Deep Graph Infomax investigations

James Singleton, 18 December 2020

What is Deep Graph Infomax, anyway?

The idea in a nutshell is to take the dataset representing a graph `[V, E]` where `V` and `E` are nodes and edges, and to map them into a new space, `X`, which is a `n x p` matrix, with `n` nodes and `p` features for each node. The p features are a dense vector representation of the original node features. Effectively, Deep Graph Infomax aims to combine the nature of the nodes with the position in the graph. This then results in a new set of node embeddings, which we should probably call "dgi embeddings", to avoid confusion with embedding of the node data alone.

How does Deep Graph Infomax arrive at this function `f()`?

This algorithm generates some competing fake data, not from the graph, but similar to the graph, by permuting the edges. A classification model is then built to classify the data into two buckets, original and fake. In the process of training this model, the desired function mapping the edge and node information is created, which is responsible for the feature matrix, necessary to build the classifier. The idea is from Google, but has been made a reality by the Australian data team "Csiro Data61" in their Conda package, "Stellargraph".

https://research.google/pubs/pub48921/

https://stellargraph.readthedocs.io/en/stable/demos/embeddings/deep-graph-infomax-embeddings.html

https://medium.com/stellargraph/do-i-know-you-flexible-unsupervised-and-semi-supervised-graph-models-with-deep-graph-infomax-96fbfd63ec31

Key concepts

- personalisation made possible by combining user journeys with feature data to create DGI embeddings
- user journeys are captured from the graph with the triple store information of start node, end node and weight, which means the algorithm is using importance sampling
- content can be recommended using the Spotify Annoy library, which is inherently fast, based on indexing the DGI embeddings
- extend the idea to heterogeneous graph structures with multiple node types using HinSage as the learner instead of GCN, where the node types could be extended to (pseudo) users
- alternatively, extend the feature data of nodes to include user data (this is potentially not feasible as the graph will double the nodes for each additional type of user data)

Acknowledgement of previous work done in this area

Previously, the Data Labs team have investigated Stellargraph algorithms. Namely, there was an effort to predict links between nodes, using GraphSage. Key changes to this process are:

- the use of the USE embeddings, particularly at the sentence level, rather than "bag of words"

- the generation of DGI embeddings which could then subsequently be used to recommend content, rather than predicting the strength of all `n**2` links in the graph of `n` nodes

https://github.com/alphagov/govuk-network-embedding/blob/master/notebooks/data_extract/make_predict_node_data.ipynb

Required ingredients to run the Deep Graph Infomax code

Requirements:

- edge data, node data
- Universal Sentence Encoder to generate node embeddings
- StellarGraph Deep Graph Infomax environment, Python 3.6, dependencies in requirements
- Annoy library to illustrate cosine similarity to find similar content

Computational requirements

Additionally, to train the Deep Graph Infomax algorithm, full batch processing occurs, which means that the entire network is processed in one batch. The machine used to do this is a i7 6700 quad core, with 64gb of RAM and GTX 1070 graphics card. Using Tensorflow-GPU, the training time is about 10 minutes, for the best model. It is hard to know exactly how long a run would take without using GPU, or on a machine lacking RAM.

Edge information is created using the Google Universal Sentence Encoder, v4 large, as illustrated previously in the document similarity work. Run time across the document contents is approximately 20 minutes on a GTX 1070, so likely to take approximately 3 hours on a CPU. RAM issues were experienced at one point, so that the maximum number of sentences per document had to be restricted to 128 (128 sentences is probably sufficient for understanding what the doc is about anyway).

Getting the data: initial_connection.py

The node data came from a downloaded version of the graph, previously provided. This is not ideal, in hindsight as it was out of date. In contrast, the edge data came from directly connecting to the graph using `py2neo`, over the VPN, and the following query:

```
result = gov_graph.run("MATCH (m:Cid) -[r:USER_MOVEMENT]-> (n:Cid) RETURN m.contentID, n.contentID, r.weight").data()
```

This could be simply extended to indlude `.description .text .first_published_at .contentID .document_type`, which would then ensure that the code pulled the necessary content simultaneously and avoid the risk of bad joins in the subsequent pipeline. Note that we require `.document_type` to see whether the DGI embeddings serve as useful feature data.

Creating the Universal Sentence Embeddings: USE_embeddings.py

In order to translate the node text into meaningful feature data, the Universal Sentence Encoder is used. The trick here is to break the document into segments, ideally sentences, but here the model is just looking at XML paragraph breaks, which roughly corresponded to sentences (opportunity for tweaking). The document is then expressed as a `n x p` of `n` sentence embeddings, dimension `p`. We then take the simple

row wise mean of the vectors, to come up with a document embedding. It seems to work, as seen before for investigations into content similarity.

Deep Graph Infomax investigations: DGI_embeddings.py

The build of the Deep Graph Infomax embeddings rests upon the creation of a local version of the directional graph, using `networkx`. Subsequently, we run the `StellarGraph.DeepGraphInfomax` function (in Python 3.6). Note that this is a Conda environment. All libraries have been pulled from Conda where possible, including Tensorflow-GPU for acceleration of the training time.

As Deep Graph Infomax is unsupervised, we have no real measure of the actual predictive power of anything. In the absence of a real target, we subsequently test the embeddings for usefulness in predicting a pseudo target of `document_type`, which was hidden from the feature generation. A logistic regression model is trained, and compared to the benchmark accuracy of a random sample (permutation test) prediction.

The overall accuracy of logistic regression based on Deep Graph Infomax embeddings is 35%. The reference permutation test is 9%. However, caution is still required here, the further question to investiate is what the logistic regression would have achieved from the original USE embeddings. Is there any value in the DGI embeddings. This has not been investigated.

Investigating the recommendations in Annoy: DGI_annoy.py

This is still work in progress. By loading the DGI embeddings into Annoy, a high dimensional space (same dimension as the DGI embedding dimension) is created and the documents are assigned to locations in this space. Annoy then indexes documents to sections of this space, so that when a query is run on the Annoy index, the Annoy functionality simply looks at the location address, and retrieves the relevant documents. The heavy lifting is done once, order O(log(n)) (probably - its splitting the space up using a decision tree like structure), in the creation of the space. The retrieval time is so fast as to be perceived as instant.

Note that the creation of the USE embeddings in the instance has included the description field in the document where the document text is null. This then means that the Annoy functionality throws an error presently, if there is no document text.