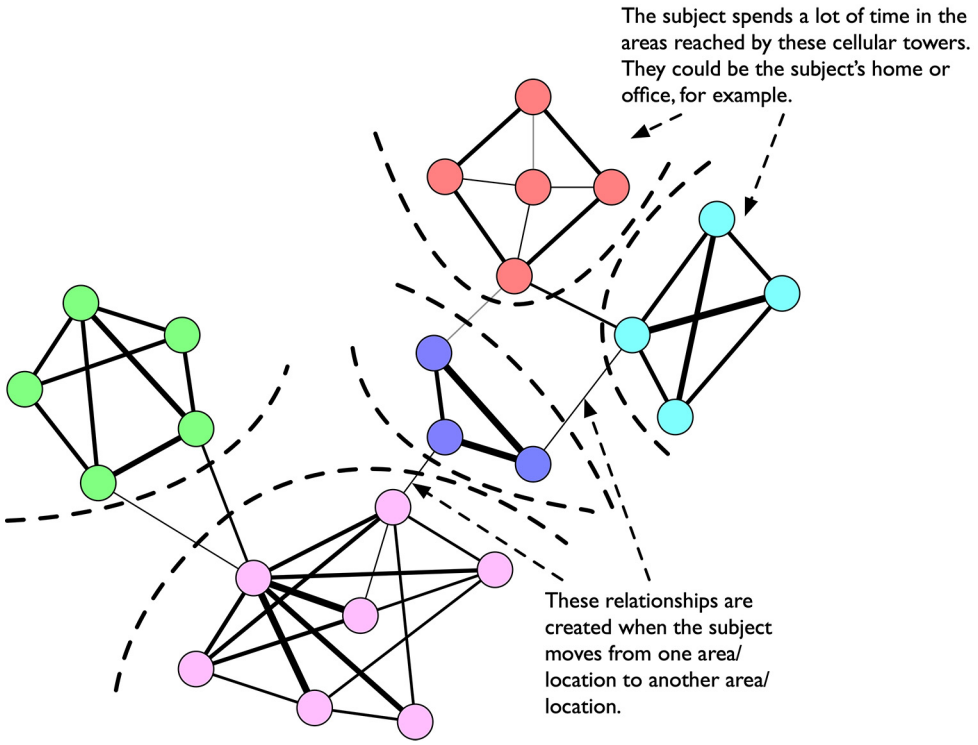


Figure 3.8 **A clustered view of the CTN.**

The graph modeling described here illustrates a graph construction technique. The resulting graph can be generalized as a *co-occurrence graph*. The nodes represent entities (in this case, cellular towers), and the relationships represent the fact that the two entities belong to a common set or group (in the CTN, the set is the row in the table that indicates that at a specific point in time the phone can reach both towers). This is a powerful technique used in many algorithms and machine learning applications as a data preprocessing step before performing the analysis. Quite often, this type of graph construction technique is used in applications related to text analysis; we'll see an example of this later.

DETECT A FRAUD

> Suppose again that you want to create a fraud-detection platform for banks that reveals the point of origin of a credit-card theft. A graph representation of the transactions can help you identify, even visually, the location of the theft.

In this scenario the data available is the credit card transactions, with details about the date (timestamp), the amount, the merchant, and whether the transaction is "disputed" or "undisputed." Once a person's credit card details have been stolen, in the

transaction history, real operations are mixed with illegal or fraudulent operations. The goal of the analysis is to identify the point where the fraud started—the shop where the theft occurred. The transactions at that shop will be real, but any transactions that have taken place afterward may be fraudulent.

The data available looks like table 3.2.

Table 3.2 A Subset of User Transactions

User identifier	Timestamp	Amount	Merchant	Validity
User A	01/02/2018	$250	Hilton Barcelona	undisputed
User A	02/02/2018	$220	AT&T	undisputed
User A	12/03/2018	$15	Burger King New York	undisputed
User A	14/03/2018	$100	Whole Foods	disputed
User B	12/04/2018	$20	AT&T	undisputed
User B	13/04/2018	$20	Hard Rock Cafe	undisputed
User B	14/04/2018	$8	Burger King	undisputed
User B	20/04/2018	$8	Starbucks	disputed
User C	03/05/2018	$15	Whole Foods	disputed
User C	05/05/2018	$15	Burger King	undisputed
User C	12/05/2018	$15	Starbucks	disputed

Our aim is to identify a common pattern that reveals at which point users start disputing their transactions, which will help us to locate the establishment where the card details were stolen. This analysis can be performed using a graph representation of the transactions. The data in table 3.2 can be modeled in a *transaction graph* as shown in figure 3.9.

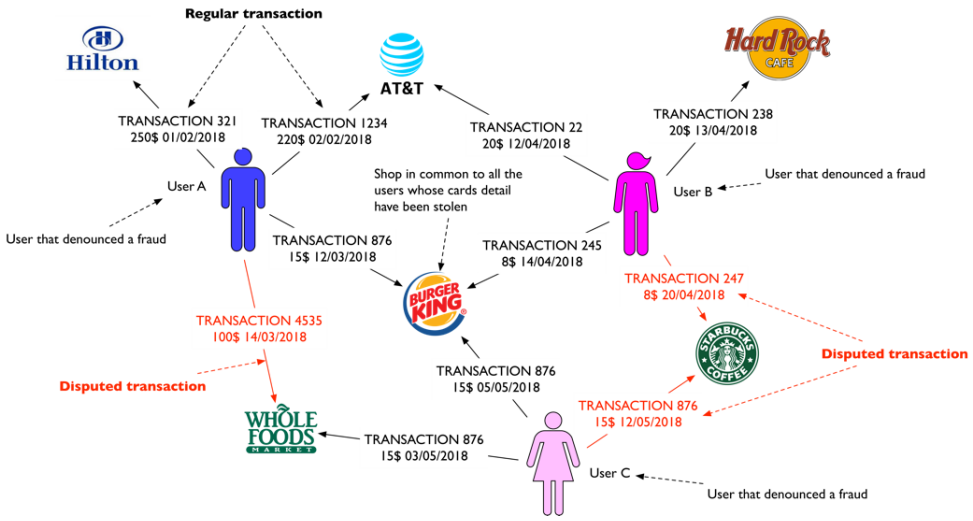


Figure 3.9 A transaction graph for credit card fraud detection.

As described in the previous chapter, by starting from this representation of the transactions and using graph queries, it's possible to determine that the theft occurred at Burger King. (The steps taken to arrive at this conclusion are described in section 2.2.2 and will not be repeated here.)

The graph of the transactions allows us to easily identify where the card thief operates. In this case the analysis is performed on the graph as it appears after the ETL phase; no other intermediate transformation is required. This data representation expresses the information in such a way that it is possible to quickly recognize behavioral patterns in a long list of transactions and spot where the issue is.

Transaction graphs such as the one shown here can represent any kind of event that involves two entities. Generally, they're used for modeling monetary transactions in an unaggregated way, which means every single operation can be related to a specific portion of the graph. For the majority of cases, in the resulting graph each transaction is represented in one of the following two ways:

- As a *directed edge* between the two entities involved in the transaction. For example, if User A makes a purchase at Shop B, this event is translated into a directed edge that starts with User A and terminates at Shop B. In this case, all the relevant details about the purchase, such as the date and amount, are stored as properties of the edge (figure 3.10 (a)).
- As a *node* that contains all the relevant information about the event and is connected via edges to the related nodes. In the case of the purchase, the transaction itself is modeled as a node, and it's then connected to the "source" and "destination" of the purchase (figure 3.10 (b)).

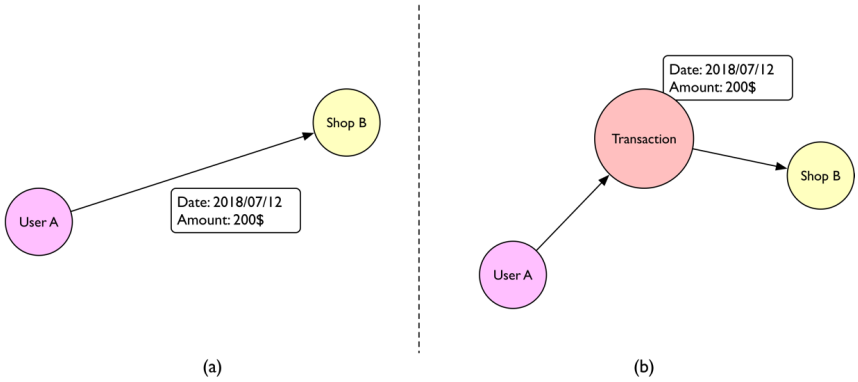


Figure 3.10 Transaction modeling examples.

The first approach is generally used when the amount of information related to the event is small or when a simpler model is preferable for the purpose of the analysis. The second approach is generally preferred when the event itself contains valuable information that could be connected to other information items, or when the event involves more than two items.

Transaction graphs are quite common in fraud detection analysis and all machine-learning projects in which each event contains relevant information that, if the events were aggregated, would be lost.

IDENTIFY RISKS IN A SUPPLY CHAIN

> Suppose that you have to implement a risk management system that identifies or predicts possible risks in a supply chain. A *supply chain network* (SCN) is a common way to represent supply chain elements and their interactions in a graph. Such a representation, together with proper graph analysis algorithms, makes it easy and fast to spot issues throughout the chain.

This scenario has become more and more relevant in recent years, for multiple reasons. For example:

- With the development of the global economy, which means any supply chain can have a global dimension, the problems that supply chain management faces are becoming more complex.
- Customers located at the end of the chain are becoming more interested in the origins of the products they buy.

Managing the disruption risk and making the chain more transparent are currently mandatory tasks in any supply chain. Supply chains are inherently fragile and face a variety of threats, from natural disasters and attacks to the contamination of raw products, delivery delays, and labor shortages [Kleindorfer and Saad, 2005]. Furthermore, because the different parts of the chain are complex and interrelated, the normal operation of one member—and the efficient operation of the chain as a whole—often relies on the normal operation of other components. The members within a supply chain include suppliers, manufacturers, distributors, customers, and so on. They're all dependent on one another and cooperate with each other through material flows, information flows, and financial flows, but they're also independent entities operating on their own, and perhaps providing the same services to multiple companies. Therefore, the data available in such a scenario will be distributed across multiple companies that have different structures. Any kind of analysis based on data in this shape is a complex task; gathering the required information from the multiple members and organizing it requires a lot of effort.

The purpose of the analysis here is to spot elements in the chain that, if compromised, can disrupt the entire network (or a large part of it), or significantly affect the normal behavior. A graph model can support such an analysis task through different network analysis algorithms. We will discuss details about the algorithms later; here, we focus on the graph construction techniques that can be used to build the graph representation from the multiple sources available.

A supply chain can be represented by a graph using the following approach:

- Each member of the supply chain is represented by a node. The members could be raw product suppliers (primary suppliers), secondary suppliers, intermediate distributors, transformation processes, organizational units in a big company, final retailers, and so on. The granularity of the graph is related to the risk evaluation required.
- Each relation in the graph represents a dependency between two members of the chain. The relationships could include transport from a supplier to an intermediate distributor, a dependency between two processing steps, the delivery to the final retailer, and so on.
- To each node it's possible to relate "temporal" data that could store historic as well as forecasting information.

The network structure might evolve over time as well. The graph model can be designed to keep track of the changes, but that would make it too complicated for the purpose of this example.

Our example graph model is shown in figure 3.11.

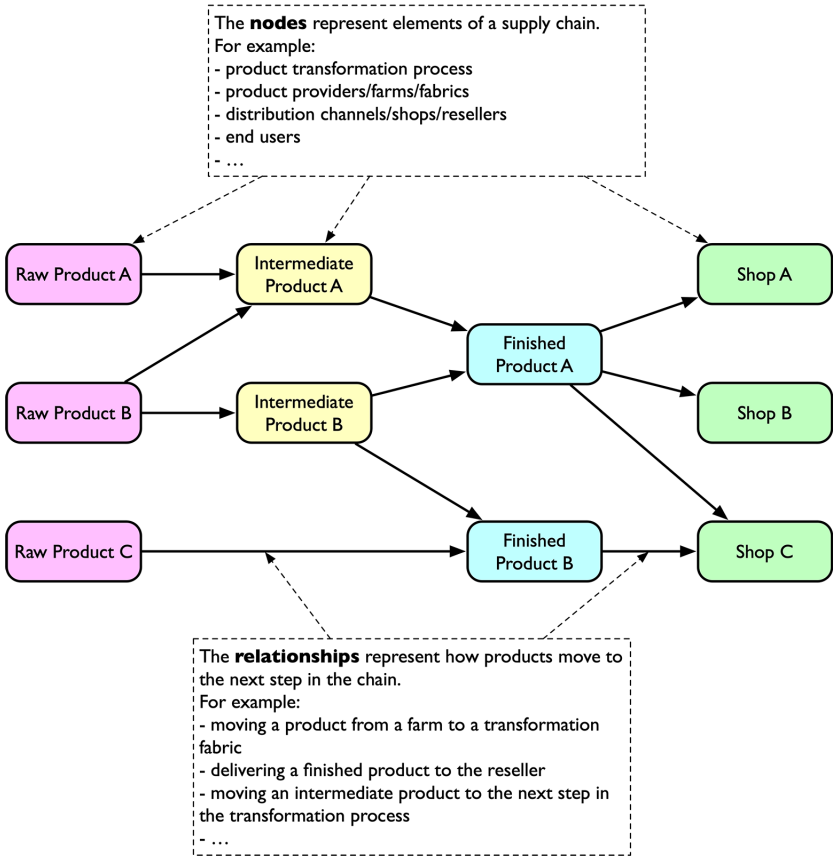


Figure 3.11 A supply chain network.

This model represents an important method for gathering data and organizing it in an organic and homogeneous way, and it provides a suitable representation for the type of analysis that risk management requires. The algorithms that allow us to perform the analysis to reveal high-risk elements in the chain will be discussed in section 3.1.2.

RECOMMEND ITEMS

> Suppose that you want to recommend items to users in an e-commerce shop, using the data you have about previous interactions (clicks, purchases, ratings). Graphs can help you store the user–item dataset in a way that speeds up access to it and storing the predictive models in the graph not only facilitates the predictions but also allows you to merge multiple models smoothly.

One of the most common use cases for graphs in machine learning is recommendations. I wrote the first recommendation engine ever built on top of Neo4j in 2012. That's how my career with graphs started, and it's why this specific topic is close to my heart. Throughout this book, using multiple examples, we'll discover the great advantages of using graphs for building multi-model recommendation engines, but here we start with a simple example by considering the most basic implementation.

It's possible to use multiple approaches to provide recommendations. In this specific example, the approach selected is based on a technique called *collaborative filtering*. The main idea of collaborative approaches to recommendations is to exploit information about the past behavior or opinions of an existing user community to predict which items the current user of the system will most probably like or be interested in [Jannach et al., 2010]. Pure collaborative approaches take a matrix of given user–item interactions of any type—views, past purchases, ratings, and so on—as input and produce the following types of output:

- A (numerical) prediction indicating the likelihood that the current user will like or dislike a certain item (the relevance score)
- An ordered list of *n* recommended items based on the value predicted

The relevance is measured with a utility function *f* that is estimated based on the user feedback [Frolov and Oseledets, 2016]. More formally, the relevance function can be defined as:

$$f: User \times Item \rightarrow Relevance\ Score$$

where *User* is the set of all users and *Item* is the set of all items. This function can then be used to compute the relevance scores for all the elements for which no information is available. The data the predictions are based on can be either directly provided by the users (through ratings, likes/dislikes, and so on) or implicitly collected by observing the users' actions (page clicks, purchases, and so on). The type of information available determines the types of techniques that can be used to build the recommendations. A content-based approach is possible if information about the users

(profile attributes, preferences) and items (intrinsic properties) can be drawn upon. If only implicit feedback is available, a collaborative filtering approach is required.

After predicting relevance scores for all the unseen (or unbought) items, we can rank them and show the top *n* items to the user, performing the recommendation.

As usual, we start our discussion from the data available. The data source in this case looks like table 3.3 (1 means a low rating while 5 means the user has a great opinion of the item).

Table 3.3 An Example of a User–Item Dataset Represented Using a Matrix

User	Item 1	Item 2	Item 3	Item 4	Item 5
Bob	-	3	-	4	?
User 2	3	5	-	-	5
User 3	-	-	4	4	-
User 4	2	-	4	-	3
User 5	-	3	-	5	4
User 6	-	-	5	4	-
User 7	5	4	-	-	5
User 8	-	-	3	4	5

This table is a classic user–item matrix that contains the interactions (in this case the ratings) between the users and the items. The cells with the symbol "-" mean that the user hasn't bought or rated that item. In an e-commerce scenario like the one we're considering, there could be a large number of users and items, so the resulting table could be quite sparse—each user will buy only a small subset of the available items, so the resulting matrix will have a lot of empty cells. In our example table, the unseen or unbought element that we'd like to predict interest in is item 5 for the user Bob.

Starting from the data available (in the shape described) and from the basic idea of collaborative filtering, multiple ways of implementing this prediction exist. For the purpose of this scenario, in this part of the book we'll consider the *item-based* algorithms. The main idea of item-based algorithms is to compute predictions using the similarity between items. Therefore, we'll consider the table column by column, with each column describing a vector of elements (called the *rating vector*) where the "-" symbol is replaced with a 0 value. Let's examine our User–Item dataset and make a prediction for Bob for item 5. We first compare all the rating vectors of the other items and look for items that are similar to item 5. The idea of item-based recommendation is now to simply look at Bob's ratings for these similar items. The item-based algorithm computes a weighted average of these other ratings and uses this to predict a rating for item 5 for the user Bob.

To compute the similarity between items, a *similarity measure* must be defined. *Cosine similarity* is the standard metric in item-based recommendation approaches: it determines the similarity between two vectors by calculating the cosine of the angle between them [Jannach et al., 2010]. In machine learning applications, this measure is often used to compare two text documents, which are represented as vectors of terms; we'll use it frequently in this book.

The formula to compute the cosine of the angle between the two vectors, and therefore the similarity between two items *a* and *b*, is as follows:

$$sim(\vec{a}, \vec{b}) = \cos(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| * |\vec{b}|}$$

The · symbol indicates the dot product of the two vectors. $|\vec{a}|$ is the Euclidian length of the vector, which is defined as the square root of the dot product of the vector with itself.

Figure 3.12 shows a representation of cosine distance in two-dimensional space.

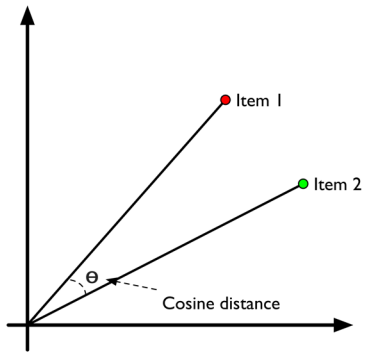


Figure 3.12 Cosine distance representation in two-dimensional space.

To further explain the formula, let's consider the cosine similarity of item 5, described by the rating vector [0, 5, 0, 3, 4, 0, 5, 5], and item 1, described by the vector [0, 3, 0, 2, 0, 0, 5, 0]. It's calculated as follows:

$$sim(\overrightarrow{I5}, \overrightarrow{I1}) = \frac{0*0+5*3+0*0+3*2+4*0+0*0+5*5+5*0}{\sqrt{5^2+3^2+4^2+5^2+5^2} * \sqrt{3^2+2^2+5^2}}$$

The numerator is the dot product between the two vectors, computed from the sum of the products of the corresponding entries of the two sequences of numbers. The denominator is the product of the Euclidian lengths of the two vectors. The *Euclidian distance* is the distance between two points in the multidimensional space of the vectors. Figure 3.13 illustrates the concept in a two-dimensional space.

The formula is as follows:

$$|x, y| = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

The Euclidian length is the Euclidian distance of the vector from the origin of the space (the vector [0,0,0,0,0,0,0,0] in our case):

$$|x| = \sqrt{{x_1}^2 + {x_2}^2}$$

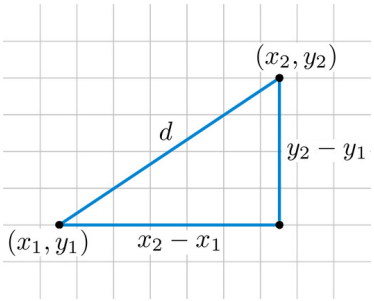


Figure 3.13 Euclidian distance in two-dimensional space.

The similarity values range between 0 and 1, with 1 indicating the strongest similarity. Consider now that we have to compute this similarity between each pair of items in the database, so if we have 1M products we need to compute 1M * 1M similarity values. We can reduce this number by half because similarity is commutative—$cos(a, b) = cos(b, a)$—but it's still many computations. In this case a graph can be a valuable helper to speed up the machine learning algorithm for the recommendations.

The User–Item dataset described previously can be converted easily into a graph like the one in figure 3.14.

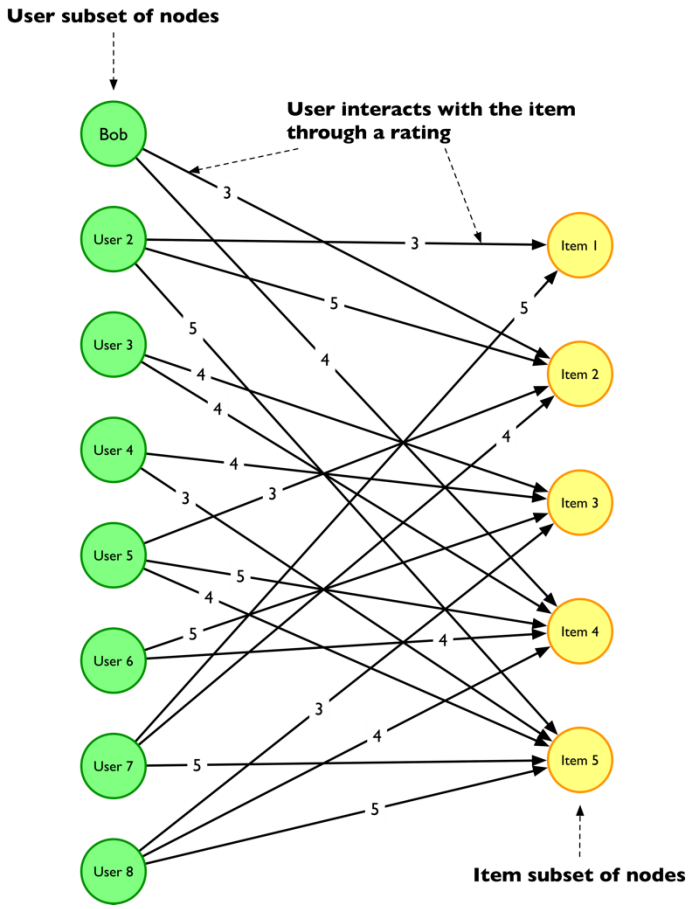


Figure 3.14 Bipartite graph representing the User–Item dataset.

In this graph representation, all the users are on the left and all the items are on the right. The relationships go only from nodes in one subset to nodes in the other subset; no relationships occur between nodes of the same set. This is an example of a *bipartite graph*, or *bigraph*.

More formally, a bigraph is a special type of graph whose vertices (nodes) can be divided into two disjoint and independent sets U and V such that every edge connects a vertex in U to one in V, or vice versa. Vertex sets U and V are usually called the *parts* of the graph [Diestel, 2017].

How can a bipartite graph representation reduce the number of similarity computations we have to perform? To understand this, it's necessary to understand cosine similarity a little better (although the principle can be extended to a wider set of similarity functions). The cosine similarity metric measures the angle between two n-dimensional vectors, so the two vectors have a cosine similarity equal to 0 when they are *orthogonal* (perpendicular). In the context of our example, this happens when there are no overlapping users between two items (users that rate both the items). In such cases, the numerator of the fraction will be 0. For example, we can compute the distance between item 2, described by the vector [3, 5, 0, 0, 3, 0, 4, 0], and item 3, described by the vector [0, 0, 4, 4, 0, 5, 0, 3], as follows:

$$sim(\overrightarrow{I2}, \overrightarrow{I3}) = \frac{3*0+5*0+0*4+0*4+3*0+0*5+4*0+0*3}{\sqrt{3^2+5^2+3^2+4^2}*\sqrt{4^2+4^2+5^2+3^2}}$$

In this case the similarity value will be 0 (see figure 3.15). In a sparse User–Item dataset the probability of orthogonality is quite high, so the number of useless computations is correspondingly high. Using the graph representation, it's easy, with a simple query, to find all the items that have at least one rating user in common. The similarity can then be computed only between the current item and the overlapping items, greatly reducing the number of computations required. In a native graph engine, the query for searching overlapping items is quite fast.

Another approach is to separate the bipartite graph into clusters and compute the distance only between the items belonging to the same cluster. In the second part, other techniques are presented for representing such a graph in a way that simplifies the identification of areas in which it is easier to find similar items or users.

In this scenario, the graph model helps to improve performance by reducing the amount of time required to compute the similarities between the items, and therefore the recommendations. Later we'll see how, starting from this graph model, it's possible to store the results of similarity computations to perform fast recommendations. Furthermore, cosine similarity will be used as a technique for graph construction.

3.1.2 Algorithms

The previous section described the role of the graph model in representing the training data that's used for the learning phase. Such a representation of the source of truth has multiple advantages, as described previously, regardless of whether the learning algorithm is graph-based or not.

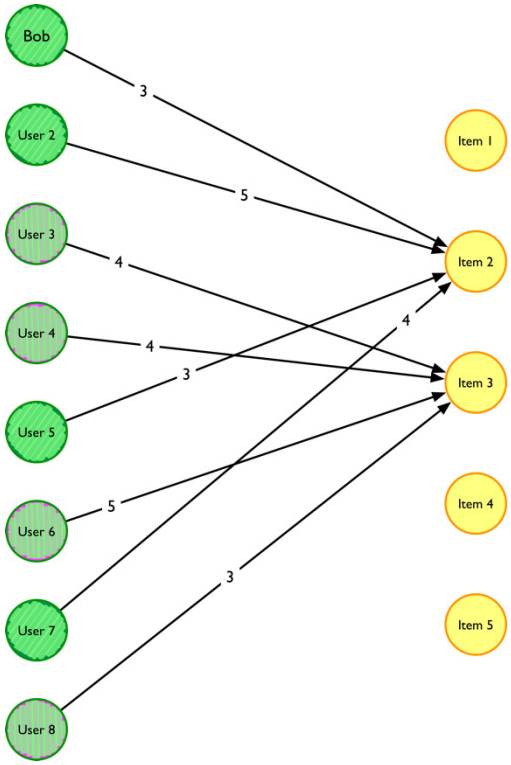


Figure 3.15 **Two nonoverlapping items: these items have no rating users in common, so the cosine similarity between them is 0.**

This section describes (see figure 3.16), again using multiple scenarios, machine-learning techniques that use graph algorithms to achieve the project's goals. We'll consider two approaches:

- The graph algorithm as the *main* learning algorithm
- The graph algorithm as a *facilitator* in a more complex algorithm pipeline

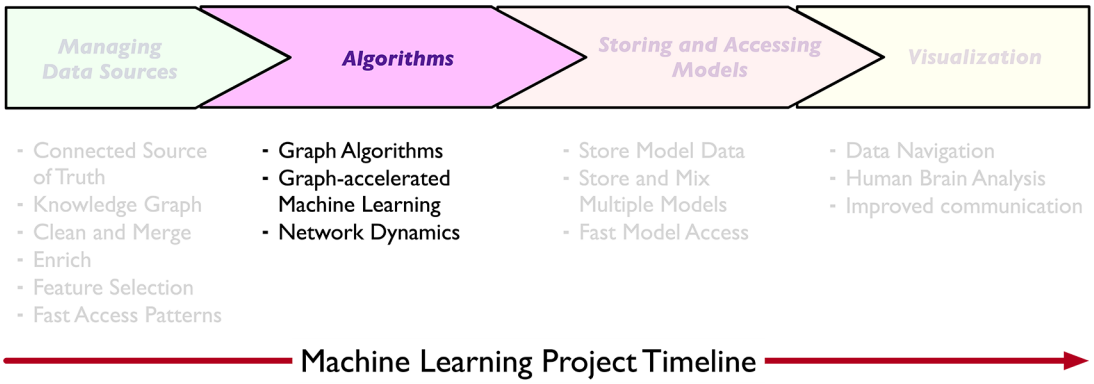


Figure 3.16 **Algorithms in the mental model**

In part II, an entire catalog of algorithms is described in detail with implementation examples. In this chapter the purpose is to highlight the role of graph algorithms for delivering predictions to the end user. These techniques, in contrast to the traditional methods, provide alternative and novel ways to solve the challenging problems posed by machine-learning use cases. The focus here (and in the rest of the book) is on showing how to use these techniques in real-world applications, but we also take into account the design of the methods introduced that are complementary or helpful in terms of performance and computational complexity.

IDENTIFY RISKS IN A SUPPLY CHAIN

Supply-chain risk management aims primarily at determining the susceptibility of the chain to disruptions, also known as the *supply chain vulnerability* [Kleindorfer and Saad, 2005]. Evaluating the vulnerability of supply chain ecosystems is challenging, because it cannot be observed or measured directly. The failure or overloading of a single node can lead to cascading failures spreading across the whole network. As a result, serious damage will occur within the supply chain system. The analysis of vulnerability must therefore take into account the entire network, evaluating the effect of a disruption of each node. This approach requires identifying nodes that, more than others, represent critical elements in the network.

If the supply chain is represented as a network, as in figure 3.17, several graph algorithms can be applied to identify nodes that expose it to greater vulnerability.

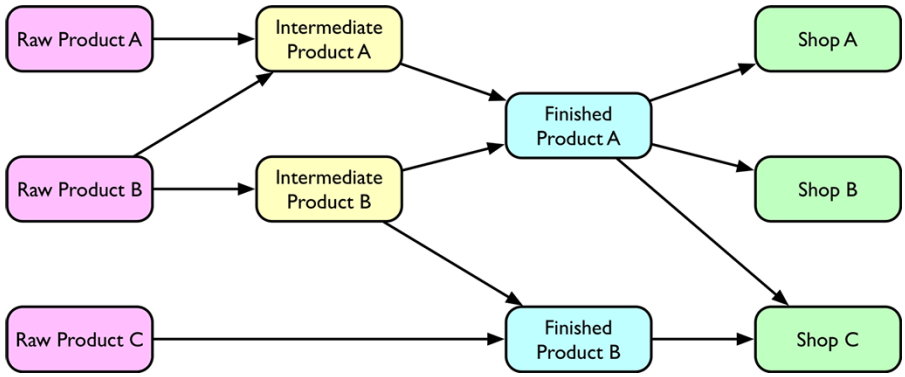


Figure 3.17 **A supply-chain network.**

The purpose of the analysis is to determine the *most important* or *central* nodes in the network. This type of analysis will reveal what the "nodes of interest" are in the supply chain—these are the most likely targets of attack and the nodes that require the most protection, because any disruption of them would gravely affect the entire supply chain and its ability to operate normally. In figure 3.17, for instance, an issue with the supply of raw product B (the disruption could be on the provider's end or in the connection) will affect the entire chain, because it is on the paths to all the shops.

There are many possible definitions of "importance," and consequently many centrality measures that could be used for the network. We'll consider two of them, not only because they're useful for the specific scenario of the supply chain, but also because they are powerful techniques used in multiple examples later in the book:

- *PageRank:* This algorithm works by counting the number and quality of edges to a node to arrive at a rough estimate of the node's importance. The basic idea implemented by the PageRank model, invented by the founders of Google for their search engine, is that of "voting" or "recommendation." When a node is connected to another node by an edge, it's basically casting a vote for that note. The more votes a node receives, the more important it is—but the importance of the "voters" matters too. Hence, the *score* associated with a node is computed based on the votes that are cast for it and the scores of the nodes casting those votes.
- *Betweenness centrality:* This algorithm measures the importance of a node by considering how often it lies on the shortest paths between other nodes. It applies to a wide range of problems in network theory. For example, in a supply chain network, a node with higher betweenness centrality will have more control over the network because more goods will pass through that node.

Figure 3.18 illustrates what these two algorithms look like.

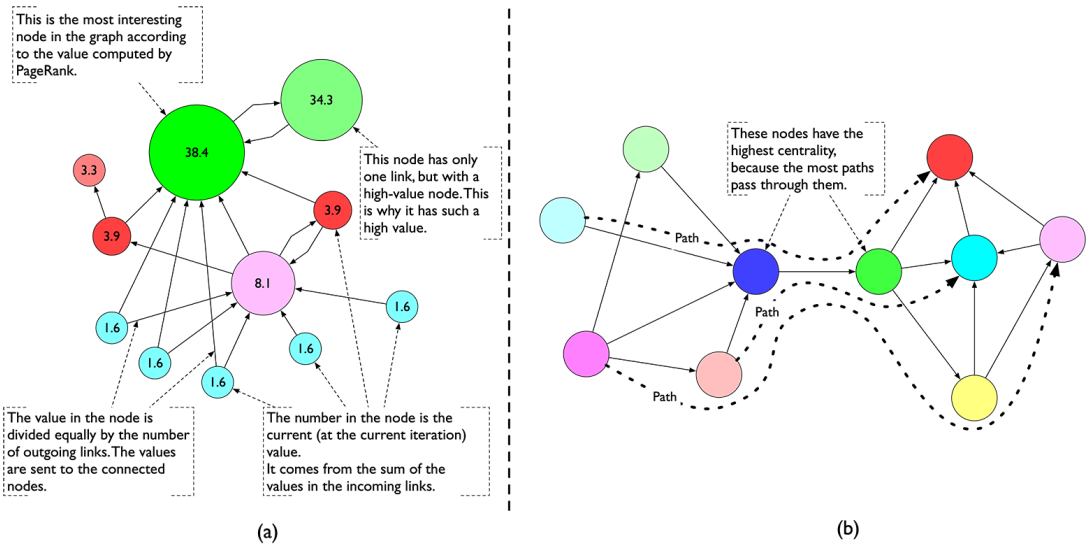


Figure 3.18 PageRank (a) and betweenness centrality (b) examples.

In the supply chain vulnerability use case, both algorithms can be used to determine the most interesting nodes in the supply chain network, but from two different perspectives:

- *Betweenness centrality* allows us to determine which nodes may have considerable influence within the supply chain by virtue of their control over the products passing through them. The nodes with highest centrality are also the ones whose removal from the supply chain network will most disrupt the products' flow, because they lie on the largest number of paths taken by the products. For example, suppose that in the chain there's a single company that's the only provider of a basic component of all the products in the supply chain, or that there's only one company operating as the sole distributor of a particular product. In both cases the greatest number of paths for that component or product pass through them, and any serious disruption of these nodes would affect the entire supply chain.
- *PageRank* allows us to identify nodes that, according to the relative importance of the nodes they're connected with, have a high value in the network. In this case, disrupting an important node might affect only a small portion of the network, but the disruption could still be significant. For example, suppose that in the chain there's a transformation process that converts a product into a form suitable only for one of the biggest end customers of the supply chain. In this case, there aren't many paths passing through the process, so the node's betweenness centrality is quite low, but the "value" of the node is high because disrupting it would affect an important element in the chain.

As these examples illustrate, graph algorithms provide a powerful analysis mechanism for supply-chain networks. This approach can be generalized to many similar scenarios, such as communication networks, social networks, biological networks, or terrorist networks.

Find keywords in a document

> Suppose that you want to identify automatically a set of terms that best describe a document or an entire corpus. Using a graph-based ranking model you can find the most relevant words or phrases in the text, via an unsupervised learning method.

Companies often need to manage and work with large amounts of data, whether to provide services to end users or for internal processes. Most of this data takes the form of text. Because of the unstructured nature of textual data, accessing and analyzing this vast source of knowledge can be a challenging and complex task. *Keywords* can provide effective access to a large corpus of documents by helping to identify the main concepts. Keyword extraction can also be used to build an automatic index for a document collection, to construct domain-specific dictionaries, or for text classification or summarization tasks [Negro et al., 2017].

Multiple techniques, some of them simple and others more complex, can be used to extract a list of keywords from a corpus. The simplest possible approach is to use a relative frequency criterion (identifying the terms that occur most frequently) to select

the "important" keywords in a document—but this method lacks sophistication and typically leads to poor results. Another approach involves using supervised learning methods, where a system is trained to recognize keywords in a text based on lexical and syntactic features—but a lot of labeled data (text with the related keywords manually extracted) is required to train a model accurate enough to produce good results.

Graphs can be your secret weapon to solve a complex problem like this, providing a mechanism to extract keywords or sentences from the text in an *unsupervised* manner by leveraging a graph representation of the data and a graph algorithm such as PageRank. *TextRank* [Mihalcea and Tarau, 2004] is a graph-based ranking model that can be used for this kind of text processing.

In this case, we need to build a graph that represents the text and interconnects words or other text entities with meaningful relations. Depending on the purpose, the text units extracted—keywords, phrases, or entire sentences for summarization—can be added as nodes in the graph. Similarly, it's the final scope that defines the types of relations that are used to connect the nodes (lexical or semantic relations, contextual overlap, and so on). Regardless of the type and characteristics of the elements added to the graph, the application of TextRank to natural language texts consists of the following steps [Mihalcea and Tarau, 2004]:

1. Identify text units relevant to the task at hand and add them as nodes in the graph.
2. Identify relations that connect the text units. The edges between nodes can be directed or undirected and weighted or unweighted.
3. Iterate the graph-based ranking algorithm until convergence or until the maximum number of iterations is reached.
4. Sort the nodes based on their final scores and use these scores for ranking/selection decisions. Eventually, merge them.

The nodes are therefore sequences of one or more lexical units extracted from text, and they're the elements that will be ranked. Any relation that can be defined between two lexical units is a potentially useful connection (edge) that can be added between the nodes. For keyword extraction, one of the most effective ways of identifying relationships is co-occurrence. In this case, two nodes are connected if they both occur within a window of a maximum of N words (an N-gram), with N typically being between 2 and 10. This is another example (maybe one of the most common) of using a co-occurrence graph; an example of the result is shown in figure 3.19. Additionally, it's possible to use syntactic filters to select only lexical units of a certain part of speech (for example, only nouns, verbs, and/or adjectives).

