Yassine MKAOUAR
yassine.mkaouar@polytechnique.edu
Mohamed Amine BESBES
mohamed-amine.besbes@polytechnique.edu
Eya JLASSI
eya.jlassi@polytechnique.edu

Generating Graphs with Specified Properties
ALTEGRAD 2024

01/14/25

1 Introduction

Graphs are powerful data structures widely used to represent complex relationships and interactions in various fields. They serve as an intuitive framework for modeling connections between entities. With the growing reliance on graphs in data science and machine learning, the ability to generate realistic and meaningful graph structures has become increasingly vital.In this spectrum, **graph generation** refers to the task of creating novel graphs that not only resemble real-world structures but also exhibit specific desired properties. In this context, **Generating Graphs with Specified Properties** competition, challenges participants to develop machine learning models that are capable of translating descriptive textual queries into corresponding graph structures that meet the specified properties.

In this challenge, we are provided with a dataset comprising graphs and their corresponding textual descriptions. Each description contains key properties of the graph, including the number of nodes, edges, average node degree, number of triangles, clustering coefficient, maximum k-core, and number of communities. The objective is to train a model that learns the latent space of the graph distribution, embeds the textual queries into this space, samples a vector that satisfies the query, and decodes it to construct the desired graph.

The training dataset includes **8,000** graph-description pairs, with **1,000** pairs reserved for validation and another **1,000** for testing.

For evaluation, the properties of each generated graph are computed and compared to the corresponding properties of the target graph. The performance metric is the **Mean Absolute Error (MAE)**, which measures the average absolute difference between the generated and target property values. The goal is to minimize the MAE, ensuring the generated graphs closely align with the specified properties.
The formula for MAE is as follows:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Where:

- n is the number of properties being evaluated,
- y_i is the actual property value of the target graph,
- $\hat{y}_i$ is the corresponding property value of the generated graph.

Minimizing the MAE ensures the model accurately captures the specified graph properties during generation.

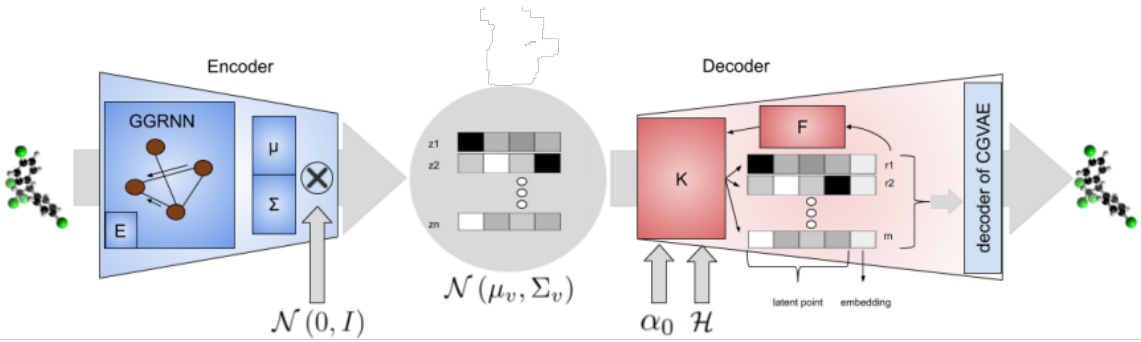

Figure 1: Conditional Constrained Graph Variational Autoencoder model structure [RNS20]

2 Architectures

2.1 Model Architectures

The implemented model consists of three main components: a text embedding module that converts the description text into a meaningful vector representation, a Variational Autoencoder (VAE) that creates a latent space to model the graph distribution, and a graph generation module that combines the text embedding with a sampled latent vector to decode and construct the desired graph based on the description.

