



## Capstone Project Phase B

### DOLORES: Deep Contextualized Knowledge Graph Embeddings

23-1-R-14

#### **Students:**

Saar Keshet - 315683854 - [Saar.Keshet@braude.ac.il](mailto:Saar.Keshet@braude.ac.il)

Evgeny Vexler - 320843881 - [Evgeny.Vexler@braude.ac.il](mailto:Evgeny.Vexler@braude.ac.il)

#### **Project supervisors:**

Dr. Renata Avros

Prof. Zeev Volkovich

**GitHub:** <https://github.com/Saark07/Final-Project>

# Table of contents

<b>Abstract</b>	<b>3</b>
<b>1. Project Review</b>	<b>3</b>
1.1. Introduction	3
1.2. The Algorithm	4
1.3. Research Process	5
1.4. Results	5
1.5. Conclusion	8
<b>2. User Documentation</b>	<b>9</b>
2.1. User Guide	9
2.2. Operation Instructions	9
2.2.1. Training Process	9
2.2.2. GUI	12
2.2.3. Gephi	16
2.3. Maintenance Guide	17
<b>3. References</b>	<b>19</b>

# Abstract

Searching for academic papers is usually performed by matching given keywords with their appearance in the paper's title and abstract description. Although this method often does not provide the needed information. In this paper, we present and discuss a new method constructed to provide reliable results by filtering irrelevant information using missing link prediction in knowledge graphs using deep and context-dependent embeddings. The methodology employs a kind of statistical modeling aimed to recognize and describe the essential links in a network.

## 1. Project Review

### 1.1. Introduction

In phase A of the final project, we discussed a new type of knowledge graph embedding that is called DOLORES which is both deep and contextualized.

Following multiple unsuccessful attempts to implement DOLORES and thorough consultation with the project supervisors, it was concluded that a change in the implementation approach to Node2Vec was necessary.

In phase B, we decided to continue using Node2Vec, in Node2Vec, representations are learned from nodes in a graph and referred to as node embeddings. These embeddings capture the relationships between nodes in a graph and the structural properties of the graph.

Node2Vec generates random walks on the graph using a starting node and traverses to the next node using transition probabilities, these probabilities are designed to balance between exploring the local neighborhood of the node and exploring the further regions of the graph. After the random walks are generated, Node2Vec learns the model embeddings by taking a node as input and predicting the nodes that are most likely to be its neighbors within the graph, by doing so, Node2Vec learns to embed the nodes in a continuous vector space, thus capturing the relationships between nodes based on their occurrence patterns.

Node2Vec uses the link prediction algorithm to predict missing connections between nodes. We have taken a graph of related academic papers as input and discarded 30% of the edges between the nodes, and using the link prediction to predict missing connections we rebuilt the edges between the nodes based on the similarity of the academic papers. After rebuilding different edges within the graph, we could ascertain the important edges connecting academic papers, resulting in connections

which are undoubtedly necessary.

## 1.2. The Algorithm

The problem we discussed in phase A is the limitation of the discrete triples which does not allow them to infer similarities and potential relations between entities that might be missing in the knowledge graph.

The goal of Node2Vec is to learn continuous representations of entities in a graph. Knowledge graph embedding aims to represent entities and relations in a knowledge graph, Node2Vec focuses specifically on learning embedding for nodes in a graph.

Following this, a popular alternative is to learn dense continuous representations of entities and relations by embedding them in latent continuous vector spaces, used to model the structure of the knowledge graph.

In Node2Vec [5], we use a two-step process which involves generating random walks in the graph and then learning the node embeddings using the Skip-gram [6] model, in the former, given an input graph, Node2Vec assigns parameters to each node, these parameters are used to control the exploration and the exploitation of the graph during the random walks. In the latter, Node2Vec performs random walks to explore the graph. A random walk starts with a starting node and iteratively moves to one of the node's neighbors. Node2Vec uses a random walk strategy that balances between two types of node transitions: breadth-first search (BFS) and depth-first search (DFS).

After the random walks are generated, Node2Vec utilizes the Skip-gram model to learn node embeddings. For each node in the sequence (target node), Node2Vec uses the nodes that appear near that node to train the Skip-gram model to maximize the probability of correctly predicting the context nodes given the target node. When the learning process is completed Node2Vec extracts the final node embeddings from the learned parameters of the Skip-gram model. Once the node embeddings are obtained, they can be used for link prediction of missing nodes in the graph.

Once the node embeddings are obtained, they can be used for missing link prediction in the graph, by using cosine similarity [7] to calculate the probability of a node being related to the target node, it can be determined what nodes are connected to the target node.

To use cosine similarity for missing link prediction, the node embeddings generated from Node2Vec were taken. Afterwards, using the corresponding embeddings for each node, the similarity between them could be calculated by using the cosine

similarity. In it, the similarity is measured by the cosine of the angle between the two vectors, which ranges from -1 to 1, where -1 is dissimilar and 1 is completely similar. After determining the value of the similarity between two vectors (nodes), we can infer the likelihood of a link between the two nodes. Using a threshold of 0.6 we inferred that cosine similarity between two nodes that is larger than the threshold indicates that a link is highly possible between the two nodes.

## **1.3. Research Process**

To build the knowledge graph, we used the CORA dataset for academic papers. CORA is a free dataset that contains a large quantity of academic papers and relations between them. We researched the Node2Vec algorithm to better understand the way it works, we also learned about the cosine similarity link prediction algorithm to help us predict the similarity between nodes and in that way we could reconstruct the edges between the nodes that we removed in our implementation.

We decided to choose Python as our implementation language of the code because there are many libraries containing useful functions that were needed for our implementation of the missing link prediction of the nodes in the graph using Node2Vec and cosine similarity.

## **1.4. Results**

In our testing of the missing link prediction using Node2Vec, we removed 30% of the nodes from the graph and checked how many edges between a set of two nodes were successfully reconstructed using the cosine similarity.



Fig 1: Partial graph visualization of nodes and edges of CORA dataset.