Once the graph has been constructed, the TextRank algorithm can be run on it to identify the most important nodes. Each node in the graph is initially assigned a value of 1, and the algorithm runs until it converges below a given threshold (usually for 20–30 iterations, at a threshold of 0.0001). Once a final score has been determined for each node, the nodes are sorted in reverse order by score and postprocessing is

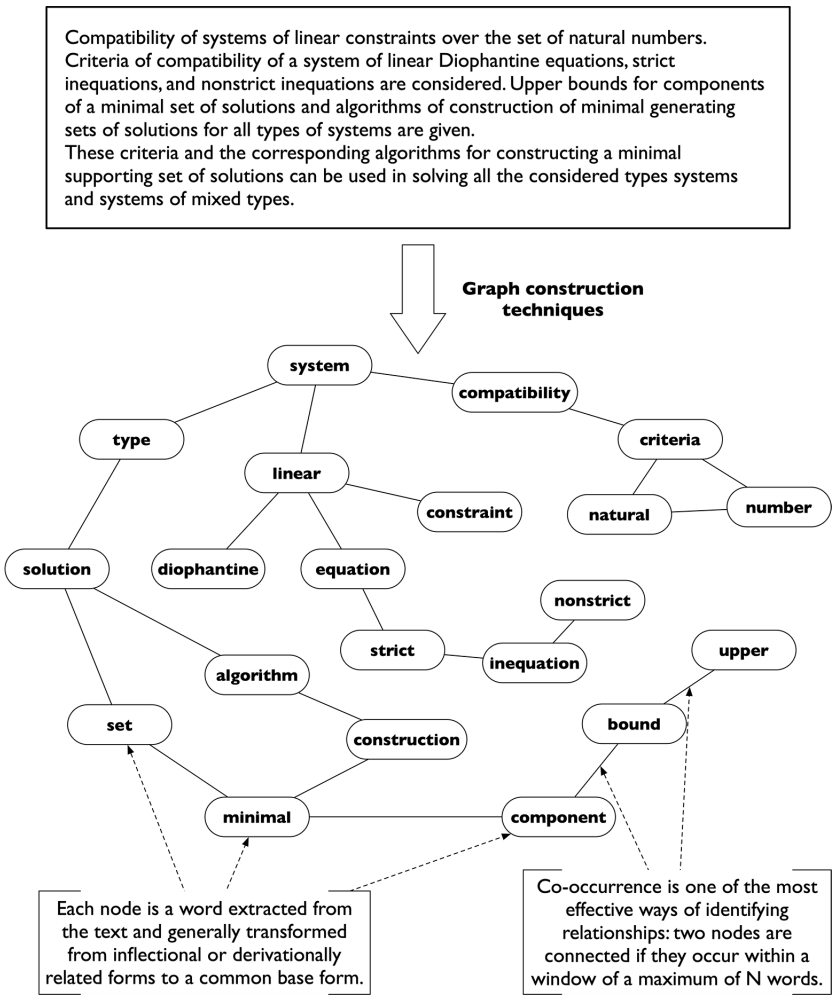


Figure 3.19 **A co-occurrence graph created by TextRank.**

performed on the top T nodes (typically between 5 and 20). During this postprocessing, words that appear one after the other in the text and are both relevant are merged into a single keyword.

The accuracy achieved by this unsupervised graph-based algorithm matches that of any supervised algorithm [Mihalcea and Tarau, 2004]. This indicates that with a graph approach it's possible to avoid the considerable effort supervised algorithms require to provide prelabeled data for a task such as the one described here.

Monitor a subject

Let's continue our discussion of how to monitor a subject's movements using cellular tower data. Earlier in this chapter, we discussed how to convert the data distributed across multiple towers or multiple phones and stored in a tabular format into a homo-

geneous graph called a cellular tower network (shown again here in figure 3.20). As explained in chapter 2, the nodes in the graph that have the highest total edge weight correspond to the towers that are most often visible to the subject's phone [Eagle, Quinn, and Clauset, 2009].

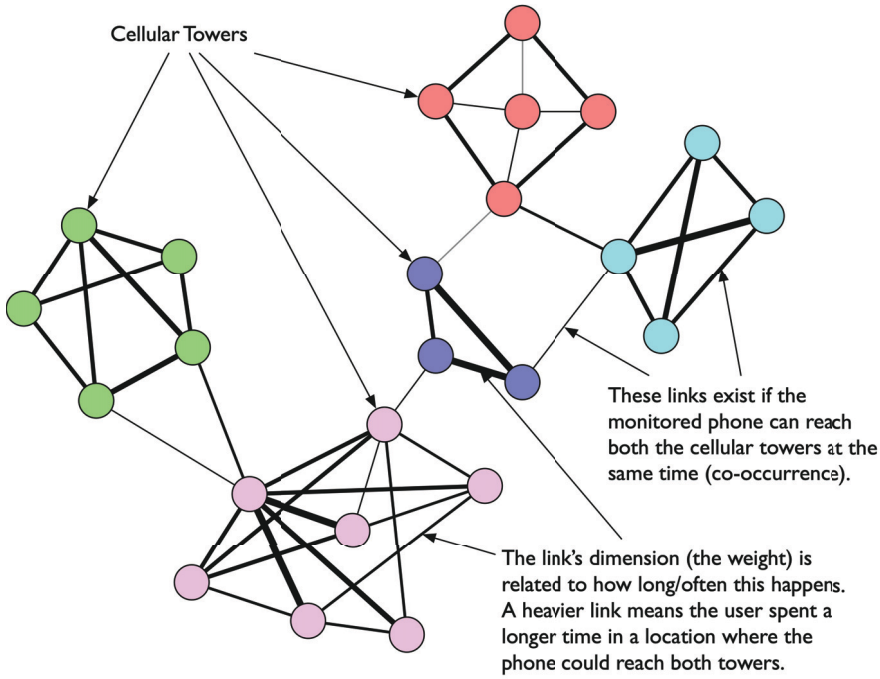


Figure 3.20 A graph representation of the CTN for a single subject.

The graph construction described earlier in this chapter was a preliminary task for using *graph clustering* algorithms that allow us to identify groups of towers. The logic here is that a group of nodes connected to one another by heavily weighted edges and to other nodes by less heavily weighted edges should correspond to a location where the monitored subject spends a significant amount of time. Graph clustering is an unsupervised learning method that aims at grouping the nodes of the graph into clusters, taking into consideration the edge structure of the graph in such a way that there should be many edges within each cluster and relatively few between the clusters [Schaeffer, 2007]. Multiple techniques and algorithms exist for this purpose, and these will be discussed extensively in the second part of the book.

Once the graph is organized into multiple subgraphs that identify locations, the next step is using this information to build a predictive model that can indicate where the subject is likely to go next based on their current position. The clusters of towers identified previously can be incorporated as states of a *dynamic model*.[2] Given a

[2] A dynamic model is used to represent or describe systems whose state changes over time.

sequence of locations visited by a subject, the algorithm learns patterns in the subject's behavior and can calculate the probability of them moving to different future locations. The algorithm used for the modeling here [Eagle, Quinn, and Clauset, 2009] is a dynamic Bayesian network. A simpler approach, the Markov chain, is introduced in the next section.

Whereas in the previous scenario applying the graph algorithm (TextRank) was the main and only necessary action, here, because the problem is more complex, the graph algorithm is used as part of a more articulated learning pipeline to create an advanced prediction model.

3.1.3 Storing and accessing machine learning models

The third step in the workflow involves delivering predictions to end users. The output of the learning phase is a model that contains the result of the inference process and allows us to make predictions about unseen instances. The model has to be stored in permanent storage or in memory so it can be accessed whenever a new prediction is required (figure 3.21). The speed at which we can access the model affects the prediction performance. This is a fundamental aspect of the machine-learning project's definition of success. If the accuracy of the resulting model is high but the prediction rate is low, the system can't accomplish the task in the right way.

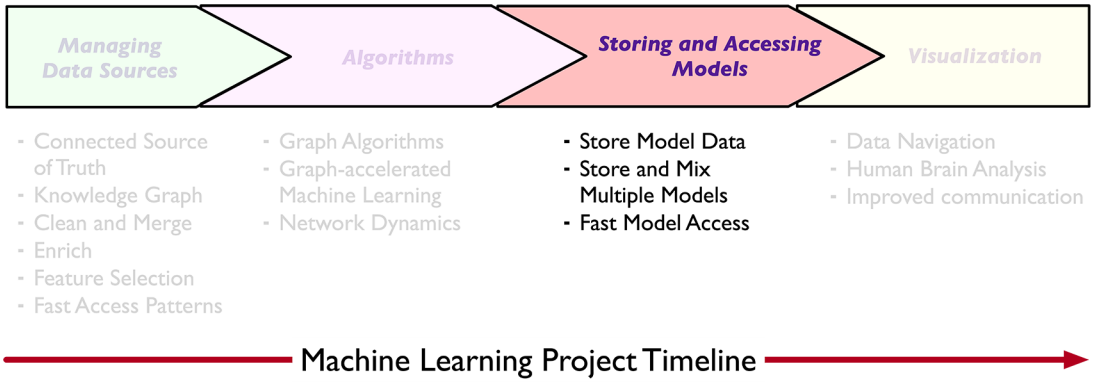


Figure 3.21 Storing and accessing models in the mental model.

For instance, consider the recommendation scenario for an e-commerce site. The user is looking for something but doesn't have a specific idea about what product to buy, so they start their navigation with a text search, then click here and there in the results list, navigating through the several options available. At this point, the system starts recommending items to the user according to the navigation path and the clicks. This is all done in a matter of moments: with a decent network the user navigates quickly, moving from one page to the next every 5 to 10 seconds, or even less.

Therefore, if the recommendation process requires 10 or more seconds, it's completely useless.

This example shows the importance of having a system that can provide predictions quickly to being effective. In this sense, providing fast access to the model is a key aspect of success, and again, graphs can play an important role here.

This section explores, through some explanatory scenarios, the use of graphs for storing prediction models and providing fast access to them.

RECOMMEND ITEMS

The item-based (or user-based) approach to collaborative filtering produces as a result of the learning phase an Item–Item matrix that contains the similarity between each pair of items in the User–Item dataset. The resulting matrix will look like table 3.4.

Table 3.4 Similarity Matrix

	Item 1	Item 2	Item 3	Item 4	Item 5
Item 1	1	0.26	0.84	0	0.25
Item 2	0.26	1	0	0.62	0.25
Item 3	0.84	0	1	0.37	0.66
Item 4	0	0.62	0.37	1	0.57
Item 5	0.25	0.25	066	0.57	1

Having determined the similarities between the items, we can predict a rating for Bob for item 5 by calculating a *weighted sum* of Bob's ratings for the items that are similar to item 5. Formally, we can predict the rating of user u for a product p as follows [Jannach et al., 2010]:

$$pred(\mathrm{u}, \mathrm{p}) = \frac{\sum_{i \in ratedItems(u)} sim(i,p) * r_{u,i}}{\sum_{i \in ratedItems(u)} sim(i,p)}$$

In this formula, the numerator contains the sum of the multiplication of the similarity value of each product that Bob rated to the target product and his rating of that product. The denominator contains only the sum of all the similarity values of the items rated by Bob to the target product.

Let's consider only the line of the User–Item dataset shown in table 3.5 (technically, a slice of the User–Item matrix).

Table 3.5 User–Item Slice for User Bob

User	Item 1	Item 2	Item 3	Item 4	Item 5
Bob	-	3	-	4	?

The preceding formula will appear as follows:

$$pred(\text{Bob}, \text{I5}) = \frac{0.25 * 3 + 0.57 * 4}{0.25 + 0.57} = 3.69$$

The Item–Item similarity matrix from table 3.4 can be stored in the graph easily. Starting from the bipartite graph created for storing the User–Item matrix, it's a simple matter of adding new relationships that connect items to other items (so it won't be a bipartite graph anymore). The weight of the relationship is the value of the similarity, between 0 (in this case, no relationship is stored) and 1. The resulting graph looks like figure 3.22.

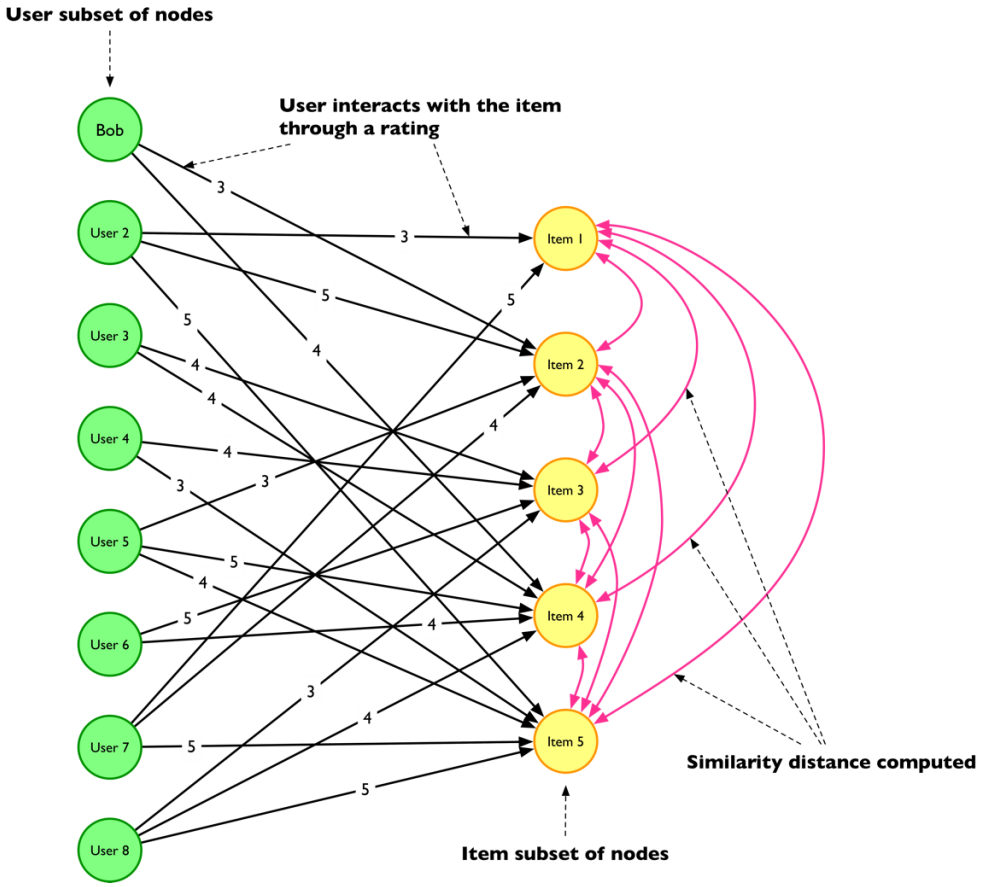


Figure 3.22 Similarity distance model stored in the original bipartite graph.

In this figure, to reduce the number of arcs connecting the nodes, bidirectional relationships between items are represented. In reality, they're two different relationships. Additionally, because the number of relationships is *N* x *N*, it could be quite difficult both in terms of reading and writing to store all the relationships. The typical

approach is to store only the top K relationships for each node. Once all the similarities are computed for each other item, they're sorted in descending order, from the most similar to the least similar, and only the first K are stored. This is because during the computation of the prediction only the top K will be used. Once the data is stored in this way, computing the topmost interesting item for a user is a matter of a few hops in the graph. According to the formula, all the items the target user rated are considered (in our case, items 2 and 4 are connected to the user Bob), and then for each of them the similarity to the target item (item 5) is taken. The information for computing the prediction is local to the user, so making the prediction using the graph model presented is fast. There's no need for long data lookups.

Furthermore, it's possible to store more type of relationships and navigate them at the same time during the prediction to provide combined predictions based on multiple similarity measures. These could be based on approaches other than pure collaborative filtering. We'll discuss other techniques for computing similarity or distance (the same concept, just a different point of view) in the second part.

MONITORING A SUBJECT

In the subject-monitoring scenario, after the identification of clusters of towers that represent locations where the subject spends significant amounts of time, the algorithm continues by learning patterns in the subject's behavior. We can then use dynamic models, like a dynamic Bayesian network, to build a predictive model for subject location.

A Bayesian network is a directed graph in which each node is annotated with quantitative probability information (like 50% or 0.5, 70% or 0.7). The Bayesian network (a.k.a. probabilistic graphical model or belief network) represents a mix of probability theory and graph theory in which dependencies between variables are expressed graphically. The graph not only helps the user to understand which variables affect which other ones, but also enables efficient computing of marginal and conditional probabilities that may be required for inference and learning. The full specification is as follows [Russell and Norvig, 2009]:

Each node corresponds to a random variable. These may be observable quantities, latent variables, unknown parameters, or hypotheses.

Edges represent conditional dependencies. If there's an edge from node X to node Y, X is said to be a *parent* of Y. The graph has no directed cycles (and hence is a *directed acyclic graph*, or DAG). Nodes that aren't connected (where there's no path between the variables in the Bayesian network) represent variables that are conditionally independent of each other.

Each node X_i has a conditional probability distribution $P(X_i, Parents(X_i))$ that quantifies the effect of the parents on the node. In other words, each node is associated with a probability function that takes (as input) a particular set of values for the node's parent variables and gives (as output) the probability, or probability distribution, if applicable, of the variable represented by the node.

To make this clearer, let's consider a simple example [Russell and Norvig, 2009]:

> *You have a new burglar alarm installed at home. It is fairly reliable at detecting a burglary, but also responds on occasion to minor earthquakes. . . . You also have two neighbors, John and Mary, who have promised to call you at work when they hear the alarm. John nearly always calls when he hears the alarm, but sometimes confuses the telephone ringing with the alarm and calls then, too. Mary, on the other hand, likes rather loud music and often misses the alarm altogether. Given the evidence of who has or hasn't called, we would like to estimate the probability of a burglary.*

The related Bayesian network for this example appears in figure 3.23. Burglaries and earthquakes have a direct effect on the probability of the alarm going off, as illustrated by the directed edges that connect the Burglary and Earthquake nodes at the top to the Alarm node. At the bottom, you can see that whether John or Mary calls depends only on the alarm (denoted by the edges connecting Alarm to the JohnCalls and MaryCalls nodes). They don't perceive the burglaries directly, or notice minor earthquakes, and they don't confer with one another before calling.

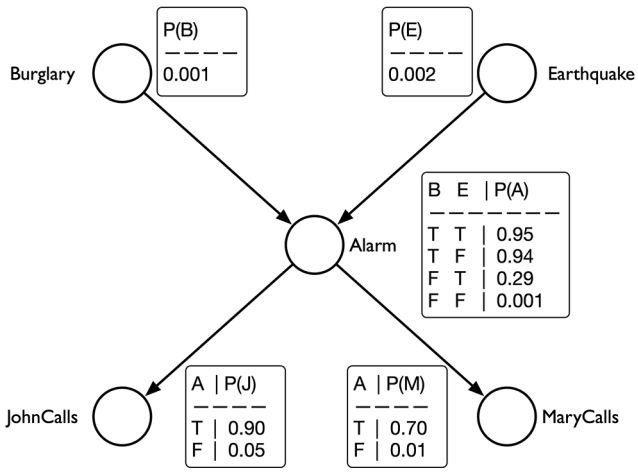


Figure 3.23 A typical Bayesian network, showing both the topology and the conditional probability tables.

In figure 3.23, the tables near each node are the conditional distributions, represented as *conditional probability tables* (CPTs). A conditional distribution is a probability distribution for a subpopulation. Each row in a CPT contains the conditional probability of each node value, given the possible combinations of values for the parent nodes. For instance, $P(B)$ represents the probability of a burglary happening, while $P(E)$ represents the probability of an earthquake happening. These are simple distributions because they don't depend on any other event. $P(J)$, the probability that John will call, and $P(M)$, the probability that Mary will call, depend on the alarm. The CPT for JohnCalls says that if the alarm is going off the probability that John will call is 90%, while the probability him calling when the alarm isn't going off (recall that John can confuse the phone ringing for an alarm) is 5%. A little more complex is the CPT for the Alarm node, which depends on the Burglary and Earthquake nodes. Here,

$P(A)$ (the probability that the alarm will go off) is 95% in the case where a burglary and an earthquake happen at the same time, while it's 94% in the case of a burglary that doesn't coincide with an earthquake and 29% in the case of an earthquake but no burglary. A false alarm is extremely rare, with a probability of only 0.1%.

A dynamic Bayesian network (DBN) is a special type of Bayesian network that relates variables to each other over adjacent time steps. Returning to our subject monitoring scenario, the simplest version of a DBN that can be used for performing a location prediction is a *Markov chain*. The example shown in figure 3.24 is a pure graph. It's a special case of the more general graph representation of a Bayesian network. Nodes in this case represent the status (in our case, the subject's location) at point t, and the weights of the relationships represent the probability of a status transition at time $t + 1$.

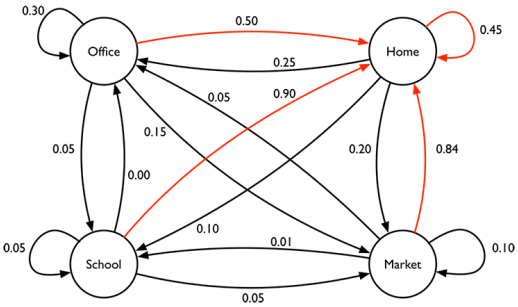


Figure 3.24 **A simple Markov chain (the most probable movement is shown in red).**

In the graph in figure 3.24, if the subject is at *Home* at time t it is most likely that they will stay at home (45%). The probability that they'll move to the *Office* is 25%, while there's a 20% likelihood that they'll instead go to the *Market* and a 10% probability that they'll go to the *School* (perhaps to drop off the kids). This is the representation of a model built using the observations. Starting from this model, computing the probability of a location ?-step-ahead is a path navigation between nodes where each node can appear more times.

This approach can be extended further. Eagle, Quinn, and Clauset [2009] noticed that patterns of movement for people in practice are dependent on the time of day and day of the week (Saturday night versus Monday morning, for example). They therefore created an extended model based on a *contextual Markov chain* (CMC) where the probability of the subject being in a location is also dependent on the hour of the day and the day of the week (which represent the context). The CMC is not described in detail here, but figure 3.25 shows the basic ideas behind it.

The context is created considering the time of day, defined as "morning," "afternoon," "evening," or "night," and the day of the week, split into "weekday" or "weekend." After learning the maximum likelihood parameters, the graphs in figure 3.25 were created. Figure 3.25(a) shows the Markov chain for a morning during the weekend, so no school for the kids, and no office. Figure 3.25(b) shows the Markov chain

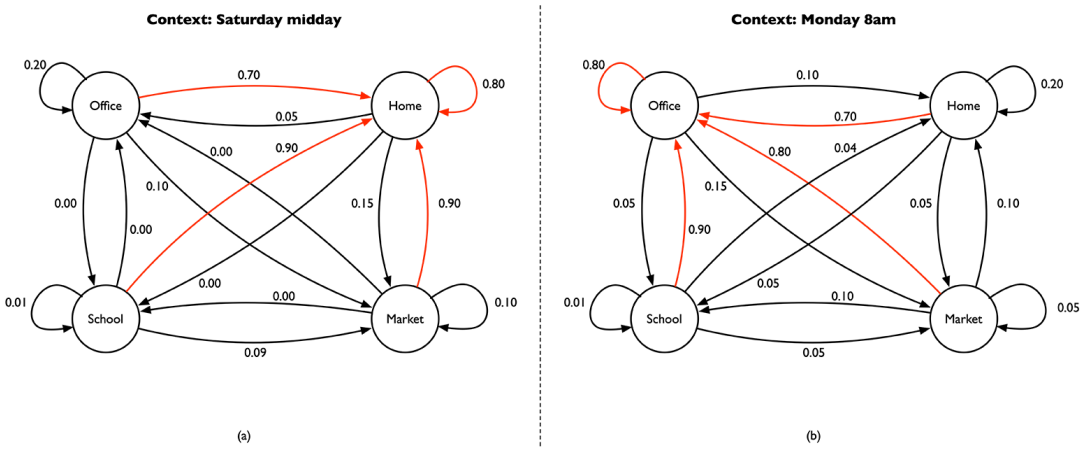


Figure 3.25 **A simple contextual Markov chain for two values of the context: C = {morning, weekend} (a) and C = {morning, weekday} (b).**

related to a morning on a weekday. Such graphs allow us to compute, through a simple query on the graph, a prediction of where the subject is most likely to go next.

Markov chains, contextual Markov chains, and, more generally, [dynamic] Bayesian networks are prediction models that work for many use cases. The subject monitoring scenario is used to illustrate here, but such models are actively used in many kinds of user modeling, and especially in web analysis to predict user intents.

3.1.4 Visualization

One of the main goals of machine learning is to make sense of data and deliver a sort of predictive capability to the end user (although, as described at the beginning of this chapter, data analysis in general aims at extracting knowledge, insights, and finally wisdom from raw data sources, and "prediction" represents a small portion of the possible uses). In this learning path data visualization (figure 3.26) plays a key role, because it

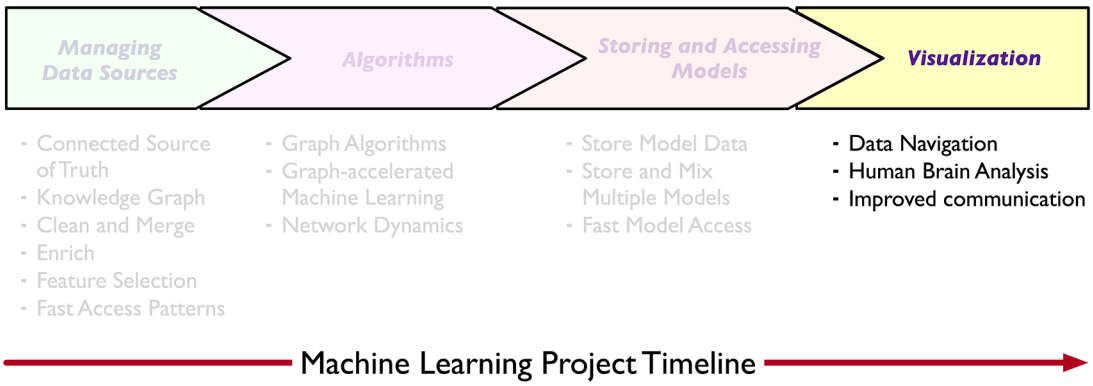


Figure 3.26 **Visualization in the mental model.**

allows us to access and analyze data from a different perspective. In our mental map of the machine-learning workflow, it's presented at the end of the process, because visualizing data after initial processing has been performed is much better than visualizing raw data, but data visualization can happen at any point in the workflow.

In this context, again, the graph approach plays a fundamental role. A growing trend in data analysis is to make sense of linked data as networks. Rather than looking solely at *attributes of the data*, network analysts also focus on the *connections and resulting structures in the data*. If graphs are a helpful way of organizing data to better understand and analyze the relationships contained within that data, visualizations help expose that organization, further simplifying understanding. Combining the two helps data scientists to make sense of the data they have. Furthermore, successful visualizations are deceptive in their simplicity, offering the viewer new insights and understanding at a glance.

Why does visualizing data, specifically in the form of a graph, make it easier to analyze? Here are several reasons:

- Humans are naturally visual creatures. Our eyes are our most powerful sensory receptors and presenting data through information visualizations makes the most of our perceptual abilities [Perer, 2010].
- Many datasets today are simply too large to be inspected without computational tools that facilitate processing and interaction. Data visualizations combine the power of computers and the power of the human mind, capitalizing on our pattern recognition abilities to enable efficient and sophisticated interpretation of the data. If we can see the data in the form of a graph, it's easier to spot patterns, outliers, and gaps [Krebs, 2010; Perer, 2010].
- The graph model exposes relationships that may be hidden in other views of the same data (tables, documents) and helps us pick out the important details [Lanum, 2016].

On the other side, choosing an effective visualization can be a challenge, because different forms have different strengths and weaknesses [Perer, 2010]:

> *Not all information visualizations highlight the patterns, gaps, and outliers important to analysts' tasks, and furthermore, not all information visualizations "force us to notice what we never expected to see" [Tukey, 1977].*

Moreover, visualizing big data requires significant effort in terms of filtering, organizing, and displaying it on a screen. But despite all of these challenges, the graph view remains appealing to researchers in a broad range of areas.

Several good examples of using graph representations to reveal insights into human behavior appear in the work of social network analyst Valdis Krebs. An interesting aspect of his work is that he can take data from any kind of source (old documents, newspapers, databases, or web pages), convert it into a graph representation, perform network analysis, and then visualize the results with his own software, called InFlow. He then analyzes the graph and comes up with some conclusions. One exam-

ple, which we saw in chapter 1, is his analysis of political book purchases on Amazon.com during the US presidential election in 2008 (figure 3.27).

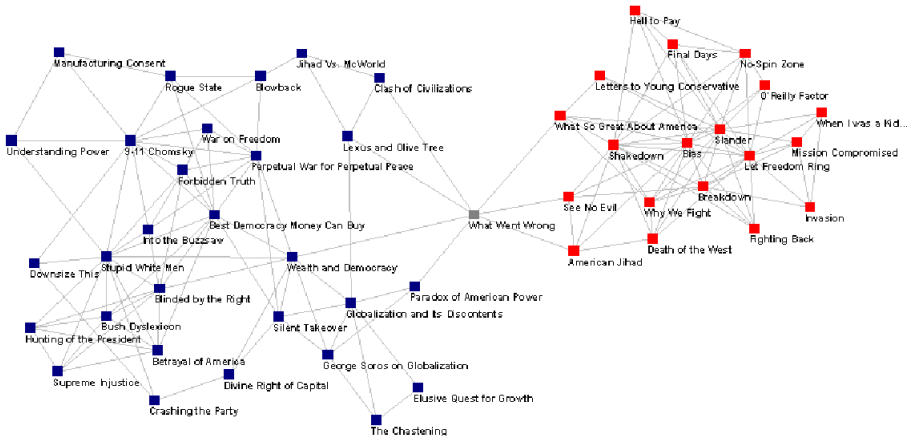


Figure 3.27 The political book networks from the 2008 US presidential election (Krebs, 2010).

Amazon provides summary purchase data that can be used for creating a co-occurrence network (a type of graph we saw earlier in some of our example scenarios). Two books are connected when a customer buys both. The more customers there are who purchase both items, the stronger the association between them is and the higher on the "customers who bought this item also bought" list the associated item appears. Consequently, using Amazon data it's possible to generate a network that provides significant insights into customers' preferences and purchasing behavior. As Krebs puts it, "With a little data mining and some data visualization, we can get great insights into the habits and choices of Amazon's customers—that is, we can come to understand groups of people without knowing about their individual choices."

In figure 3.27, it's possible to recognize two distinct political clusters: a red one designating those who read right-leaning books and a blue one designating those who read left-leaning books. Only one book holds the red and blue clusters together: ironically, that book is named *What Went Wrong*. This graph visualization provides strong evidence of how polarized US citizens were during the political election in 2008. But this "evidence" isn't so evident to a machine learning algorithm, because it requires a lot of contextual information that it's much easier for a human brain to supply, like the political orientation of the book or the book's author, the circumstances of the ongoing political election, and so on.

3.2 Graphs as processing patterns

In many machine learning projects, including many of those described in this book, the graphs that are produced are extremely large. The scale of these graphs makes processing them efficiently difficult. To deal with these challenges, a variety of distrib-

uted graph processing systems have emerged. In this section, we'll explore one of them: *Pregel*, the first computational model (and still one of the most commonly used) for processing large-scale graphs.[3]

This topic suits the purpose of the chapter for two main reasons:

- It defines a processing model that's useful for providing an alternative implementation of several of the algorithms we've discussed (both graph-based and non-graph-based).
- It shows the expressive power of the graph and presents an alternative approach to the computation based on the graph representation of the information.

3.2.1 Pregel

> Suppose that you want to execute the PageRank algorithm on a large graph, such as the whole internet.

As stated earlier in this chapter, the PageRank algorithm was originally developed by the founders of Google for their search engine—so the algorithm's primordial purpose was exactly this. We explored how the algorithm works earlier, so let's focus now on how to solve a concrete problem: processing the PageRank values for such a large graph. This will be a complex task to accomplish due to the high number of nodes (web pages) and edges (links between web pages). It requires a distributed approach.

The input to a Pregel computation is a directed graph in which each node has a unique identifier and is associated with a modifiable, user-defined value that is initialized (this is also part of the input). Each directed edge is associated with:

- A source node identifier
- A target node identifier
- A modifiable, user-defined value

In Pregel the program is expressed as a sequence of iterations (called *supersteps*), separated by global synchronization points, that run until the algorithm terminates and produces its output. In each superstep S, a node can accomplish one or more of the following tasks, conceptually conducted in parallel (Malewicz et al., 2010):

- Receive messages sent to it in the previous iteration, superstep $S - 1$.
- Send messages to other nodes that will be read at superstep $S + 1$.
- Modify its own state and that of its outgoing edges or mutate the graph topology.

Messages are typically sent along outgoing edges (to the directly connected nodes), but a message can be sent to any node whose identifier is known. In superstep 0, every node is *active*; all active nodes participate in the computation of any given superstep. At the end of each iteration a node can decide to deactivate itself by voting to *halt*. At that point it becomes *inactive* and won't participate in subsequent supersteps unless it

[3] Its name honors Leonhard Euler; the bridges of Königsberg, which inspired Euler's theorem, spanned the Pregel River (Malewicz, 2010).

receives a message from another node, at which point it is reactivated. After being reactivated, a node that wishes to halt must explicitly deactivate itself again.

This simple state machine is illustrated in figure 3.28.

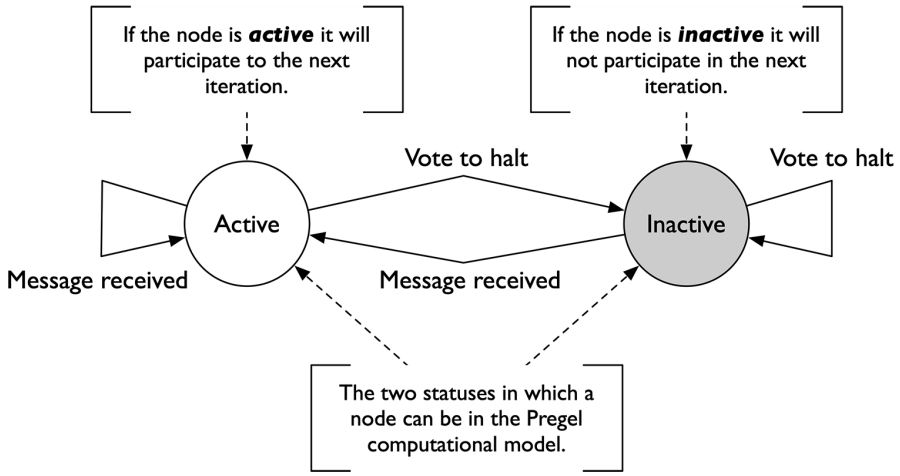


Figure 3.28 **Node statuses according to the Pregel computational model.**

The termination condition for the iterations is reached when all the nodes have voted to halt, so no further work will be done in the next superstep.

Before applying the Pregel framework to our PageRank use case, let's consider a simpler example: *given a strongly connected graph where each node contains a value, find the highest value stored in the nodes.* The Pregel implementation of this algorithm will work in this way:

- The graph and the initial values of each node represent the input.
- At superstep 0, each node sends its initial value to all its neighbors.
- In each subsequent superstep *S*, if a node has learned a larger value from the messages it received in superstep $S - 1$, it sends that value to all its neighbors. Otherwise, it deactivates itself and stops voting.
- Once all nodes have deactivated themselves and there are no further changes, the algorithm terminates.

These steps are shown in figure 3.29, with concrete numbers.
Pregel uses a pure message passing model, for two reasons:

- Message passing is expressive enough for graph algorithms; there's no need for remote reads (reading data from other machines in the processing cluster) or other ways of emulating shared memory.
- By avoiding reading values from remote machines and delivering messages asynchronously in batches it's possible to reduce latency, thereby enhancing performance.

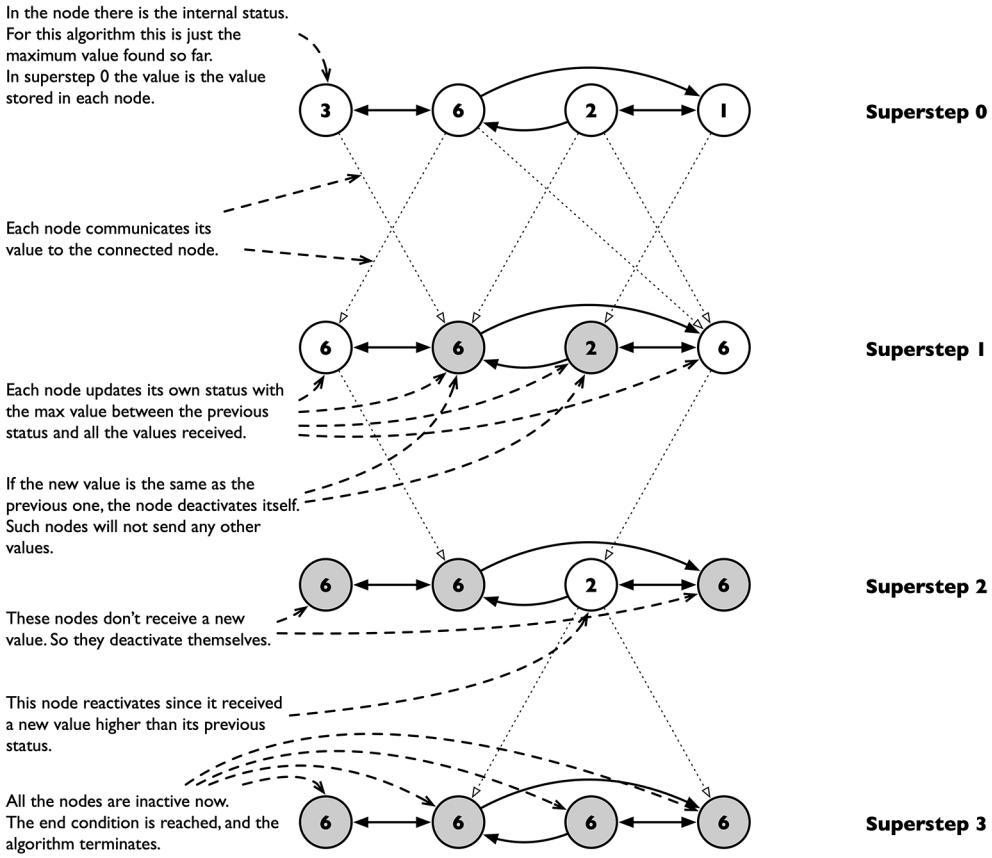


Figure 3.29 **Pregel implementation for finding the highest value stored in the nodes.**

Although Pregel's node-centric model is easy to program and has proven useful for many graph algorithms, it is also worth noting here that such a model hides the partitioning information from users and thus prevents many algorithm-specific optimizations. This often results in longer execution times due to excessive network load. To address this limitation, other approaches exist. This can be defined as a graph-centric programming paradigm. Under this graph-centric model, the partition structure is opened up to the users and can be optimized so that communication within a partition can bypass the heavy message passing [Tian et al., 2013].

