Addison Wesley Data & Analytics Series

MACHINE LEARNING IN PRODUCTION

Developing and Optimizing Data Science Workflows and Applications

ANDREW KELLEHER | ADAM KELLEHER

Machine Learning in Production

6.4.1 Layout Algorithms

It's a tricky problem to decide how to lay out a graph in a way that tells you something about the structure of it. People often hope to learn about the structure of the graph from visualizing it. In particular, they're often interested in whether there are obvious "clusters" in the graph.

By clusters, we mean a collection of objects in the graph that are far more connected to each other than they are to the rest of the graph. This is a tricky thing to measure in general, and straightforward measures of clustering can have nuanced and interesting problems. We'll dive more into this later when we talk more about graph algorithms. A useful sanity check is to visualize the graph and make sure the clusters you've detected align with the clusters that are apparent in the graph visualization (if there are any!).

How can you visualize the graph? Figure 6.21 shows what you get if you just randomly place the objects and their connections in an area.

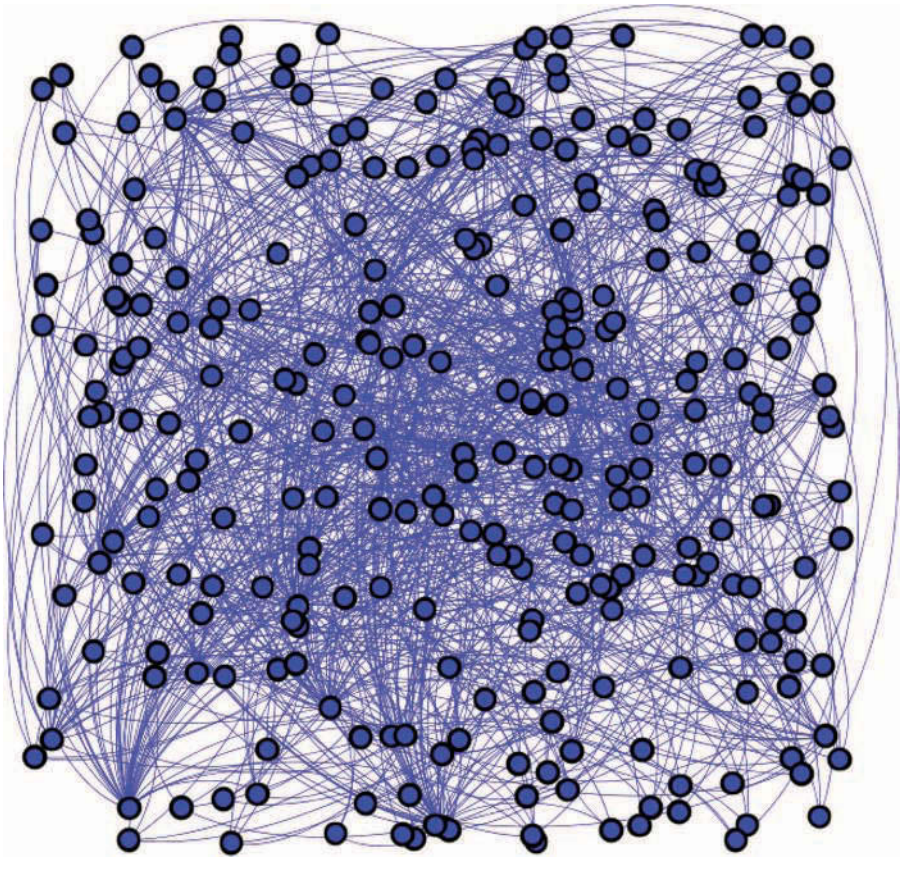

Figure 6.21 Graph visualizations that don't use the structure of the graph are unenlightening. You can do better than this!

You can see that there's nothing enlightening here. It looks like a random assortment of objects and connections. You can do a lot better!

You can imagine that each connection between objects is like a spring, pulling the objects together. Then, you can imagine each object repels every other object, so they're held closely together only by the springs. The result is that if a set of nodes are well interconnected, they'll form dense clusters or knots, while everything else tends to stretch out and fly apart. If you apply that to the graph (using ForceAtlas2 in Gephi), you can get the visualization in Figure 6.22.