After removing 30% of the edges from the graph, we used the cosine similarity on the nodes of the graph to predict any possible missing links found between two nodes, the threshold which we picked was 0.6 (60%) similarity between two nodes, if the cosine similarity was higher than the threshold, we concluded that a relation between those nodes is important and should exist, thus, using we inferred that there was a missing link between these nodes.

Following that, we started the second iteration which performs similarly, at the end of each iteration, we counted the number of reconstructed edges, meaning the edges between the nodes of the graph that had a cosine similarity of 0.6 or more, indicating these are important edges and the relations between these nodes are vital.

Our results show the reconstructed amount of each edge found within the graph between two nodes, and the number indicating the times that they were successfully reconstructed, in Fig 2, the blue bars show the top 10% of the reconstructed edges which indicates the most important edges within the graph, while the red bars show the bottom 10% of the reconstructed edges which indicates the un-important edges found within the graph.

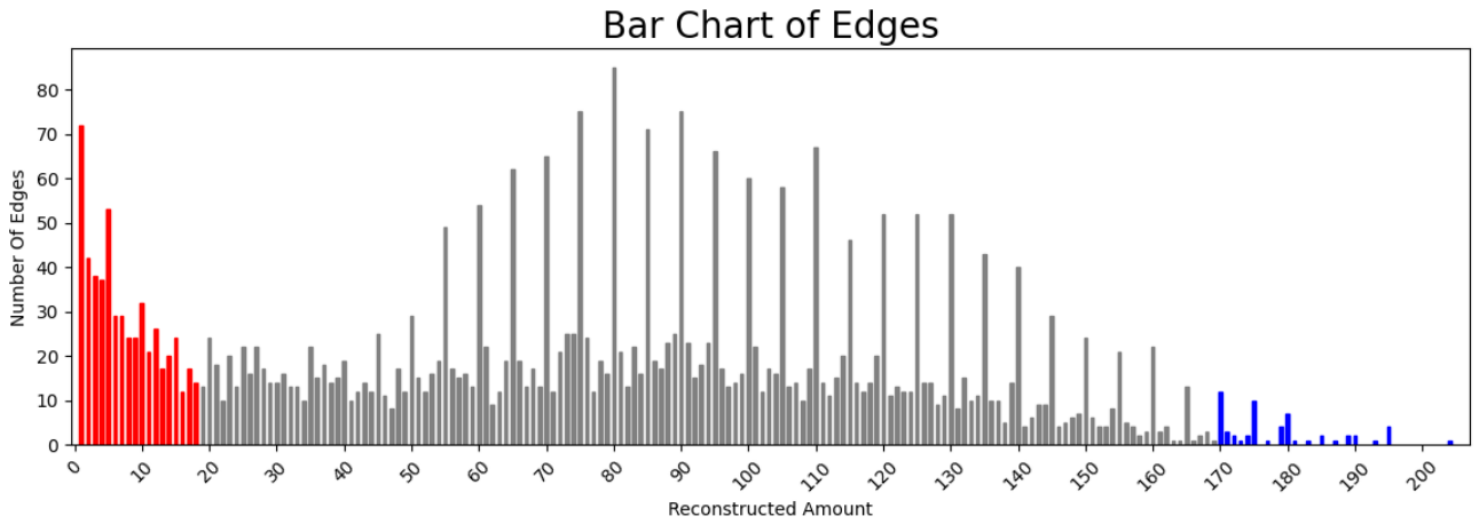


Fig 2: Bar chart of reconstructed edges.  
 The x-axis denotes the reconstructed amount of edges between two nodes.  
 The y-axis denotes the number of edges.  
 For example, approximately 30 edges were reconstructed 50 times.

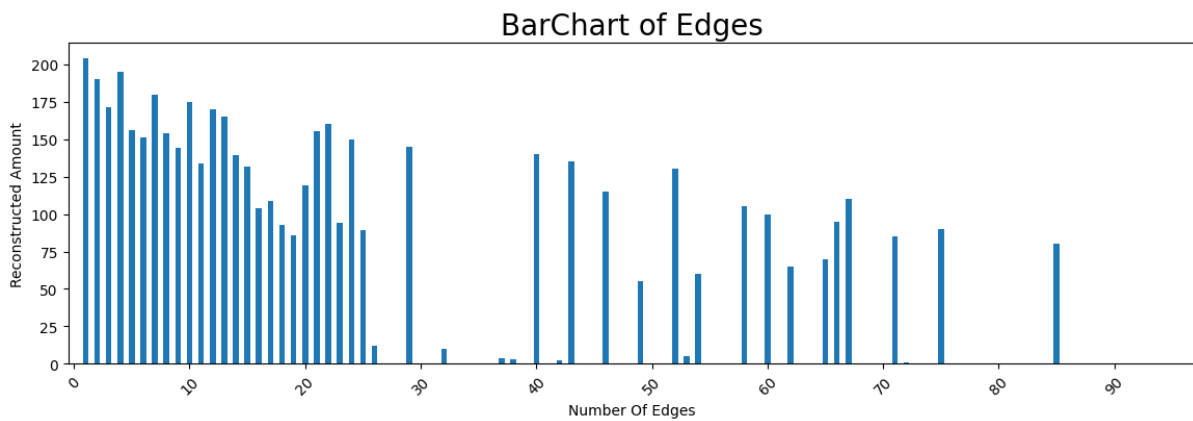


Fig 3: Bar chart of reconstructed edges of the number and edges on the x-axis and their reconstruction amount on the y-axis.