Now that the model is clear and the advantages and drawbacks have been highlighted, let's return to our scenario and examine the logical steps of the implementation of the PageRank algorithm using Pregel. It could look like figure 3.30.

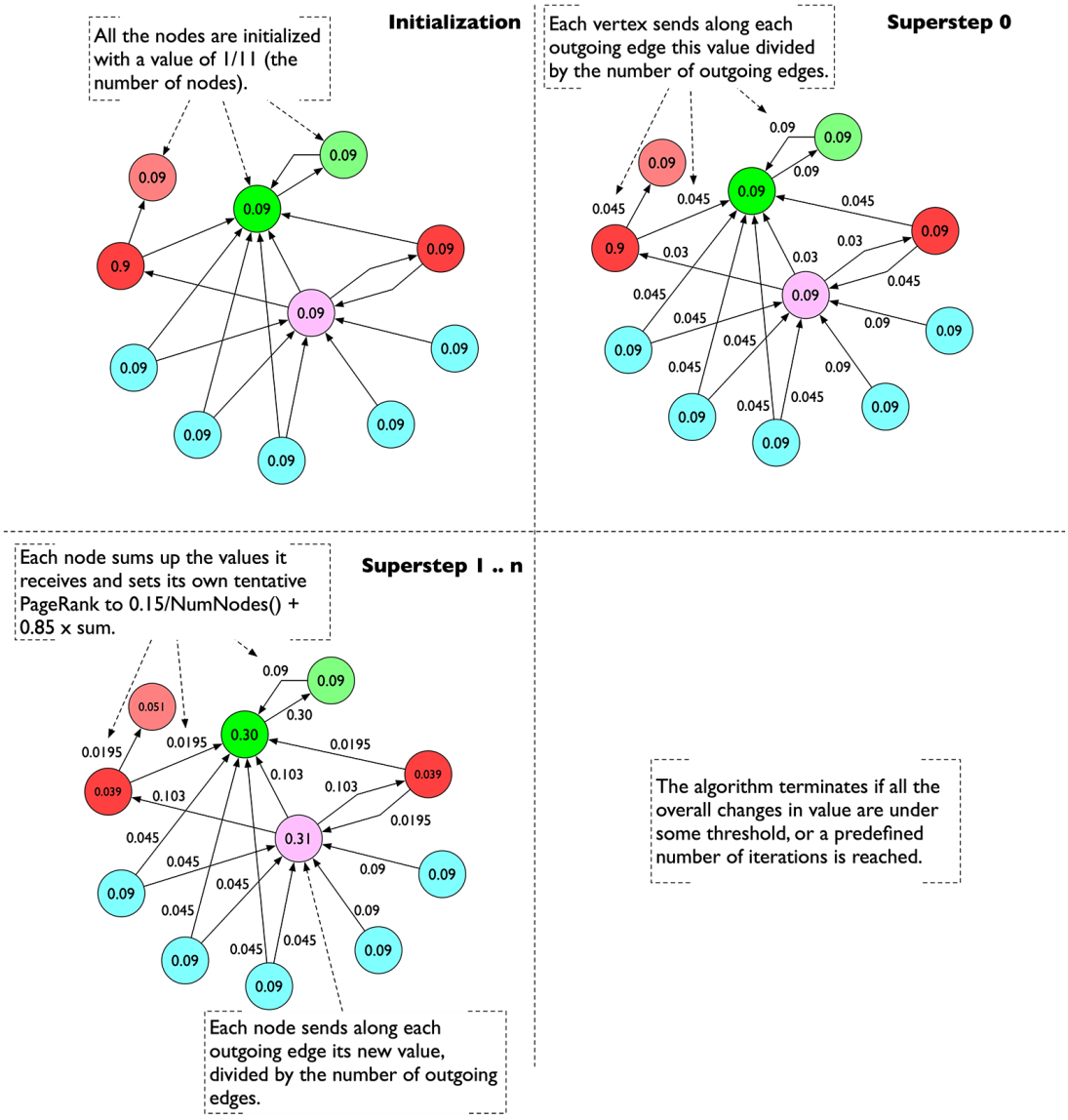


Figure 3.30 PageRank implemented using Pregel framework.

The schema in figure 3.30 can be further described as follows:

- The graph is initialized so that in superstep 0, the value of each node is 1 / NumNodes(). Each node sends along each outgoing edge this value divided by the number of outgoing edges.

- In each subsequent superstep, each node sums up the values arriving in messages into *sum* and sets its own tentative PageRank to 0.15/NumNodes() + 0.85 × *sum*. Then it sends along each outgoing edge its tentative PageRank divided by the number of outgoing edges.
- The algorithm terminates if either all of the overall changes in value are under a threshold or it reaches a predefined number of iterations.

The fun aspect of the Pregel implementation of the PageRank algorithm in the internet scenario is that we have a graph-by-nature dataset (internet links), a pure graph algorithm (PageRank), and a graph-based processing paradigm.

3.3 Graphs for defining complex processing workflows

In a machine learning project graph models can be used not only for representing complex data structures, making it easy to store, process, or access them, but also for describing, in an effective way, complex processing workflows— the sequences of subtasks that are necessary to complete bigger tasks. The graph model allows us to visualize the entire algorithm or application, simplifies the identification of issues, and makes it easy to accomplish parallelization even with an automated process.

Although this specific use of graphs won't be presented extensively in the book, it's important to introduce it because it shows the value of the graph model in representing complex rules or activities in contexts not necessarily related to machine learning.

Dataflow is a programming paradigm that uses directed graphs for representing complex applications; it's extensively used for parallel computing. In a dataflow graph, nodes represent units of computation and edges represent the data consumed or produced by a computation. TensorFlow[4] uses these graphs to represent computation in terms of the dependencies between individual operations.

3.3.1 Dataflow

> Suppose that you're expecting a baby, and during your last visit the doctor predicted the weight of the newborn to be 7.5 pounds. You want to figure out how that might differ from the baby's actual measured weight.

Let's design a function to describe the likelihood of all possible weights of the newborn. For example, you want to know if 8 pounds is more likely than 10 pounds [Shukla, 2018]. For this kind of prediction, the Gaussian (otherwise known as normal) probability distribution function is generally used. It takes as input a number and other parameters, and outputs a nonnegative number describing the probability of observing the input. The probability density of the normal distribution is given by the equation:

$$f(\mathrm{x} \mid \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

[4] https://www.tensorflow.org/

where:

- μ is the mean or expectation of the distribution (and also its median and mode).
- σ is the standard deviation.
- σ^2 is the variance.

This formula specifies how to compute the probability of *x* (the weight in our scenario) considering the median value μ (in our case 7.5 pounds) and the standard deviation (which specifies the variability from the mean). The median value isn't random; it's the actual average value of a newborn in North America.[5] This function can be represented in an XY chart as shown in figure 3.31.

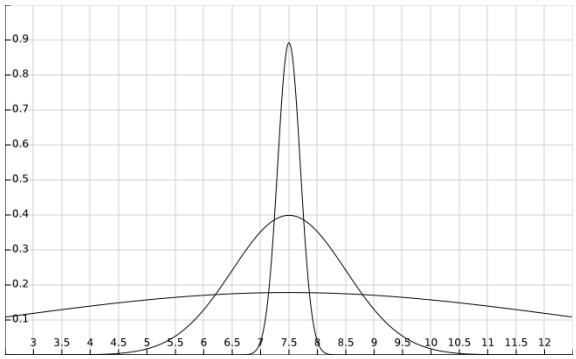


Figure 3.31 **Normal distribution curve (bell curve).**

Depending on the value of σ (the standard deviation) the curve can be taller or fatter, whereas depending on the value of μ (the mean) it can move to the left or right side of the chart. Figure 3.31 has the value of the mean centered to 7.5. According to the value of the standard deviation, the probability of the nearest values could be more or less distributed. The taller curve has a variance of 0.2 while the fatter one has a variance of 5. A smaller value of variance means that the most probable values are the closest to the mean (on both sides).

In any case, the graph presents a similar structure that causes it to be informally called the *bell curve*. This formula and the related representation mean that events closer to the tip of the curve are more likely to happen than events on the sides. In our case, if the mean expected weight of a newborn is 7.5 pounds and the variance is known, by using this function, we can get the probability of a weight of 8 pounds compared with 10 pounds. This function shows up all the time in machine learning, and it's easy to define in TensorFlow; it uses only multiplication, division, negation, and a few other fundamental operators.

[5] https://www.uofmhealth.org/health-library/te6295

To convert such a function into its graph representation in dataflow, it's possible to simplify it by setting the mean to 0 and the standard deviation to 1. With these parameter values the formula becomes:

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$$

This new function has a specific name: the *standard normal distribution.*

The conversion to the graph format requires the following steps:

- Each operator becomes a node in the graph, so we have nodes representing products, power, negation, square root, and so on.
- The edges between operators represent the composition of mathematical functions.

Starting from these simple rules, the resulting graph representation of the Gaussian probability distribution is shown in figure 3.32.

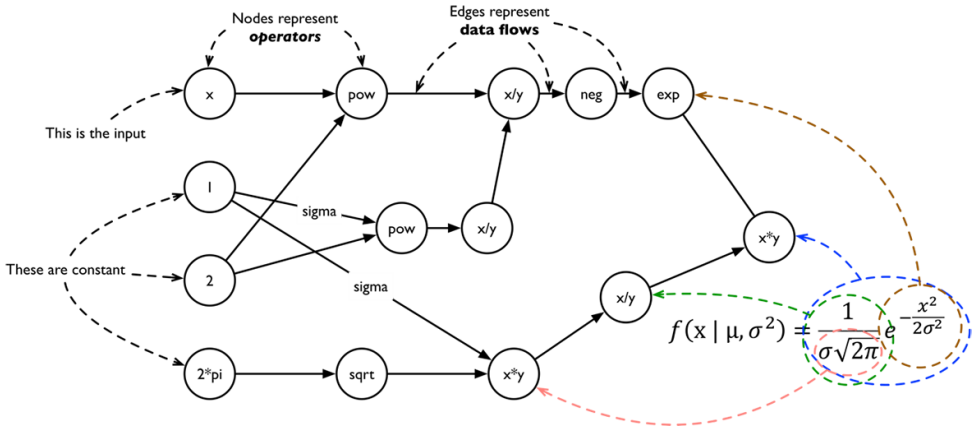


Figure 3.32 A graph representation of the normal distribution in dataflow programming.

Small segments of the graph represent simple mathematical concepts. If a node has only an inbound edge, it's a unary operator (an operator that operates on a single input, such as negation or doubling), while a node with two inbound edges is a binary operator (an operator that operates on two input variables, such as addition or exponentiation). In the graph in figure 3.32, passing 8 pounds (the weight we would like to consider for the newborn) as the input to the formula will provide the probability of this weight. The figure shows the different branches of the function clearly, which means that it's trivial to identify portions of the formula that can be processed in parallel.

In TensorFlow, this approach makes it easy to visualize and process even algorithms that appear to be quite complex. DFP was a commonly forgotten paradigm, despite its usefulness in certain scenarios, but TensorFlow revived it by showing the power of graph representations for complex processes and tasks.

The advantages of the DFP approach can be summarized as follows [Johnston, Hanna, and Millar, 2004; Sousa, 2012]:

- It provides a visual programming language with a simplified interface that enables rapid prototyping and implementation of certain systems. We've already discussed the importance of the visual sense and graphs as a way to better understand complex data structures. DFP is capable of representing complex applications and algorithms while keeping them simple to understand and modify.
- It implicitly achieves concurrency. The original motivation for research into dataflow was the exploitation of massive parallelism. In a dataflow application, internally each node is an independent processing block that works independently from all the others and produces no side effects. Such an execution model allows nodes to execute as soon as data arrives at them, without the risk of creating deadlocks, because there are no data dependencies in the system. This is an important feature of the dataflow model that can greatly increase the performance of an application being executed on a multicore CPU without requiring any additional work of the programmer.

Dataflow applications represent another example of the expressive power of graphs for decomposing complex problems into subtasks that are easy to visualize, modify, and parallelize.

Summary

This chapter presented a comprehensive array of use cases for graphs in machine learning projects. In this chapter you learned:

- How to use graphs and a graph model to manage data. Designing a proper graph model allows multiple data sources to be merged in a single connected and well-organized source of truth. This is useful not only because it creates a single knowledge base—the *knowledge graph*—that can be shared between multiple projects, but also because often it organizes the data in a way that suits the kind of analysis that has to be performed.
- How to process data using graph algorithms. Graph algorithms support a wide spectrum of analysis and can be used either in isolation or as part of a more complex and articulated analytics pipeline.
- How to design a graph that stores the prediction model resulting from training in order to simplify and speed up access during the prediction phase.
- How to visualize data in the form of graphs. Data visualization is a crucial aspect of predictive analysis. A graph can be a pattern for modeling the data so that it can be visualized by an analyst in an efficient and effective way; the human brain can do the rest.
- How to use graphs to manage data processing. Graphs are so expressive that they can be used as the processing model for distributed paradigms like dataflow and Pregel.

References

[Zhao and Silva, 2016] Zhao, Liang, and Thiago Christiano Silva. *Machine Learning in Complex Networks*. New York: Springer, 2016.

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

[Eagle, Quinn and Clauset, 2009] Eagle, Nathan, John A. Quinn, and Aaron Clauset. "Methodologies for Continuous Cellular Tower Data Analysis." *Proceedings of the 7th International Conference on Pervasive Computing* (2009): 342–353.

[Jannach et al., 2010] Jannach, Dietmar, Markus Zanker, Alexander Felfernig, and Gerhard Friedrich. *Recommender Systems: An Introduction*. Cambridge, UK: Cambridge University Press, 2010.

[Frolov and Oseledets, 2016] Frolov, Evgeny, and Ivan Oseledets. "Tensor Methods and Recommendation Engines." GroundAI, March 18, 2016. https://www.groundai.com/project/tensor-methods-and-recommender-systems/.

[Diestel, 2017] Diestel, Reinhard. *Graph Theory*. 5th ed. New York: Springer, 2017.

[Kleindorfer and Saad, 2005] Kleindorfer, Paul R., and Germaine H. Saad. "Managing Disruption Risks in Supply Chains." *Production and Operation Management* 14:1 (2005): 53–68.

[Negro et al., 2005] Negro, Alessandro, Vlasta Kus, Miro Marchi, and Christophe Willemsen. "Efficient Unsupervised Keywords Extraction Using Graphs." GraphAware, October 3, 2017. https://graphaware.com/neo4j/2017/10/03/efficient-unsupervised-topic-extraction-nlp-neo4j.html.

[Mihalcea and Tarau, 2004] Mihalcea, Rada, and Paul Tarau. "TextRank: Bringing Order into Text." *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing* (2004): 404–411.

[Schaeffer, 2007] Schaeffer, Satu Elisa. "Survey: Graph Clustering." *Computer Science Review* 1:1 (2007): 27–64.

[Russel and Norvig, 2009] Russell, Stuart J., and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Upper Saddle River, NJ: Pearson, 2009.

[Krebs, 2010] Krebs, Valdis. "Your Choices Reveal Who You Are: Mining and Visualizing Social Patterns." In *Beautiful Visualization*, edited by Julie Steele and Noah Iliinsky. Sebastopol, CA: O'Reilly, 2010. 103–121.

[Perer, 2010] Perer, Adam. "Finding Beautiful Insights in the Chaos of Social Network Visualizations." In *Beautiful Visualization*, edited by Julie Steele and Noah Iliinsky. Sebastopol, CA: O'Reilly, 2010. 157–173.

[Lanum, 2016] Lanum, Corey L. *Visualizing Graph Data*. Shelter Island, NY: Manning, 2016.

[Tukey, 1977] Tukey, John W. *Exploratory Data Analysis*. Reading, MA: Addison-Wesley, 1977.

[Malewicz, 2010] Malewicz, Grzegorz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Cjajkowski. "Pregel: A System for Large-Scale Graph Processing." *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (2010): 135–146.

[Tian, 2013] Tian, Yuanyuan, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. "From 'Think Like a Vertex' to 'Think Like a Graph.'" *Proceedings of the VLDB Endowment* 7:3 (2013): 193–204.

[Johnston et al., 2004] Johnston, Wesley M., J. R. Paul Hanna, and Richard J. Millar. "Advances in Dataflow Programming Languages." *ACM Computing Surveys (CSUR)* 36:1 (2004): 1–34.

[Sousa, 2012] Sousa, Tiago Boldt. "Dataflow Programming: Concept, Languages and Applications." Doctoral Symposium on Informatics Engineering (2012).

[Shukla, 2018] Shukla, Nishant. *Machine Learning with TensorFlow*. Shelter Island, NY: Manning, 2018.

4 Content-based recommendatsions

This chapter covers

- Presenting an overview of the most common recommendation techniques
- Designing proper graph models for a content-based recommendation engine
- Importing existing (not-graph) datasets in the graph models designed
- Implementing working content-based recommendation engines

Representation is one of the most complex and compelling tasks in machine learning, and computer science in general. Pedro Domingos, a computer science professor at the University of Washington, published an article in 2012 [Domingos, 2012] in which he decomposes machine learning into three main components: representation, evaluation, and optimization.

Representation, specifically, affects three core aspects of a machine-learning project's lifecycle:

- The formal language (or schema) in which a training dataset is expressed before passing it as input to the learning process
- The way in which the result of the learning process—the *predictive model*—is stored
- How, during the prediction phase, the training data and the prediction model are accessed during forecasting

All of these are influenced by the learning algorithm used to infer the generalization from the observed examples in the training dataset, and they affect the overall performance in terms of forecast accuracy and training and prediction performance (speed).

The second part of this book focuses on *data modeling*: the formal structures used to represent the training dataset and the inferred model (the result of the learning process) so that a computer program (the learning agent[6]) can process and access it to provide forecasting or analysis to end users. Hence, two different models are taken into account:

- The *descriptive model* is a simplified representation of reality (the training dataset) created to serve a specific learning purpose. The simplification is based on assumptions about what is and isn't relevant for the specific purpose at hand, or sometimes on constraints on the information available.
- The *predictive model* is a formula for estimating the unknown value of interest: the target. Such a formula could be mathematical, a query on a data structure (a database or a graph, for instance), a logical statement such as a rule, or any combination of these. It represents, in an efficient format, the result of the learning process on the training dataset and it is accessed to perform the actual prediction.

The chapters in this section of the book illustrate graph-based techniques for data modeling that serve both purposes. In certain cases (when the learning algorithm is a graph algorithm), this is a necessary decision. In other cases, the graph approach represents a better option than using a table or other alternatives.

Chapters 2 and 3 presented several modeling examples at a high level, such as using cellular tower networks (CTNs) for monitoring subjects, co-occurrence graphs for finding keywords, and bipartite graphs for representing a user–item dataset in a recommendation engine. Those examples showed the advantages of a graph approach for modeling purposes in those specific scenarios. In this second part of the book, we go into more depth by using four different macro-goals—recommendations, mining text, fraud analytics, and user behavior modeling—the six chapters in this part present in greater detail a selection of modeling techniques and best practices for representing the training dataset, predictive models, and access patterns. Nonetheless, the focus remains on graph modeling techniques rather than on the predictive algorithms themselves. The main purpose is to provide you with the mental tools for rep-

[6] As defined in the first chapter 1, an agent is considered as learning if, after making observations about the world, it can improve its performance on future tasks.

resenting the inputs or outputs of predictive techniques as graphs, showing the intrinsic value of the graph approach. Example scenarios are presented, and whenever possible and appropriate, the design techniques are projected—with the necessary extensions and considerations—into other scenarios where they're relevant.

From now until the end of the book, the chapters are also practical. Real datasets, pieces of code, and queries are introduced and discussed in detail. For each example scenario, datasets are selected, models are designed and ingested, and a predictive model is created and accessed to get forecasts. For queries, we'll use one of the standard graph query languages, Cypher. This SQL-like language began life as a proprietary query mechanism specifically for Neo4j databases, but since 2015 it has been an open standard. Several other companies (such as SAP HANA and Databricks) and projects (such as Apache Spark and RedisGraph) have since adopted it as a query language for graph databases.

No previous knowledge about the Cypher language is required for this chapter, and throughout the book all the queries are described and commented in detail. If you're interested in learning more about it, I recommend looking at the official documentation [Cypher Documentation] and a few books where the topic is described in great detail [Robinson et al., 2015, and Vukotic et al., 2014]. Furthermore, in order to better understand the content of this (and the following) chapter, I recommend you install and configure Neo4j and run the queries. This gives you the opportunity to learn a new query language, play with graphs, and fix the concepts presented here better in your mind. An installation guide is available on the Neo4j developer site. The queries and the code examples have been tested with the version 4.x, the latest available at the time of writing.

The first three chapters of this second part focus on data modeling as applied to recommendation engines. Because this is a common graph-related machine learning topic, several different techniques are introduced in great detail. A generic introduction to recommendation engines is presented in the next section.

4.1 Recommendation engines—an introduction

The term *recommender system* (RS) refers to all software tools and techniques that, by using the knowledge they can gather about the users and items in question and provide suggestions for items that are most likely of interest to a particular user [Ricci et al., 2015]. The suggestions can be related to various decision-making processes, such as what products to buy, which music to listen to, or which films to watch. In this context, *item* is the general term used to identify what the system recommends to users. A recommender system normally focuses on a specific type or class of items, such as books to buy, news articles to read, or hotels to book. The overall design and the techniques used to generate the recommendations are customized to provide useful and relevant suggestions for that specific type of item. Other times, it's possible to use information gathered from a class of items to provide recommendations for other types of items. For example, someone who buys suits might be interested in business books or expensive phones.

Although the main purpose of recommender systems is to help companies sell more items, they also have many advantages from the user's perspective. Users are

continually overwhelmed by choice: what news to read, products to buy, shows to watch, etc. The set of "items" offered by different providers it is growing quickly, and users can no longer sift through all of them. In this sense, recommendation engines provide a "customized" experience, helping people find what they're looking for or what could be of interest to them more quickly. The end result is that user satisfaction will be higher, because they'll get relevant results in a shorter amount of time.

There are various reasons why more and more service providers are exploiting this set of tools and techniques [Ricci et al., 2015], including to:

- *Increase the number of items sold.* This is probably the most important function for an RS: to help a provider sell more items than they would without offering any kind of recommendation. This goal is achieved because the recommended items are likely to suit the user's needs and wants. With the plethora of items (news articles, books, watches, whatever) available in most contexts, users are often flooded by so much information that they cannot find what they're looking for, and the result is that they don't conclude their sessions with a concrete action. The RS represents in this sense a valid help in refining user needs and expectations.
- *Sell more diverse items.* Another important function that an RS can accomplish is to allow the user to select items that might be hard to find without a precise recommendation. For instance, in a tourism RS the service provider might want to promote the places that could be of interest to a particular user in the area, not only the most popular ones. By providing customized suggestions, the provider dramatically reduces the risk of advertising places that are not likely to suit that user's taste. By suggesting or advertising unpopular (in the sense of not so well-known) places to users, the RS can improve the quality of their overall experience in the area and allow new places to be discovered and become popular.
- *Increase user satisfaction.* A properly designed RS improves the user's experience with the application. How many times while navigating an online bookstore such as Amazon have you looked at the recommendations and thought, "wow, that book definitely looks interesting"? If the user finds the recommendations interesting, relevant, and gently suggested by a well-designed front end, they'll enjoy using the system. The killer combination of effective, accurate recommendations and a usable interface will increase the user's subjective evaluation of the system, and most likely they'll come back again. Hence, this approach increases the system usage, the data available for the model building, the quality of the recommendations, and finally the satisfaction of the users.
- *Increase user loyalty.* Websites and other customer-centric applications appreciate and encourage loyalty by recognizing returning customers and treating them as valued visitors. Tracking returning users is a common requirement for RSs (with several exceptions that are discussed later) because the algorithms used leverage the information acquired from the users during previous interactions, such as their ratings of items for making recommendations in the course of the user's next

visit. Consequently, the more often a user interacts with the site or application, the more refined the user's model becomes: the representation of their preferences develops and the effectiveness of the recommender's output is increased.

- *Get a better understanding of what the user wants.* Another important side effect of a properly implemented RS is that it creates a model for the user's preferences, which are either collected explicitly or predicted by the system itself. The service provider can reuse the resulting new knowledge for a number of other goals, such as improving the management of the item's stock or production. For instance, in the travel domain, destination management organizations can decide to advertise a specific region to new customer sectors or use a particular type of promotional message derived by analyzing the data collected by the RS (transactions of the users).

These aspects must be taken into account during the design of a recommendation system, because they affect not only the way in which the system gathers, stores, and processes data, but also the way in which it's used for the predictions. For several of the reasons listed here, if not all, a graph representation of the data and graph-based analysis can play an important role by simplifying data management, mining, communication, and delivery. These aspects will be highlighted throughout this and the following chapters.

It's worth noting here that we're talking about *personalized recommendations*—in other words, every user receives a different list of recommendations depending on their tastes, which are inferred based on previous interactions or information gathered using different methods [Jannach et al., 2010]. The provisioning of personalized recommendations requires that the system know something (or many things) about each user and each item. Hence, the RS must develop and maintain a *user model* (or *user profile*) containing, for instance, data on the user's preferences, as well as an *item model* (or *item profile*) containing information on the item's features or other details.

The creation of user and item models is central to every recommender system. However, the way in which this information is gathered, modeled, and exploited depends on the particular recommendation technique and the related learning algorithms. According to the type of information used to build the models and the approach used to forecast user interests and provide predictions, different types of recommender systems can be implemented.

In this and the next two chapters, four main recommendation techniques are explored. This is only a sampling of the solutions available, but they've been selected to cover a wide spectrum of opportunities and modeling examples. The four approaches we will consider are:

- *Content-based recommendations:* The recommendation engine leverages the availability of (manually created or automatically extracted) item descriptions and user profiles that assign importance to different characteristics. It learns to find items that are similar in content, to the ones that the user liked (interacted with) in the past. A typical example is a news recommender that compares the

articles the user has read previously with the most recent ones available to find items that are similar in terms of content. This is the topic of this chapter.

- *Collaborative filtering:* The basic idea behind collaborative recommendations is that if users have shared the same interests in the past—for instance, if they bought similar books, or watched similar movies—they'll have the same behavior in the future. The most famous example of this approach is Amazon's recommender system, which uses user–item interaction history to provide users with recommendations. This is the topic of chapter 5.
- *Session-based recommendations:* The recommendation engine makes predictions based on session data, such as session clicks and descriptions of clicked items. A session-based approach is useful when user profiles and details of past activities aren't available. It uses information about current user interactions and matches it with other users' previous interactions. An example of this is a travel site that provides details on hotels, villas, and apartments, where users generally don't log in until the end of the process, when it's time to book. In such cases, no history about the user is available. This is one of the topics discussed in chapter 6.
- *Context-aware recommendations:* The recommendation engine generates relevant recommendations by adapting them to the specific context of the user [Adomavicius et al., 2011]. Contextual information could include location, time, or company (who the user is with). For example, many mobile applications use contextual information (location, weather, time, and so on) to refine the recommendations provided to users. This approach is also presented in chapter 6.

This isn't an exhaustive list and not the only classification available. From certain perspectives, session-based and context-aware recommendations might be considered subcategories of collaborative filtering, but it depends on the type of algorithms used to implement them. This list, however, reflects the way in which the different approaches will be described in this book.

Each of these approaches has advantages and disadvantages, which are highlighted in the next sections. *Hybrid* recommendation systems combine different approaches to overcome such issues and provide better recommendations to end users. Hybrid recommendation approaches are also discussed in chapter 6.

4.2 Content-based recommendations

Suppose you want to build a movie recommender system for your local video rental store.

Old-fashioned Blockbuster-style rental shops have largely been put out of business by the advent of new streaming platforms like Netflix, but several still exist here and there. There was one in my town where back when I was at university (a long time ago) that I used to go with my brother every Sunday to rent action movies (keep this preference in mind; it will be useful later!). That's beside the point; the important thing here is that this specific scenario inherently has many peculiarities in common with real, more complex online recommender systems. These include:

- *A small user community.* The number of users or customers is quite small. Most recommendation engines, as we'll discuss later, require there to be many active (in terms of number of interactions, such as views, clicks, or buys) users to be effective.
- *A limited set of well-curated movies.* Each item—in this case a movie—can have many associated details, such as a plot description, a list of keywords, genres, actors, and so on. These "details" aren't always available in other scenarios, where in the extreme case the item has only an identifier.
- *Knowledge of user preferences.* The owner or shop assistant knows the preferences of almost all of the customers, even if they've only rented a few movies or games.

Before moving the discussion on to technicalities and algorithms, take a moment and think about the brick-and-mortar shop. Think about the owner or clerks, and what they do to succeed. They make an effort to get to know their customers, by analyzing their previous rental habits and remembering past conversations with them. They try to create a sort of profile for each customer, containing details on their tastes (horror and action movies rather than romcoms), habits (renting generally on the weekend or during the week), item preferences (movies rather than video games), and so on. They collect information over time to build up this profile and use the mental models they create to welcome each customer in an "effective" way, suggesting something that could be of interest to them, or perhaps sending them a message when a tempting new movie becomes available in the shop.

Now let's consider a "virtual" shop assistant that welcomes a site's users, suggesting movies or games to rent or sending them an email when something new that might be of interest comes into stock. The specific conditions described earlier preclude several approaches to recommendations because they require more data. In the case we're considering (both the real and the simplified virtual shop), a valuable solution is a *content-based recommender system (CBRS)*.

CBRSs rely on item and user descriptions (content) to build item representations (or item profiles) and user profiles to suggest items similar to those a target user has usually already liked in the past (these are also known as *semantics-aware CRBSs*). This approach allows the system to provide recommendations even in cases where the amount of data available is quite small (that is, where limited number of users, items, or interactions exist).

The basic process of producing content-based recommendations consists of matching up the attributes of the target user profile, in which preferences and interests are modeled, with the attributes of the items to find similar items to what the user liked in the past. The result is a *relevance score* that predicts the target user's level of interest in those items. Usually, attributes for describing an item are features extracted from metadata associated with that item or textual features somehow related to the item—descriptions, comments, keywords, and so on. These content-rich items contain a great deal of information by themselves that can be used for making comparisons or inferring a user's interests based on the list of items they've interacted with. For these reasons, content-based recommendation engines don't require much data to be effective.

Figure 4.1 highlights the high-level architecture of a content-based recommender system. It's one of many possible architectures and is the one used in this section.

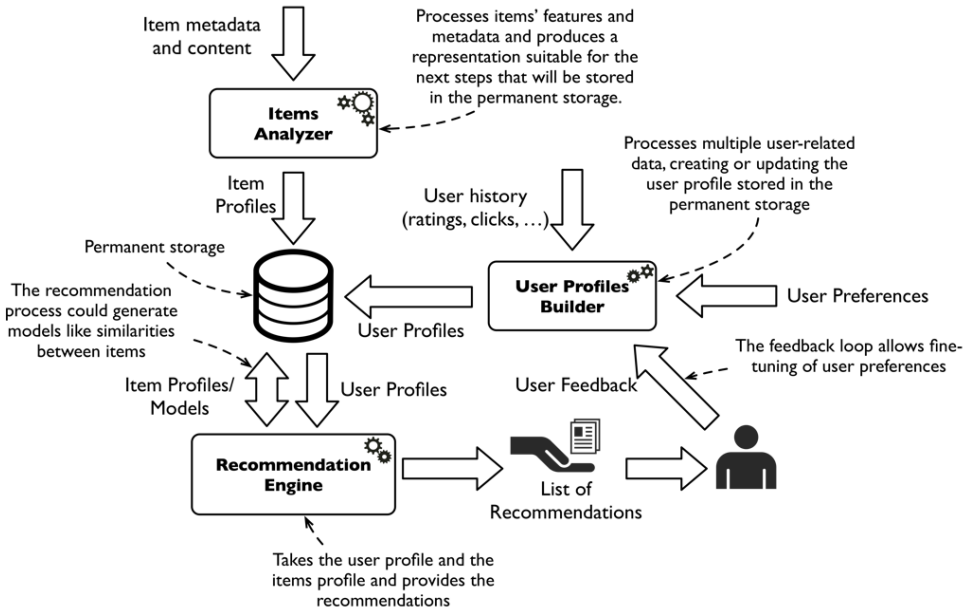


Figure 4.1 High-level architecture of a content-based recommender.

This diagram decomposes the recommendation process into three main components:

- *Items Analyzer:* The main purpose of this component is to analyze items, extract or identify relevant features, and represent the items in a form suitable for the next processing steps. It takes as input the item content (such as the contents of a book or a product description) and metainformation (such as a book's author, the actors in a movie, or movie genres) from one or more information sources and converts them into an item model that's used later for providing recommendations. In the approach described here this conversion produces graph models, which can be of different types. This graph representation is used to feed the recommendation process.
- *User Profiles Builder:* This process collects data representative of the users' preferences and infers user profiles. This may include explicit user preferences gathered by asking users about their interests or implicit feedback collected by observing and storing user behavior. The result is a model—a graph model, specifically—that represents the user's interest in some specific item, item feature, or both. In the architecture in figure 4.1, the item profiles (created during the item analysis stage) and user profiles (created in this stage) converge in the same database. Moreover, because both processes return a graph model their

outputs can be combined into a single, connected and easy-to-access graph model to be used as the input of the next phase.

- *Recommendation Engine:* This module exploits the user profiles and item representations to suggest relevant items by matching the users' interests with item features. In this phase, you build a prediction model and use it to predict for each user a relevancy score for each item. This score is used to rank and order the items to suggest to the user. Certain recommendation algorithms precompute relevant values, for instance item similarities, to make the prediction phase faster. In the approach proposed here, such new values are stored back in the graph, which is, in this way, enriched with other data inferred from the item profiles.

In the following section, each module is described in greater detail. Specifically, I describe how a graph model can be used for representing the item and user profiles that are the outputs of the item analysis and profile building stages, respectively. Such an approach simplifies the recommendation phase, described last.

As in the rest of the chapter, and most of the book from now on, real examples are presented using publicly available datasets and data sources. In this case the MovieLens dataset is used. It contains ratings of movies provided by real users, and it's a standard dataset for recommendation engine tests. However, this dataset doesn't contain much information about the movies, and a content-based recommender requires "content" to work. This is why in our examples it is used in combination with data available on the Internet Movie Database (IMDb),[7] such as plot descriptions and keywords, genres, actors, directors, writers, and so on.

4.2.1 Representing item features

In the content-based approach to recommendation, an item can be represented by a set of *features*, also called *properties* or *attributes*. Features are important or relevant characteristics of that item. In simple cases, such characteristics are easy to discover, extract, or gather. For instance, in the movie recommendation example, each movie can be described using:

- The list of genres (or categories) it belongs to (horror, action, cartoon, drama…)
- A plot description
- A list of actors
- A set of tags or keywords manually (or automatically) assigned to the movie
- The year of production
- The director
- A list of writers
- A list of producers

[7] The IMDb is an online database of information related to films, television programs, home videos, video games, and internet streams, including details on the cast and production crew, plot summaries, trivia, and fan reviews and ratings.

Consider, for example, the information presented in table 4.1 (source: IMDb).

Table 4.1 Some Examples of Movie-Related Data

Title	Genre	Director	Writers	Actors
Pulp Fiction	Action, Crime, Thriller	Quentin Tarantino	Quentin Tarantino, Roger Avary	John Travolta, Samuel Jackson, Bruce Willis, Uma Thurman
The Punisher (2004)	Action, Adventure, Crime, Drama, Thriller	Jonathan Hensleigh	Jonathan Hensleigh, Michael France	Thomas Jane, John Travolta, Samantha Mathis
Kill Bill: Volume 1	Action, Crime, Thriller	Quentin Tarantino	Quentin Tarantino, Uma Thurman	Uma Thurman, Lucy Liu, Vivica A. Fox

Such features are defined generally as *metainformation* because they aren't actually the content of the item. Unfortunately, there are classes of item where it isn't so easy to find or identify features, such as document collections, email messages, news articles, or images.

Text-based items don't tend to have readily available sets of features. Nonetheless, their "content" can be represented by identifying a set of features that describe it. A common approach is the identification of words that characterize the topic. Different techniques exist for accomplishing this task, several of which are described later in this book; the result is a list of features (keywords, tags, relevant words) that describe the content of the item. These features can be used to represent a text-based item in exactly the same way as the metainformation here, so the approach described from now on can be applied either when metainformation features are easily accessible or when features have to be extracted from content.

Extracting tags or features from images is out of the scope of this book, but once those features have been extracted, the approach used is exactly the same as that discussed in this section.

Although representing such a list of features in a graph—more precisely, a *property graph*[8]—is straightforward, several modeling best practices exist that should be taken into account while designing the item model.

Consider, as a simplistic example, the graph model in figure 4.2 of the movies in table 4.1 with their related features.

In this figure the simplest possible representation of the item is used, with the related list of attributes. For each item a single node is created, and the features are modeled as properties of the node. The following query shows the Cypher queries used to create the three movies (run them one at a time).

[8] The property graph, introduced in the chapter 2, organizes data as nodes, relationships, and properties (data stored on the nodes or relationships).