2.1.1 Feature Processing and Embedding Techniques

To enhance the representation of graph descriptions, we adopted a hybrid approach that combines numerical feature extraction and semantic embedding. Initially, the project utilized a limited set of seven numerical features extracted directly from the graph descriptions, such as the number of nodes, edges, average degree, and global clustering coefficient. These were later replaced with advanced textual embeddings generated using the BERT model, producing 768-dimensional vectors. A new function, `text_to_embedding`, was introduced to replace the `extract_feats` function in the `preprocess_dataset` pipeline, and the input dimensions of the Variational AutoEncoder (VAE) were adjusted accordingly. While this approach improved the model's performance, it was limited by the dated nature of BERT. To address this, we explored the **Gemini API**'s `text-embedding-004` model, which generates state-of-the-art continuous vector representations for words, phrases, and sentences. Recognizing that differences between descriptions are often tied to numerical feature values, we developed a comprehensive process combining both numerical and semantic approaches. Numerical features were extracted from textual descriptions using regular expressions and normalized for consistency. Simultaneously, semantic embeddings were generated with the Gemini API. These components were then concatenated, creating high-dimensional vectors that provide a rich and complete representation of the graphs, optimized for subsequent tasks.

2.1.2 Variational Auto-encoder

The Variational Autoencoder (VAE) module serves as the core component for learning a latent representation of graphs and generating graph structures conditioned on textual descriptions.

1. **The encoder** in the Variational Autoencoder (VAE) leverages a **Graph Isomorphism Network (GIN)** to extract structural information from input graphs. The GIN aggregates node features by considering neighboring nodes and applies transformations such as fully connected layers, LeakyReLU activations, and Batch Normalization. The encoder outputs two key vectors: the **mean** (μ), which represents the center of the latent space distribution for the input graph, and the **log variance** ($\log \sigma^2$), which defines the spread of this distribution. These outputs enable the VAE to model the graph's distribution in the latent space.

   $\Longrightarrow$ Instead of adhering strictly to the provided model, we enhanced it by fine-tuning its parameters and introducing additional modifications. For instance, we integrated dropout layers to mitigate overfitting. Furthermore, we utilized a powerful hyperparameter optimization library, **Optuna**, which employs advanced methods such as Tree-structured Parzen Estimators (TPE) to determine optimal hyperparameters. Through this process, we adjusted various components of the model, including:

   - Increasing the dropout rate to enhance generalization ($0.0 \rightarrow 0.3$).
   - Expanding the hidden dimensions of the encoder($64 \rightarrow 256$), decoder($256 \rightarrow 512$), and latent space ($32 \rightarrow 64$)for better representation capacity.
   - Increasing the number of hidden layers in both the encoder($2 \rightarrow 5$) and decoder($3 \rightarrow 6$) to capture more complex patterns.

   These enhancements significantly improved the performance and robustness of the model.

2. The key outputs of the encoder are used to create the **latent space**. Each graph is encoded as a Gaussian distribution parameterized by the mean (μ) and log variance ($\log \sigma^2$). The VAE employs the **reparameterization trick** to sample latent vectors while maintaining differentiability for training. This is achieved by sampling a random variable ϵ from a standard Gaussian distribution and scaling it using the latent space parameters:

$$z = \mu + \epsilon \cdot \sigma, \quad \text{where } \sigma = \exp\left(\frac{\log \sigma^2}{2}\right), \epsilon \sim \mathcal{N}(0, I).$$

In our model, the latent vector is concatenated with a text embedding derived from the graph's description, allowing the decoder to condition the graph generation on both the graph's encoded structure and its textual description.

3. The **decoder** reconstructs the graph from the latent vector and text embedding by generating an adjacency matrix. Using a **Multi-Layer Perceptron (MLP)** with Batch Normalization (added to stabilize training in the decoder) and Gumbel Softmax sampling, the decoder outputs a symmetric adjacency matrix, ensuring the generated graph adheres to the properties described in the input text. This process translates the combined latent and text features into a realistic graph structure.

4. Since both the encoder and decoder are implemented using neural networks, the VAE optimizes its parameters by minimizing a composite loss function, which consists of two key components:

   (a) **Reconstruction Loss**: This measures the accuracy of the generated graph by comparing the reconstructed adjacency matrix with the ground truth.
   $\rightarrow$ The loss used was changed from l1 loss to **MSE** to ensures that large deviations are penalized more, leading to better alignment with the true properties.

   (b) **KL Divergence**: This regularizes the latent space by aligning the learned distribution with a standard Gaussian distribution.