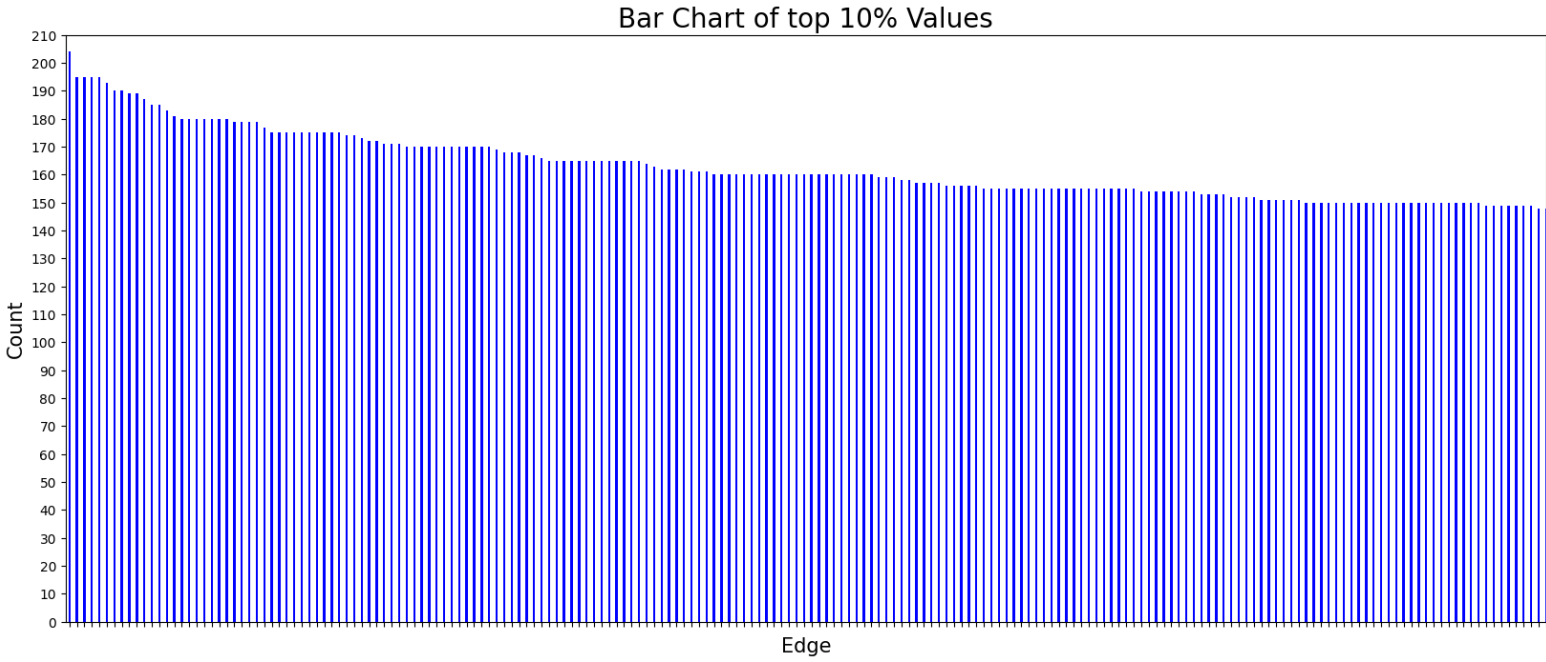


Fig 4:Bar chart of the top 10% values.

Each bar here represents an edge between two nodes.

Example: the first bar is an edge between two nodes that has been reconstructed 204 times.

Lastly, we picked the top 10% of the edges within the graph as seen in Fig 4, that had the largest number of reconstructed amounts and concluded that these edges were of the most importance, when looking at the entire graph. The following chart describes the values of the times the edges have been reconstructed, and the edges themselves.

## 1.5. Conclusion

In conclusion, our software visualizes a graph built from a collection of academic papers, the main goal of the software is to conclude which relations found between academic papers within the graph are of the most importance, indicating that they are absolutely relevant. In this way, we can find missing citations from academic papers that are relevant to the current paper we are reading, thus giving us more relevant resources when we are studying a certain academic paper and subject. Compared with other state-of-the-art link prediction methods, using Node2Vec has given the best results when trying to form the missing links between nodes.



## 2. User Documentation

### 2.1. User Guide

The “Searching for academic papers” software is intended to find the cosine similarity between nodes which denote the academic papers, by using the missing link prediction.

The software is loading the data that was created during the training process and using this data, the search for academic papers is made and the relevant results are shown.

The software that is loading the graph is Gephi, this shows a visual representation of the graph.

The algorithm is implemented in Python using the software PyCharm.

The graph is made using the library networkx and for the GUI part is implemented using the Qt Designer app and using the library PyQt5 to display the UI files.

### 2.2. Operation Instructions

#### 2.2.1. Training Process

**\*Note that the training process with Collab Pro might take approximately 7 hours.**

First, the training operation is done using the Collab Notebook that is attached to the submission.

The code block below is used to install the modules that will be used during the training:

```
pip install networkx node2vec

Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (3.1)
Collecting node2vec
  Downloading node2vec-0.4.6-py3-none-any.whl (7.0 kB)
Requirement already satisfied: gensim<5.0.0,>=4.1.2 in /usr/local/lib/python3.10/dist-packages (from node2vec) (4.3.1)
Requirement already satisfied: joblib<2.0.0,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from node2vec) (1.2.0)
Collecting networkx
  Downloading networkx-2.8.8-py3-none-any.whl (2.0 MB)
    2.0/2.0 MB 22.6 MB/s eta 0:00:00
Requirement already satisfied: numpy<2.0.0,>=1.19.5 in /usr/local/lib/python3.10/dist-packages (from networkx) (1.22.4)
Requirement already satisfied: tqdm<5.0.0,>=4.55.1 in /usr/local/lib/python3.10/dist-packages (from networkx) (4.65.0)
Requirement already satisfied: scipy<=1.7.0 in /usr/local/lib/python3.10/dist-packages (from gensim<5.0.0,>=4.1.2->node2vec) (1.10.1)
Requirement already satisfied: smart-open<=1.8.1 in /usr/local/lib/python3.10/dist-packages (from gensim<5.0.0,>=4.1.2->node2vec) (6.3.0)
Installing collected packages: networkx, node2vec
  Attempting uninstall: networkx
    Found existing installation: networkx 3.1
    Uninstalling networkx-3.1:
      Successfully uninstalled networkx-3.1
  Successfully installed networkx-2.8.8 node2vec-0.4.6

[ ] from google.colab import drive
   drive.mount('/content/drive')
```