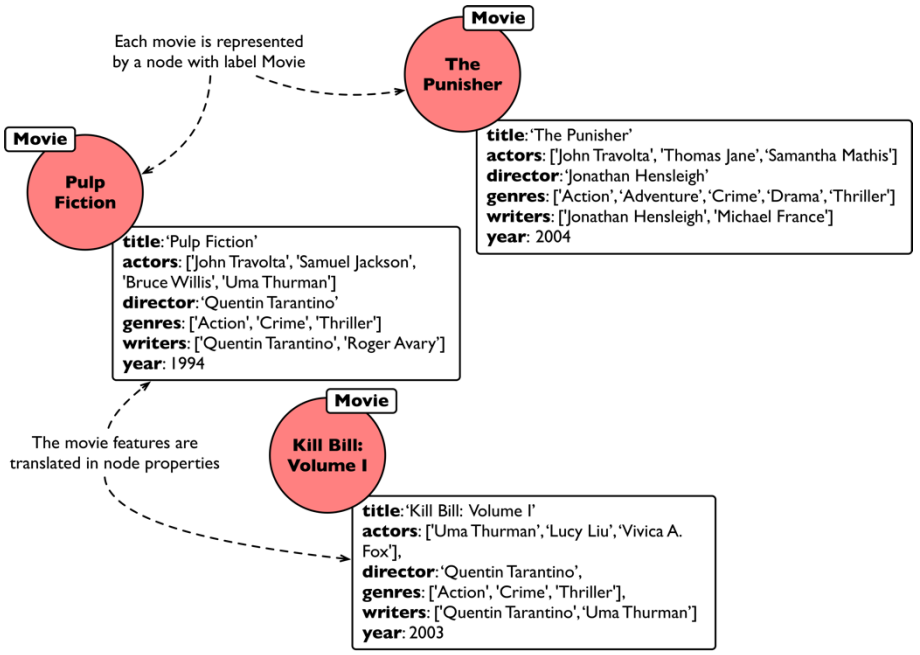


Figure 4.2 Basic graph-based item representation.

Query 4.1 Queries to create a basic model for movie representation

Each CREATE statement creates a new node with "Movie" as the label.

The curly braces define the list of key/value properties of the node, beginning with title.

Properties can be of different types: strings, arrays, integers, doubles, and so on.

The parentheses define the boundaries of the created node instance.

```
CREATE (p:Movie {
 title: 'Pulp Fiction',
 actors: ['John Travolta', 'Samuel Jackson', 'Bruce Willis', 'Uma Thurman'],
 director: 'Quentin Tarantino',
 genres: ['Action', 'Crime', 'Thriller'],
 writers: ['Quentin Tarantino', 'Roger Avary'],
 year: 1994
})

CREATE (t:Movie {
 title: 'The Punisher',
 actors: ['Thomas Jane', 'John Travolta', 'Samantha Mathis'],
 director: 'Jonathan Hensleigh',
 genres: ['Action', 'Adventure', 'Crime, Drama', 'Thriller'],
 writers: ['Jonathan Hensleigh', 'Michael France'],
 year: 2004
})

CREATE (k:Movie {
 title: 'Kill Bill: Volume 1',
 actors: ['Uma Thurman', 'Lucy Liu', 'Vivica A. Fox'],
 director: 'Quentin Tarantino',
 genres: ['Action', 'Crime', 'Thriller'],
 writers: ['Quentin Tarantino', 'Uma Thurman'],
 year: 2003
})
```

In the Cypher queries, `CREATE` allows you to create a new node (or relationship). The parentheses define the boundaries of the created node instances, which in these cases are identified by `p`, `t` and `k` and a specific label, `Movie`, is assigned to each new node. The label specifies the type of a node, or the role the node is playing in the graph. Using labels is not mandatory, but it's a common and useful practice to organize nodes in a graph (and is more performant than assigning a `type` property to each node). Labels are a bit like tables in an old-fashioned relational database, identifying classes of nodes, but in a property graph database there are no constraints on the list of attributes (such as for columns in the relational model). Each node, regardless of the label assigned to it, can contain any set of properties, or even no properties at all. Furthermore, a node can have multiple labels. These two features of the property graph database—no constraints on the list of attributes and multiple labels—make the resulting model quite flexible.

Finally, in the curly braces a set of comma-separated properties is specified.

The single-node design approach has the advantage of a one-to-one mapping between the node and the item with all the relevant attributes. With an effective index configuration, retrieving movies by feature values is quite fast. For example, the Cypher query to retrieve all the movies directed by Quentin Tarantino looks like the following query.

Query 4.2 Query to search for all the movies directed by Quentin Tarantino

The MATCH clause defines the graph pattern to match: a node with the label Movie in this case.

```
MATCH (m:Movie)
WHERE m.director = 'Quentin Tarantino'
RETURN m
```

The WHERE clause defines the filter conditions.

The return clause specifies the list of elements to return.

In this query, the `MATCH` clause is used to define the graph pattern to match. Here, we're looking for all the `Movie` nodes. The `WHERE` clause is part of `MATCH` and adds constraints—filters—to it, as in relational SQL. In the example the query is filtering by director's name. The `RETURN` clause specifies what to return. Figure 4.3 shows the result of running this query from the Neo4j browser.

The simple model just described has multiple drawbacks, including:

- *Data duplication:* In each property, data is duplicated. The same director name, for instance, is duplicated in all the movies with the same director, and the same is true for authors, genres, and so on. Data duplication is an issue in terms of the disk space required by the database and data consistency (how can we know if "Q. Tarantino" is the same as "Quentin Tarantino"?), and it makes change difficult.

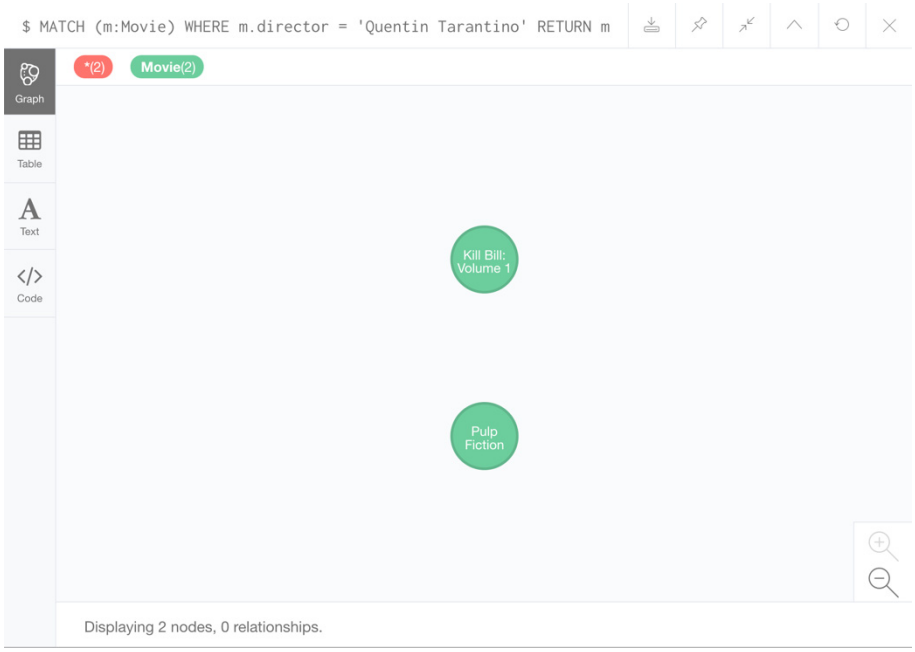


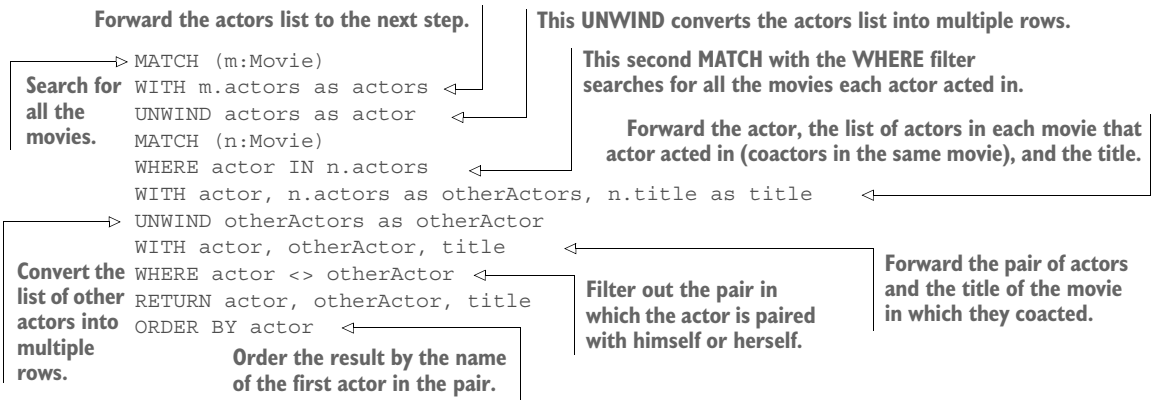
Figure 4.3 Query results from Neo4j browser for the simple model.

- *Error prone:* Particularly during data ingestion, this simple model is subject to issues such as misspelled values or property names. These errors are difficult to identify if the data is isolated in each node.
- *Difficult to extend/enrich:* If during the life of the model an extension is required, like grouping together genres to improve search capabilities or provide semantic analysis, these features are hard to provide.
- *Navigation complexity:* Any access or search is based on value comparison or, even worse, string comparison. Such a model doesn't use the real power of graphs, which enable efficient navigation of relationships and nodes.

To better understand why such a model is poor in terms of navigation and access patterns, suppose you wanted to query for "Actors who worked together in the same movie."

Such a query can be written in the following way.

Query 4.3 Finding all the actors who acted together in the same movie (simple model)



In the previous query:

1. The first `MATCH` searches for all the movies.
2. `WITH` is used to forward the results to the next step. The first one forwards only the `actors` list.
3. With `UNWIND`, you can transform any list back into individual rows. The list of actors in each movie is converted into a sequence of actors.
4. For each actor, the next `MATCH` with the `WHERE` condition finds all the movies they acted in.
5. The second `WITH` forwards the actor considered in this iteration, the list of actors in each movie they acted in, and the movie title.
6. The second `UNWIND` transforms the list of other actors and forwards the actor–other actor pair along with the title of the film they acted in together.
7. The last `WHERE` filters out pairs in which the actor is paired with himself or herself.
8. The query returns the names in each pair and the title of the movie in which both acted.
9. The results are sorted, with the clause `ORDER BY`, by the name of the first actor in the pair.

In such a query all the comparisons are based on string matches, so if a misspelling or a different format (for example, "U. Thurman" instead of "Uma Thurman") exist, the results will be incorrect or incomplete.

Figure 4.4 shows the result of running Query 4.3 on the graph database we created.

A more advanced model for representing items, which is even more useful and powerful for these specific purposes, exposes "recurring" properties as nodes. In this model, each "entity," such as an actor, a director, or a genre, has its own representation—its own node. The relationships between such entities are represented using

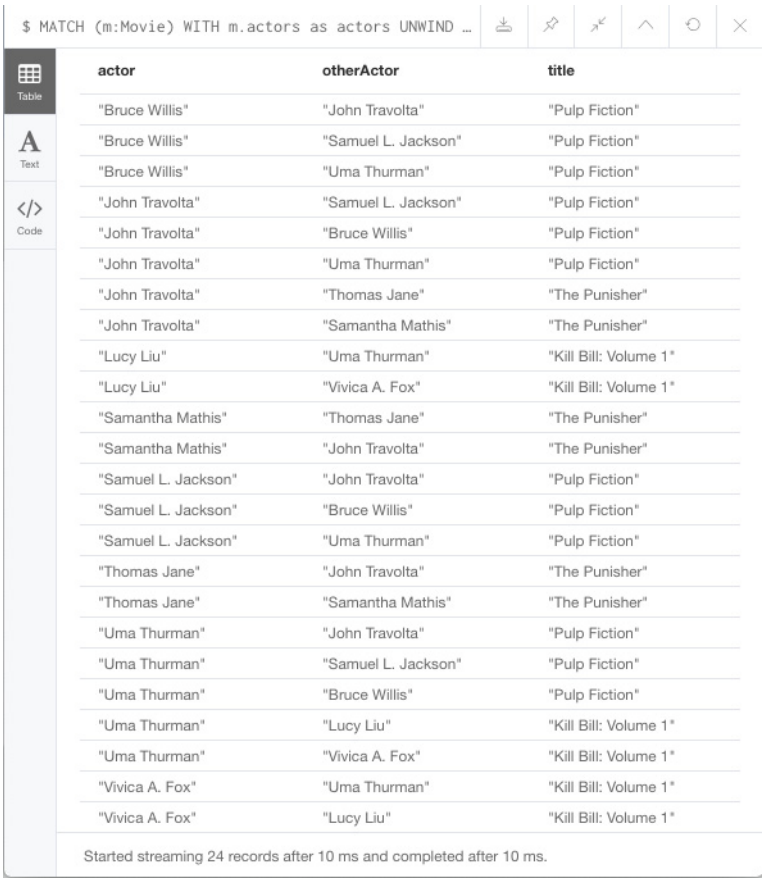


Figure 4.4 **Results from Query 4.3 with the sample database created.**

edges in the graph. The edge can also contain properties to further characterize the relationship.

Figure 4.5 shows what the new model looks like for the movie scenario.

New nodes appear in the advanced model to represent each feature value, while the feature types are specified using labels such as "Genre," "Actor," "Director," and "Writer." Certain nodes can have multiple labels, because they can have multiple roles in the same or different movies. Each node has several properties that describe it, such as `name` for actors and directors and `genre` for genres. The movies now have only the `title` property because this is specific to the item itself; there's no reason to extract it and represent as a separate node.

Figure 4.5 **Advanced graph-based item representation.**

The queries to create this new graph model for the movie example are as shown in the following query.

Query 4.4 Creating an advanced model for movie representation

```
CREATE CONSTRAINT ON (a:Movie) ASSERT a.title IS UNIQUE;
CREATE CONSTRAINT ON (a:Genre) ASSERT a.genre IS UNIQUE;
CREATE CONSTRAINT ON (a:Person) ASSERT a.name IS UNIQUE;

CREATE (pulp:Movie {title: 'Pulp Fiction'})
FOREACH (director IN ['Quentin Tarantino'] |
  MERGE (p:Person {name: director}) SET p:Director MERGE (p)-[:DIRECTED]->(pulp))
FOREACH (actor IN ['John Travolta', 'Samuel L. Jackson', 'Bruce Willis', 'Uma Thurman'] |
  MERGE (p:Person {name: actor}) SET p:Actor MERGE (p)-[:ACTS_IN]->(pulp))
FOREACH (genre IN ['Action', 'Crime', 'Thriller'] |
  MERGE (g:Genre {genre: genre}) MERGE (pulp)-[:HAS]->(g))
```

Each of these statements creates a unique constraint in the database.

Each CREATE creates the movie with only the title as a property.

FOREACH loops over a list and execute the MERGE for each element.

MERGE first checks if the node already exists, using the uniqueness of the director name in this case; if not, it creates the node.

```
| MERGE (p:Person {name: actor}) SET p:Actor MERGE (p)-[:ACTS_IN]->(pulp))
FOREACH (writer IN ['Quentin Tarantino', 'Roger Avary']
| MERGE (p:Person {name: writer}) SET p:Writer MERGE (p)-[:WRITES]->(pulp))
FOREACH (genre IN ['Action', 'Crime', 'Thriller']
| MERGE (g:Genre {genre: genre}) MERGE (pulp)-[:HAS_GENRE]->(g))

CREATE (punisher:Movie {title: 'The Punisher'})
FOREACH (director IN ['Jonathan Hensleigh']
| MERGE (p:Person {name: director}) SET p:Director MERGE (p)-[:DIRECTED]-
    >(punisher))
FOREACH (actor IN ['Thomas Jane', 'John Travolta', 'Samantha Mathis']
| MERGE (p:Person {name: actor}) SET p:Actor MERGE (p)-[:ACTS_IN]-
    >(punisher))
FOREACH (writer IN ['Jonathan Hensleigh', 'Michael France']
| MERGE (p:Person {name: writer}) SET p:Writer MERGE (p)-[:WRITES]-
    >(punisher))
FOREACH (genre IN ['Action', 'Adventure', 'Crime', 'Drama', 'Thriller']
| MERGE (g:Genre {genre: genre}) MERGE (punisher)-[:HAS_GENRE]->(g))

CREATE (bill:Movie {title: 'Kill Bill: Volume 1'})
FOREACH (director IN ['Quentin Tarantino']
| MERGE (p:Person {name: director}) SET p:Director MERGE (p)-[:DIRECTED]-
    >(bill))
FOREACH (actor IN ['Uma Thurman', 'Lucy Liu', 'Vivica A. Fox']
| MERGE (p:Person {name: actor}) SET p:Actor MERGE (p)-[:ACTS_IN]->(bill))
FOREACH (writer IN ['Quentin Tarantino', 'Uma Thurman']
| MERGE (p:Person {name: writer}) SET p:Writer MERGE (p)-[:WRITES]->(bill))
FOREACH (genre IN ['Action', 'Crime', 'Thriller']
| MERGE (g:Genre {genre: genre}) MERGE (bill)-[:HAS_GENRE]->(g))
```

Although graph databases are generally referred to as schemaless, in Neo4j it's possible to define constraints in the database. In this case, the first three queries create three constraints on the uniqueness of the `title` in a `Movie`, the value of a `Genre`, and the name of a `Person`, respectively. This will avoid, for instance, having the same person (actor, director, or writer) appear several times in the database. As described previously, in the new model the idea is to have a single entity represented by a single node in the database. The constraints help to enforce this modeling decision.

After the constraint creation, the clause (repeated three times, once for each movie in the example) works as before to create each new `Movie` with the `title` as a property. Then, the `FOREACH` clauses loop over directors, actors, writers, and genres, respectively, and for each element search for a node to connect to the `Movie` node, creating a new node if necessary. In the case of actors, writers, and directors, a generic node with label `Person` is created using a `MERGE` clause. `MERGE` ensures that the supplied pattern exists in the graph, either by reusing existing nodes and relationships that match the supplied predicates or by creating new nodes and relationships. The `SET` clause, in this case, assigns a new specific label to the node, depending on the needs. The `MERGE` in the `FOREACH` checks for (and creates if necessary) the relationship between the `Person` and the `Movie`. A similar approach has been used for genres.

The overall result is shown in figure 4.6.

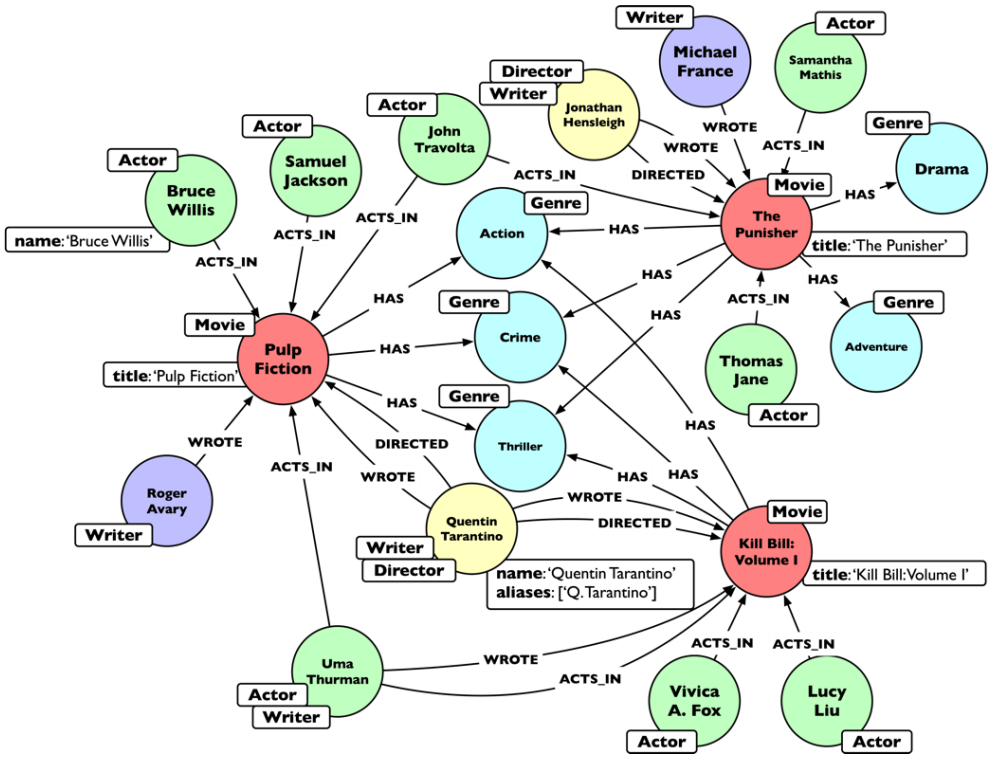


Figure 4.6 **Advanced graph-based item representation for the three movies.**

Modeling pro tip

You can use multiple labels for the same node. In this case this is both useful and necessary because in the model we want to have each person represented uniquely regardless of the role they play in the movie (actor, writer, or director). This is why we opt for `MERGE` instead of `CREATE`, and we use a common label for all of them. At the same time, the graph model assigns a specific label for each role the person has. Once assigned, it becomes assigned to the node, so it will be easier and more performant to run queries such as "find me all the producers that. . . ."

The new descriptive model not only solves all the issues described earlier, but also provides multiple advantages:

- *No data duplication:* Mapping each relevant entity (person, genre, and so on) to a specific node avoids data duplication. The same entity can play different roles and have different relationships (for example, Uma Thurman isn't only an actress in *Kill Bill: Volume I* but also one of the writers). Moreover, for each item a list of alternative forms or aliases can be stored (for example, "Q. Tarantino," "Director Tarantino," "Quentin Tarantino"). This helps with searches, and it avoids the same concept being represented in multiple nodes.

- *Error tolerant:* Avoiding data duplication guarantees a better tolerance to errors in the values. Unlike in the previous model, where a misspelled value is hard to spot because it's distributed among all the nodes as a property, here the information is centralized in isolated and not-replicated entities, making errors easy to identify.
- *Easy to extend/enrich:* Entities can be grouped together by using a common label or creating a new node and connecting it to the related nodes. This can improve the query performance or style. For instance, we can connect multiple genres, such as *Crime* and *Thriller*, under a common *Drama* node.
- *Easy to navigate:* Each node and even each relationship can be the entry point of the navigation (actors, genres, directors, and so on), whereas in the previous schema the only available entry points were the features in the nodes. This enables multiple and more efficient access patterns to the data.

Consider again the example query from earlier, for "Actors who worked together in the same movie." In the new model constructing this query is much easier, as you can see in the following query.

Query 4.5 Finding all the actors who acted together in the same movie (advanced model)

In this case, the MATCH clause specifies a more complex graph pattern.

```
MATCH (actor:Actor)-[:ACTS_IN]->(movie:Movie)<-[:ACTS_IN]-(otherActor:Actor)
WHERE actor <> otherActor
RETURN actor.name as actor, otherActor.name as otherActor,
movie.title as title
ORDER BY actor
```

The identity pair is removed.

Query 4.5 produces exactly the same results as query 4.3, but it's much simpler, clearer, and even faster. This is evidently a better use of the `MATCH` clause. Here, instead of describing a single node it describes the entire graph pattern we're looking for: we're looking for two actors who worked on the same movie, `movie`, and the `WHERE` filters out the original actor. The result is shown in figure 4.7.

It's worth noting that, unlike before, here we have no string comparison. Furthermore, the query is much simpler, and on a bigger database it will execute faster. If you recall our discussion about the native graph database and how Neo4j implements adjacency-free indexes for node relationships, it will be much faster than the index lookups on strings that are necessary in listing 4.3 instead.

We've now designed our final graph-based model for representing items. In a real machine-learning project, the next step would be to create the database, importing data from one or more sources. As stated at the beginning of this section, the MovieLens dataset has been selected as a testing dataset. You can download the dataset from GroupLens. Depending on how long you're willing to wait to see a first graph database, you can choose a suitable dataset size (if you're really impatient, choose the smallest). The dataset contains only a little information about each movie, such as the

```
$ MATCH (a:Actor)-[:ACTS_IN]->(m)<-[:ACTS_IN]-(o:Actor) WHERE a <> o RETURN a.name as…
```

Table | Text | Code

actor	otherActor	title
"Bruce Willis"	"John Travolta"	"Pulp Fiction"
"Bruce Willis"	"Samuel L. Jackson"	"Pulp Fiction"
"Bruce Willis"	"Uma Thurman"	"Pulp Fiction"
"John Travolta"	"Samuel L. Jackson"	"Pulp Fiction"
"John Travolta"	"Bruce Willis"	"Pulp Fiction"
"John Travolta"	"Uma Thurman"	"Pulp Fiction"
"John Travolta"	"Thomas Jane"	"The Punisher"
"John Travolta"	"Samantha Mathis"	"The Punisher"
"Lucy Liu"	"Uma Thurman"	"Kill Bill: Volume 1"
"Lucy Liu"	"Vivica A. Fox"	"Kill Bill: Volume 1"
"Samantha Mathis"	"John Travolta"	"The Punisher"
"Samantha Mathis"	"Thomas Jane"	"The Punisher"
"Samuel L. Jackson"	"John Travolta"	"Pulp Fiction"
"Samuel L. Jackson"	"Bruce Willis"	"Pulp Fiction"
"Samuel L. Jackson"	"Uma Thurman"	"Pulp Fiction"
"Thomas Jane"	"John Travolta"	"The Punisher"

Started streaming 24 records after 6 ms and completed after 6 ms.

Figure 4.7 Results from listing 4.5 with the sample database created.

title and a list of genres, but it also contains a reference to its IMDb ID, where it's possible to access all sorts of details about the movie: plot, directors, actors, writers, and so on. This is exactly what we need.

The following two code listings contain the Python code necessary for reading the data from the MovieLens dataset, storing the first nodes in the graph, then enriching them using the information available on IMDb.

Listing 4.1 Importing basic movie information from MovieLens

```
def import_movies(self, file):
    with open(file, 'r+') as in_file:
        reader = csv.reader(in_file, delimiter=',')
        next(reader, None)
        with self._driver.session() as session:
            self.executeNoException(session,
                "CREATE CONSTRAINT ON (a:Movie) ASSERT a.movieId IS UNIQUE; ")
            self.executeNoException(session,
                "CREATE CONSTRAINT ON (a:Genre) ASSERT a.genre IS UNIQUE; ")

            tx = session.begin_transaction()

            i = 0;
            j = 0;
            for row in reader:
                try:
                    if row:
                        movie_id = strip(row[0])
                        title = strip(row[1])
                        genres = strip(row[2])
```

Reading the values from a CSV file (movies.csv).

Starting a new session connecting to Neo4j.

Creating contraints to guarantee the uniqeness of people and genres. The function executeNoException wrap the exception generated if the constraint already exist.

Beginning a new transaction, which will allow the atomicity (all in or all out) of the operations on the database.

Creating the movie and the genres (the merge avoids creating the same genre multiple times) and connecting them.

```
            query = """
                CREATE (movie:Movie {movieId: $movieId, title:
                  $title})
                with movie
                UNWIND $genres as genre
                MERGE (g:Genre {genre: genre})
                MERGE (movie)-[:HAS_GENRE]->(g)
            """
            tx.run(query, {"movieId": movie_id, "title": title,
                "genres": genres.split("|")})
            i += 1
            j += 1

            if i == 1000:
            tx.commit()
            print(j, "lines processed")
            i = 0
            tx = session.begin_transaction()
        except Exception as e:
            print(e, row, reader.line_num)
    tx.commit()
    print(j, "lines processed")
```

Pro tip: To avoid a huge commit at the end, this check ensures the commit to the database happens for every 1,000 lines processed.

Listing 4.2 Enriching the database with details available on IMDb

```
def import_movie_details(self, file):
    with open(file, 'r+') as in_file:
        reader = csv.reader(in_file, delimiter=',')
        next(reader, None)
        with self._driver.session() as session:
            self.executeNoException(session, "CREATE CONSTRAINT ON (a:Person)
                ASSERT a.name IS UNIQUE;")
            tx = session.begin_transaction()
            i = 0;
            j = 0;
            for row in reader:
                try:
                    if row:
                        movie_id = strip(row[0])
                        imdb_id = strip(row[1])
                        movie = self._ia.get_movie(imdb_id)
                        self.process_movie_info(movie_info=movie, tx=tx,
                            movie_id=movie_id)
                        i += 1
                        j += 1
                    if i == 10:
                        tx.commit()
                        print(j, "lines processed")
                        i = 0
                        tx = session.begin_transaction()
```

Creating a new constraint to make people unique.

Getting movie details from IMDb.

Processing information from IMDb and storing it in the graph.

```
                except Exception as e:
                    print(e, row, reader.line_num)
            tx.commit()
            print(j, "lines processed")

    def process_movie_info(self, tx, movie_info, movie_id):
        query = """
            MATCH (movie:Movie {movieId: $movieId})
            SET movie.plot = $plot
            FOREACH (director IN $directors | MERGE (d:Person {name: director})
                   SET d:Director MERGE (d)-[:DIRECTED]->(movie))
            FOREACH (actor IN $actors | MERGE (d:Person {name: actor}) SET
                   d:Actor MERGE (d)-[:ACTS_IN]->(movie))
            FOREACH (producer IN $producers | MERGE (d:Person {name: producer})
                   SET d:Producer MERGE (d)-[:PRODUCES]->(movie))
            FOREACH (writer IN $writers | MERGE (d:Person {name: writer}) SET
                   d:Writer MERGE (d)-[:WRITES]->(movie))
            FOREACH (genre IN $genres | MERGE (g:Genre {genre: genre}) MERGE
                   (movie)-[:HAS_GENRE]->(g))
        """
        directors = []
        for director in movie_info['directors']:
            if 'name' in director.data:
                directors.append(director['name'])

        genres = ''
        if 'genres' in movie_info:
            genres = movie_info['genres']

        actors = []
        for actor in movie_info['cast']:
            if 'name' in actor.data:
                actors.append(actor['name'])

        writers = []
        for writer in movie_info['writers']:
            if 'name' in writer.data:
                writers.append(writer['name'])

        producers = []
        for producer in movie_info['producers']:
            producers.append(producer['name'])

        plot = ''
        if 'plot outline' in movie_info:
            plot = movie_info['plot outline']

        tx.run(query, {"movieId": movie_id, "directors": directors, "genres":
                   genres, "actors": actors, "plot": plot,
                       "writers": writers, "producers": producers})
```

Same as Query 4.4, except here the movie already exists

Take the plot value from the movie info to create a plot property on the nodet.

This code is oversimplified, and it takes ages to complete because accessing and parsing IMDb pages requires time. In the book's code repository, in addition to the complete implementation of the code there's also a "parallel" version of the function

`import_movie_details`, where multiple threads are created to download and process several IMDb pages at the same time. After it completes, the resulting graph has the structure described in Figure 4.6.

EXERCISES

Play with the newly created database and write queries to:

1. Search for pairs of actors who worked on the same movie. Tip: Use Query 4.3 but add `LIMIT 50` at the end of the query; otherwise it will produce a lot of results.
2. Count, for each actor, how many movies they acted in.
3. Get a movie (by `movieId`) and list all the features.

The items (movies, in this scenario) are now properly modeled and stored in a real graph database. In the next section, we're going to model the users.

4.2.2 User modeling

In a content-based recommender system, several methods exist to gather and model user profiles. The selected design model will vary according to how the preferences are collected (implicitly or explicitly) and the type of filtering strategy, or recommendation approach. A straightforward way of collecting user preferences is by asking the user. For example, the user might express interest in specific genres or keywords, or particular actors or directors.

From a high-level perspective, the purpose of the user profile and the defined model is to help the recommendation engine to assign a score to each item or item feature. The score helps to rank the items suggested to the specific user—they're ordered from high to low. This is why recommender systems belong to the area of machine learning called *learning to rank*.

We can add preferences or interests to the model we are designing by adding nodes for users and connecting them to the features of interest. The resulting schema will look like figure 4.8.

The graph model defined for modeling user preferences extends the model previously described for the items, adding a new node for each user and connecting it to the "features" of interest for the user.

Modeling note

The advanced model designed for items fits better in this scenario because the features are nodes in the graph and so can be connected to the users by edges. This is another advantage of such a model in comparison with the simpler one, in which modeling interests would have been much harder and more painful.

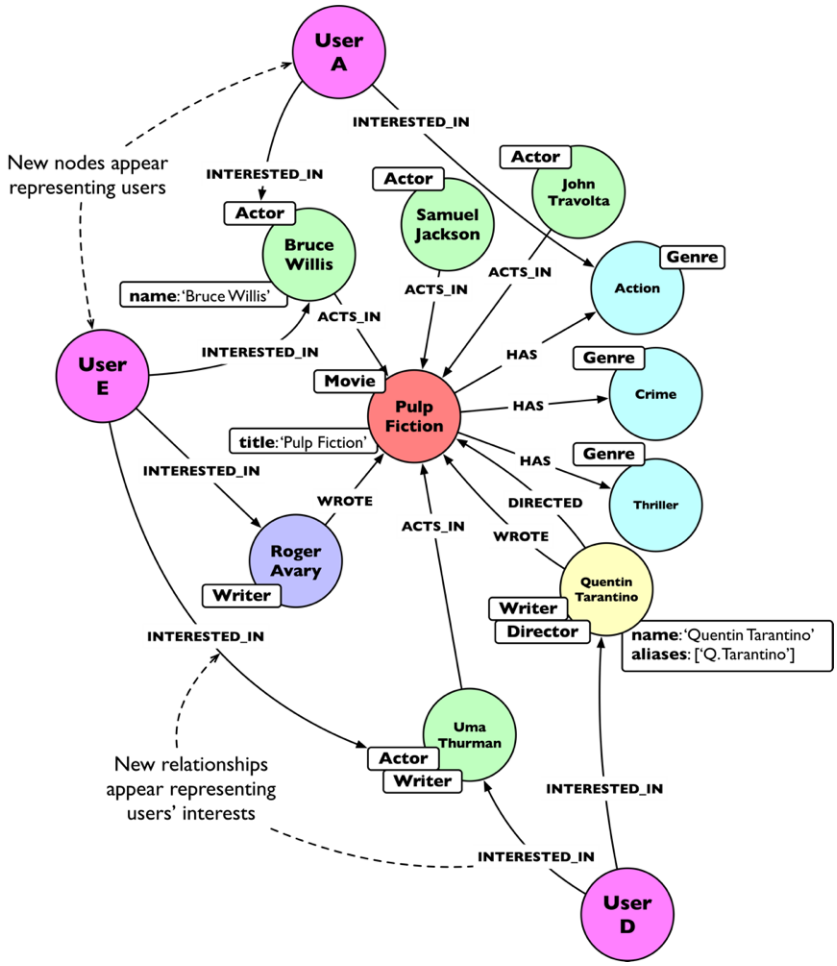


Figure 4.8 **Graph model with user interests pointing to metainformation.**

Alternatively, the system can explicitly ask the user to rate certain items. The optimal approach is to select items that will help to understand, in the broadest sense, the user's tastes. The resulting graph model looks like figure 4.9.

In this case, the user nodes are connected to the movies. The ratings are stored on the edges as a property.

These approaches are called *explicit* because the system asks the users to manifest their own tastes and preferences.

On the other side of the spectrum, another approach is to infer the users' interests, tastes, and preferences *implicitly* by considering the interactions each user has with items. For example, if I buy soy milk, it's highly probable that I'm interested in similar products, such as soy yogurt. "Soy" in this case is the "relevant" feature. Similarly, if a user watches the first episode of the *Lord of the Rings* trilogy, it's highly proba-

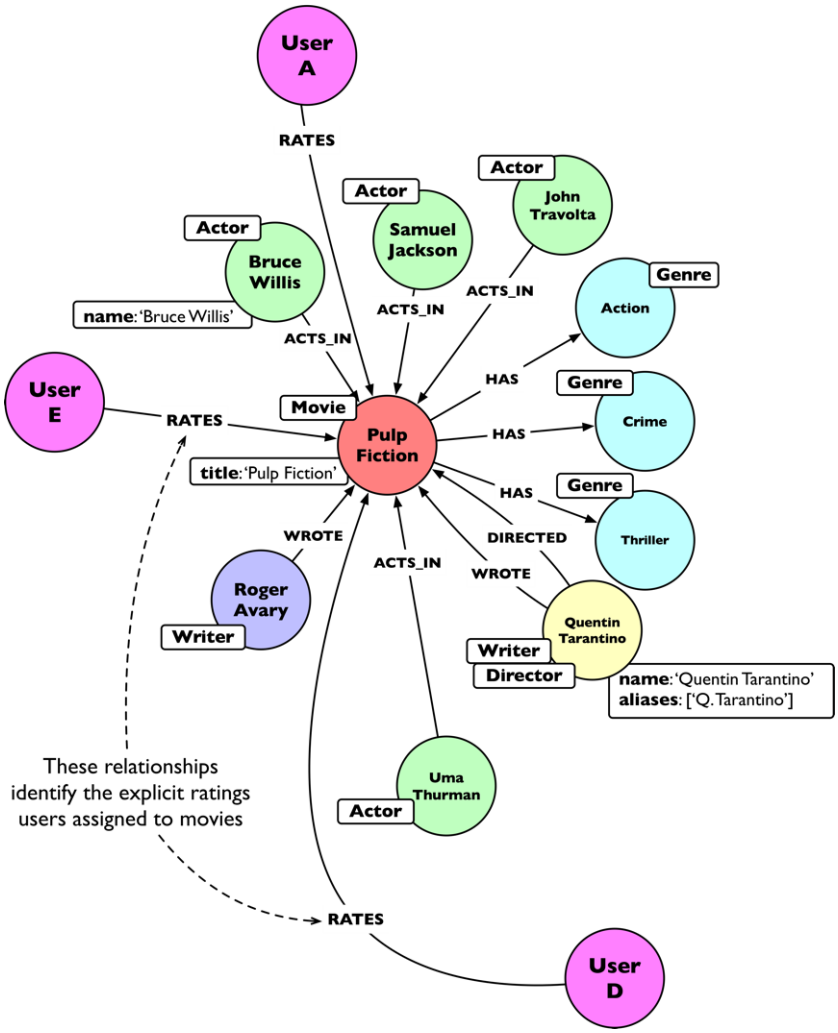


Figure 4.9 Graph model with users' explicit item ratings.

ble that they'll be interested in the other two episodes, or other movies of the same fantasy–action genre. The resulting model looks like figure 4.10.

The model in figure 4.10 is exactly the same as the previous one in figure 4.9; the only difference is that here the system collects and stores data on user behavior in order to infer users' interests implicitly.

It's worth noting that when the system models relationships between users and items, regardless of whether information on users' interests is collected implicitly or explicitly, it's possible to infer the users' interests in specific item features using different approaches. Starting from the graph as depicted in figures 4.9 and 4.10, the following Cypher query computes new relationships between users and features and then *materializes* (that is, stores as new relationships to improve access performance) them by creating new edges in the graph. See the following query.

