4

Improving Embeddings with Biased Random Walks in Node2Vec

Node2Vec is an architecture largely based on DeepWalk. In the previous chapter, we saw the two main components of this architecture: random walks and Word2Vec. How can we improve the quality of our embeddings? Interestingly enough, not with more machine learning. Instead, Node2Vec brings critical modifications to the way random walks themselves are generated.

In this chapter, we will talk about these modifications and how to find the best parameters for a given graph. We will implement the Node2Vec architecture and compare it to using DeepWalk on Zachary's Karate Club. This will give you a good understanding of the differences between the two architectures. Finally, we will use this technology to build a real application: a movie **recommender system** (**RecSys**) powered by Node2Vec.

By the end of this chapter, you will know how to implement Node2Vec on any graph dataset and how to select good parameters. You will understand why this architecture works better than DeepWalk in general, and how to apply it to build creative applications.

In this chapter, we'll cover the following topics:

- Introducing Node2Vec
- Implementing Node2Vec
- Building a movie RecSys

Technical requirements

All the code examples from this chapter can be found on GitHub at **https://github.com/PacktPublishing/Hands-On-Graph-Neural-Networks-Using-Python/tree/main/Chapter04**.

Installation steps required to run the code on your local machine can be found in the *Preface* of this book.

Introducing Node2Vec

Node2Vec was introduced in 2016 by Grover and Leskovec from Stanford University [1]. It keeps the same two main components from DeepWalk: random walks and Word2Vec. The difference is that instead of obtaining sequences of nodes with a uniform distribution, the random walks are carefully biased in Node2Vec. We will see why these **biased random walks** perform better and how to implement them in the two following sections:

- Defining a **neighborhood**
- Introducing biases in random walks

Let's start by questioning our intuitive concept of neighborhoods.

Defining a neighborhood

How do you define the neighborhood of a node? The key concept introduced in Node2Vec is the flexible notion of a neighborhood. Intuitively, we think of it as something close to the initial node, but what does "close" mean in the context of a graph? Let's take the following graph as an example:

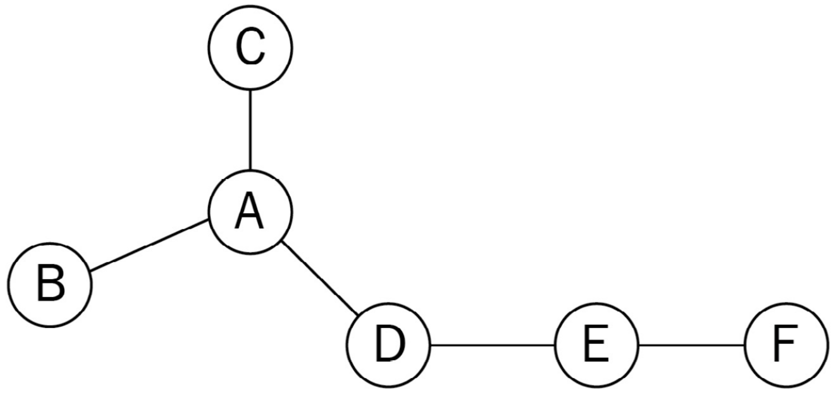


Figure 4.1 – Example of a random graph

We want to explore three nodes in the neighborhood of node **A**. This exploration process is also called a **sampling strategy**:

- A possible solution would be to consider the three closest nodes in terms of connections. In this case, the neighborhood of A, noted $N(A)$, would be $N(A) = \{B, C, D\}$:
- Another possible sampling strategy consists of selecting nodes that are not adjacent to previous nodes first. In our example, the neighborhood of A would be $N(A) = \{D, E, F\}$:

In other words, we want to implement a **Breadth-First Search** (**BFS**) in the first case and a **Depth-First Search** (**DFS**) in the second one. You can find more information about these algorithms and implementations in *Chapter 2, Graph Theory for Graph Neural Networks.*

What is important to notice here is that these sampling strategies have opposite behaviors: BFS focuses on the local network around a node while DFS establishes a more macro view of the graph. Considering our intuitive definition of a neighborhood, it is tempting to simply discard DFS. However, Node2Vec's authors argue that this would be a mistake: each approach captures a different but valuable representation of the network.

They make a connection between these algorithms and two network properties:

- **Structural equivalence**, which means that nodes are structurally equivalent if they share many of the same neighbors. So, if they share many neighbors, their structural equivalence is higher.
- **Homophily**, as seen previously, states that similar nodes are more likely to be connected.

