Three-Graph Natural Language Processer: A Solution for Graph-Based Text Analysis

Jesse Coulson, Primrose Johns (She||Her)
Github

Abstract

In both academia and industry, natural language processing is a topic that has received much attention for both its current and potential use cases. One important use case for natural language processing is the analysis of a corpus to which words therein have significant relationships with each other. We have developed a python library that seeks to provide effective solutions to the problem of text analysis by generating graphs that represent semantic, syntactic, and sequential relationships between words in a provided corpus.

1 Introduction

There are many different use cases that involve some kind of text analysis. It is common to use large text databases to analyze trends among users, or to summarize reviews of a product or service. Large collections of smaller text entries, like a user's search history, are useful tools for trend analysis and advertising. A very popular contemporary use case is in the field of AI, specifically for training models that can parse and generate natural-language text. One of the most interesting developments in NLP and graph theory has been progress made with GCN's, or graph convolutional neural networks. These deep learning models are trained using graphs as input, and when applied to the problems of text comprehension or classification this necessitates a pre-processing step in order for the provided training corpus to be converted into a graph.

This particular subfield of graph theory has produced several graph generation methodolgies that seek to create the most meaningful posssible graphs out of a corpus in order to give their model the best possible input(s). This work led us to consider an alternative use case of these methodologies. Instead of these graphs being the training data for an AI model, what if they were an end in themselves? We supposed that these same methods could be modified to create useful analytical data about a provided corpus.

2 Related Work

The general structure of our system is derived from the methods discussed by Xien Lu et al.[1]. This paper put forth their system for developing a state-of-the-art GCN for text classification. One notable feature of their system (referred to as TensorGCN) that distinguished it from prior methodology was their development of a way to propagate multiple graphs together in order to train a model on analytically heterogeneous information. Specifically, a single model could be trained on semantic, syntactic, and sequential data all at once. This paper was an invaluable resource for our work, and in many ways the primary systems we developed were our attempts to generalize and simplify their methodologies into something that could be an effective general-purpose tool for text analysis. Where their work was attempting to synthesize different kinds of information, we observed that this system could also produce three independant graphs of useful data for corpus analysis.

It should be noted that our work with this topic was initially inspired by the work of Usman Nassem et al [2]. Their successful work incorporating temporal relationships into the structure of a TensorGCN-like text classifier was our introduction to this particular methodology for training deep-learning text classifiers.

Note: The order of author Names in the title of this paper is alphabetical, and is not representative of seniority or work performed.

3 Preprocessing

Text Parser

Our methodology uses three objects as inputs: A list of words in the order they appear (including duplicates), a list of sentences in the order they appear, and a dictionary of word frequencies. An example of the word and sentence lists are shown below:

Text:
Fee fi fo fum! I smell the blood of an englishmun!
Word list:

```
["fee", "fi", "fo", "fum", "i", "smell",
"the", "blood", "of", "an", "englishmun"]
```

Sentence List:

```
["fee fi fo fum",
"i smell the blood of an englishmun"]
```

Figure 1: Examples of the word and Sentence Lists Data Objects TGNLP Uses for Graph Generation

The dictionary (not shown above) is very simple. Keys are words, and values are integers representing the frequency of the word in a corpus.

The Corpus Class

In order to make our system more user-friendly, we have created a "corpus" class that users use instead of calling the text parser and word-frequency dict creation functions directly. Instead, the user calls the tgnlp.Corpus() class constructor with their raw data as input, and the constructor handles generating and storing the required objects.

4 Semantic Graph

The semantic graph uses word2vec to generate embeddings of each word. The system is given the corpus as a list of sentences, rather than as a list of words. Once the embeddings are generated, we utilize the `most_similar(n)` function in word2vec in order to find the n most similar words to the provide word. By default, n is 20. The cosine similarity between these words become the edge weight, which can be normalized if necessary. An alternative approach we considered would be to check the cosine similarity between every node and every other node and then trim the graph down, but this proved to be to heavily resource intensive and lacked meaningful improvement over the approach of using `most_similar(n)`. An example subgraph demonstrating the types of connections this generates is provided.