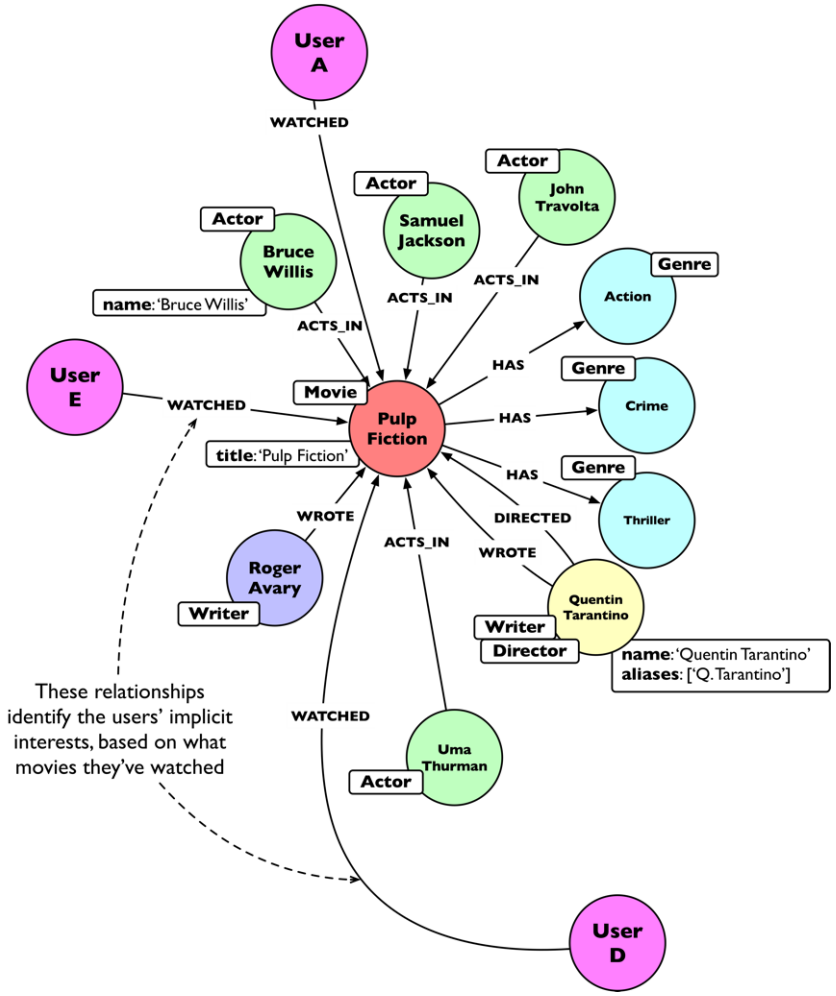


Figure 4.10 Graph model with user–item interactions.

Query 4.6 Computing relationships between users and item features[9]

The | allows you to specify multiple relationship types in the MATCH pattern.

The WITH clause aggregates users and features, counting the occurrences.

```
MATCH (user:User)-[:WATCHED|RATES]->(movie:Movie)-
 [:ACTS_IN|WRITES|DIRECTED|PRODUCES|HAS_GENRE]-(feature)
WITH user, feature, count(feature) as occurrences
WHERE occurrences > 2
MERGE (user)-[:INTERESTED_IN]->(feature)
```

This WHERE clause allows you to consider only features that occur in at least three movies the user watched.

This creates the relationship; using MERGE instead of CREATE avoids having multiple relationships between the same pairs of nodes.

[9] This query can be executed only after the import of the user rating has been done as showed in Listing 4.3.

This query searches for all the graph patterns (`(u:User)-[:WATCHED|RATES]->(m:Movie)`) that represent all the movies watched or rated by the user. It identifies the features with

```
(movie:Movie)-[:ACTS_IN|WRITES|DIRECTED|PRODUCES|HAS_GENRE]-(feature)
```

For each user–feature couple, the output of `WITH` also indicates how often a user watched a movie with that specific feature (which could be an actor, a director, a genre, and so on). The `WHERE` clause filters out all the features that appear less than three times, to keep only the most relevant and not fill the graph with useless relationships. Finally, the `MERGE` clause creates the relationships, avoiding storing multiple relationships between the same pairs of nodes (which would happen if you used `CREATE` instead).

The resulting model looks like figure 4.11.

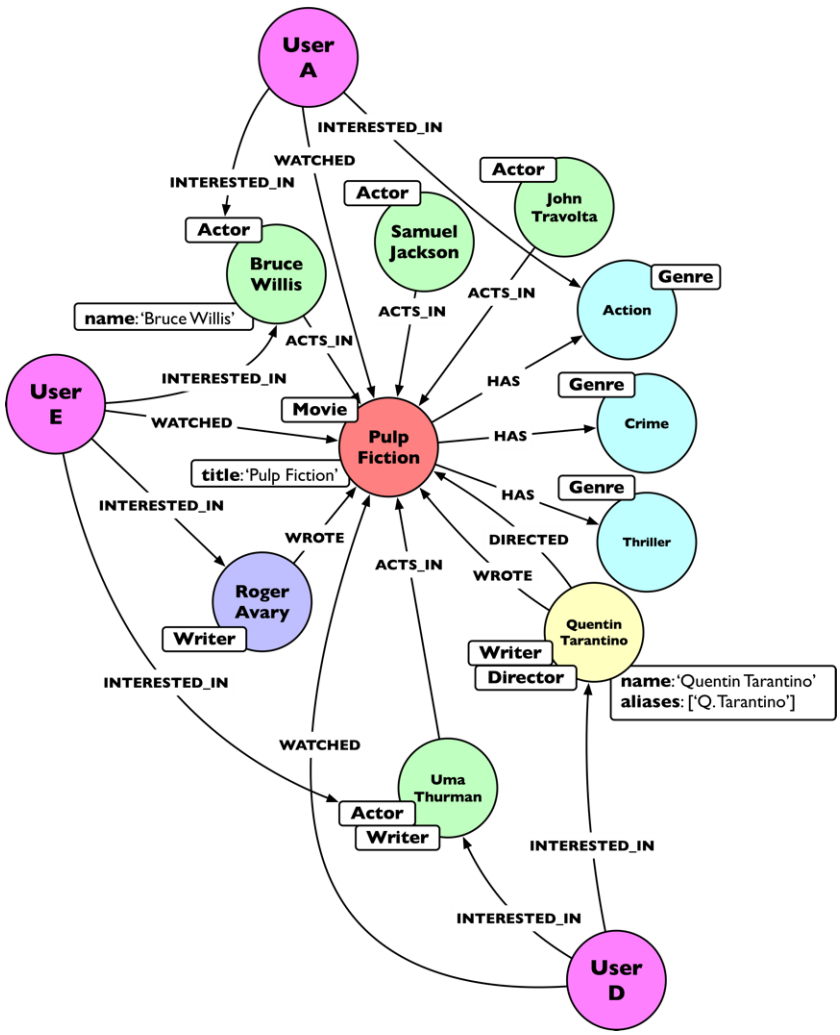


Figure 4.11 The graph model after inferring relationships INTERESTED_IN.

The model depicted in figure 4.11 contains relationships between:

- Users and items (In the modeling example, we used explicit "watched" relationships but the same would have held for explicit ratings.)
- Users and features

The second type were computed starting from the first, using a simple query. This shows another possible extension of the starting model. In this case, instead of using an external source of knowledge, the new information is inferred from the graph itself. In this specific case a graph query is used to *distill* the knowledge and convert it into a new relationship for better navigation.

The MovieLens dataset contains explicit user–item pairings based on the users' ratings (they're considered explicit because the users decided to rate the items). In listing 4.3 the ratings are used to build a graph as modeled in figure 4.10, with the only difference that here `WATCHED` is replaced with `RATED` because it represents what the user explicitly rated. The function reads from a CSV file, creates users, and connects them to the movies they rated.

Listing 4.3 Importing user–item pairings from MovieLens

```
def import_user_item(self, file):
    with open(file, 'r+') as in_file:
        reader = csv.reader(in_file, delimiter=',')
        next(reader, None)
        with self._driver.session() as session:
            self.executeNoException(session, "CREATE CONSTRAINT ON (u:User)
            ASSERT u.userId IS UNIQUE")
            tx = session.begin_transaction()
            i = 0;
            for row in reader:
                try:
                    if row:
                        user_id = strip(row[0])
                        movie_id = strip(row[1])
                        rating = strip(row[2])
                        timestamp = strip(row[3])
                        query = """
                            MATCH (movie:Movie {movieId: $movieId})
                            MERGE (user:User {userId: $userId})
                            MERGE (user)-[:RATES {rating: $rating, timestamp:
                              $timestamp}]->(movie)
                        """
                        tx.run(query, {"movieId":movie_id, "userId": user_id,
                              "rating":rating, "timestamp": timestamp})
                        i += 1
                    if i == 1000:
                        tx.commit()
```

Create a constraint to guarantee User uniqueness.

The query searches for a Movie by movieId then creates the User if it doesn't exist and connects them.

```
                i = 0
                tx = session.begin_transaction()
        except Exception as e:
            print(e, row, reader.line_num)
tx.commit()
```

At this point, the graph model we've designed is capable of representing both items and users properly and also accommodating several variations or extensions, such as semantic analysis and either implicit or explicit information.

We've now created and filled a real graph database using data obtained by combining the MovieLens dataset and information from the IMDb.

EXERCISES

Play with the database and write queries to:

1. Get a user (by `userId`) and list all the features that user is interested in.
2. Find pairs of users that share common interests.

The next section discusses how to leverage this model to deliver recommendations to the end users in the movie rental scenario we're considering.

4.2.3 Providing recommendations

During the recommendation phase, a content-based recommendation engine uses user profiles to match users with the items that are most likely to be of interest to them. Depending on the information available and the models defined for both users and items, different algorithms or techniques can be used for this. Starting from the models described previously, several techniques are described here for predicting user interests and providing recommendations. They're presented in order of growing complexity and recommendation accuracy.

The first approach is based on the model presented in figure 4.12, in which users are explicitly asked to indicate their interest in features, or interest is inferred from user–item interactions.

This approach is applicable when:

- The *items* are represented using a list of *features* that are related to the items, such as tags, keywords, genres, or actors. Such features may be manually curated by users (for instance, tags) or professionals (for instance, keywords), or generated automatically through an extraction process.
- The *user profiles* are represented by connecting the users to features that are "of interest" to them. These connections are described in a binary form: *like* (in the graph, represented with an edge between the user and the feature) and *don't like/unknown* (represented by no edge between the user and the feature). In the case in which explicit information about user interests is not available, interests can be inferred from other sources (explicit or implicit), as described previously, using a graph query.

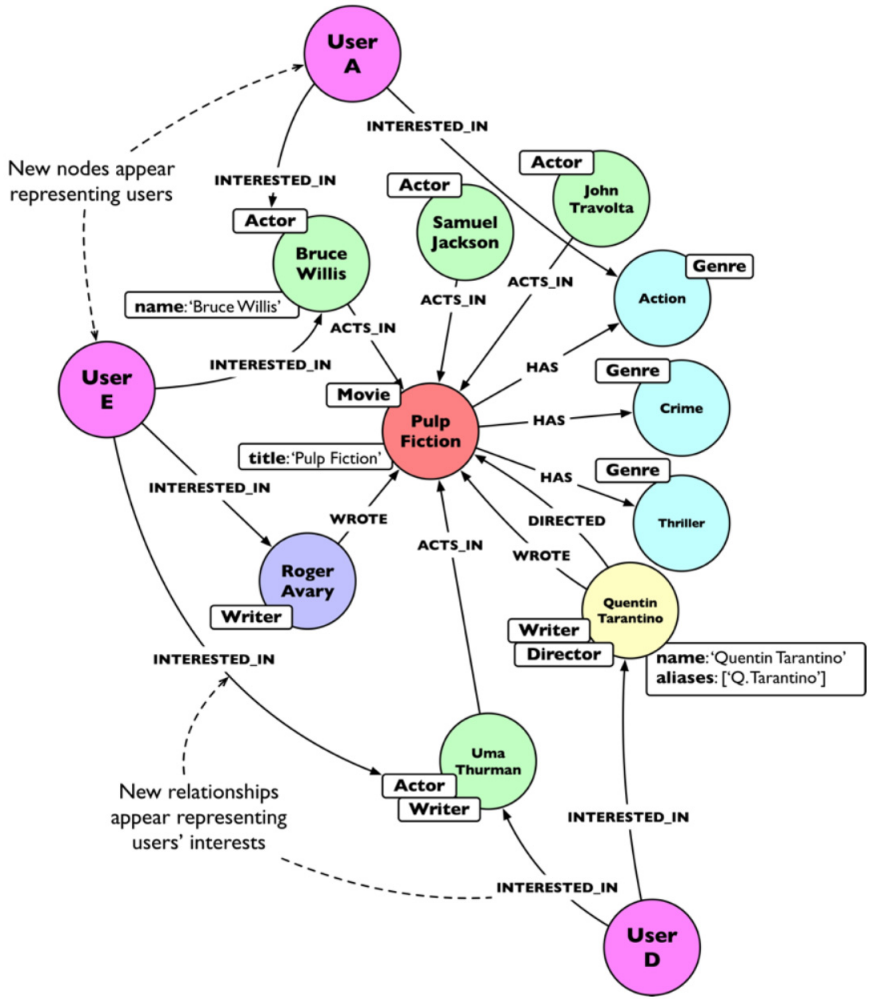


Figure 4.12 **Graph model with user interests pointing to metainformation.**

This approach is highly relevant for scenarios like the movie rental example considered here, in which metainformation is available and better describes the items themselves. The entire recommendation process in this scenario can be summarized as illustrated in figure 4.13.

This high-level diagram highlights how the entire process can be based on graphs. This approach doesn't require complex or fancy algorithms for providing recommendations. With a proper graph model, a simple query can do the job. The data already contains enough information, and the graph structure helps to compute the score

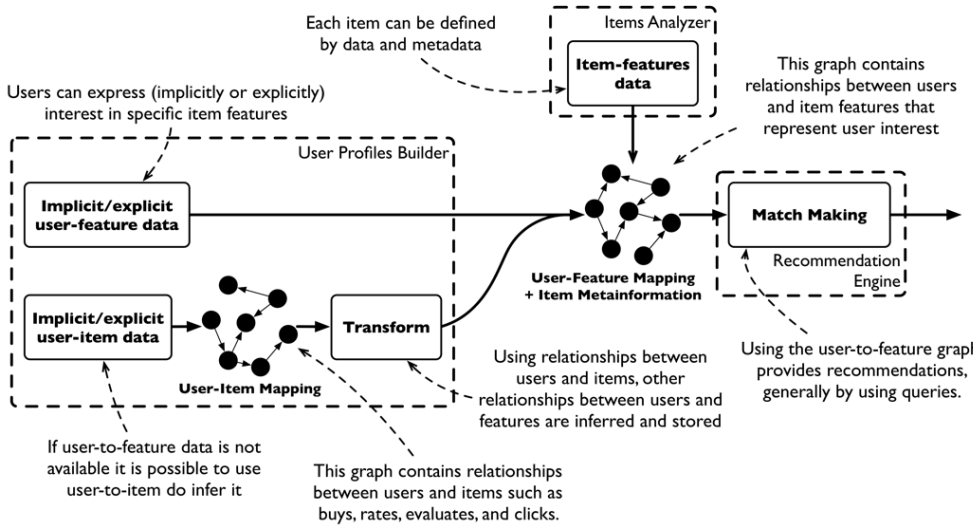


Figure 4.13 **Recommendation process for the first scenario in content-based recommenders.**

and returns the ordered list to the user with no need to prebuild any model: *the description and prediction models overlap.*

This pure graph-based approach is simple, but it has many advantages:

- *It produces good results.* The quality of the recommendations is quite high considering the limited effort required by this method.
- *It's simple.* It doesn't require complex computations or complex code that reads and preprocesses the data before providing recommendations. If the data is modeled properly in the graph, as shown previously, it's possible to perform the queries and reply to users in real time.
- *It's extensible.* The graph can contain other information that can be useful to refine the results according to other data sources or contextual information. The queries can easily be changed to take new aspects into account.

The task of providing recommendations is accomplished using queries like the following one.

Query 4.7 Providing recommendations to a user

Starting from a user, the MATCH clause searches for all the movies that have features of interest for that user.

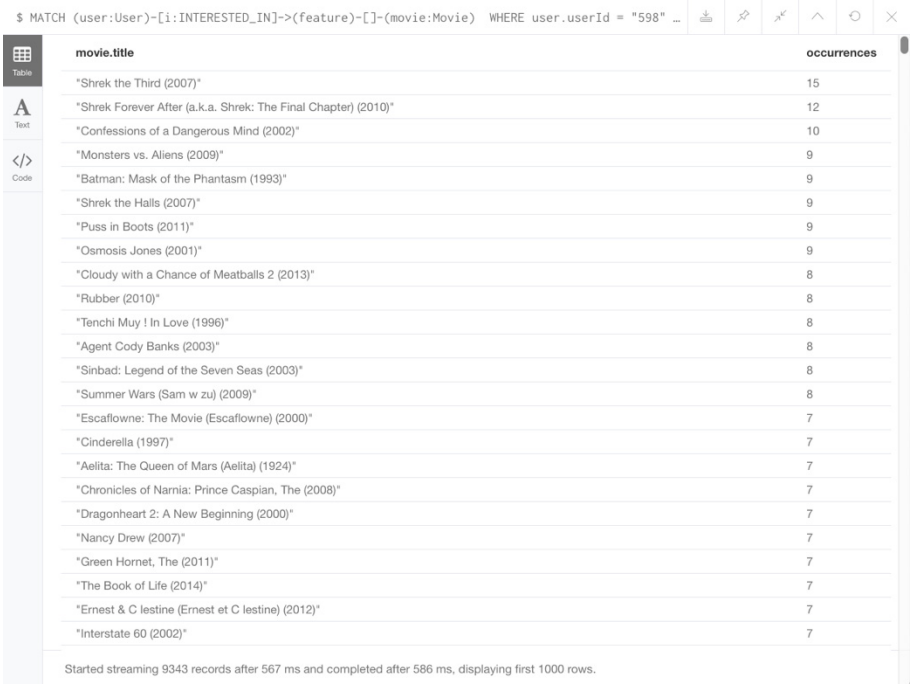
The NOT exists() filters out all the movies already WATCHED or RATED by the user.

```
MATCH (user:User)-[i:INTERESTED_IN]->(feature)-[]-(movie:Movie)
WHERE user.userId = "<user Id>" AND NOT exists((user)-[]->(movie))
RETURN movie.title, count(i) as occurrences
ORDER BY occurrences desc
```

Sorting in reverse order helps to bring to the top the movies shared more with the selected user.

This query starts from the user (the `WHERE` clause specifies a `userId` as string), identifies all the features of interest to the user, and finds all the movies that contain them. For each movie, the query counts the overlapping features, and it orders the movies according to this value: the higher the number of overlapping features, the higher the likelihood is that the item might be of interest to the user.

This approach can be applied to the database we created earlier. The MovieLens dataset contains connections between users and items, but no relationships between users and features of interest for the users—these aren't available at all in the dataset. We enriched it using the IMDb as a source of knowledge for movie features, and by applying query 4.6 it's possible to compute the missing relationships between users and item features. Use the code and the queries to play with the graph database and provide recommendations. It will not be fast, but it will work properly. Figure 4.14 shows the result of running query 4.7 on the imported database. It's worth noting that user 598 in the example shown here had already rated *Shrek* and *Shrek 2*.



movie.title	occurrences
"Shrek the Third (2007)"	15
"Shrek Forever After (a.k.a. Shrek: The Final Chapter) (2010)"	12
"Confessions of a Dangerous Mind (2002)"	10
"Monsters vs. Aliens (2009)"	9
"Batman: Mask of the Phantasm (1993)"	9
"Shrek the Halls (2007)"	9
"Puss in Boots (2011)"	9
"Osmosis Jones (2001)"	9
"Cloudy with a Chance of Meatballs 2 (2013)"	8
"Rubber (2010)"	8
"Tenchi Muy ! In Love (1996)"	8
"Agent Cody Banks (2003)"	8
"Sinbad: Legend of the Seven Seas (2003)"	8
"Summer Wars (Sam w zu) (2009)"	8
"Escaflowne: The Movie (Escaflowne) (2000)"	7
"Cinderella (1997)"	7
"Aelita: The Queen of Mars (Aelita) (1924)"	7
"Chronicles of Narnia: Prince Caspian, The (2008)"	7
"Dragonheart 2: A New Beginning (2000)"	7
"Nancy Drew (2007)"	7
"Green Hornet, The (2011)"	7
"The Book of Life (2014)"	7
"Ernest & C lestine (Ernest et C lestine) (2012)"	7
"Interstate 60 (2002)"	7

Started streaming 9343 records after 567 ms and completed after 586 ms, displaying first 1000 rows.

Figure 4.14 The result of running query 4.7 on the imported MovieLens database.

Later in this chapter and the book, different techniques and methods for improving performance are described; here the focus is on different graph modeling techniques and design options.

EXERCISE

Rewrite query 4.7 to consider only movies of a specific genre or from a specific year. Tip: Add a condition to the `WHERE` clause using `EXISTS`.

The approach in this section described works quite well and is simple. But with a small amount of effort, it can be greatly improved. The second approach extends the previous one by considering two main aspects that can be improved:

1. In the user profile, interest in an item feature is represented by a Boolean value. It's *binary*, representing only the fact that the user is interested in the feature. It doesn't ascribe any "weight" to this relationship.
2. Counting the overlapping features between user profiles and items isn't enough. We need a function that computes the similarity or commonalities between user interests and items.

Regarding the first point, as is stated often in this book, models are representations of reality, and the reality is that we're modeling a user who's likely to be interested more in certain features than others (I like action movies, but I love movies with Jason Statham). This information can improve the quality of the recommendations.

Regarding the second point, instead of counting the overlapping features, a better method to find interesting items for a specific user consists of measuring the similarity between the user profile and the item's features—the closer, the better. This requires:

- A *function* that measures the similarity
- A *common representation* for both items and user profiles so that their similarity is measurable

The selected function defines the required representation for items and user profiles. Different functions are available; one of the most accurate is *cosine similarity*, introduced in chapter 3. Here, we recall the formula:

$$sim(\vec{a}, \vec{b}) = \cos(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| * |\vec{b}|}$$

Like most of the commonly used similarity functions, this requires that each item and each user profile be projected into a common *vector space model (VSM)*, which means that each element has to be represented using a fixed-dimension vector.

The entire recommendation process in this second scenario can be summarized using the high-level diagram in figure 4.15.

Compared with the previous approach, this case has an intermediate step before the recommendation process where the items and user profiles are projected into the VSM. To describe this process of converting items and user profiles into their representations in the VSM, let's consider our movie recommendation scenario. Suppose our movie dataset is as represented in figure 4.16.

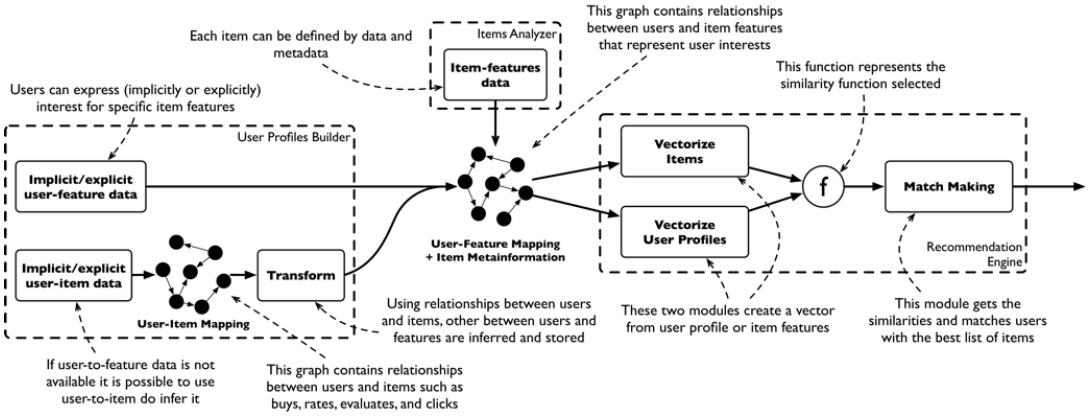


Figure 4.15 Recommendation process for the second scenario in content-based recommenders.

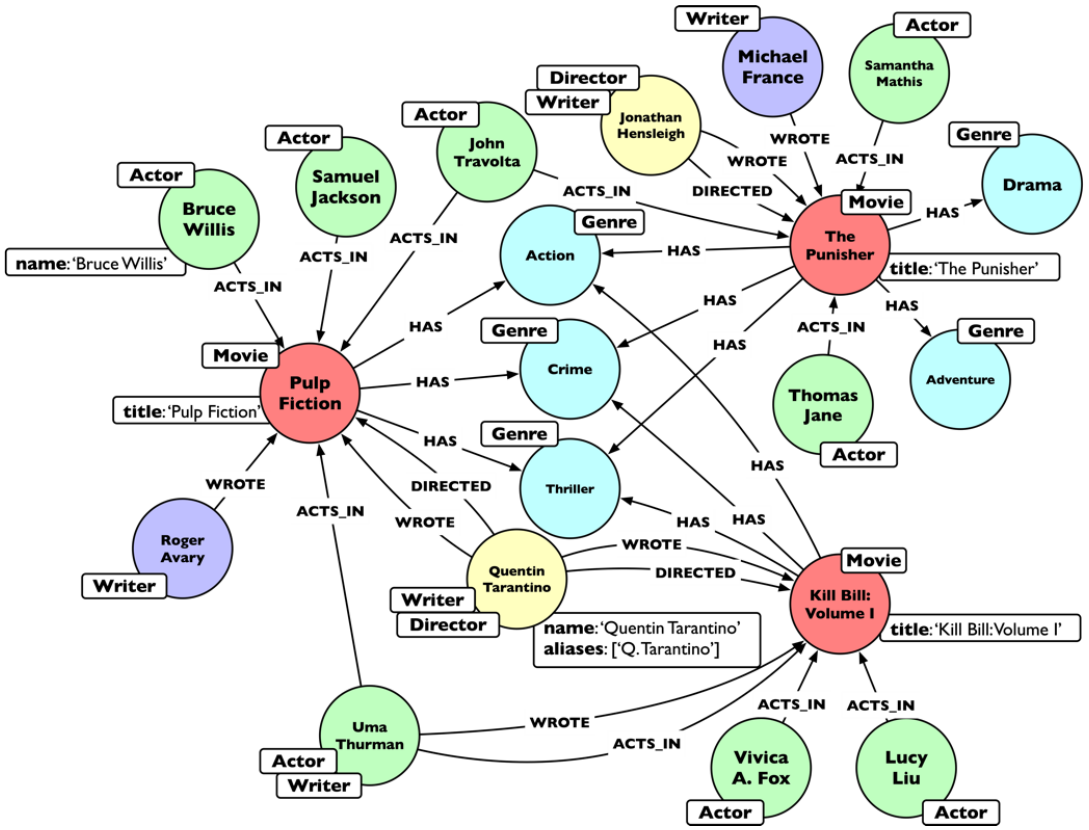


Figure 4.16 Movie advanced model.

Each item can be represented as a vector considering metainformation such as the genres and director (as an example—we could use all the metainformation available, but that would make the next table too big). The dimensions of each vector, in this case, are defined by the list of all possible values for genres and directors. Table 4.2 shows what this looks like for the simple dataset we created manually.

Table 4.2 Converting Items to Vectors

	Action	Drama	Crime	Thriller	Adventure	Quentin Tarantino	Jonathan Hensleigh
Pulp Fiction	1	0	1	1	0	1	0
The Punisher	1	1	1	1	1	0	1
Kill Bill: Vol I	1	0	1	1	0	1	0

These are Boolean vectors, because the values can be only 0, which means absence, and 1, which means presence. The vectors representing the three movies are then:

$$Vector(Pulp\ Fiction) = [1,0,1,1,0,1,0]$$

$$Vector(The\ Punisher) = [1,1,1,1,1,0,1]$$

$$Vector(Kill\ Bill{:}\ Volume\ I) = [1,0,1,1,0,1,0]$$

These binary vectors can be extracted from the graph model in figure 4.16 through the following query.

Query 4.8 Extracting Boolean vectors for movies

Search for all the features that are Director or Genre using the labels function to get the list of labels assigned to a node.

```
MATCH (feature)
WHERE "Genre" in labels(feature) OR "Director" in labels(feature)
WITH feature
ORDER BY id(feature)
MATCH (movie:Movie)
WHERE movie.title STARTS WITH "Pulp Fiction"
OPTIONAL MATCH (movie)-[r:DIRECTED|HAS_GENRE]-(feature)
RETURN CASE WHEN r IS null THEN 0 ELSE 1 END as Value,
CASE WHEN feature.genre IS null THEN feature.name ELSE feature.genre
     END as Feature
```

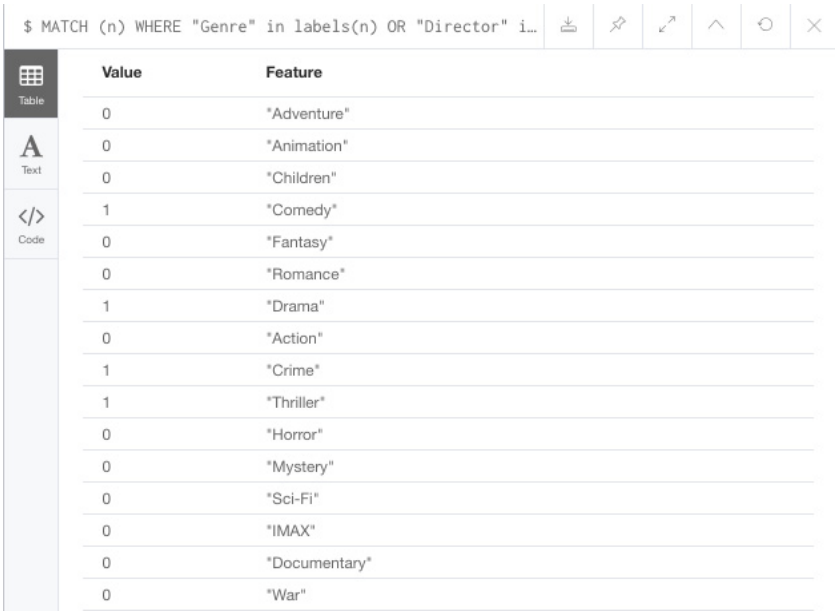
Search for the movie Pulp Fiction; using STARTS WITH is preferable to an exact string comparison because the movies generally have the year in the title.

This CASE clause returns 0 if no relationship exists, and 1 otherwise.

OPTIONAL MATCH allows us to consider all the features, even if they aren't related to the movie selected.

This CASE clause returns either the name of the director or the genre.

The query starts by looking for all the nodes that represent genres or directors, and it returns all of them ordered by node identifier. The order is important because in each vector the specific genre or director must be represented in the same position. It then searches for a specific movie by title, and `OPTIONAL MATCH` checks whether the movie is connected to the feature. Unlike `MATCH`, which filters out the nonmatching elements, `OPTIONAL MATCH` returns null if the relationship doesn't exist. In the `RETURN`, the first `CASE` clause returns 0 if no relationship exists and 1 otherwise; the second returns the name of either the director or the genre. Figure 4.17 shows the result of the query run against the database imported from MovieLens.



Value	Feature
0	"Adventure"
0	"Animation"
0	"Children"
1	"Comedy"
0	"Fantasy"
0	"Romance"
1	"Drama"
0	"Action"
1	"Crime"
1	"Thriller"
0	"Horror"
0	"Mystery"
0	"Sci-Fi"
0	"IMAX"
0	"Documentary"
0	"War"

Figure 4.17 Result of running query 4.8 on the MovieLens dataset.

As is evident from the screenshot in figure 4.17, the real vectors are quite large because many possible "dimensions" exist. Although this full representation is manageable by the implementation discussed here, the next chapter introduces a way to represent such long vectors in a better way.

It's possible to generalize this vector approach to all sorts of features, including those that have numerical values, like the average ratings in our movie scenario.[10] In the vector representation the related components hold the exact values of these features.

In our example, the vector representations for the three movies become:

[10] The average rating isn't a valuable feature, but it will serve the purpose in our example.

Adding an index

Running this query on the MovieLens database can take long time. The time is spent in the filter condition, `movie.title STARTS WITH "Pulp Fiction"`. Adding an index can greatly improve the performance. Run the following command and then try the query again:

```
CREATE INDEX ON :Movie(title)
```

Much faster, isn't it?

$$Vector(Pulp\ Fiction) = [1,0,1,1,0,1,0,4]$$
$$Vector(The\ Punisher) = [1,1,1,1,1,0,1,3.5]$$
$$Vector(Kill\ Bill{:}\ Volume\ I) = [1,0,1,1,0,1,0,3.9]$$

The last element represents the average rating. It doesn't matter if part of the components of the vectors are Boolean and others are real-valued or integer-valued [Ullman and Rajaraman, 2011]. It's still possible to compute the cosine distance between vectors, although if we do so, we should consider some appropriate scaling of the non-Boolean components so that they neither dominate the calculation nor are irrelevant. To do this, we multiply the values by a *scaling factor*:

$$Vector(Pulp\ Fiction) = [1,0,1,1,0,1,0,4\alpha]$$
$$Vector(The\ Punisher) = [1,1,1,1,1,0,1,3.5\alpha]$$
$$Vector(Kill\ Bill{:}\ Volume\ I) = [1,0,1,1,0,1,0,3.9\alpha]$$

In this representation, if α is set to 1 the average rating will dominate the value of the similarity, while if it's set to 0.5 the effect will be reduced by half. The scaling factor can be different for each numerical feature and depends on the weight assigned to that feature in the resulting similarity.

With the proper vector representation of the items in hand, we need to project the user profile into the same VSM, which means that we need to create vectors with the same components in the same order as in the item vectors that describe the user's preferences. As described in section 4.2.2, the information available in the content-based case regarding user preferences or tastes can be either a *user–item* pair or a *user–feature* pair. Both can be collected implicitly or explicitly. Because the vector space has the feature values as dimensions, the first step in the projection is to migrate the user–item matrix to the user–feature space (unless it is already available). Different techniques can be used for this conversion, including aggregating by counting the occurrences of each feature in a user's list of previously "liked"[11] items. This option works well for Boolean values; another option is computing the average values for numerical features.

[11] "Liked" here means any kind of interaction between user and item, such as "watched", "rated", and so on.

In the movie scenario, each user profile can be represented as shown in table 4.3.

Table 4.3 User Profiles Represented in the Same Vector Space as Movies

	Action	Drama	Crime	Thriller	Adventure	Quentin Tarantino	Jonathan Hensleigh	Total
User A	3	1	4	5	1	3	1	9
User B	0	10	1	2	3	0	1	15
User C	1	0	3	1	0	1	0	5

Each cell represents how many movies the user watched that have that specific feature. So, for example, User A has watched three movies directed by Quentin Tarantino, while User B hasn't watched any movies he directed. The table in this case also contains a new column that represents the total number of movies watched by each user; this value will be useful later in the creation of the vectors to normalize the values.

These user–feature pairs with the related counts are easy to obtain from the graph model we've used so far for representing user–item interactions. To simplify the next steps, it's recommended to materialize these values by storing them properly in the graph itself. In a property graph database, the "weight" representing how much the user is interested in a specific item feature can be modeled with a property on the relationship between the user and the feature. Modifying query 4.6, used earlier for inferring the relationship between users and features, it's possible to extract this information, create new relationships, and add these weights to the edges. The new query looks like the following.

Query 4.9 Extracting weighted relationship between users and features

```
MATCH (user:User)-[:WATCHED|RATES]->(m:Movie)-
     [:ACTS_IN|WRITES|DIRECTED|PRODUCES|HAS_GENRE]-(feature)
WITH user, feature, count(feature) as occurrence
WHERE occurrence > 2
MERGE (user)-[r:INTERESTED_IN]->(feature)
SET r.weight = occurrence
```

SET add or modify the weight property on the INTERESTED_IN relationship.

In this version `occurrence` is stored as a property on the relationship `INTERESTED_IN`, whereas in query 4.6 it was used only as a filter. Figure 4.18 shows the resulting model.

On their own, the numbers in the table could lead to incorrect computations of the similarities between the user profile vector and the item vector. They have to be *normalized* to better represent the real interest of a user in a specific feature. For example, if a user has watched 50 movies and only 5 are dramas, we might conclude that that user is less interested in this genre than a user who watched 3 dramas out of a total of only 10 movies, even though the first user watched more movies in total of this type.

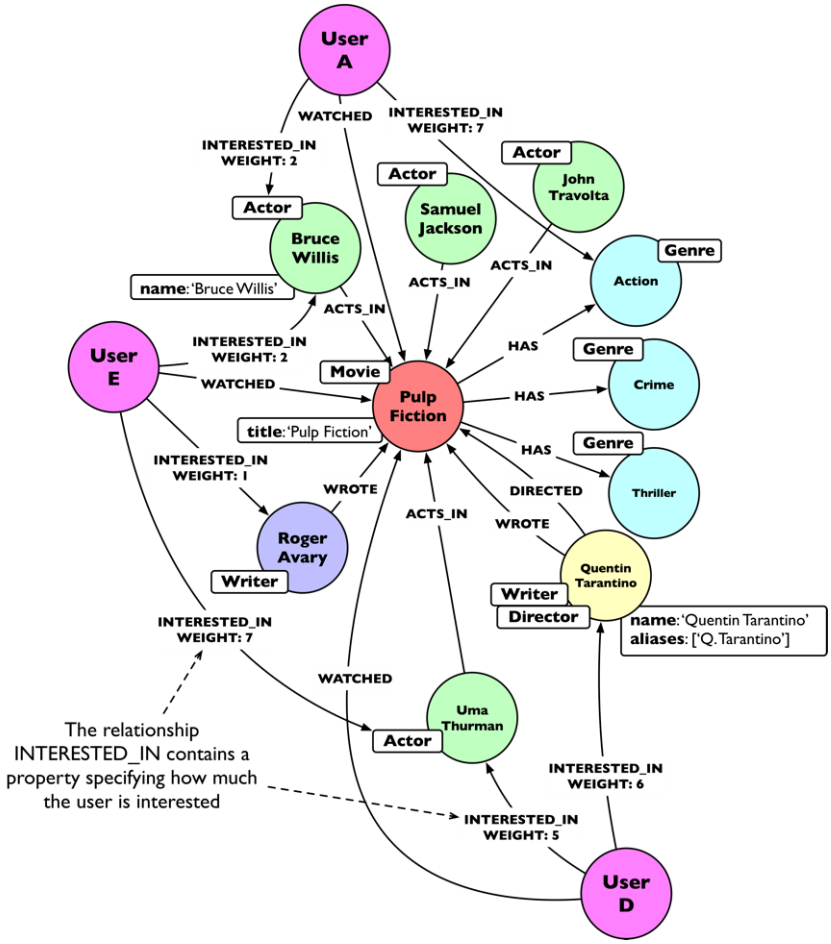


Figure 4.18 The graph model after inferring INTERESTED_IN relationships with the weights.