They argue that BFS is ideal to emphasize structural equivalence since this strategy only looks at neighboring nodes. In these random walks, nodes are often repeated and stay close to each other. DFS, on the other hand, emphasizes the opposite of homophily by creating sequences of distant nodes. These random walks can sample nodes that are far from the source and thus become less representative. This is why we're looking for a trade-off between these two properties: homophily may be more helpful for understanding certain graphs and vice versa.

If you're confused about this connection, you're not alone: several papers and blogs wrongly assume that BFS emphasizes homophily and DFS is connected to structural equivalence. In any case, we consider graphs that combine homophily and structural equivalence to be the desired solution. This is why, regardless of these connections, we want to use both sampling strategies to create our dataset.

Let's see how we can implement them to generate random walks.

Introducing biases in random walks

As a reminder, random walks are sequences of nodes that are randomly selected in a graph. They have a starting point, which can also be random, and a predefined length. Nodes that often appear together in these walks are like words that appear together in sentences: under the homophily hypothesis, they share a similar meaning, hence a similar representation.

In Node2Vec, our goal is to bias the randomness of these walks to either one of the following:

- Promoting nodes that are not connected to the previous one (similar to DFS)
- Promoting nodes that are close to the previous one (similar to BFS)

Let's take *Figure 4.2* as an example. The current node is called j, the previous node is i, and the future node is k. We note π_{jk}, the unnormalized transition probability from node j to node k. This probability can be decomposed as $\pi_{jk} = \alpha(i,k) \cdot \omega_{jk}$, where $\alpha(i,k)$ is the **search bias** between nodes i and k and ω_{jk} is the weight of the edge from j to k.

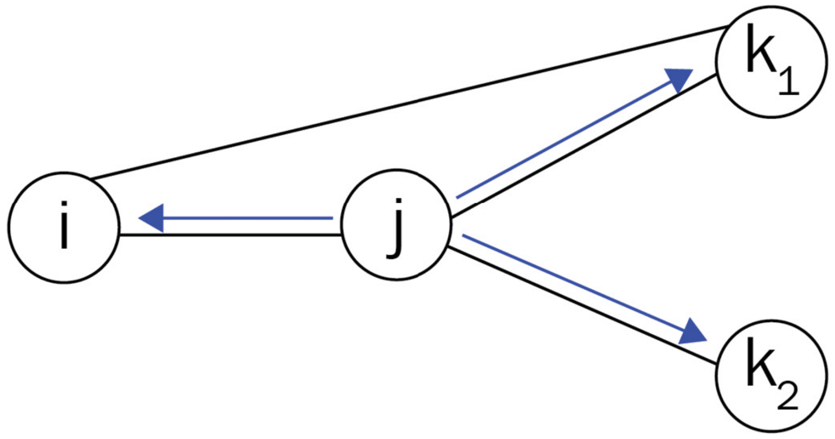


Figure 4.2 – Example of a random graph

In DeepWalk, we have $\alpha(a,b) = 1$ for any pair of nodes a and b. In Node2Vec, the value of $\alpha(a,b)$ is defined based on the distance between the nodes and two additional parameters: p, the return parameter, and q, the in-out parameter. Their role is to approximate DFS and BFS, respectively.

Here is how the value of $\alpha(a, b)$ is defined:

$$\alpha(a, b) = \begin{cases} \frac{1}{p} \ if \ d_{ab} = 0 \\ 1 \ if \ d_{ab} = 1 \\ \frac{1}{q} \ if \ d_{ab} = 2 \end{cases}$$

Here, d_{ab} is the shortest path distance between nodes a and b. We can update the unnormalized transition probability from the previous graph as follows:

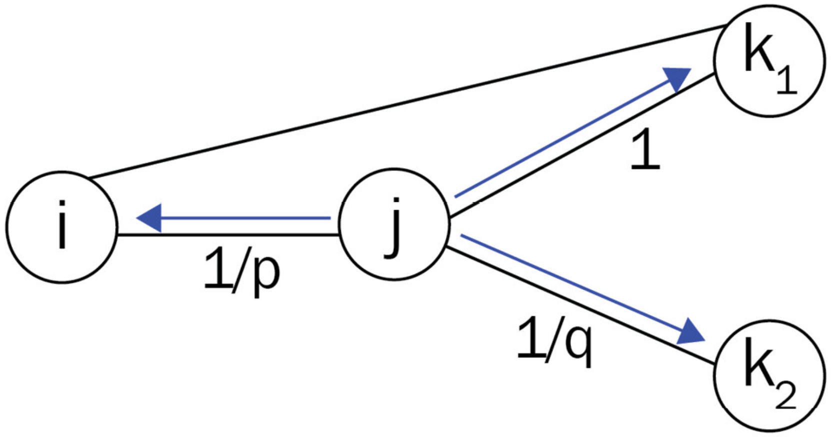


Figure 4.3 – Graph with transition probabilities

Let's decrypt these probabilities:

- The walk starts from node i and now arrives at node j. The probability of going back to the previous node i is controlled by the parameter p. The higher it is, the more the random walk will explore new nodes instead of repeating the same ones and looking like DFS.
- The unnormalized probability of going to k_1 is 1 because this node is in the immediate neighborhood of our previous node, i.
- Finally, the probability of going to node k_2 is controlled by the parameter q. The higher it is, the more the random walk will focus on nodes that are close to the previous one and look like BFS.

The best way to understand this is to actually implement this architecture and play with the parameters. Let's do it step by step on Zachary's Karate Club (a graph from the previous chapter), as shown in *Figure 4.4*:

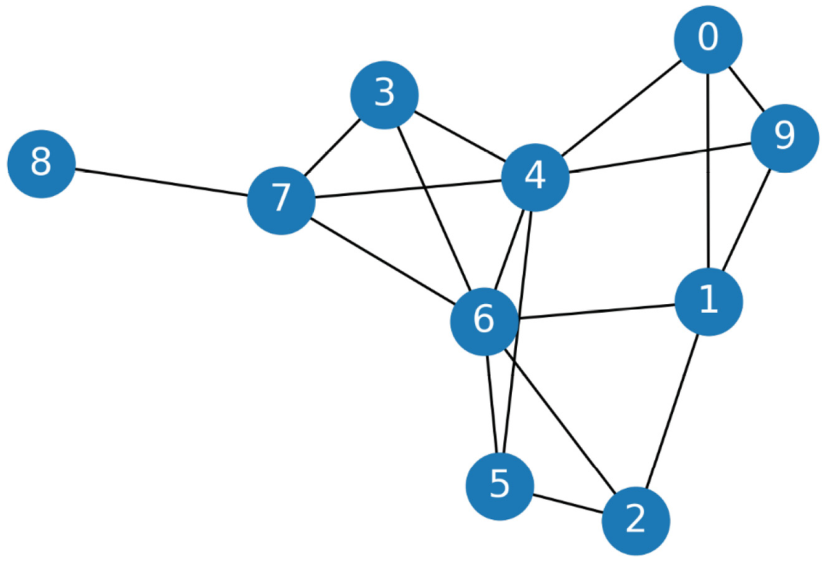


Figure 4.4 – Zachary's Karate Club

Note that it is an unweighted network, which is why the transition probability is only determined by the search bias.

First, we want to create a function that will randomly select the next node in a graph based on the previous node, the current node, and the two parameters p and q.

1. We start by importing the required libraries: **networkx**, **random**, and **numpy**:

   ```
   import networkx as nx
   import random
   random.seed(0)
   import numpy as np
   np.random.seed(0)
   G = nx.erdos_renyi_graph(10, 0.3, seed=1, directed=False)
   ```

2. We defined the **next_node** function with the list of our parameters:

   ```
   def next_node(previous, current, p, q):
   ```

3. We retrieve the list of neighboring nodes from the current node and initialize a list of alpha values:

   ```
           neighbors = list(G.neighbors(current))
           alphas = []
   ```

4. For each neighbor, we want to calculate the appropriate alpha value: $1/p$ if this neighbor is the previous node, 1 if this neighbor is connected to the previous node, and $1/q$ otherwise:

   ```
       for neighbor in neighbors:
           if neighbor == previous:
               alpha = 1/p
           elif G.has_edge(neighbor, previous):
               alpha = 1
           else:
               alpha = 1/q
           alphas.append(alpha)
   ```

5. We normalize these values to create probabilities:

   ```
   probs = [alpha / sum(alphas) for alpha in alphas]
   ```

6. We randomly select the next node based on the transition probabilities calculated in the previous step using **`np.random.choice()`** and return it:

```
        next = np.random.choice(neighbors, size=1, p=probs)[0]
        return next
```

Before this function can be tested, we need the code to generate the entire random walk.