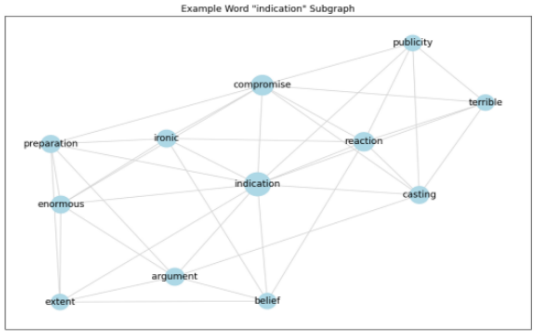

Figure 2: Example Semantic Subgraph

5 Syntactic Graph

Our methodology for syntactic graph generation made use of the spaCy library's dependancy parser. This can be compared to the way that the Stanford CoreNLP parser is used by TensorGCN. However, unlike CoreNLP the spaCy parser allows for custom toeknizers to be used. This is useful if your corpus has words that a standard tokenizer may not know. SpaCy uses our sentence corpus in order to generate and label the different kind of dependencies in each sentence.

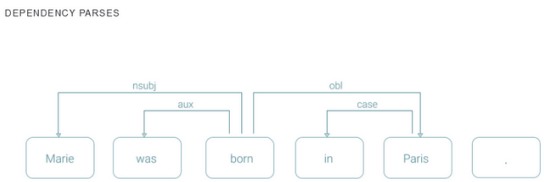

Figure 3: Different kinds of word dependancies in a sentence. source

In our methodology, we generalize these dependencies by of affording them equal importance in the graph. The formula for calculating edge weights in the graph is $W_{i,j} = \frac{freq_{i,j}}{min\{freq_i, freq_j\}}$ where $freq_{i,j}$ is the number of dependencies that the two words have over the corpus and $freq_i$ is the frequency of that

word over the corpus. By dividing co-occurrence values by the lesser word's frequency, words that have few co-occurrences in corpus will have high weights with the few words they are near. This means that uncommon co-occurrences between common words will have very small weights comparatively, and be more likely to be dropped if the graph is trimmed. An example subgraph demonstrating the types of connections this generates is provided.

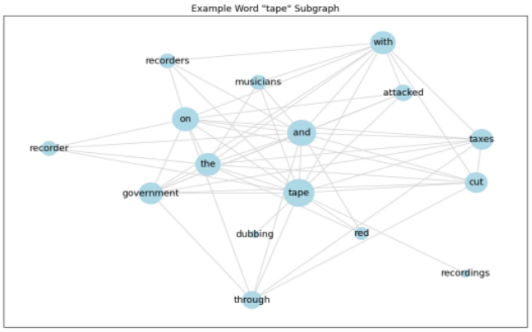

Figure 4: Example Syntactic Subgraph

6 Sequential Graph

The sequential graph was probably the most straightforward to implement. The idea of the sequential graph is to connect nodes (which are words) to one another with weighted edges that represent how close together those words are in the corpus on average. The implementation keeps track of the frequency of word co-occurrences in a sliding window that passes over the corpus. This sliding window approach has the benefit of counting the same two words more than once if they are closer together, since the window only moves one word at a time. The frequency of word co-occurrences are tracked in a dictionary similar to the way word frequencies are. The keys in the co-occurrence dictionary are generated as the "[word1],[word2]". These words are always in alphabetical order in the keys, so that there aren't entries for both "amber,waves" and "waves,amber" in the dictionary.

Once we have used the sliding windows technique to generate our co-occurrence dictionary, generating the untrimmed graph from it is simple. The dictionary is iterated through, and at each loop we grab the two words referenced from the key and the frequency of the co-occurrence from the value. Then, we generate the weight of the edge between the two nodes as $W_{i,j} = \frac{freq_{i,j}}{min\{freq_i, freq_j\}}$ (the same as with the syntactic graph). An example subgraph demonstrating the types of connections this generates is provided.

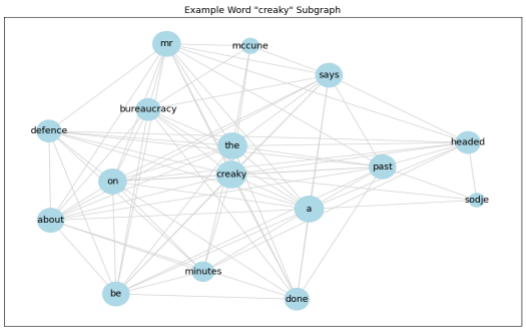

Figure 5: Example Sequential Subgraph

7 Graph Processing

We developed a dual-purpose function called `trim_norm_graph()` in order to help remove unimportant edges and prevent hairballs. It performs two functions, scaling and trimming.

Scaling

Before the graph can be trimmed, it has to be scaled. This is because our trimming method works using numerical integration, and will not work well if the lower bound of our edge weights are too high. To do this, we use the sci-kit-learn function `min_max_scaler()` to transform the edge weights to values between 0 and 1. The weights are then updated within the graph in a canonical way.