If we *normalize* each value in table 4.3 with the total number of movies watched, we see that the first user has 0.1 interest in the drama genre, while the second has 0.6. Table 4.4 shows the normalized user profiles.

Table 4.4 Normalized Version of Table 4.3

	Action	Drama	Crime	Thriller	Adventure	Quentin Tarantino	Jonathan Hensleigh
User A	0.33	0.11	0.44	0.55	0.11	0.33	0.11
User B	0	0.66	0.06	0.13	0.2	0	0.06
User C	0.2	0	0.6	0.2	0	0.2	0

Modeling pro tip

It isn't recommended to store the results of the normalization process as weights in the graph, because these are affected by the total number of movies watched by the user. Storing such values would require us to recompute each weight every time a user watched a new movie. If we store only the counts as weights, when a user watches a new movie only the affected features have to be updated. For example, if a user watches an adventure movie only the count for that genre has to be updated.

In the explicit scenario this "weight" information can be collected by asking the user to assign rates to a set of possible of item features. The related values can be stored in the `weight` property on the edges between users and features.

At the end of this process we have both items and user profiles represented in a common and comparable way. The recommendation task for each user can be accomplished by computing the similarities between the user profile vector representation and each not-yet-seen movie, ordering them from highest to lowest and returning the top *N*, where *N* can be 1, 10, or whatever the application requires.

In this scenario, the recommendation task requires complex operations that cannot be accomplished by a query, because they require complex computations, looping, transformation and so on. The following listing shows how to provide recommendations once the data is stored as depicted in figure 4.18.

The full code is available in the code repository as `ch04/recommendation/content_based_recommendation_second_approach.py`.

Listing 4.4 Method to provide recommendations using the second approach

This function provides recommendations.

```
def recommendTo(self, userId, k):
    user_VSM = self.get_user_vector(userId)
    movies_VSM = self.get_movie_vectors(userId)
    top_k = self.compute_top_k (user_VSM, movies_VSM, k);
    return top_k

def compute_top_k(self, user, movies, k):
    dtype = [ ('movieId', 'U10'),('value', 'f4')]
    knn_values = np.array([], dtype=dtype)
    for other_movie in movies:
        value = cosine_similarity([user], [movies[other_movie]])
        if value > 0:
            knn_values = np.concatenate((knn_values, np.array([(other_movie,
     value)], dtype=dtype)))
    knn_values = np.sort(knn_values, kind='mergesort', order='value' )[::-1]
    return np.array_split(knn_values, [k])[0]

def get_user_vector(self, user_id):
    query = """
            MATCH p=(user:User)-[:WATCHED|RATES]->(movie)
            WHERE user.userId = $userId
            with count(p) as total
            MATCH (feature:Feature)
            WITH feature, total
            ORDER BY id(feature)
```

This function computes the similarities between the user profile vector and the movie vectors and gets back the top k movies that best match the user profile.

We use the cosine_similarity function provided by scikit-learn.

This function creates the user profile; note how it is provided with a single query and mapped with a vector.

The order is critical because it allows us to have comparable vectors.

```
                    MATCH (user:User)
                    WHERE user.userId = {userId}
                    OPTIONAL MATCH (user)-[r:INTERESTED_IN]-(feature)
                    WITH CASE WHEN r IS null THEN 0 ELSE
                           (r.weight*1.0f)/(total*1.0f) END as value
                    RETURN collect(value) as vector
                """
        user_VSM = None
        with self._driver.session() as session:
            tx = session.begin_transaction()
            vector = tx.run(query, {"userId": user_id})
            user_VSM = vector.single()[0]
        print(len(user_VSM))
        return user_VSM;

    def get_movie_vectors(self, user_id):
        list_of_moview_query = """
                    MATCH (movie:Movie)-[r:DIRECTED|HAS_GENRE]-(feature)<-
                           [i:INTERESTED_IN]-(user:User {userId: $userId})
                    WHERE NOT EXISTS((user)-[]->(movie)) AND EXISTS((user)-[]-
                           >(feature))
                    WITH movie, count(i) as featuresCount
                    WHERE featuresCount > 5
                    RETURN movie.movieId as movieId
                """

        query = """
                    MATCH (feature:Feature)
                    WITH feature
                    ORDER BY id(feature)
                    MATCH (movie:Movie)
                    WHERE movie.movieId = {movieId}
                    OPTIONAL MATCH (movie)-[r:DIRECTED|HAS_GENRE]-(feature)
                    WITH CASE WHEN r IS null THEN 0 ELSE 1 END as value
                    RETURN collect(value) as vector;
                """
        movies_VSM = {}
        with self._driver.session() as session:
            tx = session.begin_transaction()

            i = 0
            for movie in tx.run(list_of_moview_query, {"userId": user_id}):
                movie_id = movie["movieId"];
                vector = tx.run(query, {"movieId": movie_id})
                movies_VSM[movie_id] = vector.single()[0]
                i += 1
                if i % 100 == 0:
                    print(i, "lines processed")
            print(i, "lines processed")
        print(len(movies_VSM))
        return movies_VSM
```

This function provides the movie vectors.

This query gets only relevant not-seen-yet movies for the user, which speeds up the process.

This query creates the movie vectors.

If you run this code for user 598 (the same user as in the previous scenario) you'll see that the list of recommended movies isn't that different from the results obtained in the previous case, but these new results should be better in terms of prediction accu-

racy. Thanks to the graph, it's possible to easily get movies that contain at least five features in common with the user profile.

Also be aware that this recommendation process takes a while to produce results. Don't be worried; the goal here is to show the concepts in the simplest way, and various optimization techniques will be discussed later in the book. If you're interested, in the code repository you'll find an optimized version of this code that uses a different approach to vector creation and similarity computation.

EXERCISES

Considering the code in listing 4.4:

1. Rewrite the code to use a different similarity function, such as the Pearson correlation, instead of cosine similarity. Tip: Search for a Python implementation and replace the `cosine_similarity` function.
2. Look at the optimized implementation in the code repository and figure out how the new vectors are created. How much faster is it now? In the next chapter the concept of "sparse vectors" will be introduced.

The third approach we will consider for content-based recommendations can be described as "recommend items that are similar to those the user liked in the past" [Jannach et al., 2010]. This approach works well in general and is the only option when it's possible to compute relevant similarities between items, but difficult or not relevant to represent user profiles in the same way.

Consider our training dataset, as represented in figures 4.9 and 4.10. The user preferences are modeled by connecting users to items, instead of users to items metainformation. This could be necessary when metainformation for each item is either not available, limited, or not relevant, so it isn't possible (or necessary) to extract data on the user's interest in certain features. Nonetheless, the "content" or "content description" related to each item is somehow available—otherwise, the content-based approach wouldn't be applicable. Even in the case where metainformation is available, this third approach greatly outperforms the previous one in terms of recommendation accuracy.

This is known as the *similarity-based retrieval* approach, and it has been selected as a valuable approach to cover for several reasons:

- It was already introduced in chapter 3. Here, different item representations are used for computing similarities.
- Similarities are easy to store back in the graph as relationships between items. This represents a prefect use case for graph modeling and navigating similarity relationships provides for fast recommendations.
- It's one of the most common and powerful approaches to content-based recommendation engines.
- It's flexible and general enough to be used in many different scenarios, regardless of the type of data/information available for each item.

The entire recommendation process in this scenario can be summarized using the high-level diagram in figure 4.19.

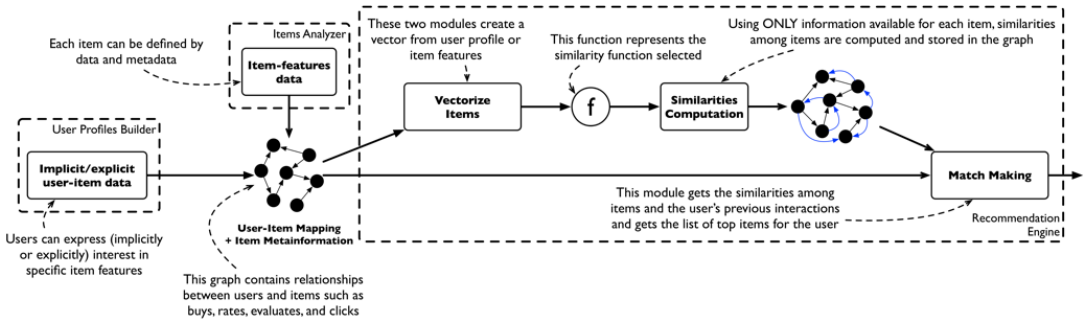


Figure 4.19 Recommendation process for the third approach in content-based recommenders.

It's worth noting here, because this is the biggest difference with the collaborative filtering approach described in the next chapter, that the similarities between items are computed only using the item-related data, whatever that is. The user–item interaction is used only during the recommendation phase.

According to the diagram in figure 4.19, three key elements are necessary in this scenario:

1. *User profile:* This is represented by modeling the interactions the user has with items, such as rated, bought, or watched. In the graph these interactions are represented as relationships between the user and the items.
2. *Item representation/description:* To compute similarities between items, it's necessary to represent each item in a measurable way. How this is done depends on the function selected for measuring similarities.
3. *Similarity function:* A function is needed that, given two item representations, computes the similarity between them. We described applying the cosine similarity metric from chapter 3 to a simplified example of collaborative filtering. Here, different techniques are described in more detail, applied to content-based recommendations.

As in the second approach, points 2 and 3 are strictly related because each similarity formula requires a specific item representation. And conversely, according to the data available for each item, certain functions can be applied while others cannot.

A typical similarity metric, which is suitable for multivalued characteristics, is the Dice coefficient [Dice, 1945]. It works as follows. Each item I_i is described by a set of features *features*(I_i)—for example, a set of keywords. The Dice coefficient measures the similarity between items I_i and I_j as:

$$dice\_coefficient(I_i, I_j) = \frac{2 \times \left|keywords(I_i) \cap keywords(I_j)\right|}{\left|keywords(I_i)\right| + \left|keywords(I_j)\right|}$$

In this formula, `keywords` return the list of keywords that describe the item. In the numerator, the formula computes the number of overlapping/intersecting keywords and multiplies this by 2. In the denominator, it sums the number of keywords in each item. This is a simple formula, where "keywords" can be replaced with anything: in our movie example it could be genres, actors, and so on (see figure 4.20).

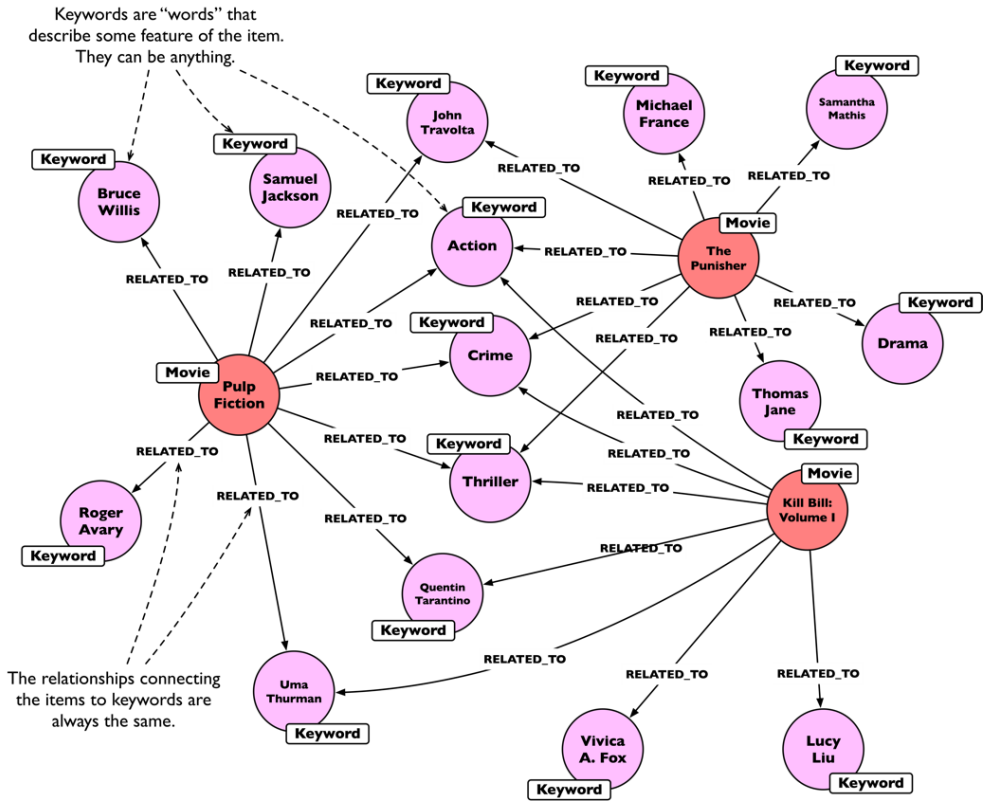


Figure 4.20 **The graph model for representing keywords.**

Once the similarities are computed they can be stored back in the graph, as shown in figure 4.21.

It isn't necessary to store these neighbor relationships between each pair of nodes (although it's necessary to compute all of them). Generally, only a small number of them are stored. You can define a minimum similarity threshold, or you can define a *k* value and keep only the *k* topmost similar items. For this reason, the methods described in this approach are known as *k-nearest neighbor (k-NN)* methods regardless of the similarity function selected.

The Dice coefficient is simple, but the quality of the resulting recommendations is quite poor, because it leverages a small amount of information for computing the

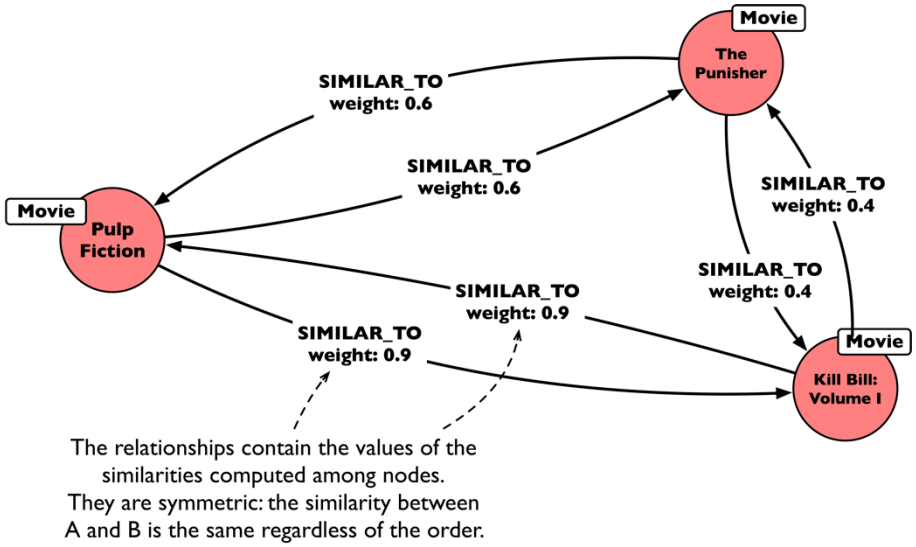


Figure 4.21 **Storing similarities back in the graph.**

similarities. A much more powerful method for computing the similarities between items is based on cosine similarity. The items can be represented exactly as in the second approach. The difference here is that, instead of computing cosine similarities between user profiles and items, the cosine function is used to compute similarities between items.

This similarity is computed for each pair of items, and then the top k matches for each item are stored back in the graph as similarity relationships. As an example, consider the similarities listed in table 4.5.

Table 4.5 **Cosine Similarities Between Movies**

	Pulp Fiction	The Punisher	Kill Bill: Volume I
Pulp Fiction	1	0.612	1
The Punisher	0.612	1	0.612
Kill Bill: Volume I	1	0.612	1

The table's contents can be stored in the graph as shown in figure 4.21.

It's important to note here that, unlike in the first and second approaches, where the recommendation process uses the data as it is, here the recommendation process requires an intermediate step: this k-NN computation and storing. In this case the descriptive model and the prediction model don't match completely.

The following listing shows a Python script for computing *k*-NN and storing this data back in the graph. It works on the graph database we imported from the MovieLens dataset.

Listing 4.5 Code for creating the k-NN network

```
def compute_and_store_similarity(self):
    movies_VSM = self.get_movie_vectors()
    for movie in movies_VSM:
        knn = self.compute_knn(movie, movies_VSM.copy(), 10);
        self.store_knn(movie, knn)

def get_movie_vectors(self):
    list_of_moview_query = """
                MATCH (movie:Movie)
                RETURN movie.movieId as movieId
            """

    query = """
                MATCH (feature:Feature)
                WITH feature
                ORDER BY id(feature)
                MATCH (movie:Movie)
                WHERE movie.movieId = $movieId
                OPTIONAL MATCH (movie)-[r:DIRECTED|HAS_GENRE]-(feature)
                WITH CASE WHEN r IS null THEN 0 ELSE 1 END as value
                RETURN collect(value) as vector;
            """
    movies_VSM = {}
    with self._driver.session() as session:
        tx = session.begin_transaction()

        i = 0
        for movie in tx.run(list_of_moview_query):
            movie_id = movie["movieId"];
            vector = tx.run(query, {"movieId": movie_id})
            movies_VSM[movie_id] = vector.single()[0]
            i += 1
            if i % 100 == 0:
                print(i, "lines processed")
        print(i, "lines processed")
    print(len(movies_VSM))
    return movies_VSM

def compute_knn(self, movie, movies, k):
    dtype = [ ('movieId', 'U10'),('value', 'f4')]
    knn_values = np.array([], dtype=dtype)
    for other_movie in movies:
        if other_movie != movie:
            value = cosine_similarity([movies[movie]], [movies[other_movie]])
            if value > 0:
                knn_values = np.concatenate((knn_values,
     np.array([(other_movie, value)], dtype=dtype)))
    knn_values = np.sort(knn_values, kind='mergesort', order='value' )[::-1]
```

Overall function that performs all the tasks for all the movies.

This function projects each movie in the VSM.

This function computes the KNN for each movie.

Here it used the cosine_similarity available in scikit.

```
    return np.array_split(knn_values, k)[0]

def store_knn(self, movie, knn):
    with self._driver.session() as session:
        tx = session.begin_transaction()
        test = {a : b.item() for a,b in knn}
        clean_query = """MATCH (movie:Movie)-[s:SIMILAR_TO]-()
            WHERE movie.movieId = $movieId
            DELETE s
        """
        query = """
            MATCH (movie:Movie)
            WHERE movie.movieId = $movieId
            UNWIND keys($knn) as otherMovieId
            MATCH (other:Movie)
            WHERE other.movieId = otherMovieId
            MERGE (movie)-[:SIMILAR_TO {weight: $knn[otherMovieId]}]-(other)
        """
        tx.run(clean_query, {"movieId": movie})
        tx.run(query, {"movieId": movie, "knn": test})
        tx.commit()
```

This function stores the KNN on the graph database.

Before storing the new similarity it deletes the old.

As in the previous case, this code may take a while to complete. Here I am presenting the basic ideas; later in the book we'll discuss several optimization techniques for real projects.

EXERCISES

Once the *k*-NN have been computed using the code in listing 4.5, write a query to:

1. Get a movie (by `movieId`) and get the list of the top 10 most similar items.
2. Search for the top 10 most similar pairs of items.

The next step in the recommendation process for this third approach, as depicted in figure 4.19, consists of making the recommendations, which we do by drawing upon the *k*-NN network and the user's implicit/explicit preferences for items. The goal is to predict those not-yet-seen/bought/clicked-on items that could be of interest to a user.

This task can be accomplished in different ways. In the simplest approach [Allan, 1998], the prediction for a not-yet-seen item *d* for a user *u* is based on a "voting" mechanism considering the *k* most similar items (in our scenario, movies) to the item *d*. Each of them expresses a "vote" for *d* if the user *u* watched or rated it. For instance, if the current user liked 4 out of k=5 of the most similar items to *d*, the system may guess that the chance that the user will also like *d* is relatively high.

Another (more accurate) approach is inspired by collaborative filtering, and specifically item-based collaborative filtering recommendations [Sarwar et al., 2001, and Deshpande and Karypis, 2004]. This approach involves predicting the interest of a user in a specific item by considering the sum of all the similarities of the target item to the other items the user interacted with before:

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

Here, *Items(u)* returns all the items the user has interacted with (liked, watched, bought, clicked on). The returned value can be used to rank all the not-yet-seen items and return the top *k* to the user as recommendations.

In the following listing, the final step of providing recommendations for this third scenario is implemented.

Listing 4.6 Code for getting a ranked list of items for the user

```
def recommendTo(self, user_id, k):
    dtype = [('movieId', 'U10'), ('value', 'f4')]
    top_movies = np.array([], dtype=dtype)
    query = """
        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_TO]->(targetMovie)
        RETURN targetMovie.movieId as movieId, sum(r.weight)/count(r) as
     relevance
        order by relevance desc
        LIMIT %s
    """
    with self._driver.session() as session:
        tx = session.begin_transaction()
        for result in tx.run(query % (k), {"userId": user_id}):
            top_movies = np.concatenate((top_movies,
     np.array([(result["movieId"], result["relevance"])], dtype=dtype)))

    return top_movies
```

This function provides the recommendations to the user.

This query returns the recommendations; it requires the model built previously.

When you run this code, you'll notice that it's extremely fast. Once the model is created, providing recommendations takes only a few milliseconds.

Other approaches can also be used, but they're out of the scope of this chapter and this book. The main purpose here was to show how, once you've defined a proper model for items, users, and the interaction between them, multiple approaches can be used for providing recommendations without changing the base graph model defined.

EXERCISE

Rewrite the method that computes the similarity between items to use a different function than cosine similarity, like the Jaccard index, Dice coefficient, or Euclidian distance.

4.2.4 Advantages of the graph approach

In this chapter we discussed how to create a content-based recommendation engine using graphs and graph models for storing different types of information useful as input and output of several steps of the recommendation process.

Specifically, the main aspects and advantages of the graph-based approach to content-based recommendations are:

- Meaningful information must be stored as unique node entities in the graph so that these entities can be "shared" across items and users.
- Converting user–item data into user–feature data is a trivial task when the metainformation is available and is meaningful. It only requires a query to compute and materialize it.
- It's possible to extract several vector representations for both items and user profiles from the same graph model. This improves the feature selection, because it reduces the effort required to try different approaches.
- It's possible to store different similarity values computed using different functions and use them in combination.
- The code presented also showed how easy it is to switch between different models or even combine them if they're described by a proper graph model.

The great advantage here is the flexibility provided by the graph representation of the information, enabling the same data model to serve many use cases and scenarios with small adaptations. Furthermore, all the scenarios can coexist in the same database. This frees the data scientists and data engineers from having to deal with multiple representations of the same information. These advantages are shared by all the methods described in the following chapters about recommendations.

Summary

This chapter introduced you to graph-based data modeling techniques. In this first chapter on the topic, we focused on recommendation engines, exploring how to model data sources used for training, how to store the resulting model, and how to access it to make predictions.

In this chapter you learned:

- How to design a graph model for a user–item as well as a user–feature dataset
- How to import data from the original format into the graph model you've designed
- How to project user profile and item data and metadata into a vector space model
- How to compute similarities between user and item profiles and among pairs of items using cosine similarity and other functions
- How to store item similarities in the graph model
- How to query the resulting model to perform predictions and recommendations
- How to design and implement a graph-powered recommendation engine from end to end, using different approaches of increasing complexity

References

[Domingos, 2012] Domingos, Pedro. "A Few Useful Things to Know About Machine Learning." *Communications of the ACM* 55:10 (2012): 78–87. doi: http://dx.doi.org/10.1145/2347736.2347755

[Cypher Documentation] Cypher Query Language Reference, Version 9. https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf.

[Robinson et al., 2015], Robinson, Ian, Jim Webber, and Emil Eifrim. *Graph Databases*. 2nd ed. Sebastopol, CA: O'Reilly, 2015.

[Vukotic et al., 2014] Vukotic, Aleksa, Dominic Fox, Jonas Partner, Nicki Watt, and Tareq Abedrabbo. *Neo4j in Action*. Shelter Island, NY: Manning, 2014.

[Ricci et al., 2015] Ricci, Lior Rokach, and Bracha Shapira. *Recommender Systems Handbook*. 2nd ed. New York: Springer, 2015.

[Jannach et al., 2010] Jannach, Dietmar, Markus Zanker, Alexander Felfernig, and Gerhard Friedrich. *Recommender Systems: An Introduction*. Cambridge, UK: Cambridge University Press, 2010. doi: http://dx.doi.org/10.1017/CBO9780511763113

[Adomavicius et al., 2011] Adomavicius, Gediminas, Bamshad Mobasher, Francesco Ricci, and Alexander Tuzhilin. "Context-Aware Recommender Systems." *AI Magazine* 32:3 (2011): 67–80, 2011. doi: https://doi.org/10.1609/aimag.v32i3.2364

[Ullman and Rajaraman, 2011], Ullman, Jeffrey David, and Anand Rajaraman. *Mining of Massive Datasets*. New York: Cambridge University Press, 2011.

[Dice, 1945] Dice, Lee Raymond. "Measures of the Amount of Ecologic Association Between Species." *Ecology* 26:3 (1945): 297–302. doi: http://dx.doi.org/10.2307/1932409. JSTOR 1932409.

[Allan, 1998] Allan, James. "Topic Detection and Tracking Pilot Study Final Report." *Proceedings of the DARPA Broadcast News Transcription and Understanding Workshop* (1998): 194–218.

[Deshpande and Karypis, 2004] Deshpande, Mukund, and George Karypis. "Item-Based Top-*N* Recommendation Algorithms." *ACM Transactions on Information Systems* 22:1 (2004): 143–177. DOI: http://dx.doi.org/10.1145/963770.963776

[Sarwar et al., 2001] Sarwar, Badrul, George Karypis, Joseph Konstan, and John Riedl. "Item-Based Collaborative Filtering Recommendation Algorithms." *Proceedings of the 10th International World Wide Web Conference* (2001): 285–295. DOI: https://doi.org/10.1145/371920.372071

7 Context-aware and hybrid recommendation

This chapter covers

- Implementing a recommendation engine that takes into account the user's context
- Designing graph models for context-aware recommendation engines
- Importing existing datasets into the graph models designed
- Combining multiple recommendation approaches

This chapter introduces into the recommendation scenario another variable that the previous approaches ignored: *context*. The specific conditions in which the user expresses a desire, preference, or need have a strong influence on their behavior and expectations. Different techniques exist to consider the user's context during the recommendation process. We'll cover the main ones in this chapter.

Furthermore, to complete our overview of recommendation engine models and algorithms, we'll see how it's possible to use a hybrid approach that combines the different types of systems presented so far. Such an approach enables us to create a

unique and powerful recommendation ecosystem capable of overcoming all the issues, limitations, and drawbacks of each individual recommendation method.

7.1 The context-based approach

> Suppose you want to implement a mobile application that provides recommendations about movies to watch at the cinema (we'll call it Reco4.me). Using context-aware techniques, you can take into account environmental information during the recommendation process, suggesting, for instance, movies playing at cinemas close to the user's current location.

Let's further refine the scenario with a concrete example. Suppose you're in London, and you want to find a movie to watch at a nearby cinema. You take out your phone, open the Reco4.me app, and hope for good recommendations. What kinds of recommendations do you expect? You want to know about movies *currently* playing in cinemas *close* to where you are. Ideally you also want to have recommendations that suit your preferences. I don't know about you, but for me the context changes my preferences a lot. When I'm alone at home, I love to watch action or fantasy movies. When I'm with my kids, I prefer to watch cartoons or family movies. When I'm with my wife, "we" prefer to watch chick flicks or romcoms. The app should take into account this environmental information and provide accurate recommendations that suit the user's current context.

This example shows how essential it can be to consider context in a recommender system, because it may have a subtle but powerful influence on user behaviors and needs. Considering the context can therefore dramatically affect the quality of the recommendations, converting what might be a very good tip in some circumstances into a useless suggestion in others. This is true not only in scenarios such as the one described here, but in many others too. Think, for instance, about how you use an e-commerce site like Amazon. You might use it to buy a book for yourself, or a gift for your fiancé, or a toy for your kids. You have a single account, but your behavior and your preferences are driven by the specific needs you have while you're navigating the site. Although it could be useful to see recommendations about books that might be of interest to you while you're looking for a skateboard for your son, it would be more effective to get suggestions that suit your current needs, based on previous gifts you've bought for your kids.

Traditional recommender systems, such as those based on the content-based and collaborative filtering approaches discussed in the chapters 4 and 5, tend to use fairly simple user models. For example, user-based collaborative filtering models the user as a vector of item ratings. As additional observations are made about users' preferences, the user models are extended, and the full collection of user preferences is used to generate recommendations or make predictions. This approach, therefore, ignores the notion of "situated actions" [Suchman, 1987]—the fact that users interact with the system within a particular context or specific scope, and that preferences for items within one context may be different from those in another context. In many applica-

tion domains, a context-independent representation may lose predictive power because potentially useful information from multiple contexts is aggregated.

More formally, interactions between users and items exhibit a multifaceted nature. User preferences are typically not fixed and may change with respect to a specific situation. Going back to the example of the Reco4.me app, a simplified schema of the possible contextual information is depicted in figure 7.1.

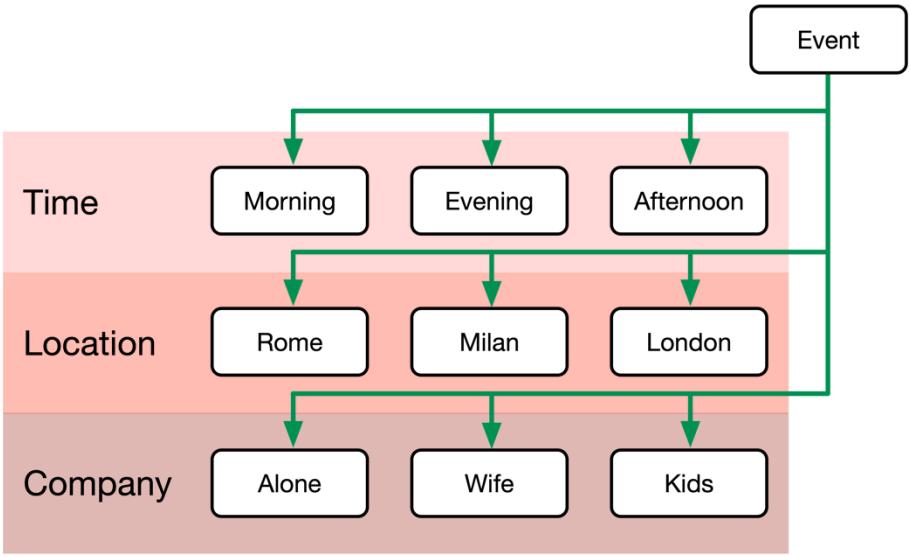


Figure 7.1 **Contextual information for the app Reco4.me.**

This is a small subset of the types of contextual information that could be considered. Context might include the season of the year or day of the week, the type of electronic device the user is using, the user's mood—it can be almost anything [Bazire and Brézillon, 2005; Doerfel at al., 2016].

It's worth mentioning here too that the contextual information is defined by what the system knows or can guess about the specific conditions in which an action or interaction occurs.

In content-based and collaborative filtering approaches, the recommendation problem is defined as a prediction problem in which, given a user profile (defined in different ways) and a target item, the recommender system's task is to predict that user's rating of or interest in that item, reflecting the degree of user's preference for the item. Specifically, a recommender system tries to estimate a rating function:

$$f: \text{User x Item} \rightarrow \text{ratings}$$

Such a function maps user–item pairs to an ordered set of score values. Note that f can be viewed as a general-purpose utility (or preference) measure for user–item pairs. The ratings for all user–item pairs aren't known, and therefore, must be inferred. This is why we talk about "prediction." Once an initial set of ratings have been collected,

implicitly or explicitly, a recommender system tries to estimate the rating values for items that haven't yet been rated by the users. From now on we'll refer to these traditional recommender systems as *two-dimensional (2D)*, because they consider only the *Users* and *Items* dimensions as input in the recommendation process.

In contrast, *context-aware* recommender systems try to incorporate or use additional environmental evidence (beyond information about users and items) to estimate user preferences for unseen items. When such contextual evidence can be incorporated as part of the input to the recommender system, the rating function can be viewed as "multidimensional":

$$f: \text{User x Item x } Context_1 \text{x } Context_2 \text{ x ... x } Context_n \rightarrow \text{ratings}$$

In this formula, *Context* represents a set of factors that further delineate the conditions under which the user–item pair is assigned a particular rating. The underlying assumption of this extended model is that user preferences for items aren't only a function of the items themselves, but of the context in which the items are being considered.

Context information represents a set of explicit variables that model contextual factors in the underlying domain (time, location, surroundings, device, occasion, and so on). Regardless of how the context is represented, context-aware recommenders must be able to obtain contextual information that corresponds to the user's activity (for example, making a purchase or rating an item). Such information, in a context-aware recommender system, has a twofold purpose:

- It's part of the learning and modeling process (used, for example, for discovering rules, segmenting users, or building regression models).
- For a given target user and target item, the system must be able to identify the values of specific contextual variables as part of the user's ongoing interaction with the system. This information is used to ensure the right recommendation is delivered, considering the context.

Contextual information can be obtained in many different ways, either explicitly or implicitly. Explicit contextual information may be obtained from users themselves or from sensors designed to measure specific physical or environmental information [Frolov and Oseledets, 2016]. In certain cases, however, contextual information must be derived or inferred from other observed data. Here some examples:

- *Explicit:* The application may ask an individual looking for a restaurant recommendation to specify whether they're going on a date or going out with coworkers for a business dinner.
- *Explicit/implicit:* If the restaurant recommender is a mobile app, additional contextual information can be obtained through the device's GPS or other sensors about the location, time, and weather conditions.
- *Implicit:* An e-commerce system may attempt, using previously learned models of user behavior, to distinguish (for example) whether the user is likely to be purchasing a gift for their spouse or a work-related book.

Approaches to implicitly infer contextual information typically require building predictive models from historical data [Palmisano et al., 2008].

Figure 7.2 shows the mental model of a context-aware recommendation engine. The user's events, input of the system, are contextualized. They're converted in a graph, then the process of building a model and provide recommendation can start.

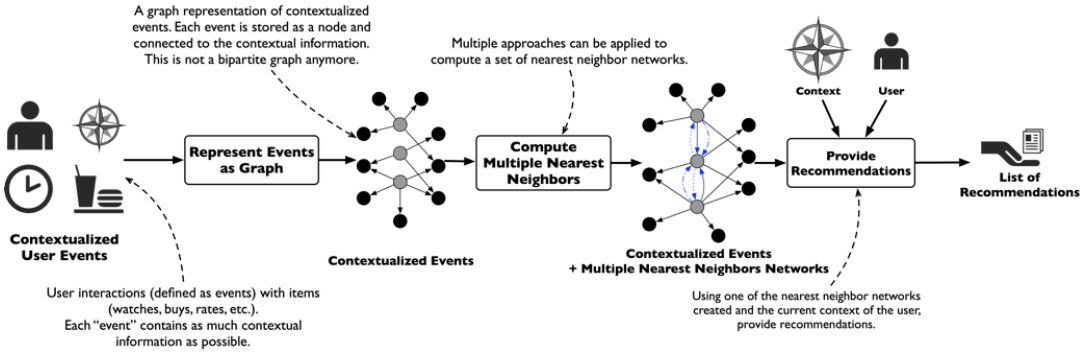


Figure 7.2 A graph-powered context-aware recommender system diagram.

7.1.1 Representing contextual information

In content-based and collaborative filtering approaches, the user–item interactions—buy, click, view, rate, watch, and so on—are represented as a two-dimensional matrix that we've defined as the User x Item (U x I) dataset. Such a matrix can easily be represented as a bipartite graph, where one set of vertices represents the users and the other set represents the items. The interaction is modeled via a relationship between the user (the subject of the event) and the item (the object of the event).

In context-aware recommendation systems, each interaction event brings more information with it. It's described not only by the user and the item, but also by all the environmental information that contextualizes the situated action. For example, if a user is watching a movie at home with their kids in the evening, the contextual information is composed of:

- *Time:* Evening, weekday
- *Company:* Kids
- *Location:* Home

This is only a subset of the relevant information that can describe the event "watch." Other information could include the device being used, the mood of the users, the ages of the viewers, or the occasion (date night, party, kids' bedtime movie). Certain variables may be discrete (contextual information with defined sets of values, such as device and location), while others are continuous (numerical values like age). In the latter case, it's generally preferable to discretize them somehow. For instance, in the case of age you might have "buckets" of 0–5, 6–14, 15–21, 22–50, and over 50. It depends on the specific requirements of the recommendation engine.

The resulting dataset, which represents the input for the recommendation process, can no longer be represented as a simple two-dimensional matrix. It requires an *N*-dimensional matrix, where two dimensions are the users and the items, and the others represent contexts. In the example considered here, the dataset is a five-dimensional matrix:

$$\text{dataset} = \text{User x Item x Location x Company x Time}$$

Each interaction or event cannot be described simply by two elements and the relationship between them. In the best case, when all the contextual information is available, it requires three other elements. This means that we cannot use a simple relationship in a bipartite graph to represent an event. To represent relationships among five vertices, we need a *hypergraph*. In mathematics, a hypergraph is a generalization of the graph in which one edge can connect any number of vertices. However, in most graph databases (including Neo4j) it isn't possible to represent an *n*-vertex relationship.

The solution is to "materialize" events as nodes and connect each event node with all the elements, or dimensions, that describe the event. The result looks like figure 7.3.