   The total loss combines these terms with a hyperparameter β to balance reconstruction accuracy and latent space regularization:

$$\text{Loss} = \text{Reconstruction Loss} + \beta \cdot \text{KL Divergence}.$$

5. A **denoise model** is then trained to refine noisy latent graph representations generated during the forward diffusion process. At each stage, the noisy latent graph representation is combined with the conditional vector, allowing the model to accurately predict the added noise. During the reverse diffusion process, this predicted noise is subtracted step-by-step, enabling the reconstruction of the original latent graph representation. This ensures the generated graphs are not only structurally realistic but also semantically aligned with the textual descriptions.

   $\Longrightarrow$ The steps implemented enabled us to achieve a significant improvement, skyrocketing our score from **0.89274**, which was obtained using the professor's provided code, to **0.14016**. However, during this process, we focused on minimizing the **Reconstruction Loss** and the **KL Divergence**, rather than directly optimizing the loss metric specified in the Kaggle challenge. To address this discrepancy, an additional step was introduced to align our optimization process with the challenge's evaluation criteria, ensuring that the model performance aligns with the desired metric. This refinement played a crucial role in bridging the gap between the training objective and the competition requirements.

2.2 Graph Generation

After training our model, we can generate a multitude of graphs corresponding to the given textual descriptions. However, like any deep learning model, our approach may occasionally produce graphs that fail to fully meet the specified properties due to potential inaccuracies at certain steps. To address this, it is prudent to introduce a quality control mechanism that selects, from the set of generated graphs, the one that minimizes the Mean Absolute Error (MAE) between its property vector and the target property vector described in the input. To achieve this, we developed an "evaluation.py" file, designed to identify and return the graph with the lowest **z-score normalized MAE**. This ensures that the final output aligns as closely as possible with the specified properties, enhancing the reliability of the graph generation process. To implement this quality check, we began by creating a vector containing the 7 required properties based on the text description. Simultaneously, we analyzed each generated graph, and using ***NetworkX python library*** we extracted the number of nodes, edges, triangles, average degree, maximum k-core, global clustering coefficient, and the number of communities. For the last property, we employed the **Louvain algorithm**, known for its efficiency with a complexity of $O(n \log n)$, making it suitable for large-scale graph analysis. To handle missing values, we assigned a value of -100 to such cases, effectively increasing the MAE and penalizing graphs with incomplete or invalid properties, ensuring their exclusion from selection.
While one might suggest directly using the raw MAE score to select the best graph, it is more robust to rely on the **z-score normalized MAE** for several reasons:

1. **Fair Comparison Across Properties:** Different properties, such as the number of nodes and clustering coefficient, have vastly different scales. Z-score normalization ensures that each property is evaluated on a common scale, preventing any single property from disproportionately influencing the selection process.

2. **Handles Variability:** By normalizing based on the mean and standard deviation of each property, the z-score MAE accounts for natural variability in the dataset, offering a more balanced evaluation.

This being said, we proceeded to calculating the mean and standard deviation for each property:

$$\text{normalized\_property}_i = \frac{y_i - \text{mean}_i}{\text{std}_i + \epsilon}$$

At the end, we return the graph that has the smallest **z-score normalized MAE**:

$$\text{mae\_st} = \frac{1}{n} \sum_{i=1}^{n} |\text{normalized\_gen}[i] - \text{normalized\_true}[i]|$$

$\Longrightarrow$ This approach increased the overall computation time, as it required generating n graph samples for the same description, significantly extending the processing time, especially considering the evaluation of 1000 test graphs. Despite the increased time consumption, this method effectively reduced the public MAE error by half, improving it from **0.14016** to **0.07508**.

3 What Did Not Work

This project provided a valuable opportunity to experiment with multiple approaches. Although not all attempts were successful, each contributed to our learning experience. Below, we highlight some of the methods we tried and the challenges we encountered:

1. **Implementing Different Encoders and Decoders:** Initially, we explored various encoder-decoder architectures, even before incorporating a text embedding module. For instance, we experimented with Diffusion Convolutional Networks (DCN), Graph Convolutional Networks (GCN), and Graph Attention Networks (GAT) as encoders, tuning their hyperparameters. However, these changes showed minimal improvement in the public MAE score, moving from 0.89274 (using GIN) to approximately 0.88838—an insignificant difference. After integrating text embeddings, we revisited DCN as an encoder. While this provided a slight improvement, the MAE score only dropped marginally from 0.14604 to 0.14540, which was not impactful enough to justify the computational cost.