Trimming

The trimming portion of the function attempts to remove the very small edges that are in the graph, while keeping most intact. This is somewhat challenging, since the absolute value or the percentage of edges that we want to trim can change from corpus to corpus. Instead, we use numerical integration. By summing up all weights in the graph, and then determine how much total weight we want to trim (10% by default). The function then iterates through the edges in ascending order of edge weight, removing each one until the total amount trimmed is less than or equal the target total amount. The graph is now scaled and trimmed, and is ready for text analysis.

8 Analytics

Graph Report

We developed several functions that allow the user to easily analyze specific parts of the graph data. These functions are used collectively by `generate_graph_report()`. This function uses Bibtex to generate a pdf report showing a few useful metrics on the graph and its highest- and lowest-degree nodes, visualizations of linear and logarithmic degree distributions, and an example subgraph using a typical word in the corpus. The word chosen is the first node found in the graph S.T. $10 < k < 15$ for the node's degree k. An example logarithmic degree distribution visualization generated from BBC new articles is shown.

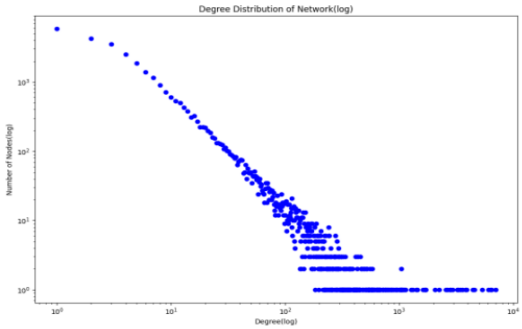

Figure 6: Example Degree Distribution From a Trimmed TGNLP Syntactic Graph

Graph and Node Metrics

The metrics we chose to focus on were a combination of what data we felt was most useful to user's data at a glance and what data could be gathered without requiring prolonged runtimes when the number of nodes in the graph is very large. We report on:

- The number of nodes and edges
- The highest and lowest degree nodes along with their degrees and degree centralities
- The average degree of nodes in the network, the average centrality
- the degree assortativity coefficient of the graph.

These values are be collected and returned as a dictionary using our `generate_metrics()` function.

Subgraph Visualization

We developed a function to generate a visualization of any word in the corpus and its nth-degree neighbors. These visualizations are done in `Matplotlib.pyplot`, and by default only first-degree neighbors are shown. An example for the word "creaky" is shown, with its neighbors being the result of a graph analyzing BBC new articles:

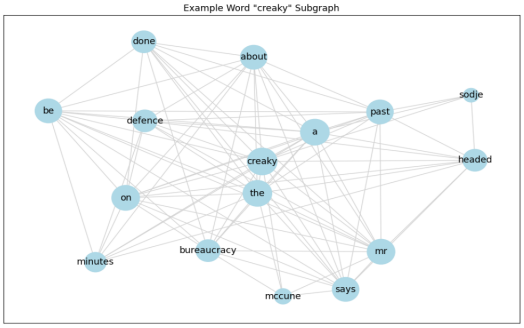

Figure 7: Example Subgraph From a TGNLP Report

9 Testing and Coverage

We did our testing and code coverage using pytest for the unit tests and the tool "coverage" for code coverage. We developed five tests, which are all passing as of this writing. Total code coverage is at 94%.

10 Future Work

Primarily, we would like to improve on this work by fully reviewing the project, writing documentation, and adding more functionality. We would like to explore potential features that could assist users looking to use these graphs as inputs to GCNs. We would also be interested in adding more features to help summarize and visualize the potentially massive graphs that users can generate. These two paths represent what we believe are the main use cases of our project, and we have ideas for how to generally improve both.

11 Conclusion

We have succeeded in creating a useful tool that allows a user to analyze large corpuses of text by generating graph(s) that capture important relationships between words. Several different graph types are available, they can be generated easily using our API, and a comprehensive report on each graph can also be generated using a one-line API call. We feel this is a strong foundation upon which we can build further, increasing our functionality and accessibility to users.

References

[1] Xien Liu, Xinxin You, Xiao Zhang, Ji Wu, and Ping Lv. Tensor graph convolutional networks for text classification. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 8409–8416, 2020.

[2] Usman Naseem, Jinman Kim, Matloob Khushi, and Adam Dunn. Graph-based hierarchical attention network for suicide risk detection on social media. In *Companion Proceedings of the ACM Web Conference 2023*, pages 995–1003, 2023.