The way we generate these random walks is similar to what we saw in the previous chapter. The difference is that the next node is chosen by the **`next_node()`** function, which requires additional parameters: p and q, but also the previous and current nodes. These nodes can easily be obtained by looking at the two last elements added to the **`walk`** variable. We also return strings instead of integers for compatibility reasons.

Here is the new version of the **`random_walk()`** function:

```
def random_walk(start, length, p, q):
    walk = [start]
    for i in range(length):
        current = walk[-1]
        previous = walk[-2] if len(walk) > 1 else None
        next = next_node(previous, current, p, q)
        walk.append(next)
    return [str(x) for x in walk]
```

We now have every element to generate our random walks. Let's try one with a length of 5, $p = 1$, and $q = 1$:

```
random_walk(0, 8, p=1, q=1)
```

This function returns the following sequence:

```
[0, 4, 7, 6, 4, 5, 4, 5, 6]
```

This should be random since every neighboring node has the same transition probability. With these parameters, we reproduce the exact DeepWalk algorithm.

Now, let's bias them toward going back to the previous node with $q = 10$:

```
random_walk(0, 8, p=1, q=10)
```

This function returns the following sequence:

```
[0, 9, 1, 9, 1, 9, 1, 0, 1]
```

This time, the random walk explores more nodes in the graph. You can see that it never goes back to the previous node because the probability is low with $p = 10$:

```
random_walk(0, 8, p=10, q=1)
```

This function returns the following sequence:

```
[0, 1, 9, 4, 7, 8, 7, 4, 6]
```

Let's see how to use these properties in a real example and compare it to DeepWalk.

Implementing Node2Vec

Now that we have the functions to generate biased random walks, the implementation of Node2Vec is very similar to implementing DeepWalk. It is so similar that we can reuse the same code and create sequences with $p = 1$ and $q = 1$ to implement DeepWalk as a special case of Node2Vec. Let's reuse Zachary's Karate Club for this task:

As in the previous chapter, our goal is to correctly classify each member of the club as part of one of the two groups ("Mr. Hi" and "Officer"). We will use the node embeddings provided by Node2Vec as input to a machine learning classifier (Random Forest in this case).

Let's see how to implement it step by step:

1. First, we want to install the **gensim** library to use Word2Vec. This time, we will use version 3.8.0 for compatibility reasons:

    ```
    !pip install -qI gensim==3.8.0
    ```

2. We import the required libraries:

    ```
    from gensim.models.word2vec import Word2Vec
    from sklearn.ensemble import RandomForestClassifier
    from sklearn.metrics import accuracy_score
    ```

3. We load the dataset (Zachary's Karate Club):

    ```
    G = nx.karate_club_graph()
    ```

4. We transform the nodes' labels into numerical values (**0** and **1**):

    ```
    labels = []
    for node in G.nodes:
        label = G.nodes[node]['club']
        labels.append(1 if label == 'Officer' else 0)
    ```

5. We generate a list of random walks as seen previously using our **random_walk()** function 80 times for each node in the graph. The parameters p and q as specified here (2 and 1, respectively):

    ```
    walks = []
    for node in G.nodes:
        for _ in range(80):
            walks.append(random_walk(node, 10, 3, 2))
    ```

6. We create an instance of Word2Vec (a skip-gram model) with a hierarchical **softmax** function:

    ```
    node2vec = Word2Vec(walks,
                    hs=1,   # Hierarchical softmax
                    sg=1,   # Skip-gram
    ```

```
                vector_size=100,
                window=10,
                workers=2,
                min_count=1,
                seed=0)
```

7. The skip-gram model is trained on the sequences we generated for `30` epochs:

```
node2vec.train(walks, total_examples=node2vec.corpus_count, epochs=30, report_delay=1)
```

8. We create masks to train and test the classifier:

```
train_mask = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]
train_mask_str = [str(x) for x in train_mask]
test_mask = [0, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 26, 27, 28, 29, 30, 31, 32, 33]
test_mask_str = [str(x) for x in test_mask]
labels = np.array(labels)
```

9. The Random Forest classifier is trained on the training data:

```
clf = RandomForestClassifier(random_state=0)
clf.fit(node2vec.wv[train_mask_str], labels[train_mask])
```