This code block is used to load the dataset, and validate the path of the file cora.cites or any other dataset:

```
import random
import networkx as nx
import numpy as np
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from node2vec import Node2Vec
from operator import itemgetter

# Load the CORA dataset and create the graph
edgelist_file = '/content/drive/MyDrive/Colab Notebooks/cora.cites'
g = nx.read_edgelist(edgelist_file, nodetype=int)

# Print the number of nodes and edges before removal
num_nodes_before = g.number_of_nodes()
num_edges_before = g.number_of_edges()
print("Number of nodes before removal:", num_nodes_before)
print("Number of edges before removal:", num_edges_before)

# Define the fraction of edges to remove and the random seed
frac_edges_to_remove = 0.3
random_seed = 42
num_iterations = 500

# Get the list of edges to remove
num_edges_to_remove = int(frac_edges_to_remove * g.number_of_edges())
# Initialize a dictionary to count the number of times each edge has been successfully reconstructed
reconstruction_counts = {edge: 0 for edge in g.edges()}
```

The number of iterations is 500, the user can change the value to the number of iterations that is suitable for him, as well as the percentage of nodes that would be removed in each iteration from the graph, in the code above the value is 0.3 (30%).

This code block is the creation of the graph [Fig 1](#), validate that the path is correct:

```
[ ] # Save the graph as a GraphML file
output_file = "/content/drive/MyDrive/Colab Notebooks/cora_graph.gexf"
nx.write_gexf(g, output_file)
print("Graph saved successfully as", output_file)
```

This code block is the training part:

```
for iteration in range(num_iterations):
    print("Iteration:", iteration, "\n")
    # Get the list of edges to remove
    num_edges_to_remove = int(frac_edges_to_remove * g.number_of_edges())
    random.seed(iteration) # Set the random seed based on the iteration
    edges_to_remove = random.sample(g.edges(), num_edges_to_remove)

    # Create a new graph with the removed edges
    g_removed = g.copy()
    g_removed.remove_edges_from(edges_to_remove)

    # Train the node2vec model on the graph with removed edges
    node2vec = Node2Vec(g_removed, dimensions=64, walk_length=30, num_walks=200, workers=4)
    model = node2vec.fit(window=2, min_count=1)

    # Predict the removed edges
    for edge in edges_to_remove:
        try:
            # Get the node embeddings for the source and target nodes of the edge
            src_emb = model.wv[str(edge[0])]
            tgt_emb = model.wv[str(edge[1])]

            # Use cosine similarity to predict whether the edge should exist or not
            score = np.dot(src_emb, tgt_emb) / (np.linalg.norm(src_emb) * np.linalg.norm(tgt_emb))
            #print("Edge: ", edge)
            #print("Score: ", score)

            # If the score is above a threshold, consider the edge successfully reconstructed
            if score > 0.6:
                if edge in reconstruction_counts:
                    reconstruction_counts[edge] += 1
                else:
                    reconstruction_counts[edge] = 1
            except KeyError:
                # If one of the nodes is not in the vocabulary, skip this edge
                pass

    # Print the number of times each edge has been successfully reconstructed
    print("Reconstruction Counts:")
    edges_count=0
    for edge, count in reconstruction_counts.items():
        print(f"Edge: {edge} Count: {count}")
        edges_count+=1
```

This code block saves the result into a CSV file, validate the path of the saved file:

```
file_path = '/content/drive/MyDrive/Colab Notebooks/reconstruction_counts.csv'

with open(file_path, 'w', newline='') as csv_file:
    writer = csv.writer(csv_file)
    writer.writerow(['Edge', 'Count']) # Write the header row

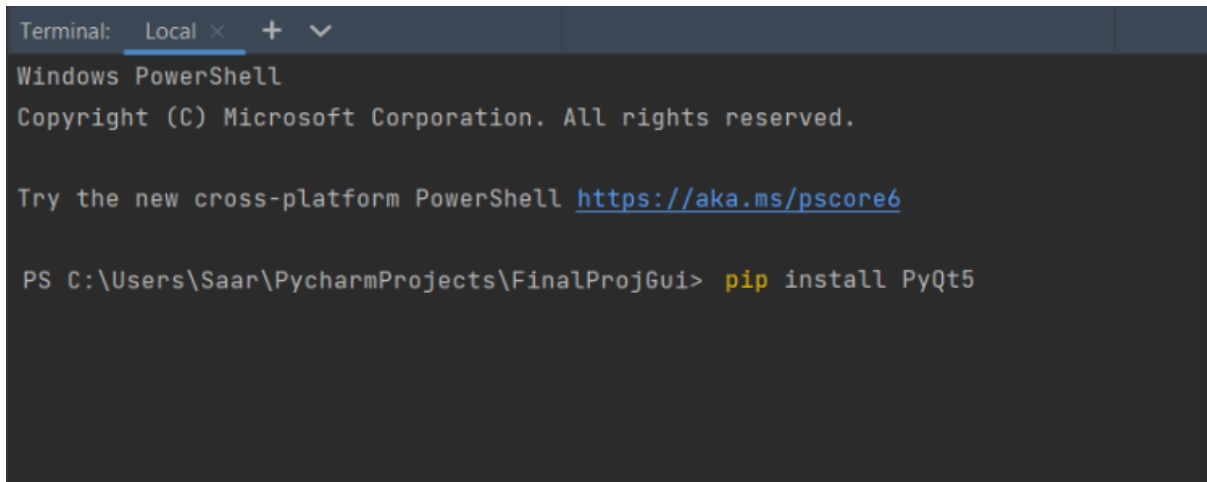
    for key, value in reconstruction_counts.items():
        writer.writerow([key, value]) # Write each row
```

The results are stored in reconstruction\_counts.csv.

### 2.2.2. GUI

To run the software it is recommended to use PyCharm.