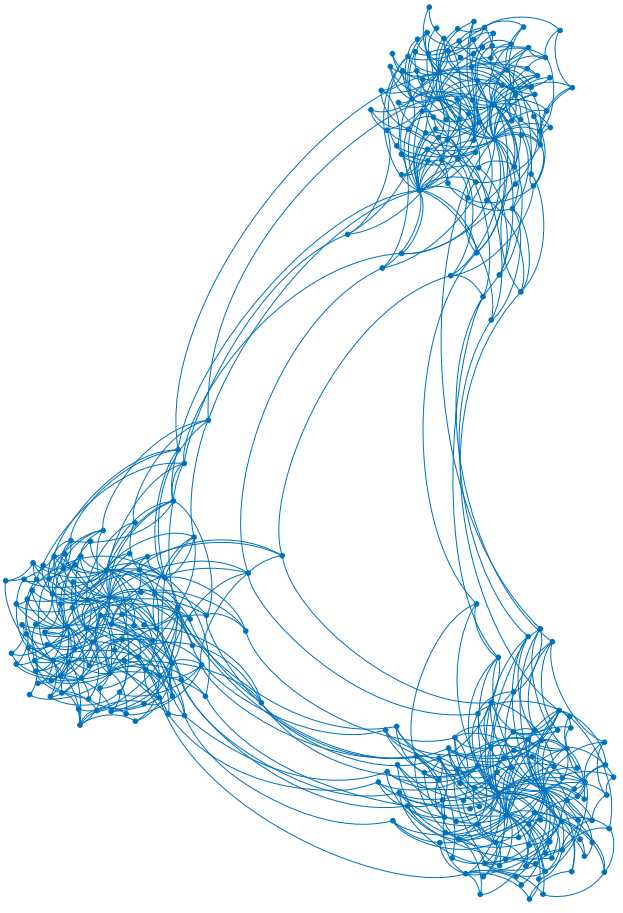

Figure 6.22 This visualization relies on the edges to attract nodes to each other, and the nodes themselves repel each other. This results in interconnected nodes getting grouped together, and the groups tend to push one another apart. You can get a nice picture of the structure from this layout, but it can be computationally expensive. It's intractable for large graphs.

You can see clearly here that there are three clusters in the graph that are loosely connected by a few extra connections. This graph was built using three random, disconnected components that were

attached together by adding a few more connections at random. This is clear just from the visualization.

These are great algorithms for small graphs, but you'll see that they can be intractable for moderately sized or large graphs. Generally, they're good for graphs up to around a few hundred thousand nodes. After that, you need to use a different technique, like h3-layout.

6.4.2 Time Complexity

Most small-scale layout algorithms run with something like the following procedure:

```
# main loop
for step in num_steps:
    for node in graph:
        force = calculate_force(node)
        new_position = update_position(force)

# force calculation
def calculate_force(node):
    force = zero_vector
    for other_node in graph:
        force += update_force(node, other_node)
    return force
```

You can see from this algorithm, each time `calculate_force` is called, you have to iterate over every object (node) in the graph. Within each step, you have to call this function once for each node in the graph! This results in N^2 calls to `update_force`. Even if this function has a fast runtime, these N^2 calls to it cause the algorithm to slow down tremendously for even moderately sized graphs. Even if the call takes just a microsecond, a graph with 100,000 nodes would make 10^{10} calls, resulting in about 10,000 seconds runtime. A graph with 500,000 nodes would take around 70 hours!

This problem, where the runtime grows with the square of the input, is our first look at the *time complexity* of an algorithm. Generally, when the runtime grows like the square of the input, the algorithm is not considered scalable. Even if you parallelize it, you only reduce the runtime by a constant factor. You might make it scale to slightly larger inputs, but the scaling problem still exists. The runtime will still be long for just slightly larger inputs. This can be okay in some applications, but it doesn't solve the problem of quadratic time complexity.

6.5 Conclusion

In this chapter, we covered some of the basics of data visualization. These can be useful when you're presenting results to your team, writing reports or automated emails to send to stakeholders, or sharing information with the rest of your team.

When you're making data visualizations, always consider your audience. A fellow data scientist might be interested in error bars, detailed labels and captions, and much more information contained in the plots. Stakeholders might be interested in only one quantity being bigger than another or the graph going "up and to the right." Don't include more detail than necessary. A plot can easily become overwhelming and push attention away from itself rather than ease the presentation of the data. Always consider removing tick marks and lines, moving labels and numbers down into captions, and simplifying labels however you can. Less is more.

This page intentionally left blank

II

Algorithms and Architectures