Predicting Initial Public Offerings Using Graph Convolutional Neural Networks

May 20th, 2020

About a year ago, I was introduced to the concept of graph databases, and how they represent data differently compared to a tabular, relational database. I was fascinated that there was a way to store and find relationships in data in a manner that I found more intuitive instead of computing JOINs on tabular data. If two pieces of data are related, in a graph database, you simply create an edge between them. Since the data is in a graph, you can perform all the standard graph algorithms on your database, such as breadth and depth-first search, shortest path algorithms, and similarity algorithms. All of these algorithms work off the edges (relationships) between the various data points. Turns out, there are machine learning algorithms that also work with the relationships between the data. This article will walk through the steps to get your own graph machine learning pipeline up and running — from database to predictions. Photo by Rick Tap on Unsplash

First, we will set up a TigerGraph Cloud instance with an example dataset from Crunchbase. This data has about 200,000 companies in various stages of funding, and contains information such as if they have achieved an Initial Public Offering (IPO), the key investors of the company, the founders, their headquarters location, etc. We will then connect to the database using pyTigerGraph and Giraffle inside of a Jupyter Notebook. Finally, we will set up a Graph Convolutional Neural Network (GCN) to predict whether a given company in our dataset will IPO. To follow along with the code, check out the repository here.

Setting Up TigerGraph Cloud

I’m not going to go that in-depth with setting up a TigerGraph Cloud instance, as this article does a really good job walking you through the steps of provisioning an instance. On step 1, simply choose the “Enterprise Knowledge Graph (Crunchbase)” starter kit. Once you have your starter kit up and running, we will have to get a SSL certificate to access the server via Gradle. In your project directory, type this into your terminal:

```
openssl s_client -connect <YOUR_HOSTNAME_HERE>.i.tgcloud.io:14240
< /dev/null 2> /dev/null | \
openssl x509 -text > cert.txt
```

We also need to create two other files. First, lets create a gradle-local.properties file in the base project directory. This should contain:

```
gsqlHost=YOUR_HOSTNAME_HERE.i.tgcloud.io
gsqlUserName=tigergraph
gsqlPassword=YOUR_PASSWORD_HERE
gsqlAdminUserName=tigergraph
gsqlAdminPassword=YOUR_PASSWORD_HERE
gsqlCaCert=./cert.txt
```

The other should be placed in the py_scripts/ directory and be named cfg.py. This should contain:

```
secret = "YOUR_SECRET_HERE"
token = ""
password = "YOUR_PASSWORD_HERE"
```

The secret key can be obtained in Graph Studio under the admin page.

Installing Queries and Pulling Data

We will use two different tools to interface with our TigerGraph cloud instance: Giraffle and pyTigerGraph. Giraffle will allow us to install queries we need to run on the database, while pyTigerGraph will provide an interface for us to pull the results of those queries into Python. Giraffle

Giraffle is a plugin for Gradle, a build system. This allows you to easily package up code to be deployed on various different platforms and use version control software such as Git to keep track of the queries you write for the database. For more information, check out its project page here.

The Queries

We install the queries in the first few cells of the Jupyter Notebook through a couple of terminal commands. What each query does is outlined below: - **companyLinks** computes relationships between companies in TigerGraph and returns them in a JSON payload that we can parse with Python. This oversimplifies the graph here. The query returns pairs of companies that have something in common. This hurts accuracy, as some common elements (founders, investors, etc.) might be more important than location or industry. It is possible to create a GCN that has multiple types of vertices, (known as a Relational Graph Convolutional Notebook) but it is more complex. A good way to get started is to simplify the graph until you only have relations between the same type of thing. - **getAllCompanies** does exactly what the name implies — it returns a list of all companies found in the dataset. The reason why we need this will become apparent in the next section. - **getAllIpo** gets all the companies that have IPOed found in the dataset. This is useful in the next section, as well as checking our accuracy of predictions.

pyTigerGraph

In order to get the results from the queries we installed, we will use pyTigerGraph. For more information, check out the package on my GitHub here.

Undersampling Data

Alright, now that we got all of the queries installed and the data pulled into our notebook, notice a few things. First, the number of IPOed companies is minuscule (about 1,200) compared to the total number of companies (about 200,000). This means that the dataset is extremely unbalanced, and that will lead to the GCN predicting every company not to IPO (that way it would be 99.4%

accurate). Another thing to take into account is that most computers will not have enough memory to run a GCN on the full graph. Conveniently, the unbalanced data means that we should undersample the non-IPOed companies in order to make a more evenly balanced dataset. This results in a graph that has about 2,000 vertices, evenly split between companies that have IPOed and ones that have not. There is a drawback to this approach, however. Since these companies are randomly sampled from the non-IPOed and IPOed list, we cannot guarantee that there are many edges between each company, which hurts our accuracy quite a bit.

The Graph Convolutional Neural Network

GCNGif Classification using a Graph Convolutional Neural Network (Source: https://docs.dgl.ai/en/latest/tutorials/basics/1_first.html)

A Graph Convolutional Neural Network (GCN) is a semi-supervised classification algorithm that works off of the connections in the graph, as well as the features of each vertex. The concept is similar to a traditional image-based convolutional neural network, but instead of looking at adjacent pixels, the GCN looks at vertices that are connected via an edge. The vertex features can be any vector, such as a doc2vec representation of the various attributes of the vertex, or simply just a one-hot encoded vector, which is what I chose to do here. We then label two different vertices, one that is known to have IPOed and another that hasn't.

Our neural network architecture is pretty straight forward. It consists of two layers, with an input dimension equal to the number of entries in the feature vector (it also happens to be the number of vertices in the graph since we one-hot encoded them). We then pass the input through a layer with 32 neurons, and then out through 2 neurons that will provide our output. We use Adam as our optimizer for the training process. We then begin the training loop. Unfortunately, due to our undersampling of the graph earlier, the GCN does not always have enough edges in the graph to reliably make predictions accurately. I usually get about 60% accuracy, but it does vary quite a bit due to the random sample of companies.

Conclusion

The GCN is not a great method to predict if a company will IPO or not, due to the memory constraints and the need for undersampling the graph. Other graph machine learning methods, such as node2vec might fair better. Another way that accuracy might improve is using a relational graph convolutional neural network (R-GCN) which would work on graphs with multiple different types of vertices.

Credits

Article and notebook written by Parker Erickson, a student at the University of Minnesota pursuing a B.S. in Computer Science. His interests include graph databases, machine learning, traveling, playing the saxophone, and watching Minnesota Twins baseball. Feel free to reach out! Find him at:

- LinkedIn: https://www.linkedin.com/in/parker-erickson/
- GitHub: https://github.com/parkererickson
- Email: parker.erickson30@gmail.com

GCN Resources:

- DGL Documentation: https://docs.dgl.ai/
- GCN paper by Kipf and Welling https://arxiv.org/abs/1609.02907
- R-GCN paper: https://arxiv.org/abs/1703.06103

Notebook adapted from: https://docs.dgl.ai/en/latest/tutorials/basics/1_first.html