You need to install PyQt5 from the terminal the installation is very simple all needs to do is to run the command: `pip install PyQt5` from the terminal:

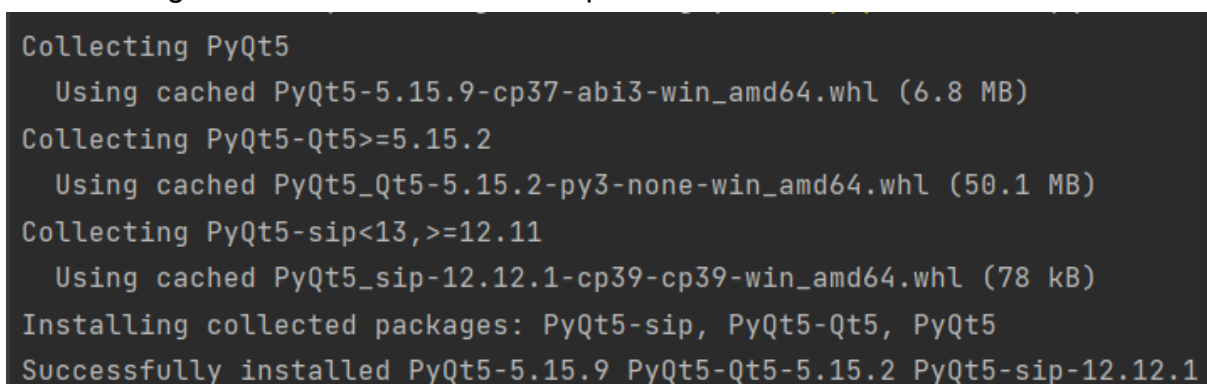


```
Terminal: Local x + v
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

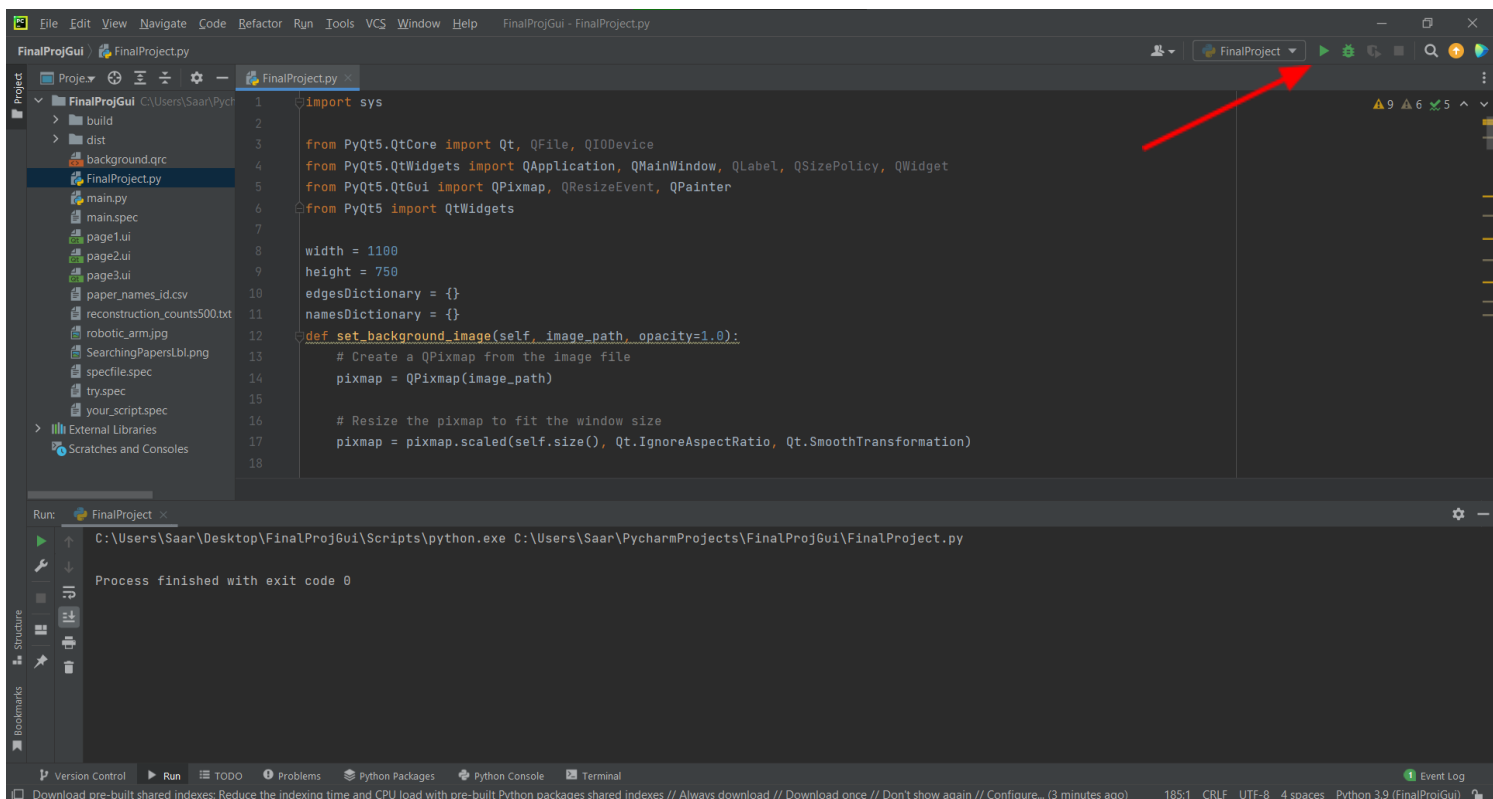
PS C:\Users\Saar\PycharmProjects\FinalProj6Gui> pip install PyQt5
```

After running the command this is the output that should be:

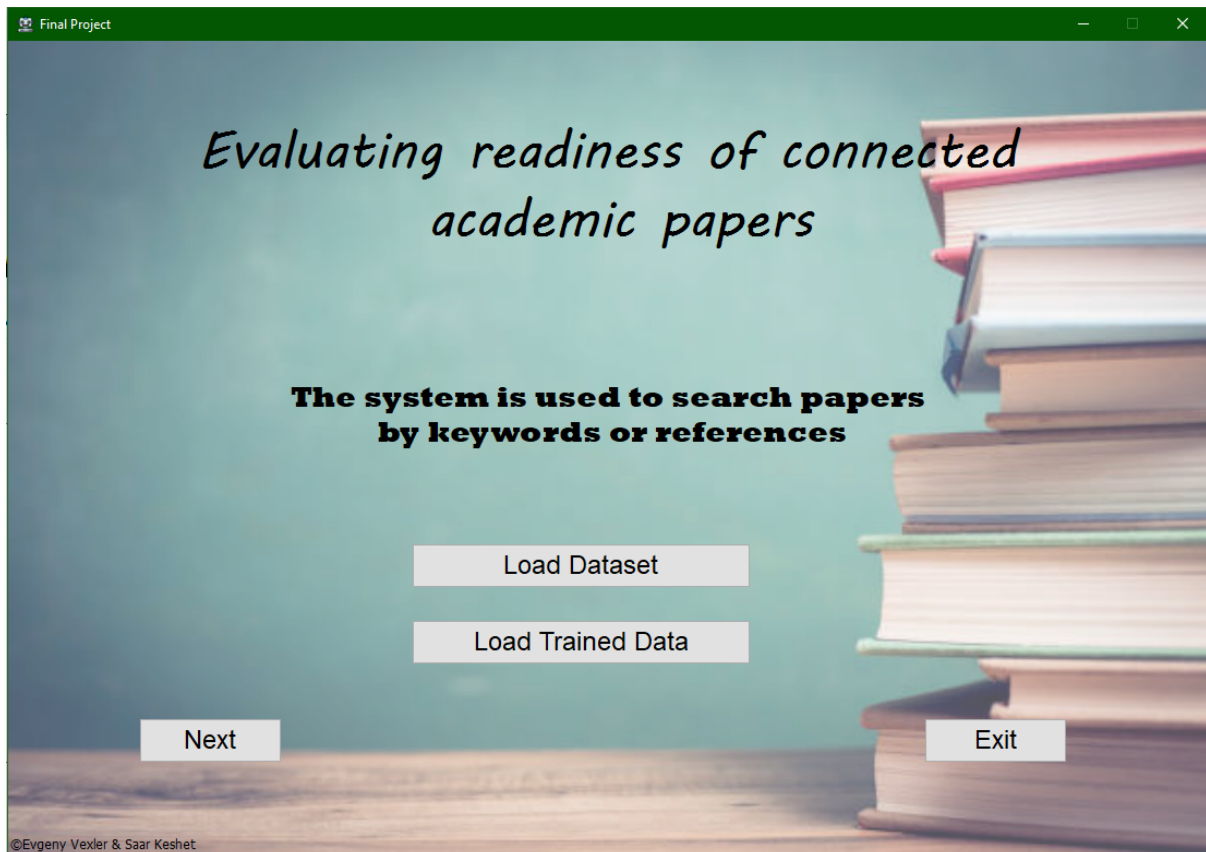


```
Collecting PyQt5
  Using cached PyQt5-5.15.9-cp37-abi3-win_amd64.whl (6.8 MB)
Collecting PyQt5-Qt5>=5.15.2
  Using cached PyQt5_Qt5-5.15.2-py3-none-win_amd64.whl (50.1 MB)
Collecting PyQt5-sip<13,>=12.11
  Using cached PyQt5_sip-12.12.1-cp39-cp39-win_amd64.whl (78 kB)
Installing collected packages: PyQt5-sip, PyQt5-Qt5, PyQt5
Successfully installed PyQt5-5.15.9 PyQt5-Qt5-5.15.2 PyQt5-sip-12.12.1
```

Now the user can start and run the GUI:



Clicking on the 'Run' will open our main window of the GUI:



Clicking on 'Load Dataset' will open a popup window and then select the desired dataset csv, the csv should be in this format:

ID	Title
35	Genetic Algorithms in Search, Optimization and Machine Learning.
40	Dynamic control of genetic algorithms using fuzzy logic techniques.
114	Learning to Act using Real- Time Dynamic Programming.
117	Dynamic Programming and Markov Processes.
128	Reinforcement Learning Algorithms for Average-Payoff Markovian Decision Processes.
130	Ok. Scaling up average reward reinforcement learning by approx-imating the domain models and the value function.
164	Learning polynomials with queries: The highly noisy case.
288	Memory-based Stochastic Optimization,
424	A hybrid nearest-neighbor and nearest-hyperrectangle algorithm.
434	Learning Analytically and Inductively.
463	"Abstraction and Decomposition in Hill-climbing Design Optimization".
504	Constructive belief and rational representation.
506	Rationality and its Roles in Reasoning (extended version),
887	Some studies in machine learning using the game of Checkers.
906	Regularization thory and neural networks architectures.
910	A Theory of Networks for Approximation and Learning,
936	The Use of Explicit Goals for Knowledge to Guide Inference and Learning.

The first column is the ID the second is the Title\Name of the paper.

Clicking on 'Load Trained Data' also open a popup window, select a csv file in this format:

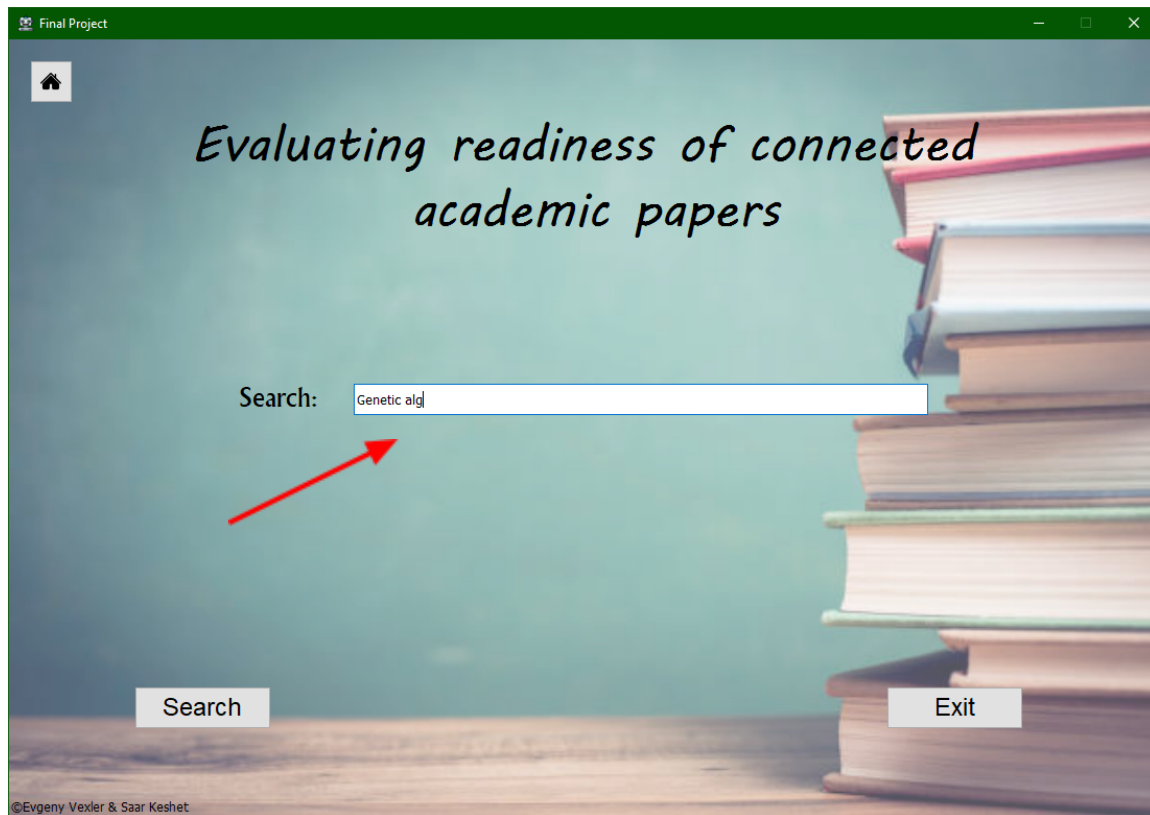
	A	B
1	Edge	Count
2	(35, 1033)	5
3	(35, 103482)	8
4	(35, 103515)	11
5	(35, 1050679)	5
6	(35, 1103960)	7
7	(35, 1103985)	44
8	(35, 1109199)	20
9	(35, 1112911)	10
10	(35, 1113438)	0
11	(35, 1113831)	36
12	(35, 1114331)	22
13	(35, 1117476)	14
14	(35, 1119505)	0
15	(35, 1119708)	59
16	(35, 1120431)	0
17	(35, 1123756)	14

The first column represents the edge between 2 nodes the second column represents how many times this edge has been reconstructed.

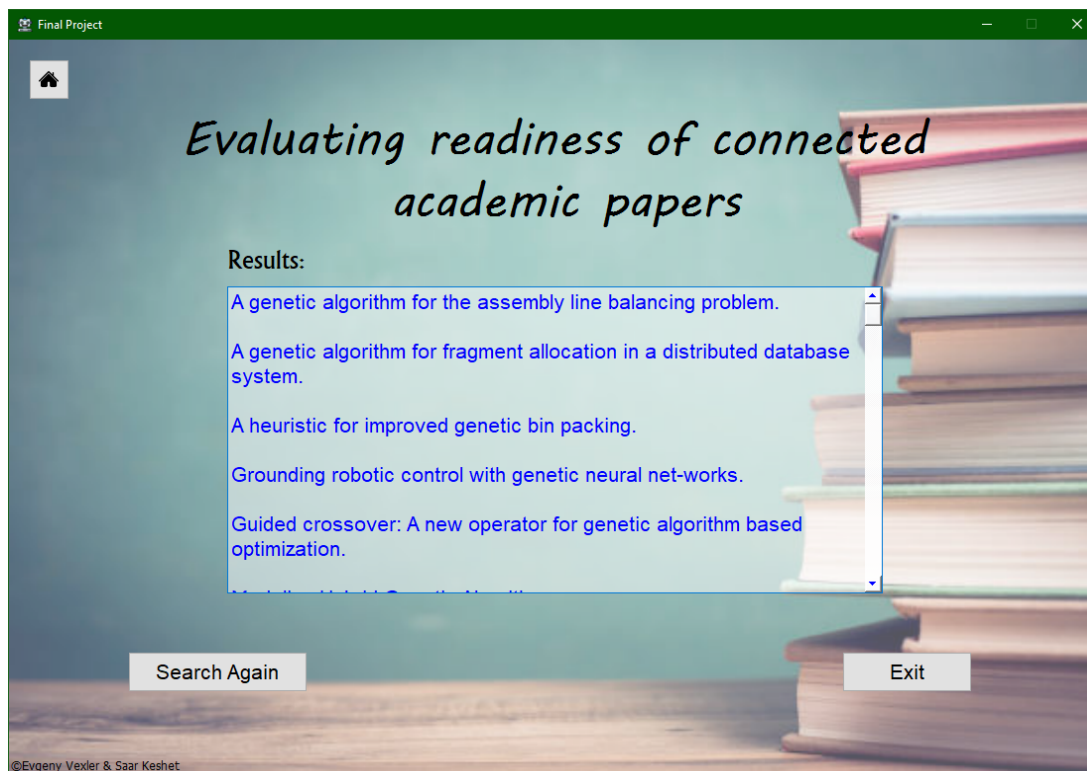
**\*Note: the csv files that are used for this project are attached in the csv\_files folder.**

Clicking on Next will move to the next screen and here we can enter the text to search:

**\*Note: this might take a few seconds to load the data (depending on the data size)**



And after clicking the 'Search' button we get the results:

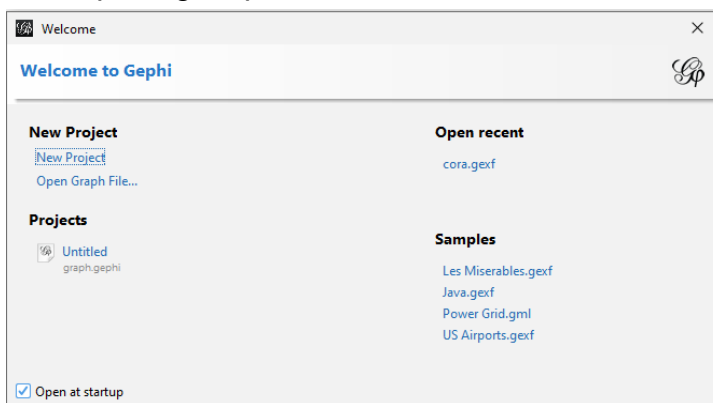


Clicking ‘Search Again’ will transfer the user to the second page.  
And in each page, there is a Home button to return to the first page and load new data.