10. We evaluate it in terms of accuracy for the test data:

```
y_pred = clf.predict(node2vec.wv[test_mask_str])
acc = accuracy_score(y_pred, labels[test_mask])
print(f'Node2Vec accuracy = {acc*100:.2f}%')
```

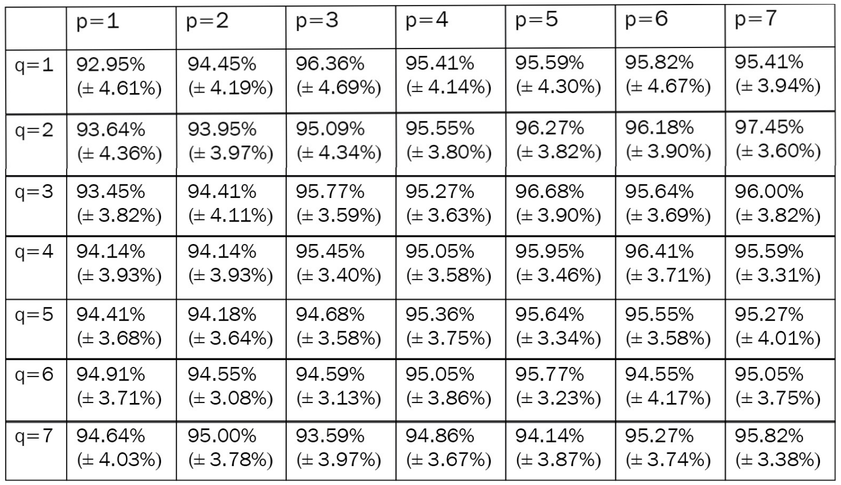
To implement DeepWalk, we can repeat the exact same process with $p = 1$ and $q = 1$. However, to make a fair comparison, we cannot use a single accuracy score. Indeed, there are a lot of stochastic processes involved – we could be unlucky and get a better result from the worst model.

To limit the randomness of our results, we can repeat this process 100 times and take the mean value. This result is a lot more stable and can even include the standard deviation (using `np.std()`) to measure the variability in the accuracy scores.

But just before we do that, let's play a game. In the previous chapter, we talked about Zachary's Karate Club as a homophilic network. This property is emphasized by DFS, which is encouraged by increasing the parameter p. If this statement and the connection between DFS and homophily are true, we should get better results with higher values of p.

I repeated the same experiment for values of p and q between 1 and 7. In a real machine learning project, we would use validation data to perform this parameter search. In this example, we use the test data because this study is already our final application.

The following table summarizes the results:



	p=1	p=2	p=3	p=4	p=5	p=6	p=7
q=1	92.95% (± 4.61%)	94.45% (± 4.19%)	96.36% (± 4.69%)	95.41% (± 4.14%)	95.59% (± 4.30%)	95.82% (± 4.67%)	95.41% (± 3.94%)
q=2	93.64% (± 4.36%)	93.95% (± 3.97%)	95.09% (± 4.34%)	95.55% (± 3.80%)	96.27% (± 3.82%)	96.18% (± 3.90%)	97.45% (± 3.60%)
q=3	93.45% (± 3.82%)	94.41% (± 4.11%)	95.77% (± 3.59%)	95.27% (± 3.63%)	96.68% (± 3.90%)	95.64% (± 3.69%)	96.00% (± 3.82%)
q=4	94.14% (± 3.93%)	94.14% (± 3.93%)	95.45% (± 3.40%)	95.05% (± 3.58%)	95.95% (± 3.46%)	96.41% (± 3.71%)	95.59% (± 3.31%)
q=5	94.41% (± 3.68%)	94.18% (± 3.64%)	94.68% (± 3.58%)	95.36% (± 3.75%)	95.64% (± 3.34%)	95.55% (± 3.58%)	95.27% (± 4.01%)
q=6	94.91% (± 3.71%)	94.55% (± 3.08%)	94.59% (± 3.13%)	95.05% (± 3.86%)	95.77% (± 3.23%)	94.55% (± 4.17%)	95.05% (± 3.75%)
q=7	94.64% (± 4.03%)	95.00% (± 3.78%)	93.59% (± 3.97%)	94.86% (± 3.67%)	94.14% (± 3.87%)	95.27% (± 3.74%)	95.82% (± 3.38%)

Figure 4.5 – Average accuracy score and standard deviation for different values of p and q

There are several noticeable results:

- DeepWalk ($p = 1$ and $q = 1$) performs worse than any other combination of p and q that is covered here. This is true for this dataset and shows how useful biased random walks can be.