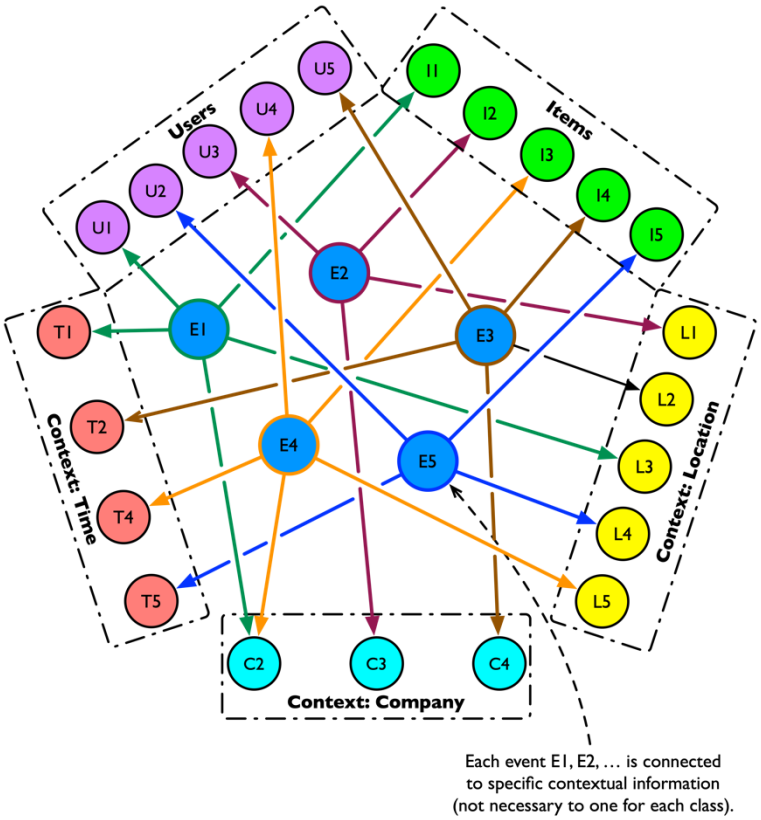


Figure 7.3
An *n*-partite graph representing contextual information about events.

The new graph representation is a six-partite graph, because we have users, items, location information, time information, and company information plus the events. This graph represents the input for the next steps in the recommendation process depicted in figure 7.2.

Passing from a two-dimensional representation to an n-dimensional representation ($n = 5$ in our case) makes data sparsity an even bigger concern. It will be hard to find many events for multiple users that happen in the exact same circumstances. This problem is exacerbated when we have detailed contextual information (higher values of n), but it can be mitigated by introducing hierarchies in the contextual information. Figure 7.4 shows some examples of possible hierarchies—represented in the

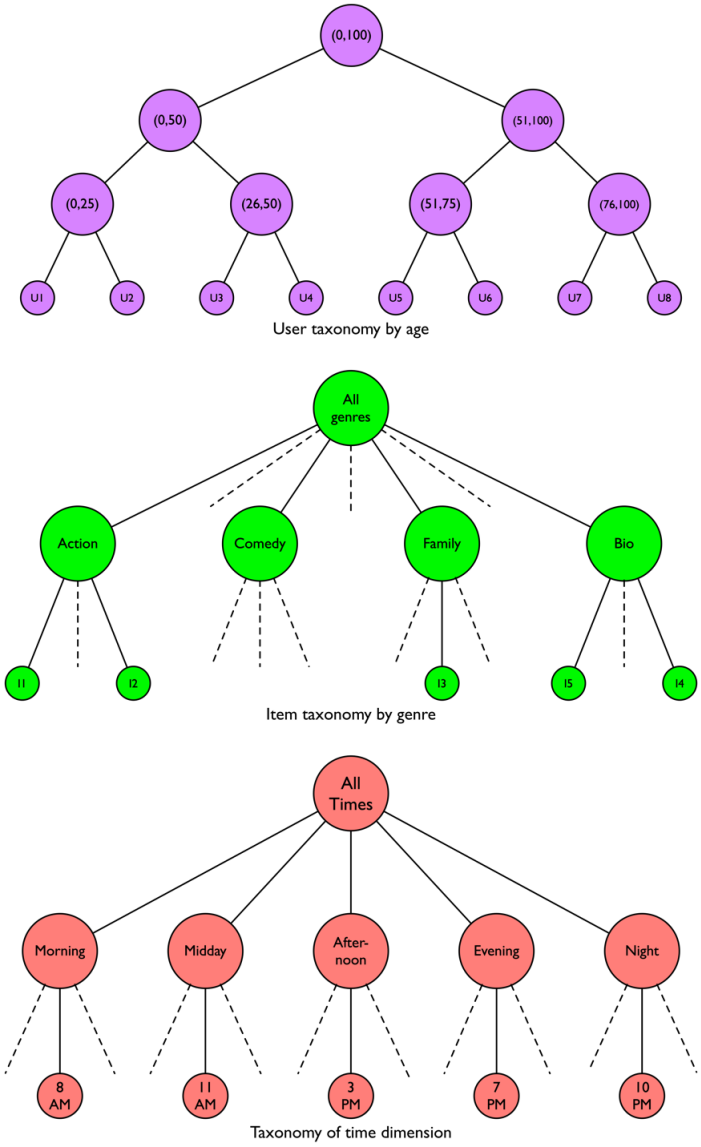


Figure 7.4 Taxonomies of users, items, and times.

form of a graph—considering part of the contextual information for our specific scenario. These hierarchies are defined as taxonomies.

These taxonomies will be used during the recommendation phase to solve the sparsity problem and enable us to provide recommendations even when we don't have much information about the current user's specific context.

For this section we'll use the DePaulMovie dataset[12] [Zheng et al., 2015], which contains data collected from surveys conducted with students. It contains data from 97 users about 79 movies, rated in different contexts (time, location, and companion). Such a dataset matches perfectly with our needs, and it's often used to perform comparisons of different context-aware recommender systems [Ilarri et al., 2018].

To begin, let's import the data from the DePaulMovie dataset selected for the following listing. Please run the code using a fresh database, you can clean it up[13] or decide to use a different one and keep the previous you created in a previous chapters for further experimenting.

Listing 7.1 Importing data from the DePaulMovie dataset

```
def import_event_data(self, file):                          # Entry point for importing the data from the CSV file.
    with self._driver.session() as session:
        self.executeNoException(session, "CREATE CONSTRAINT ON (u:User)
           ASSERT u.userId IS UNIQUE")                      # Queries which create the constraints in the database to avoid duplicates and speed up access.
        self.executeNoException(session, "CREATE CONSTRAINT ON (i:Item)
           ASSERT i.itemId IS UNIQUE")
        self.executeNoException(session, "CREATE CONSTRAINT ON (t:Time)
           ASSERT t.value IS UNIQUE")
        self.executeNoException(session, "CREATE CONSTRAINT ON
           (l:Location) ASSERT l.value IS UNIQUE")
        self.executeNoException(session, "CREATE CONSTRAINT ON
           (c:Companion) ASSERT c.value IS UNIQUE")

        j = 0;
        with open(file, 'r+') as in_file:
            reader = csv.reader(in_file, delimiter=',')
            next(reader, None)
            tx = session.begin_transaction()
            i = 0;
            query = """                                     # Query which in one shot creates the events and connects them to the related dimensions.
                    MERGE (user:User {userId: $userId})
                    MERGE (time:Time {value: $time})
                    MERGE (location:Location {value: $location})
                    MERGE (companion:Companion {value: $companion})
                    MERGE (item:Item {itemId: $itemId})
                    CREATE (event:Event {rating:$rating})
                    CREATE (event)-[:EVENT_USER]->(user)
                    CREATE (event)-[:EVENT_ITEM]->(item)
                    CREATE (event)-[:EVENT_LOCATION]->(location)
                    CREATE (event)-[:EVENT_COMPANION]->(companion)
                    CREATE (event)-[:EVENT_TIME]->(time)
```

[12] https://github.com/JDonini/DePaulMovie-Recommender-System

[13] To clean the existing database, you could run "MATCH (n) DETACH DELETE n" buti t could take longer. Another option is to stop the database and purge the data directory.

```
            """

        for row in reader:
            try:
                if row:
                    user_id = row[0]
                    item_id = strip(row[1])
                    rating = strip(row[2])
                    time = strip(row[3])
                    location = strip(row[4])
                    companion = strip(row[5])
                    tx.run(query, {"userId": user_id, "time": time,
                      "location": location, "companion": companion,
                      "itemId": item_id, "rating": rating})
                    i += 1
                    j += 1
                    if i == 1000:
                        tx.commit()
                        print(j, "lines processed")
                        i = 0
                        tx = session.begin_transaction()
            except Exception as e:
                print(e, row)
        tx.commit()
        print(j, "lines processed")
print(j, "lines processed")
```

In the complete version in the code repository, you'll notice that I've also imported information about the movies. This will be useful for getting a sense of the results and also for the following exercises.

Exercises

Once you've imported the data, play with the graph database. Here are several things to try:

- Look for the most frequent contextual information—the most frequent time for watching a movie, for instance.
- Look for the most active users and check the variability of their contextual information.
- Try adding taxonomies and see if the results of the preceding queries change.
- Search for movies or genres that are commonly watched during the week and those that are more often watched at the weekend.

7.1.2 Providing recommendations

Classical recommender systems provide recommendations using limited knowledge of user preferences (that is, user preferences for some subset of the items), and the input data for these systems is typically based on records of the form <user, item, rating>. As described in the previous chapters, the recommendation processes generally use the U x I matrix to create a model and provide recommendations based only on user interaction and preferences.

In contrast, context-aware recommender systems typically deal with data records of the form <user, item, context1, context2, …, rating>, where each specific record

includes not only how much a given user liked a specific item, but also contextual information about the conditions in which the user interacted with the item (for example, context1 = Saturday, context2 = wife, and so on). This "rich" information is used (we'll see how later) to create the model. Furthermore, information about the user's current context can be used in various stages of the recommendation process, leading to several different approaches to context-aware recommender systems.

From an algorithmic perspective, the vast majority of the context-aware recommendation approaches:

- Take as input the contextualized (extended) User x Item dataset in the form U x I x C_1 x C_2 x … x C_n, where C_i is an additional contextual dimension.
- Produce a list of contextual recommendations i_1, i_2, i_3, ... for each user u, based on the current context of the user.

Based on how the contextual information, the current user, and the current item are used during the recommendation process, context-aware recommendation systems can take one of the three forms shown in figure 7.5.

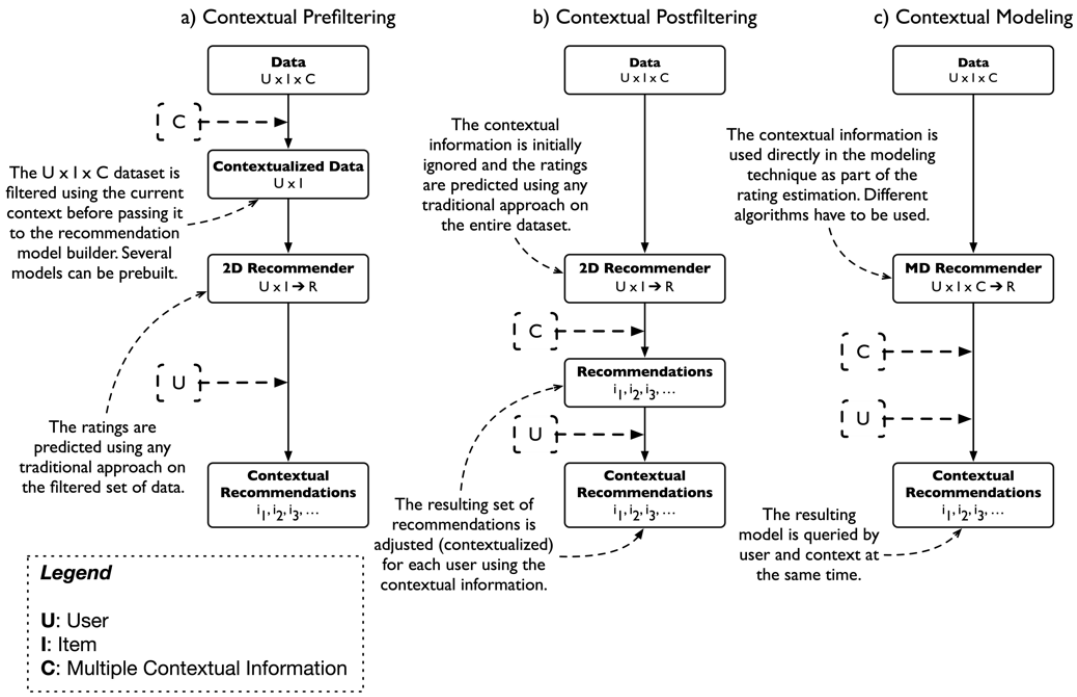


Figure 7.5 The three forms of a context-aware recommender system.

The three types of context-aware recommendation system are [Ilarri et al., 2018]:

- *Contextual prefiltering (or contextualization of recommendation input):* In this paradigm information about the current context, c is used only for selecting the relevant set of data, and ratings are predicted using any traditional 2D recommender

system on the selected data. For efficiency, several models must be precomputed, considering the most probable combinations of contexts.

- *Contextual postfiltering (or contextualization of recommendation output):* In this paradigm contextual information is initially ignored, and the ratings are predicted using any traditional 2D recommender system on the entire dataset. Then the resulting set of recommendations is adjusted (contextualized) for each user using the contextual information. Only one model is built, so it's easier to manage, and the contextual information is used only during the recommendation phase.
- *Contextual modeling (or contextualization of recommendation function):* In this paradigm contextual information is used directly in the modeling technique as part of the model building.

The following subsections describe the three paradigms in more detail, highlighting the role of the graph approach for each (especially the first two).

CONTEXTUAL PREFILTERING

As shown in figure 7.6, the contextual prefiltering approach uses contextual information to select the most relevant User x Item matrices and create models from them. It then generates recommendations through the inferred models.

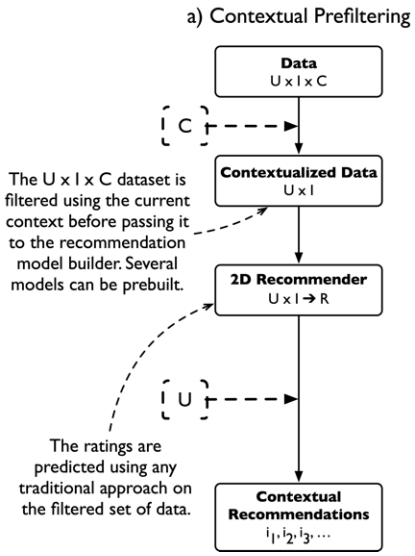


Figure 7.6 Contextual prefiltering.

Once the User x Item datasets are extracted, any of the numerous traditional recommendation techniques proposed in the literature (some of them have been discussed here in the previous chapters on recommendation) can be used to build the model and provide recommendations. This represents one of the biggest advantages of this first approach to context-aware recommendation engines.

Note that the prefiltering approach is related to the task of building multiple local models in machine learning and data mining based on the most relevant combination

of contextual information. Rather than building the global rating estimation model utilizing all the available ratings, the prefiltering approach builds (prebuilds in the real scenario) a local rating estimation model that uses only the ratings pertaining to the user-specified criteria for the recommendation (for example, Saturday or weekday).

When using this approach, the context c essentially serves as a filter for selecting relevant rating data. An example of a contextual data filter for a movie recommender system would be if a person wants to see a movie on Saturday, only the Saturday rating data is used to recommend movies. Of course, extracting the relevant dataset, building the model, and providing the recommendations requires time, especially if the dataset is big. For this reason, multiple versions are precomputed using most relevant combinations of contextual information.

In the graph approach proposed and considering the model depicted in figure 7.3, performing such prefiltering consists in selecting the relevant events by running a query such as the following.

Query 7.1 Filtering events based on relevant contextual information

```
MATCH (event:Event)-[:EVENT_ITEM]->(item:Item)
MATCH (event)-[:EVENT_USER]->(user:User)
MATCH (event)-[:EVENT_TIME]->(time:Time)
MATCH (event)-[:EVENT_LOCATION]->(location:Location)
MATCH (event)-[:EVENT_COMPANION]->(companion:Companion)
WHERE time.value = "Weekday"
AND location.value = "Home"
AND companion.value = "Alone"
RETURN user.userId, item.itemId, event.rating
```

In this query we're considering only the events that happen during a weekday alone at home. The output is a "slice" of our multidimensional matrix. If we instead want to get a User x Item matrix for the context <Weekend, Cinema, Partner>, the query would look like the following.

Query 7.2 Filtering events based on different contextual information

```
MATCH (event:Event)-[:EVENT_ITEM]->(item:Item)
MATCH (event)-[:EVENT_USER]->(user:User)
MATCH (event)-[:EVENT_TIME]->(time:Time)
MATCH (event)-[:EVENT_LOCATION]->(location:Location)
MATCH (event)-[:EVENT_COMPANION]->(companion:Companion)
WHERE time.value = "Weekend"
AND location.value = "Cinema"
AND companion.value = "Partner"
RETURN user.userId, item.itemId, event.rating
```

The resulting matrices will be totally different.

Of course, it isn't necessary to specify all the contextual information. Several of the dimensions can be ignored. For instance, we could have a context <Cinema, Partner> in which the time dimension is considered irrelevant. The query in this case will look like the following.

Query 7.3 Filtering events considering only two items of contextual information

```
MATCH (event:Event)-[:EVENT_ITEM]->(item:Item)
MATCH (event)-[:EVENT_USER]->(user:User)
MATCH (event)-[:EVENT_LOCATION]->(location:Location)
MATCH (event)-[:EVENT_COMPANION]->(companion:Companion)
WHERE location.value = "Cinema"
AND companion.value = "Partner"
RETURN user.userId, item.itemId, event.rating
```

The graph model is highly flexible. As mentioned previously, once the data is filtered, any classic method can be applied for building the model and providing recommendations. Suppose that we want to use the collaborative approach, and more specifically the nearest neighbor approach. We have to compute similarities among items, users, or both. The resulting similarities can be stored as "simple" relationships between items and/or users, but in this way the information about the prefiltering condition is lost. A property can be added on the relationships to keep track of the sources used for computing them, but it's difficult to query and, most importantly, this approach doesn't leverage the graph capabilities to speed up navigation through nodes and relationships.

The best modeling choice in this case is to *materialize* the similarities using nodes and connect them to the relevant contextual information used for computing them: the prefiltering conditions. At this point the resulting graph model will look like figure 7.7.

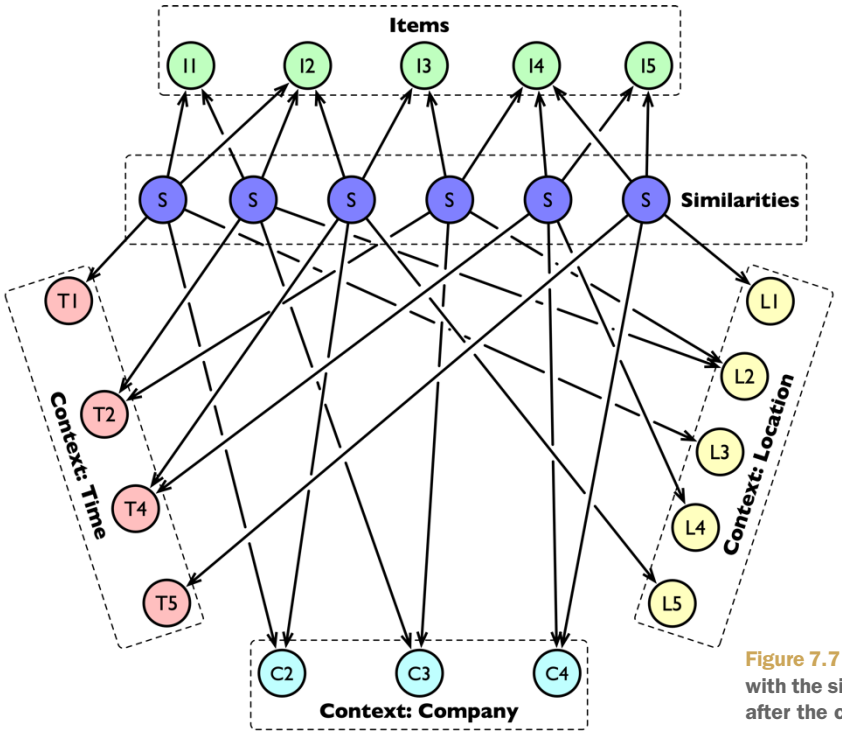


Figure 7.7 Graph model with the similarity nodes after the computation.

This model is easy to navigate during the recommendation process. We can assign an ID to each set of contextual information to make querying easier; this isn't mandatory, but it's helpful because it allows a faster and simpler access to it. We can then get the *k*-NN for a specific context using a query such as the following.[14]

Query 7.4 Getting the k-NN given specific contextual information

```
MATCH p=(n:Similarity)-->(i)
WHERE n.contextId = 1
RETURN p
limit 50
```

Assigning IDs to specific sets of contextual information allows us to query by context ID.

The following listing allows you to create such a graph model.

Listing 7.2 Code for computing and storing similarities in the prefiltering approach

Entry point for computing similarities in prefiltering. The context parameter specifies the contextual information. This function has to be run multiple times for multiple combinations of contextual information.

```
def compute_and_store_similarity(self, contexts):
    for context in contexts:
        items_VSM = self.get_item_vectors(context)
        for item in items_VSM:
            knn = self.compute_knn(item, items_VSM.copy(), 20);
            self.store_knn(item, knn, context)
```

Computes the similarities (the function is the same of previous listing)

```
def get_item_vectors(self, context):
    list_of_items_query = """
                MATCH (item:Item)
                RETURN item.itemId as itemId
            """
    context_info = context[1].copy()
    match_query = """
                MATCH (event:Event)-[:EVENT_ITEM]->(item:Item)
                MATCH (event)-[:EVENT_USER]->(user:User)
            """
    where_query = """
                WHERE item.itemId = {itemId}
            """
    if "location" in context_info:
        match_query += "MATCH (event)-[:EVENT_LOCATION]->(location:Location) "
        where_query += "AND location.value = {location} "

    if "time" in context_info:
        match_query += "MATCH (event)-[:EVENT_TIME]->(time:Time) "
        where_query += "AND time.value = {time} "

    if "companion" in context_info:
        match_query += "MATCH (event)-[:EVENT_COMPANION]->(companion:Companion) "
        where_query += "AND companion.value = {companion} "

    return_query = """
```

Prefilters the dataset, considering the relevant contextual information, and returns the usual item list with related sparse vectors

if statements that change the query according to the contextual information.

[14] The query can be run after the code completed the KNN creation. Here the purpose is to show how to query the model.

```
                WITH user.userId as userId, event.rating as rating
                ORDER BY id(user)
                RETURN collect(distinct userId) as vector
            """

    query = match_query + where_query + return_query
    items_VSM_sparse = {}
    with self._driver.session() as session:
        i = 0
        for item in session.run(list_of_items_query):
            item_id = item["itemId"];
            context_info["itemId"] = item_id
            vector = session.run(query, context_info)
            items_VSM_sparse[item_id] = vector.single()[0]
            i += 1
            if i % 100 == 0:
                print(i, "rows processed")
        print(i, "lines processed")
    print(len(items_VSM_sparse))
    return items_VSM_sparse

def store_knn(self, item, knn, context):
    context_id = context[0]
    params = context[1].copy()
    with self._driver.session() as session:
        tx = session.begin_transaction()
        knnMap = {a: b for a, b in knn}
        clean_query = """
            MATCH (s:Similarity)-[:RELATED_TO_SOURCE_ITEM]->(item:Item)
            WHERE item.itemId = $itemId AND s.contextId = $contextId
            DETACH DELETE s
        """

        query = """
            MATCH (item:Item)
            WHERE item.itemId = $itemId
            UNWIND keys($knn) as otherItemId
            MATCH (other:Item)
            WHERE other.itemId = otherItemId
            CREATE (similarity:Similarity {weight: $knn[otherItemId],
               contextId: $contextId})
            MERGE (item)<-[:RELATED_TO_SOURCE_ITEM]-(similarity)
            MERGE (other)<-[:RELATED_TO_DEST_ITEM ]-(similarity)
        """

        if "location" in params:
            query += "WITH similarity MATCH (location:Location {value:
               $location}) "
            query += "MERGE (location)<-[:RELATED_TO]-(similarity) "

        if "time" in params:
            query += "WITH similarity MATCH (time:Time {value: $time}) "
            query += "MERGE (time)<-[:RELATED_TO]-(similarity) "

        if "companion" in params:
            query += "WITH similarity MATCH (companion:Companion {value:
               $companion}) "
            query += "MERGE (companion)<-[:RELATED_TO]-(similarity) "
```

Query that cleans up the previous stored model

Query that creates the new similarity nodes and connects them to the related items and contextual information

if statements which modify the query according to the filter conditions.

```
        tx.run(clean_query, {"itemId": item, "contextId": context_id})
        params["itemId"] = item
        params["contextId"] = context_id
        params["knn"] = knnMap
        tx.run(query, params)
        tx.commit()

def compute_knn(self, item, items, k):
    knn_values = []
    for other_item in items:
        if other_item != item:
            value = cosine_similarity(items[item], items[other_item])
            if value > 0:
                knn_values.append((other_item, value))
    knn_values.sort(key=lambda x: -x[1])
    return knn_values[:k]
```

As mentioned previously, sometimes the exact context can be too narrow. Consider, for example, the context of watching a movie with your partner in a cinema on Saturday—or, more formally, c = <Partner, Cinema, Saturday>. Using this exact context as a data-filtering query may be problematic, because there may not be enough data available for accurate rating prediction. To address this issue, Adomavicius and Tuzhilin [2005] suggest generalizing the filtering conditions by aggregating more narrow context details that may not be significant. These generalizations are the taxonomies we discussed earlier, several examples of which were shown in figure 7.4. For instance, Saturday can become Weekend, while Monday to Friday are considered Weekday. Not only is it easy to represent such hierarchies or aggregations in a graph, but it's also easy to query them. Using broader concepts while filtering data can deliver better results.

When considering the prefiltering approach, it's important to determine whether the local (specific to part of the contextual information) model it generates outperforms the global model of the traditional 2D technique, where all the information associated with the contextual dimensions is simply ignored. For example, it's possible that it's better to use contextual prefiltering to recommend movies to watch in a movie theater on the weekend but use the traditional 2D technique (ignoring the contextual information) to recommend movies to watch at home on demand.

The trade-off during the calculation of an unknown rating in this case is between:

- Using more specific (in the sense of narrower contextual information) but relevant data (the prefiltering)
- Using all the data available (the traditional 2D recommendation)

There's no rule for this—which approach will be more successful depends on many factors, such as the type of contextual information, the application domain, the user behaviors, the amount and sparsity of data available, and so on. That's why the prefiltering recommendation method may outperform traditional 2D recommendation techniques in some contexts but not others. Based on this observation, Adomavicius and Tuzhilin [2005] propose combining a number of contextual prefilters with the traditional 2D technique (where no filtering is done).

CONTEXTUAL POSTFILTERING

As shown in figure 7.8, the contextual postfiltering approach ignores contextual information completely during model generation.

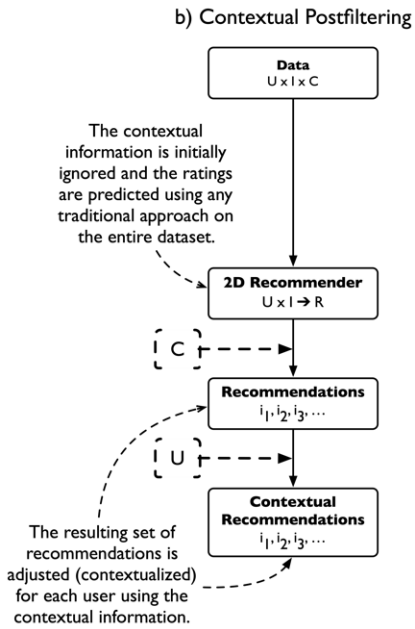


Figure 7.8 Contextual postfiltering.

Furthermore, the ranked list of all candidate items is computed regardless of the context. The postfiltering approach uses contextual information in a later phase to adjust the obtained recommendation list for each user. The adjustment to the top *N* items can be performed in two ways:

- *Filtering out* recommendations that are irrelevant in a given context
- *Adjusting* the ranking of recommendations in the list

For example, in our movie recommendation application Reco4.me, if the user only watches comedies at the weekend, the recommendation system could filter out all noncomedies in the recommendation list for weekend viewing or penalize them by reducing their ratings.

Which method is preferable will depend on the application. For example, Panniello et al. [2009] performed an experimental comparison of the exact (that is, nongeneralized) prefiltering method with postfiltering methods they called *Weight* and *Filter*, using several real-world e-commerce datasets. Their results showed that the Weight postfiltering method outperformed the exact prefiltering approach, which in turn outperformed the Filter method. Depending on your application, however, your results may vary.

The methods for filtering or adjusting rankings can be classified as *heuristic-based* or *model-based*. Heuristic postfiltering approaches focus on finding common item characteristics (attributes) for a given user in a given context (for example, preferred actors to watch Saturday in a cinema) and then uses these attributes to adjust the recommen-

dations. This method requires storing metadata about each item and searching for common patterns in user preferences.

In the graph model representing item metadata is straightforward, and multiple modeling techniques have been presented in the previous chapters (specifically for the content-based approach). Mixing such models with the User x Item x Contexts graph representation is a simple exercise; a possible result is presented in figure 7.9.

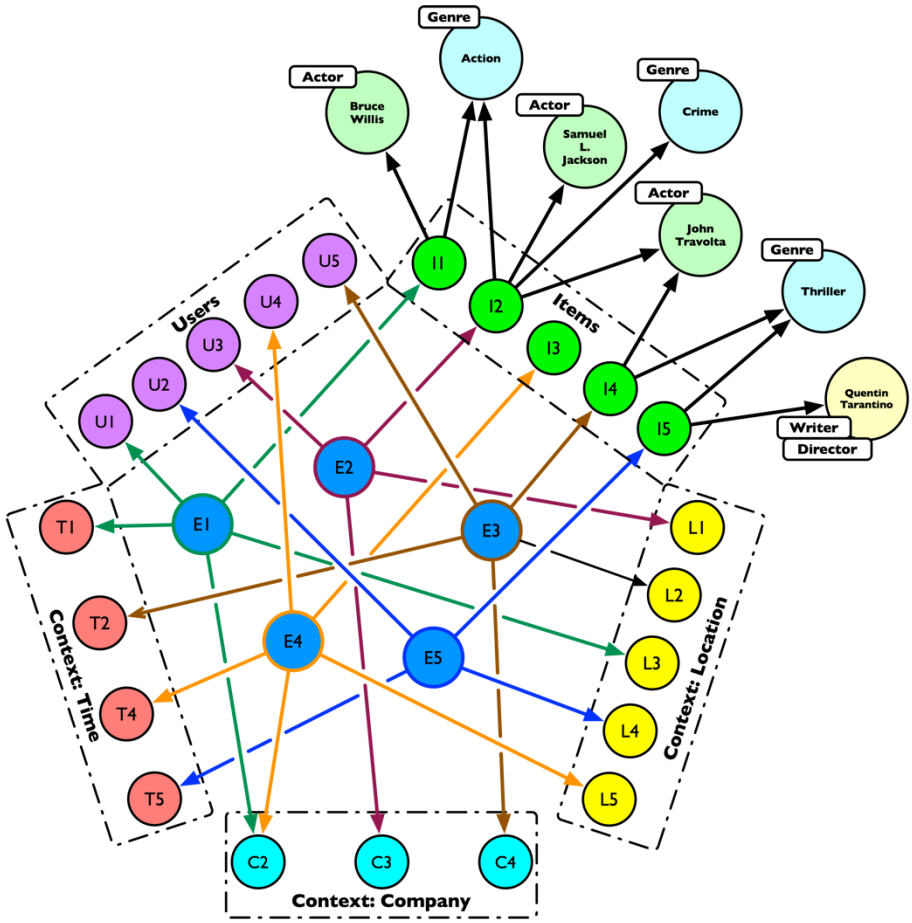


Figure 7.9 An *n*-partite graph representing contextual information of events plus the item attributes.

The DePaulMovie dataset contains references to IMDb IDs for each movie, so we can reuse the code we implemented in chapter 4 to get and add information from the IMDb. The code is presented in the file import_depaulmovie.py in the repository for this chapter.

After the import, queries such as the following can be used to compute "commonalities" based on contextual information for users. Note that the queries shown here focus on a specific user to prove the concept. They consider only two contexts, of all the possible combinations: <Cinema, Partner> and <Home, Alone>. We'll start with the <Cinema, Partner> context shown in the following query.

Query 7.5 Getting user preferences/profile for the context <Cinema, Partner>

```
MATCH (user:User)<-[:EVENT_USER]-(event:Event)
MATCH (event)-[:EVENT_ITEM]->(item:Item)-[]-(feature:Feature)
MATCH (event)-[:EVENT_LOCATION]->(location:Location)
MATCH (event)-[:EVENT_COMPANION]->(companion:Companion)
WHERE user.userId = "1032"
AND location.value = "Cinema"
AND companion.value = "Partner"
RETURN CASE 'Genre' IN labels(feature)
    WHEN true THEN feature.genre
    ELSE feature.name END AS feature, count(event) as occurrence
ORDER BY occurrence desc
```

The results of query 7.5 are shown in figure 7.10.

feature	occurrence
"Comedy"	4
"Romance"	4
"Drama"	4
"Action"	4
"Adventure"	3
"Roland Emmerich"	3
"Dan Brown"	3
"Tom Hanks"	2
"Al Jean"	2
"Mike Scully"	2
"Matt Groening"	2
"James L. Brooks"	2
"Animation"	2
"Dan Aykroyd"	2
"Fantasy"	2
"Family"	2
"Michael Bay"	2
"Graham Beckel"	2
"Randall Wallace"	2
"Chiwetel Ejiofor"	2
"Harald Kloser"	2
"Ron Howard"	2
"Thriller"	2
"Mystery"	2

Figure 7.10 Results of query 7.5.

From the results it's clear that when this user is watching a movie at the cinema with his partner, "he" prefers comedies, romances, dramas, and action movies. The preferred actors/directors follow the same logic. Now let's take a look at the <Home, Alone> context in the following query.

Query 7.6 Getting user preferences/profile for the context <Home, Alone>

```
MATCH (user:User)<-[:EVENT_USER]-(event:Event)
MATCH (event)-[:EVENT_ITEM]->(item:Item)-[]-(feature:Feature)
MATCH (event)-[:EVENT_LOCATION]->(location:Location)
MATCH (event)-[:EVENT_COMPANION]->(companion:Companion)
WHERE user.userId = "1032"
AND location.value = "Home"
AND companion.value = "Alone"
RETURN CASE 'Genre' IN labels(feature)
    WHEN true THEN feature.genre
    ELSE feature.name END AS feature, count(event) as occurrence
ORDER BY occurrence desc
```

The results of this query are shown in figure 7.11.
Is this the same user as before? The results here are totally different, showing the extent to which, the context plays a role in the user's preferences.

Results obtained in this way can be used to postfilter/fine-tune the results of a traditional collaborative filtering approach.

These preferences based on context can be precomputed and stored back in our graph model. The result will look like figure 7.12.

The types of nodes and relationships presented in the model in figure 7.12 can be created by running queries such as the following.

Query 7.7 Creating user preferences

```
MERGE (userPreference:UserPreference {userId: "1032", location:"Home",
     companion: "Alone"})
WITH userPreference
MATCH (user:User)<-[:EVENT_USER]-(event:Event)
MATCH (event)-[:EVENT_LOCATION]->(location:Location)
MATCH (event)-[:EVENT_COMPANION]->(companion:Companion)
WHERE user.userId = userPreference.userId
AND location.value = userPreference.location
AND companion.value = userPreference.companion
WITH userPreference, user, collect(distinct event) as events
MERGE (userPreference)<-[:HAS_PREFERENCE]-(user)
WITH userPreference, events, size(events) as size
UNWIND events as event
MATCH (event)-[:EVENT_ITEM]->(item:Item)-[]-(feature:Feature)
WITH feature, userPreference, 1.0f*count(event)/(1.0f*size) as
     preferenceValue
MERGE (userPreference)-[:RELATED_TO {value: preferenceValue}]->(feature)
```

feature	occurrence
"Comedy"	12
"Adventure"	9
"Action"	9
"Sci-Fi"	6
"Crime"	6
"Drama"	6
"Jonah Hill"	5
"Thriller"	5
"Animation"	4
"Family"	4
"Fantasy"	4
"Steve Carell"	4
"Andrew Stanton"	4
"Judd Apatow"	4
"Romance"	4
"Michael Bay"	3
"Mitchell Tanen"	3
"Allison Janney"	3
"Ronnie Del Carmen"	3
"Pete Docter"	3
"Leonardo DiCaprio"	3
"Christopher Nolan"	3
"Quentin Tarantino"	3

Figure 7.11 Results of query 7.6.

It's worth noting that this query uses properties to represent the contextual information of the user preferences. This is a slight deviation from the model design shown in figure 7.12, but it's a valid option. In the following exercises, you're invited to create an equivalent query that matches the model perfectly.

