COL761 Assignment 1 Question 3

Kartik Arora (2021CS50124) Piyush Chauhan(2021CS11010)

Ayushman Kumar Singh (2021CS51004)

February 11, 2025

§1. Introduction

Given a graph database, identified the subgraphs crucial for classification. Used them to convert each database graph into a binary presence/absence feature vector consisting of at most 100 dimensions. The various approaches we tried are mentioned below.

§2. The Approaches we tried:

2.1. FastProbe Algorithm From Paper: LTS: Discriminative subgraph mining by learning from search history

- We implemented the fastProbe library from the paper. The memory requirements were exceeding the limit for NCI-H23 due to Depth first extensive search ingrained in the algorithm.
- We also tested their official implementation in JAVA. That also exceeded the memory limits.
- The reason was lots of:
  - subgraph isomorphism checks.
  - generating CCAM codes.
  - and extending a candidate graph by one extra edge.
- We tried various optimization techniques including systems and algorithmic techniques (like preprocessing the graphs using embedding codes and parallelization) but could not get much improvement on performance.
- We present below the algorithm we implemented:

Algorithm: *fast-probe* (G^+, G^-)
G^+: positive graph set
G^-: negative graph set
Candidate_list: the set of subgraph patterns to be extended
1. Put all single-edge subgraph patterns into *candidate_list*
2. while (*candidate_list* is not empty)
3. $p \leftarrow$ get next pattern and remove it from *candidate_list*
4. *updated* $\leftarrow$ *false*
5. for each graph g in G^+
6. if $s(p)$ > optimal score for g so far
7. update the optimal pattern and optimal score for g
8. *updated* $\leftarrow$ *true*
9. if (not *updated*)
10. continue
11. $C \leftarrow$ all subgraph patterns with one more edge attached to p
12. for each pattern q in C
13. if q has not been generated before
14. put q into *candidate_list*
15. return the optimal pattern for each g in G^+

2.2. Mining based on Custom Discriminative Score:

- **Train Test Splitting:** We performed all the experiments by randomly splitting the graph dataset into **80:20 train:test split.**
- **Positive Vs Negative graphs:** Graphs contained in class of less cardinality (ideally label 1) were categorized as positive graphs, and negative otherwise.
- We ran gaston with m=5(minimum number of nodes as 5) and s=0.1(minimum support of 10 percent) to mine the **most frequent subgraphs in the positive graphs.**
- **Scoring Funtion:** The **discriminative score** normally used is

$$\log(PositiveSupport) - \log(NegativeSupport)$$

We created a new scoring function which **weights the positive support along with the discriminative score**, and gave better performance:

$$2 \times \log(PositiveSupport) - \log(NegativeSupport)$$

- We then selected top 100 subgraphs based on the above scoring function to create or feature vectors. We present below the ROC AUC s on the train and test datasets for this approach:

Dataset	Train ROC	Test ROC
NCI-H23	0.891	0.831
Mutagenicity	0.882	0.813

2.3. Final Approach: Mining based on frequency and Using rustworkx for Isomorphism Checks:

- **Train Test Splitting:** We performed all the experiments by randomly splitting the graph dataset into **80:20 train:test split.**
- **Positive Vs Negative graphs:** Graphs contained in class of less cardinality (ideally label 1) were categorized as positive graphs, and negative otherwise.
- We ran gaston with m=5(minimum number of nodes as 5) and s=0.1(minimum support of 10 percent) to mine the **most frequent subgraphs in the positive graphs.**
- Then we selected top 100 most frequent patterns to consitute our features.
- **Grid Search to find the best parameters:** We searched over the space of m = [2,3,4,5,6,7,100] and s = [0.005, 0.01, 0.05, 0.1, 0.2]. We present test and train accuracies for **some of the data points:**

  Grid Search For Mutagenicity:

m	s	Train ROC	Test ROC
2	0.1	0.730	0.709
3	0.05	0.847	0.798
4	0.2	0.865	0.802
5	0.01	0.878	0.809
6	0.005	0.859	0.800
7	0.01	0.838	0.797
4	0.01	0.893	0.832

  On Mutagenicity best parameters found were: m = 4 and s = 0.01.
  Similarly On NCI-H23 best parameters found were: m = 5 and s = 0.1.

- Finally we selected **m = 4 and s = 0.01.**

§3. Performance on Different Datasets

We tested on various datasets of comparable sizes to NCI-H23 and having 2 classes picked up from the **TUDataset** .

Dataset	No. of Graphs	Train ROC	Test ROC
NCI-H23	41472	0.944	0.869
OVCAR-8	40516	0.945	0.869
P388	41472	0.917	0.855
NCI-H23H	40353	0.951	0.850
Mutagenicity	4337	0.893	0.832

§4. Conversion to Feature vectors(*convert.py*)

- **Using Rustworkx:** The provided Python script is a graph feature extraction tool that utilizes the **rustworkx library** for efficient graph processing and parallel computing. We observer around **100x time improvement** using **rustworkx library** without compromising any correctness.
- We **used rustwokx** based on rust which signicantly improved the performance compared to networkx. We also used joblib.Parallel for parallel computing, distributing subgraph matching tasks across CPU cores.

Dataset	Rustworkx (in s)	Networkx (in s)
NCI-H23	100	2940
Mutagenicity	27	720

- It is designed to analyze input graphs and determine the presence of predefined subgraph patterns. The script reads graph data from an input file, validates its structure, and performs subgraph isomorphism checks to generate binary feature vectors representing the presence or absence of subgraphs in each graph.
- **Prechecks to perform subgraph isomorphisms:** We added some **pre-checks to reject the prospective subgraphs** based on matching number of nodes, edge labels and neighbouring edge and node labels. This **prevented costly subgraph isomorphism checks** between all pairs of graph and subgraph features thus decreasing the time required.
- To optimize performance, the script uses **joblib.Parallel** for parallel computing, distributing subgraph matching tasks across CPU cores.

§5. Classification(*classify.py*)

This script trains a Support Vector Machine (SVM) model to classify graphs based on extracted feature vectors and evaluates its performance using the macro-averaged ROC-AUC score. It takes feature vectors and corresponding labels for training and testing datasets as input.