However, it is not always the case: non-biased random walks can also perform better on other datasets.

- High values of p lead to better performance, which validates our hypothesis. Knowing that this is a social network strongly suggests that biasing our random walks toward homophily is a good strategy. This is something to keep in mind when dealing with this kind of graph.

Feel free to play with the parameters and try to find other interesting results. We could explore results with very high values of p (> 7) or, on the contrary, values of p and q between 0 and 1.

Zachary's Karate Club is a basic dataset, but we'll see in the next section how we can use this technology to build much more interesting applications.

Building a movie RecSys

One of the most popular applications of GNNs is RecSys. If you think about the foundation of Word2Vec (and, thus, DeepWalk and Node2Vec), the goal is to produce vectors with the ability to measure their similarity. Encode movies instead of words, and you can suddenly ask for movies that are the most similar to a given input title. It sounds a lot like a RecSys, right?

But how to encode movies? We want to create (biased) random walks of movies, but this requires a graph dataset where similar movies are connected to each other. This is not easy to find.

Another approach is to look at user ratings. There are different techniques to build a graph based on ratings: bipartite graphs, edges based on pointwise mutual information, and so on. In this section, we'll implement a simple and intuitive approach: movies that are liked by the same users are connected. We'll then use this graph to learn movie embeddings using Node2Vec:

1. First, let's download a dataset. **`MovieLens`** [2] is a popular choice, with a small version of the latest dataset (09/2018) comprising 100,836 ratings, 9,742 movies, and 610 users. We can download it with the following Python code:

```
from io import BytesIO
from urllib.request import urlopen
from zipfile import ZipFile
url = 'https://files.grouplens.org/datasets/movielens/ml-100k.zip'
with urlopen(url) as zurl:
    with ZipFile(BytesIO(zurl.read())) as zfile:
        zfile.extractall('.')
```

2. We are interested in two files: **`ratings.csv`** and **`movies.csv`**. The first one stores all the ratings made by users, and the second one allows us to translate movie identifiers into titles.
3. Let's see what they look like by importing them with **`pandas`** using **`pd.read_csv()`**:

```
import pandas as pd
ratings = pd.read_csv('ml-100k/u.data', sep='\t', names=['user_id', 'movie_id', 'rating', 'unix_t
ratings
```

4. This gives us the following output:

```
      user_id movie_id rating unix_timestamp
0       196      242      3      881250949
1       186      302      3      891717742
2        22      377      1      878887116
...     ...      ...     ...       ...
99998   13       225      2      882399156
99999   12       203      3      879959583
100000 rows × 4 columns
```

5. Let's import **`movies.csv`** now:

```
movies = pd.read_csv('ml-100k/u.item', sep='|', usecols=range(2), names=['movie_id', 'title'], en
```

6. This dataset gives us this output:

```
movies
     movie_id      title
0       1      Toy Story (1995)
1       2      GoldenEye (1995)
```

```
2      3      Four Rooms (1995)
...      ...      ...
1680      1681      You So Crazy (1994)
1681      1682      Scream of Stone (Schrei aus Stein) (1991)
1682 rows × 2 columns
```

7. Here, we want to see movies that have been liked by the same users. This means that ratings such as 1, 2, and 3 are not very relevant. We can discard those and only keep scores of 4 and 5:

```
ratings = ratings[ratings.rating >= 4]
ratings
```

8. This gives us the following output:

```
     user_id   movie_id    rating      unix_timestamp
5      298      474      4      884182806
7      253      465      5      891628467
11     286      1014     5      879781125
...      ...      ...      ...      ...
99991      676      538      4      892685437
99996      716      204      5      879795543
55375 rows × 4 columns
```

9. We now have 48,580 ratings made by 610 users. The next step is to count every time that two movies are liked by the same user. We will repeat this process for every user in the dataset.
10. To simplify things, we will use a **`defaultdict`** data structure, which automatically creates missing entries instead of raising an error. We'll use this structure to count movies that are liked together:

```
from collections import defaultdict
pairs = defaultdict(int)
```

11. We loop through the entire list of users in our dataset:

```
for group in ratings.groupby("userId"):
```

12. We retrieve the list of movies that have been liked by the current user:

```
user_movies = list(group[1]["movieId"])
```