2. **Clustering:** Another approach involved clustering the graphs and optimizing models and hyperparameters for each cluster. We implemented the k-means algorithm and used the Elbow method to determine the optimal number of clusters, which was identified as 25. Despite our efforts, as shown in the figure,

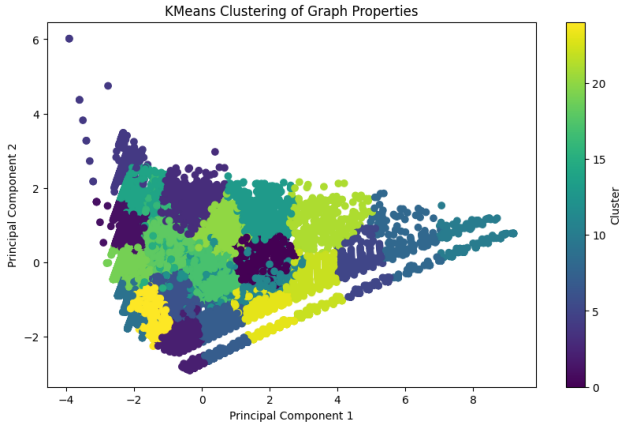

Figure 2: Graph Clustering

the clusters were very compact, making it challenging to distinguish meaningful groupings based on graph characteristics. Additionally, the dataset contained some outliers whose removal would have further reduced the size of an already limited training set, making this approach less feasible.

3. **Graph Attention Networks (GAT):** We also experimented with Graph Attention Networks. While GAT has the potential to improve performance by assigning varying importance to node connections, it was computationally expensive. Training the autoencoder using GAT took an excessive amount of time; for example, completing 78 epochs required over 5 hours. Moreover, this method demanded high computational resources, such as a powerful CPU and sufficient RAM, which were not available locally, further limiting its practicality.

4 What can be tried

Despite the challenges faced in earlier experiments, several promising directions can be explored to further improve the performance and efficiency of the model:

1. **Generative Adversarial Networks (GANs)** Generative Adversarial Networks (GANs) are a powerful tool for generating high-quality, diverse graph structures by leveraging a generator-discriminator setup. The generator creates graphs based on noise and text embeddings, while the discriminator evaluates the realism of the generated graphs. GANs were planned as a potential approach to enhance graph generation in this project, given their ability to address issues like mode collapse and improve output diversity. However, due to time constraints and the computational complexity involved in implementing and fine-tuning GANs, this method could not be fully explored within the project's timeline.

2. **Optimizing Computation with Cloud Resources and Platforms**: One approach to optimize and accelerate computation is using GCP, which provides the resources required for training complex models like GATs or GANs, significantly reducing training time with GPUs or TPUs.

3. **Hybrid Architectures:** Another approach that can be tried is combining multiple architectures like Graph Convolutional Networks (GCNs), Graph Attention Networks (GATs), and Diffusion Convolutional Networks (DCNs)ca n leverage their individual strengths to improve performance. For instance, a hybrid encoder could utilize GAT to highlight important nodes and edges using attention mechanisms, while integrating DCN to capture global diffusion patterns within the graph.

5 Conclusion

In this competition, we had the chance to explore the intersection of text and graph data in a unique formulation, moving beyond conventional tasks like classification and regression. We started by diving into relevant literature to identify potential areas of improvement and to guide our experimentation. Throughout the process, we tested various approaches, learning what worked and what didn't along the way.
Ultimately, our solution involved leveraging a combination of advanced models and robust evaluation techniques to improve the quality and reliability of the generated graphs. While we are satisfied with the progress made, there are still ideas and methodologies we would have liked to explore further. However, time constraints and the limited computational resources available presented challenges. Nonetheless, this experience provided valuable insights into graph generation and laid the foundation for future improvements in similar tasks.

References

[RNS20] Davide Rigoni, Nicolò Navarin, and Alessandro Sperduti. *Conditional Constrained Graph Variational Autoencoders for Molecule Design*. 2020. arXiv: 2009.00725 [cs.LG]. URL: https://arxiv.org/abs/2009.00725.