## 2.2.3. Gephi

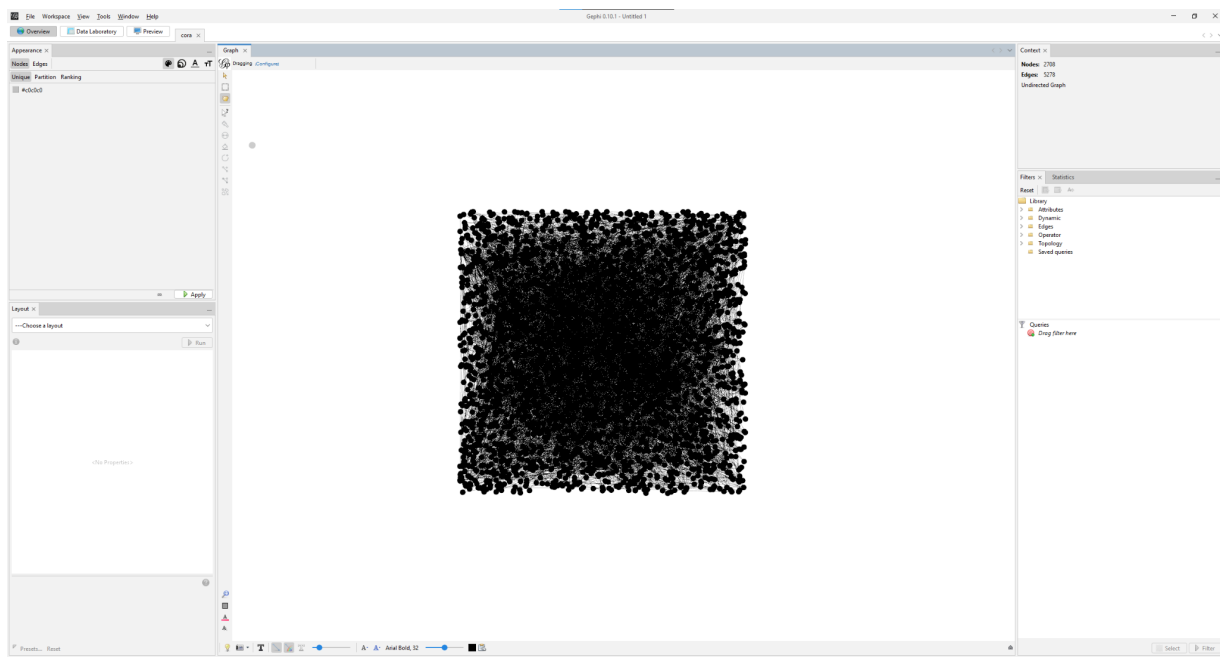
Gephi is the leading visualization and exploration software for all kinds of graphs and networks. Gephi is open-source and free.

After opening Gephi, the user is welcomed with the following screen:



And select the “Open Graph File...” option and open the \*.gexf file which was created during the training process, e.g. cora.gexf.

After loading the graph file, a visual representation of the edges and nodes of the graph can be viewed in Gephi:





By hovering with the mouse button on the nodes, the nodes that are connected to each node can be viewed, as well as other options such as: showing the names of the nodes, different visual representations of the graph, etc.

## 2.3. Maintenance Guide

The library networkx's node2vec module is used to implement the Node2Vec algorithm during the software development and training. The dataset file needs to be written in the following format to be compatible with the algorithm:

ID	Title					
35	Genetic Algorithms in Search, Optimization and Machine Learning.					
40	Dynamic control of genetic algorithms using fuzzy logic techniques.					

The dataset file for the training process needs to be in this format:

35	1033
35	103482
35	103515
35	1050679
35	1103960

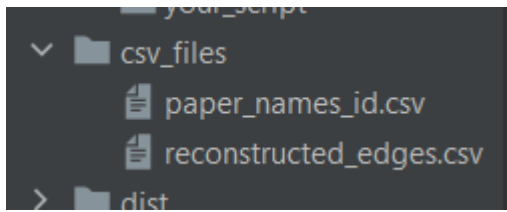
After training the dataset with the model, a Trained file is obtained, which also needs to follow a specific format e.g:

Edge	Count
(35, 1033)	5
(35, 103482)	8

This format ensures that the trained data can be easily used in subsequent processes and applications.

For visual representation of the trained data, Gephi, an open-source and free software, is employed. The graph file obtained prior to the training process is loaded into Gephi, enabling the visualization of nodes and their connections in the form of a graph. Gephi provides an intuitive interface and a range of tools to explore and interpret the relationships between nodes.

The GUI uses QT Designer app for the UI files and also uses the library PyQt5 if there is a need to use different csv files then add them to the project in the “csv\_files” folder:



### 3. References

1. Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, Luke Zettlemoyer, "Deep contextualized word representations"
2. Matthew E. Peters, Waleed Ammar, Chandra Bhagavatula, Russell Power, "Semi-supervised sequence tagging with bidirectional language models"
3. Hanwen Liu, Huaizhen Kou, Chao Yan, Lianyong Qi, "Link prediction in paper citation network to construct paper correlation graph"
4. <https://www.connectedpapers.com/main/cab46caf83a9e0390c6ca4d8603187969c9a53ad/DOLORES%3A-Deep-Contextualized-Knowledge-Graph-Embeddings/graph>
5. A. Grover and J. Leskovec, "node2vec: Scalable Feature Learning for Networks," in Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16), San Francisco, CA, USA, 2016, pp. 855-864.
6. Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space.
7. Manning, C. D., Raghavan, P., & Schütze, H. (2008). Introduction to Information Retrieval. Cambridge University Press. Chapter 6: "Similarity-based retrieval."