13. We increment a counter specific to a pair of movies every time they are seen together in the same list:

```
for i in range(len(user_movies)):
            for j in range(i+1, len(user_movies)):
                pairs[(user_movies[i], user_movies[j])] += 1
```

14. The **pairs** object now stores the number of times two movies have been liked by the same user. We can use this information to build the edges of our graph as follows.
15. We create a graph using the **networkx** library:

```
G = nx.Graph()
```

16. For each pair of movies in our **pairs** structure, we unpack the two movies and their corresponding score:

```
for pair in pairs:
    movie1, movie2 = pair
    score = pairs[pair]
```

17. If this score is higher than 10, we add a weighted link to the graph to connect both movies based on this score. We don't consider scores lower than 10 because that would create a large graph in which connections were less meaningful:

```
if score >= 20:
    G.add_edge(movie1, movie2, weight=score)
```

18. The graph we created has 410 nodes (movies) and 14,936 edges. We can now train Node2Vec on it to learn the node embeddings!

We could reuse our implementation from the previous section, but there is actually an entire Python library dedicated to Node2Vec (also called **node2vec**). Let's try it in this example:

1. We install the **node2vec** library and import the **Node2Vec** class:

```
!pip install node2vec
from node2vec import Node2Vec
```

2. We create an instance of **Node2Vec** that will automatically generate biased random walks based on p and q parameters:

```
node2vec = Node2Vec(G, dimensions=64, walk_length=20, num_walks=200, p=2, q=1, workers=1)
```

3. We train a model on these biased random walks with a window of 10 (5 nodes before, 5 nodes after):

```
model = node2vec.fit(window=10, min_count=1,
batch_words=4)
```

The Node2Vec model is trained and we can now use it the same way we use the Word2Vec object from the **gensim** library. Let's create a function to recommend movies based on a given title:

4. We create the **recommend()** function, which takes a movie title as input. It starts by converting the title into a movie ID we can use to query our model:

```
def recommend(movie):
    movie_id = str
        movies.title == movie].movie_ id.values[0])
```

5. We loop through the five most similar word vectors. We convert these IDs into movie titles that we print with their corresponding similarity scores:

```
    for id in model.wv.most_similar(movie_id)[:5]:
        title = movies[movies.movie_id == int(id[0])].title.values[0]
        print(f'{title}: {id[1]:.2f}')
```

6. We call this function to obtain the five movies that are the most similar to Star Wars in terms of cosine similarity:

```
recommend('Star Wars (1977)')
```

7. We receive the following output:

```
Return of the Jedi (1983): 0.61
Raiders of the Lost Ark (1981): 0.55
Godfather, The (1972): 0.49
Indiana Jones and the Last Crusade (1989): 0.46
White Squall (1996): 0.44
```

The model tells us that **Return of the Jedi** and **Raiders of the Lost Ark** are the most similar to **Star Wars**, although with a relatively low score (<

0.7). Nonetheless, this is a good result for our first step into the RecSys world! In later chapters, we'll see more powerful models and approaches to building state-of-the-art RecSys.

Summary

In this chapter, we learned about Node2Vec, a second architecture based on the popular Word2Vec. We implemented functions to generate biased random walks and explained the connection between their parameters and two network properties: homophily and structural equivalence. We showed their usefulness by comparing Node2Vec's results to DeepWalk's for Zachary's Karate Club. Finally, we built our first RecSys using a custom graph dataset and another implementation of Node2Vec. It gave us correct recommendations that we will improve even more in later chapters.

In ***Chapter 5***, *Including Node Features with Vanilla Neural Networks*, we will talk about one overlooked issue concerning DeepWalk and Node2Vec: the lack of proper node features. We will try to address this problem using traditional neural networks, which cannot understand the network topology. This dilemma is important to understand before we finally introduce the answer: graph neural networks.

Further reading

- [1] A. Grover and J. Leskovec, *node2vec: Scalable Feature Learning for Networks*. arXiv, 2016. DOI: 10.48550/ARXIV.1607.00653. Available: **https://arxiv.org/abs/1607.00653**.
- [2] F. Maxwell Harper and Joseph A. Konstan. 2015. *The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS)* 5, 4: 19:1–19:19. **https://doi.org/10.1145/2827872**. Available: **https://dl.acm.org/doi/10.1145/2827872**.