Exercises

Using query 7.7 as the basis, create the following queries:

- The same query, but for different a context
- An equivalent query that uses relationships to contextual information instead of using properties for specifying the context
- A query only for actors

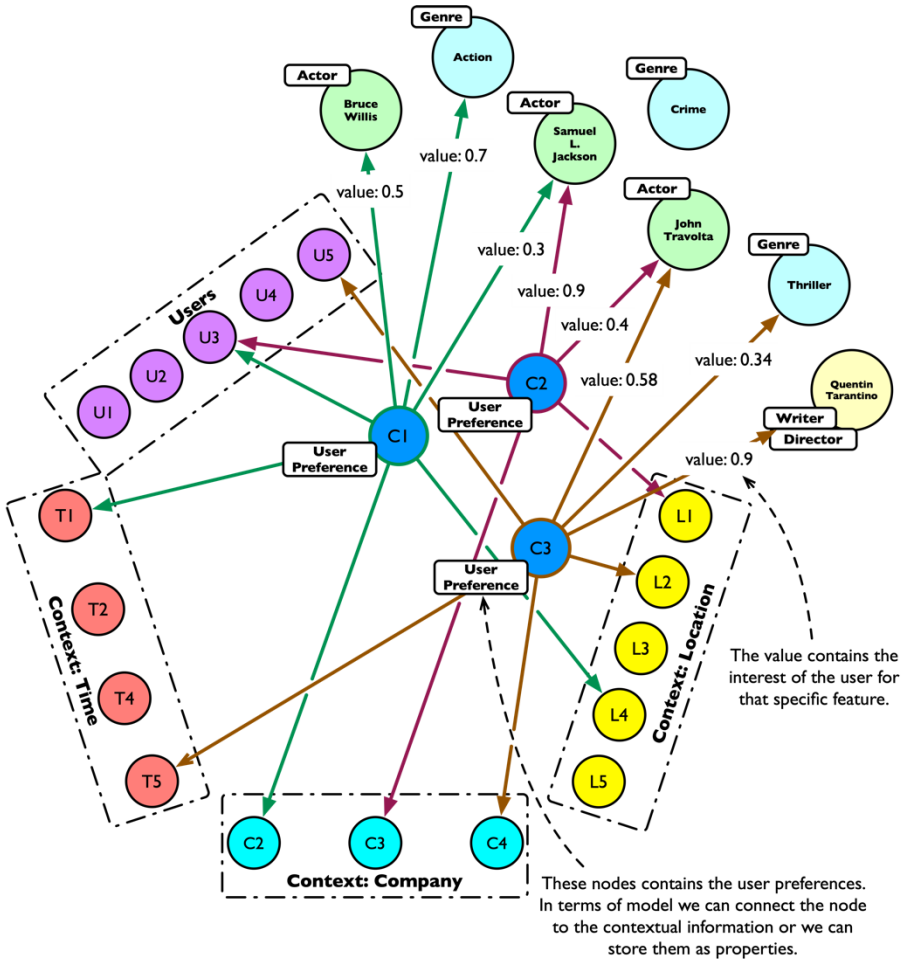


Figure 7.12 Graph model for the contextualized user preferences.

- A query only for directors
- A query only for writers
- A query only for genres

During the recommendation process, we can use this information about user preferences to determine how to adjust the results obtained using the first approach. The query to get this information is simple, as you can see here.

Query 7.8 Getting the boosting factor for the features

```
MATCH (user:User)-[:HAS_PREFERENCE]->(userPreference:UserPreference)-
     [r:RELATED_TO]->(feature:Feature)
WHERE user.userId = "1032"
AND userPreference.location = "Home"
AND userPreference.companion = "Alone"
```

```
RETURN CASE 'Genre' IN labels(feature)
    WHEN true THEN feature.genre
    ELSE feature.name END AS feature, r.value
```

This query returns values we can use as boosting factors after the first "generic" recommendation list has been obtained using one of the classic approaches.

The alternative to the heuristic approach to postfiltering is the model-based approach. Here, we build predictive models that calculate the probability of the user choosing a certain type of item in a given context (for example, the likelihood of choosing movies of a certain genre when alone and at home) and then use these probabilities to adjust the recommendations. The algorithms for computing probability are out of the scope of this chapter, but once computed they can be stored in the graph model exactly as shown in figure 7.12.

It's important to note that, as was the case with contextual prefiltering, the biggest advantage of the contextual postfiltering approach is that it allows the use of any traditional recommendation technique.

CONTEXTUAL MODELING

The third type of context-aware recommendation system is based on contextual modeling. This approach, as illustrated in figure 7.13, uses contextual information directly during the model creation, giving rise to truly multidimensional recommendation functions representing either predictive models (such as decision trees, regressions, and so on) or heuristic calculations that incorporate contextual information in addition to the user and item data.

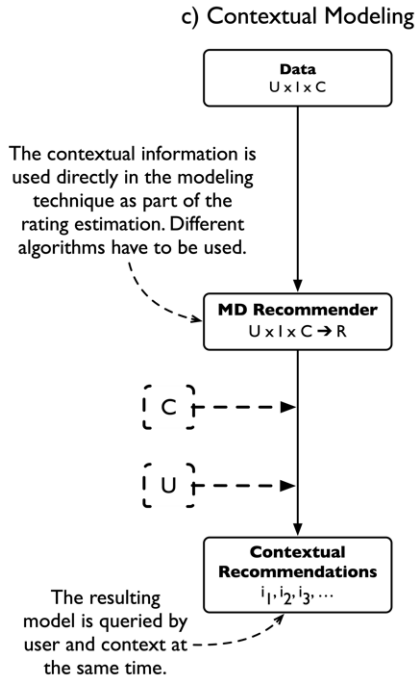


Figure 7.13 Contextual modeling.

In the last few years, a large number of recommendation algorithms based on a variety of heuristics as well as predictive modeling techniques have been developed. Several of these techniques can be considered an extension of the 2D to the multidimensional recommendation settings. Frolov and Oseledets [2016] show how to represent the User x Item x Contexts dataset as a multidimensional matrix, or tensor.[15] Such a tensor can be represented as in figure 7.14, where each event represents an element and the contexts, users, and items represent the dimensions. In such a representation, certain operations on tensors, such as slicing, are easy to do with simple queries.

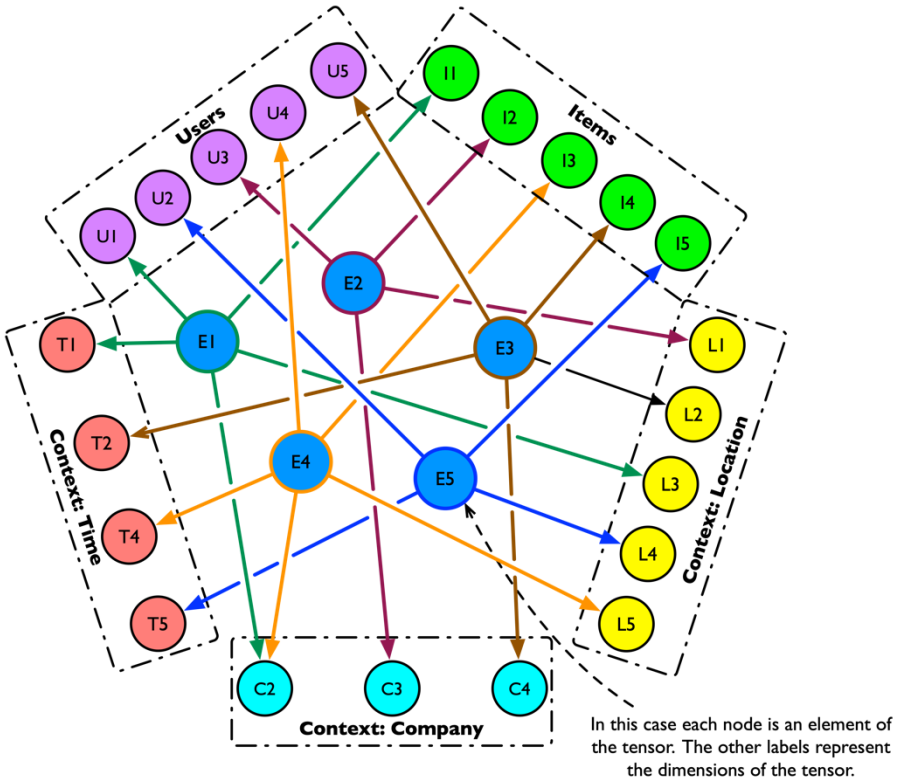


Figure 7.14 **Tensor representation in a graph model.**

Other researchers have addressed the task of contextual modeling with a pure graph-based approach, considering context-aware recommendation as a searching problem to find interesting items for a user given a so-called *context graph* [Wu et al., 2015]. The previously suggested method for creating the graph would not work in this case,

[15] A matrix is a two-dimensional grid of numbers. A tensor is a generalization that can have any number of dimensions: 0 (a single number), 1 (a vector), 2 (a traditional matrix), 3 (a cube of numbers), or more. These higher-dimensional structures are difficult to visualize. The dimension of a tensor is known as its *rank* (aka *order* or *degree*).

where the model design is completely different. Instead, the graph is created as follows. Given a context graph G = {V, E} the vertices and the edges are defined so that:

- The vertex set V is divided into several distinct sets, such as a set of users U, a set of items I, a set of attributes A, and a set of contexts C. C represents the combination of contextual information in a node—for instance, <Home, Alone, Weekday> is a node. While nodes A represent the static features or attributes of users or items—information that doesn't change for different ratings, unlike the contextual information.
- The edge set E consists of the existing connections of the Cartesian product: V x V. Edges with diverse types have distinct semantics. U x A connects users and their attributes (user interests), U x I connects users with the items they have interacted with (this is the old User x Item dataset), and U x C connects users and contexts. The submatrix U x U, which stores social network information, may exist or not.

The context graph G can be represented as an adjacent matrix where submatrices are all configured as symmetric (for example, UI^T is the transport matrix of UI), as shown in table 7.1.

Table 7.1 An Adjacent Matrix Representation of Contextual User–Item Interaction

	Users	Items	Contexts	Attributes
Users	UU	UI	UC	UA
Items	UI^T	0	IC	IA
Contexts	UC^T	IC^T	0	0
Attributes	UA^T	IA^T	0	0

The resulting graph is shown in figure 7.15.

Avoiding too many details, a random walk approach (more specifically, the Personalized PageRank or "PageRank with restart" algorithm) is then used to compute the relevance of the nodes in the graph. The recommendation process uses these relevance scores to estimate the likelihood of an unseen item *i* being accessed by a user *u*. For a detailed description, see Wu et al. [2015].

Pros and cons

Each of the three techniques discussed for context-based recommendation has advantages and disadvantages. These include:

- *Prefiltering*:
  - *Pros:* This method not only is easy to implement but allows you to use any of the traditional recommendation techniques. It can deliver quite accurate results if relevant data for the user's current context is available.

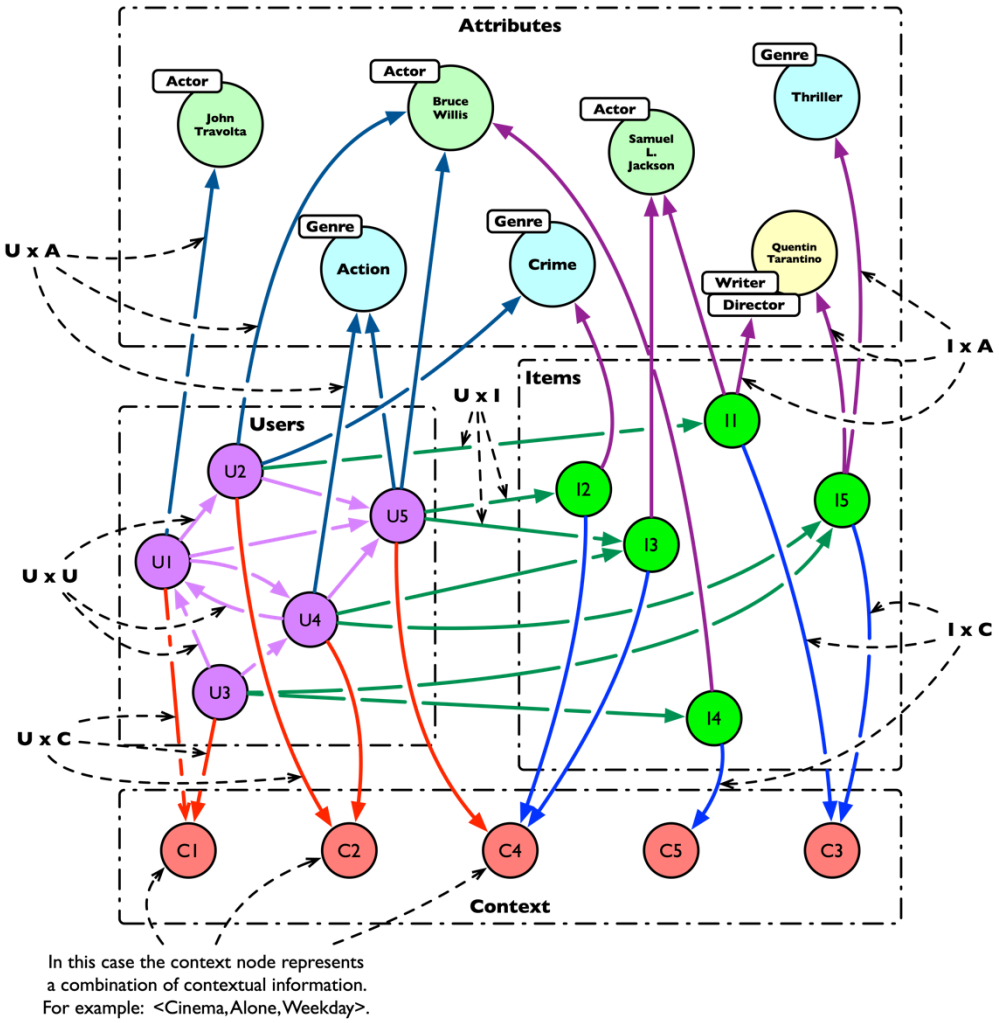


Figure 7.15 Representation of a context graph.

  - *Cons:* The data sparsity problem is common here, because it's highly probable that for some contexts there won't be enough data available for accurate recommendations. Moreover, to be performant this approach requires you to prebuild a high number of models and keep them all updated.
- *Postfiltering:*
  - *Pros:* This method is even easier to implement. You use a traditional technique (such as collaborative filtering) to generate the recommendations, then apply the filter to the result.
  - *Cons:* The postfiltering filters out or reduces the ratings for elements not relevant for the user's current context. The prediction accuracy is almost indepen-

dent from the context, and it's mostly aligned with the traditional methods. Data sparsity is a problem here too, as it is for the traditional methods—it may be that all the resulting elements are irrelevant for the current context.

- *Contextual modeling:*
  - *Pros:* The methods belonging to this category are the most recent and tend to be the most accurate. The main disadvantage of the previous approach is that context isn't integrated tightly into the recommendation algorithm, preventing you from taking full advantage of the relationships between the various user–item combinations and contextual values. Contextual modeling takes context into account from the beginning, enabling the creation of precise models that can be queried using user, item, and contextual information.
  - *Cons:* Most of the methods available for contextual modeling are complex to implement, and much computational power is required to create and update the model.

Which technique to choose actually depends on weighing these pros and cons. More specifically, it depends on the type and quantity of data available, the frequency of new data, and how closely the model should be aligned with the current data.

7.1.3 Advantages of the graph approach

In this section we've discussed the different approaches available for creating a context-aware recommendation engine: prefiltering, postfiltering, and contextual modeling. The different methods and algorithms presented can all use the graph representation of the User x Item x Contexts dataset, which simplifies accessing and navigating this complex data.

Specifically, the main aspects and advantages of the graph-based approach to context-aware recommendation systems are:

- The User x Item x Contexts multidimensional matrix, which represents the input of any such system, can be represented by a graph materializing the "interaction event." This data model speeds up the filtering phase and avoids the data sparsity problem, which can be problematic in this scenario.
- A proper graph model can store the multimodel results of contextual prefiltering. Specifically, in the case of the nearest neighbor approach to prefiltering, which results in different sets of similarities among the items or users, graphs can store the results of multiple models by materializing the similarity nodes.
- During the recommendation phase, graph access patterns simplify the selection of relevant data based on the current user and the current context.
- In the contextual modeling approach, graphs provide a suitable method to store tensors, simplifying certain operations. Additionally, specific approaches not only leverage a graph representation of the data (the context graph described earlier), but also use graph algorithms such as random walk and specifically PageRank for building models and then providing recommendations.

7.2 Hybrid recommendation engines

The recommendation approaches discussed in this book exploit different sources of information and follow different paradigms to make recommendations. Although they produce results that are considered to be personalized based on the assumed interests of their recipients, they perform with varying degrees of success in different application domains. Collaborative filtering exploits a specific type of information (item ratings) from a user model to derive recommendations, whereas content-based approaches rely on product features and textual descriptions as well as on the user profile. Session-based approaches use the clickstreams of "anonymous" users, whereas context-aware methods leverage contextual information together with item ratings to fine-tune the recommendations according to the current needs of the user.

Each of these approaches has its pros and cons (they were highlighted in detail in this and previous chapters)—for instance, the ability to handle the data sparsity and cold start problems, or the amount of effort required for content or context acquisition and processing.

Figure 7.16 sketches a recommendation system as a black box that transforms input data into a ranked list of items as output. Potential inputs, based on the approaches discussed here, include user models and contextual information as well as session data and item data; other inputs, required by other recommendation models, could be included as well. However, none of the basic approaches is able to fully exploit all of these. Consequently, building *hybrid* systems that combine the strengths of different algorithms and models to overcome some of the aforementioned shortcomings and problems has become the target of recent research.

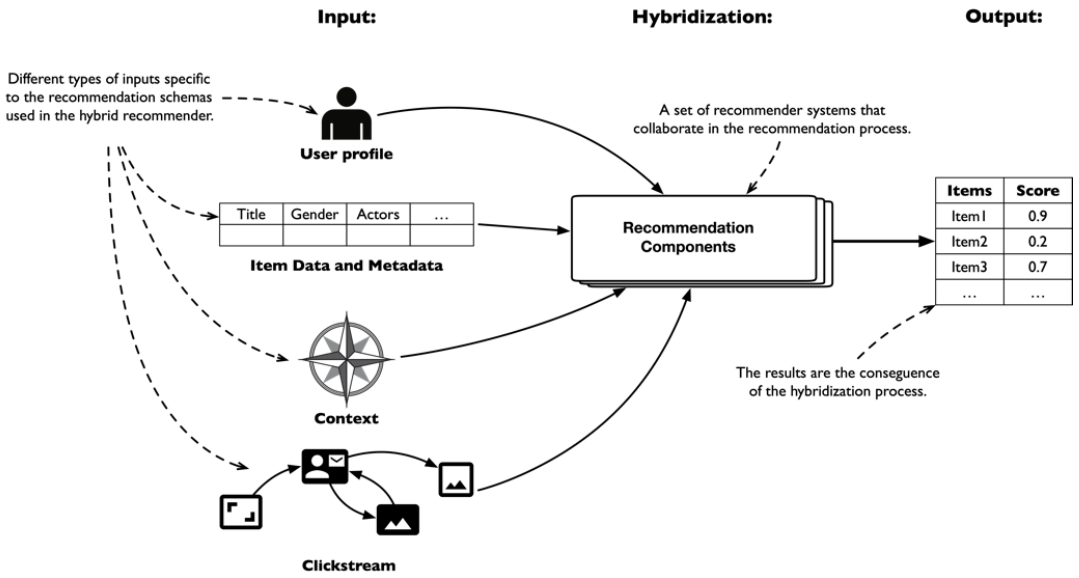


Figure 7.16 Hybrid recommendation system as a black box.

Hybrid recommender systems are technical implementations that combine multiple algorithms or recommendation components. Burke's [2002] taxonomy distinguishes among seven different hybridization strategies. From a more general perspective, however, the seven variants can be abstracted into three base designs:

- *Monolithic:* This hybridization design incorporates aspects of several recommendation strategies in one algorithm implementation. Several recommenders contribute virtually because the hybrid uses additional input data that's specific to another recommendation algorithm, or the input data is augmented by one technique and factually exploited by another. *Feature combination* and *feature augmentation* strategies can be assigned to this category. Feature combination uses a diverse range of input data. For instance, it can combine collaborative features, such as a user's likes and dislikes, with content features of catalog items. Feature augmentation applies complex transformation steps: the output of a contributing recommender system augments the feature space of the actual recommender by preprocessing its knowledge sources. See figure 7.17a.
- *Parallelized:* This approach requires at least two separate recommender implementations, which are subsequently combined (see figure 7.17b). Parallelized hybrid recommender systems operate independently of one another and produce separate recommendation lists. In a subsequent hybridization step, their output is combined into a final set of recommendations. Following Burke's taxonomy, the *weighted*, *mixed*, and *switching* strategies require recommendation components to work in parallel.
- *Pipelined:* In this case, several recommender systems are joined together in a pipeline architecture (see figure 7.17c). The output of one recommender becomes part of the input of the subsequent one. Optionally, the subsequent recommender components may use parts of the original input data, too. The *cascade* and *meta-level* hybrids, as defined by Burke, are examples of such architectures.

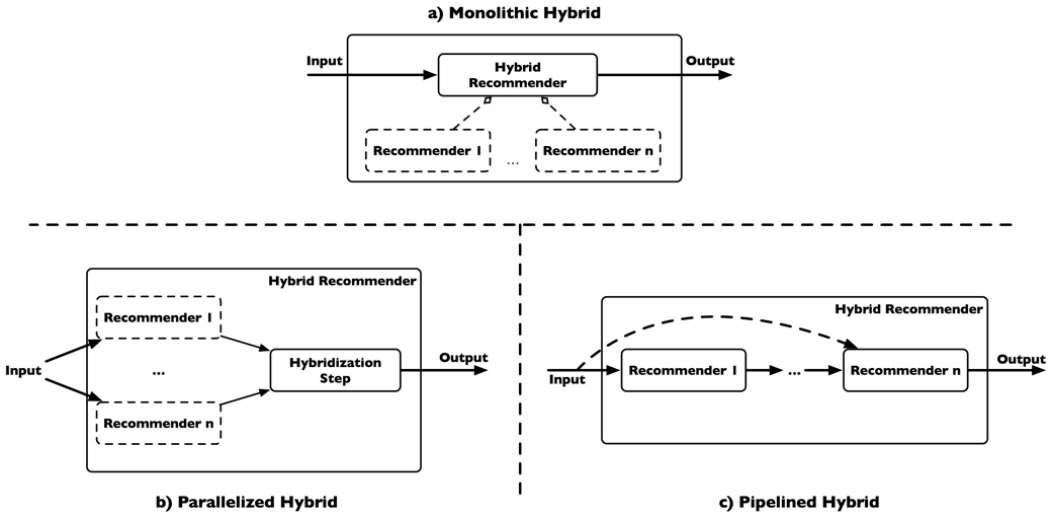


Figure 7.17 Hybridization design techniques.

For the purposes of this chapter we'll focus on the parallelized hybridization technique, which allows multiple recommender systems to operate in parallel, each using their own input and producing their own output model. The resulting models have to be stored somewhere so that they can be accessed and "mixed" or "merged" easily during the recommendation phase. In this context, graphs provide:

- A suitable representation for storing in a single, homogeneous, and connected data source all the different sets of information required by each recommender
- A model for storing the results of the training process so that they can be queried easily in parallel and then merged according to the hybridization strategy

7.2.1 Multiple models, a single graph

Let's take a closer look at the parallelized hybrid approach (figure 7.18). Suppose you have two types of recommender systems to be hybridized: one content-based, such as the ones described in chapter 4, and one collaborative, such as those described in chapter 5. This is quite a common scenario: it's often useful to merge these types of recommender systems because each can solve the issues of the other. The content-based approach mitigates the cold start problem that occurs when data is missing, such as in the case of a new user, a new item, or a totally new platform, whereas the collaborative filtering approach not only provides more accurate results but also works without information or metadata about users and items.

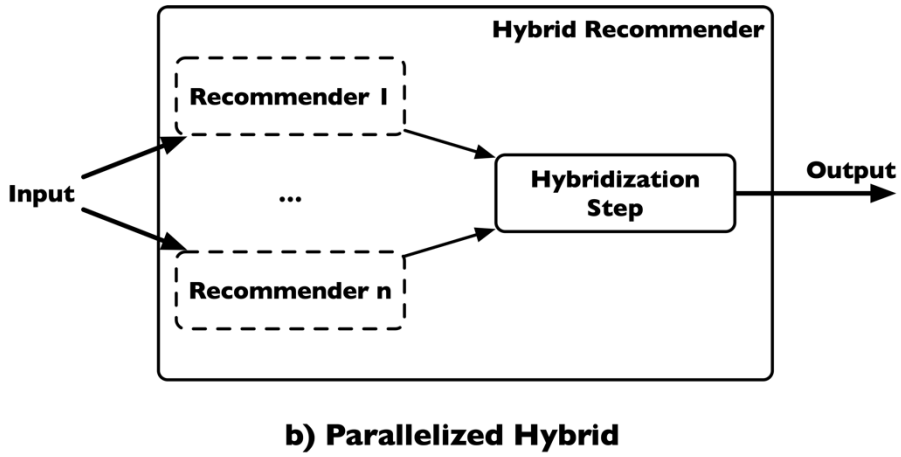


Figure 7.18 Parallelized approach.

The graph used as input for the parallelized hybrid recommendation system using these two types of recommenders as input will look like figure 7.19.

Specifically, in this case it's important to note how the "rated" connection is used by both recommender systems, in different ways.

Once the models are computed they can be stored back in the graph, as depicted in figure 7.20.

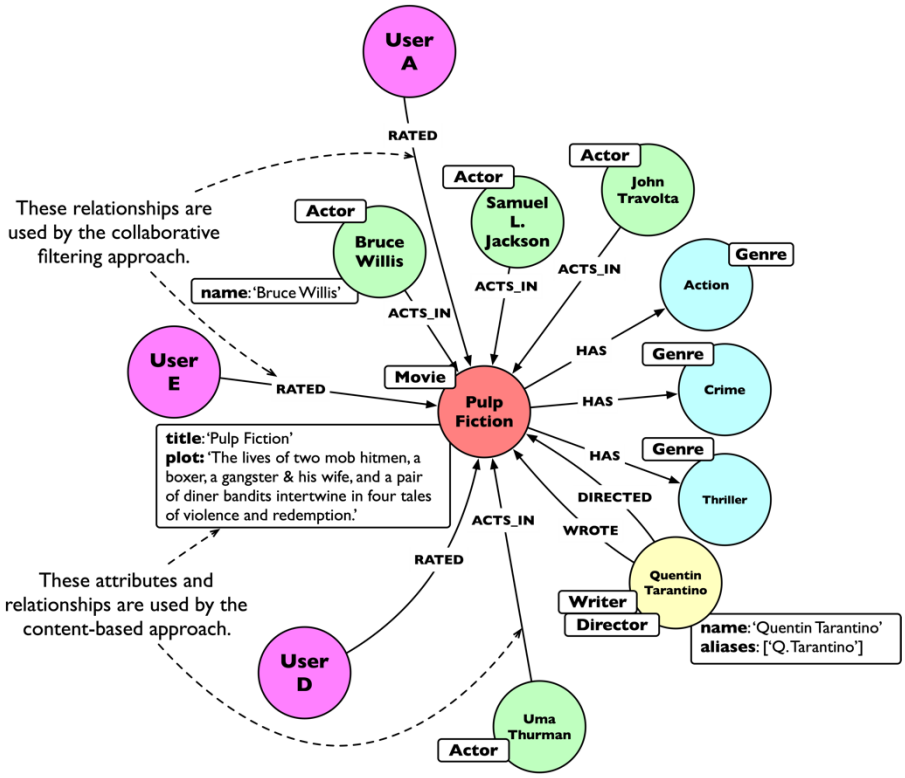


Figure 7.19 Example of graph model that combines collaborative filtering and content-based approaches.

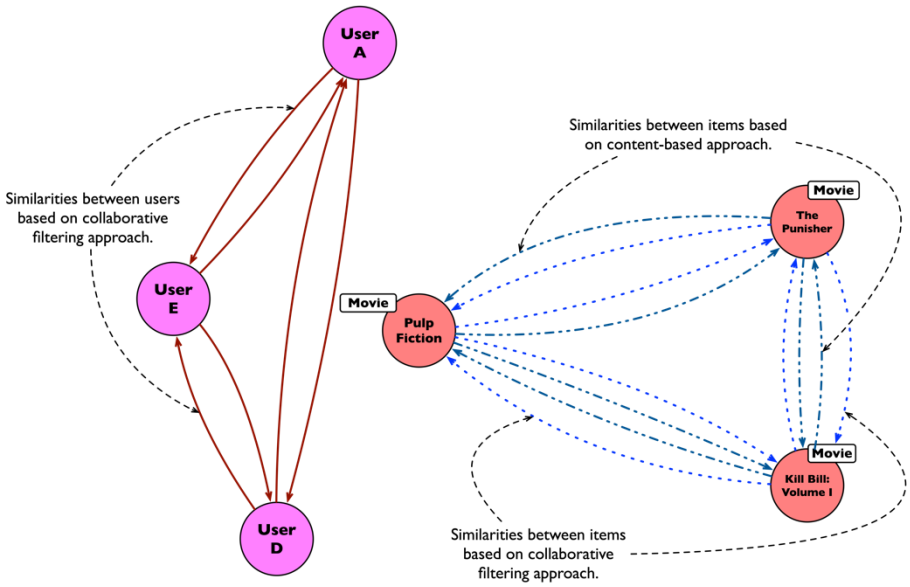


Figure 7.20 Mixing multiple recommendation models in the same graph.

7.2.2 Providing recommendations

Now that we've built the models and stored them in the graph, We can combine their outputs to obtain a unique list (or sometimes multiple lists) of items to recommend to the user. As described earlier, parallelized hybridization designs employ several recommenders side by side and use a specific hybridization mechanism to aggregate their outputs. The hybridization mechanism defines the strategy to provide recommendations to the user. According to Burke's [2002] classification, three main strategies can be applied: mixed, weighted, and switching. However, additional combination strategies for multiple recommendation lists, such as majority voting schemes, may also be applicable.

MIXED

The *mixed* hybridization strategy combines the results of different recommender systems at the level of the user interface. Results from different techniques are presented together; therefore, the recommendation result for user u is a set of lists of items.

The top-scoring items for each recommender are then displayed to the user next to each other, generally specifying to the user the "criteria" behind each of them. Sometimes in the mixed approach, a type of conflict resolution is necessary to avoid too many overlaps in the multiple lists.

WEIGHTED

A *weighted* hybridization strategy combines the recommendations of two or more recommender systems by computing weighted sums of their scores. Figure 7.21 is a graphical model showing how it works.

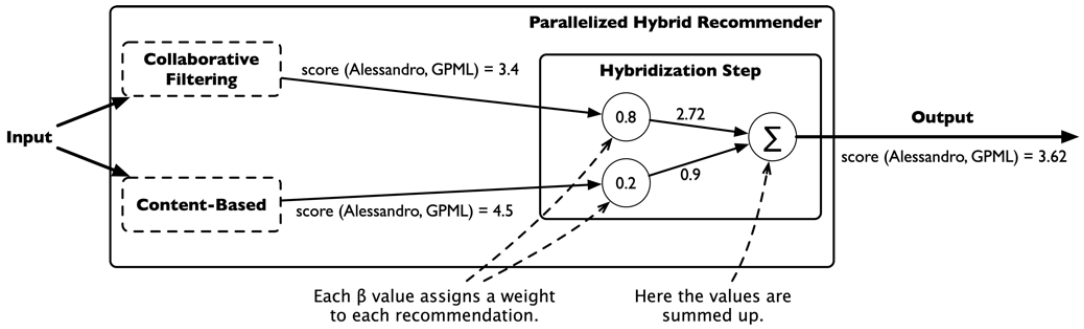


Figure 7.21 Weighted method explained with graphical model.

Thus, given n different recommendation functions $score_k(u, i)$ with associated relative weights β_k, the final score will be computed according to the following formula:

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

where n is the number of recommenders whose outputs have to be mixed. Furthermore, the item scores need to be restricted to the same range for all recommenders, and the sum of all β_k must be 1. This technique is quite straightforward and is thus a popular strategy for combining the predictive power of different recommendation techniques in a weighted manner.

It's worth noting here that the value of β_k can be dynamic. It can change over the life of the recommendation system, privileging, for instance, the content-based approach over collaborative filtering in the early stages when not enough information is available for the latter to be effective and then gradually giving it more weight once more data has been gathered. Moreover, the values can be dynamic per user assigning a higher value of β_k to the content-based recommendations until the system has acquired enough data for the collaborative filtering approach to be effective. Different techniques can be applied for evaluating how to set and then evolve the values of the weights.

SWITCHING

Switching hybrids require an "oracle" that decides which recommender should be used in a specific situation, depending on the user profile and/or the quality of recommendation results.

Figure 7.22 is a graphical model describing how it works.

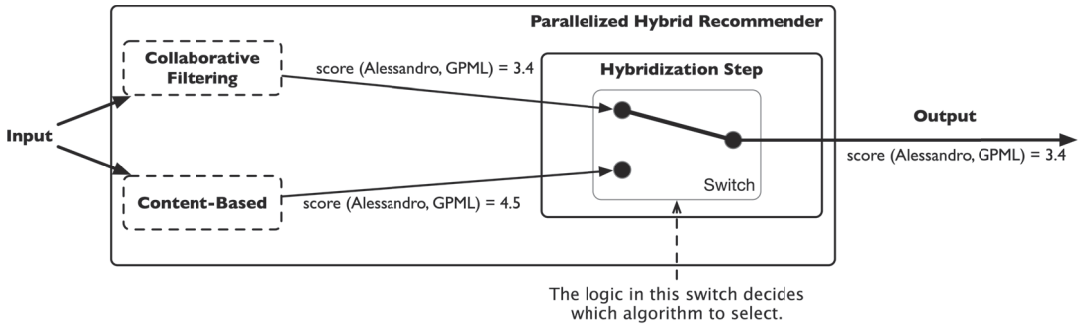


Figure 7.22 Switching method explained using a graphical model.

Such an evaluation could be carried out as follows:)

$$score_{switching}(\mathrm{u},\mathrm{i}) = score_k(u,i)$$

where k is determined by the switching condition. For instance, to overcome the cold start problem, a content-based and collaborative switching hybrid could initially make content-based recommendations until enough rating data is available. When the collaborative filtering component can deliver recommendations with sufficient confidence, the recommendation strategy could be switched. In the extreme case, dynamic weight adjustment could be implemented as a switching hybrid. There, the weights of

all but one dynamically selected recommender are set to 0, and the output of the single remaining recommender is assigned a weight of 1.

7.2.3 Advantages of the graph approach

In this section we've discussed how to create a hybrid recommendation engine, focusing on the different parallelized hybridization approaches that are available: mixed, weighted, and switching. The methods presented here can all take advantage of a graph representation of the data, both for training and in the resulting models .

Specifically, the main aspects and advantages of the graph-based approach to hybrid methods are:

- Various sets of information can coexist in the same data structure, making it easier to meet the data management needs of a hybrid recommender.
- The several independent models resulting from each recommender can be stored together and then accessed easily during the recommendation phase.

Summary

This chapter presented the latest advanced techniques to implement recommendation engines using contextual information and showed how to combine different approaches for greater effect. The various data models illustrate the usefulness and flexibility of graphs for satisfying different requirements in terms of training data and model storage.

In this chapter, you learned:

- How to improve the quality of the recommendations by embedding contextual information in the model and the related graph model
- How to leverage the graph for feeding the different context-aware design approaches: pre/postfiltering and contextual modeling
- How to combine multiple algorithms in a single recommendation engine and how to mix different training datasets and models in a single big graph

References

[Suchman, 1987] Suchman, Lucy. *Plans and Situated Actions*. Cambridge, UK: Cambridge University Press, 1987.

[Bazire and Brézillon, 2005] Bazire, Mary, and Patrick Brézillon. "Understanding Context Before Using It." *Proceedings of the 5th International and Interdisciplinary Conference on Modeling and Using Context* (2005): 29–40.

[Doerfel et al., 2016] Doerfel, Stephan, Robert Jäschke, and Gerd Stumme. "The Role of Cores in Recommender Benchmarking for Social Bookmarking Systems." *ACM Transactions on Intelligent Systems and Technology* 7:3 (2016): Article 40.

[Palmisano et al., 2008] Palmisano, Cosimo, Alexander Tuzhilin, and Michele Gorgoglione. "Using Context to Improve Predictive Modeling of Customers in Personalization Applications." *IEEE Transactions on Knowledge and Data Engineering* 20:11 (2008): 1535–1549.

[Zheng et al., 2015] Zheng, Yong, Bamshad Mobasher, and Robin Burke, "CARSKit: A Java-Based Context-Aware Recommendation Engine." *Proceedings of the 15th IEEE International Conference on Data Mining (ICDM) Workshops* (2015): 1668–1671.

[Ilarri et al., 2018] Ilarri, Sergio, Raquel Trillo-Lado, and Ramon Hermoso. "Datasets for Context-Aware Recommender Systems: Current Context and Possible Directions." *Proceedings of the IEEE 34th International Conference on Data Engineering Workshops* (2018).

[Adomavicius and Tuzhilin, 2005] Adomavicius, Gediminus, and Alexander Tuzhilin. "Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions." IEEE Transac- tions on Knowledge and Data Engineering 17(6): 734–749.

[Panniello et al., 2009] Panniello, Umberto, Alexander Tuzhilin, Michele Gorgoglione, Cosimo Palmisano, and Anto Pedone. "Experimental Comparison of Pre- vs. Post-Filtering Approaches in Context-Aware Recommender Systems." *Proceedings of the 3rd ACM Conference on Recommender Systems* (2009): 265–268.

[Frolov and Oseledets, 2016] Frolov, Evgeny, and Ivan Oseledets. "Tensor Methods and Recommender Systems." arXiv preprint arXiv:1603.06038 (2016).

[Wu et al., 2015] Wu, Hao, Kun Yue, Xiaoxin Liu, Yijian Pei, and Bo Li. "Context-Aware Recommendation via Graph-Based Contextual Modeling and Postfiltering." *International Journal of Distributed Sensor Networks – Special Issue on Big Data and Knowledge Extraction for Cyber-Physical Systems* (2015): Article 16.

[Burke, 2002] Burke, Robin. "Hybrid Recommender Systems: Survey and Experiments." *User Modeling and User-Adapted Interaction* 12:4 (2002): 331–370.

conclusion

Thank you for reading this excerpt from Graph-Powered Machine Learning. I hope you've learned a lot, but this isn't the end; it's just the beginning of a new journey.

Graph databases can be your best friend, not only in solving traditional data problems in a completely new way, but also for overcoming new data challenges required by machine learning.

Having a graph database is key for storing and operationalizing the connections in your data, but I'd also like to highlight the new Neo4j Graph Data Science Library. I've used the GDS library in my own machine learning projects. I believe this new set of data science tools addresses better than ever the core issues and challenges of the machine learning field.

—Graphs are an ML-empowering technology whether you're using the Neo4j graph database, the Graph Data Science Library, or both. So, if you're embarking on a new machine learning project, I encourage you to consider using graph technology to make it a success.

Thanks again for reading,
—Alessandro Negro

index

A

B

C

D

E

F

G

H

I

K

L

M

N

P

R

S

T

U

V

W