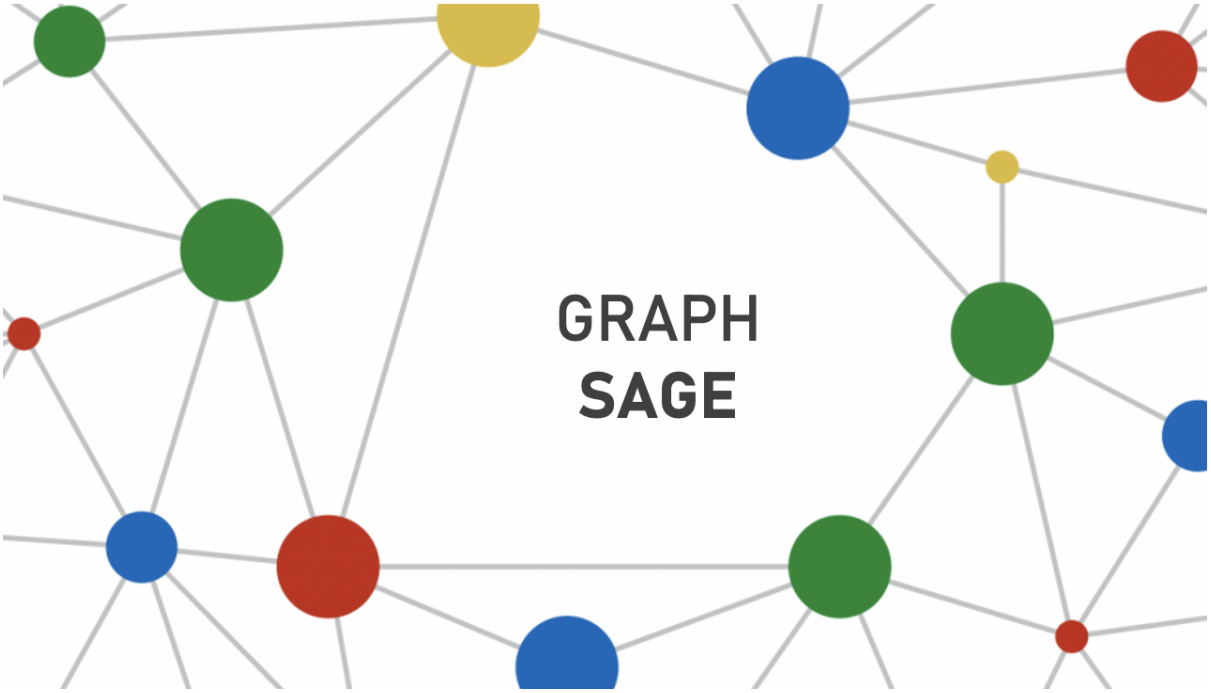
ESILV
ENGINEERING SCHOOL
DE VINCI PARIS

Scientific paper explanation

Inductive Representation Learning on Large Graphs



DATA & AI - Graphs and mining

BEN SOUNA Nacima

BENJIRA Wissal

HAOUAS Lies

FADILI Yanis

Professors

TRAVERS Nicolas

OULED DLALA Imen

A5 - DIA2

Table of content

1. Resume of the article

The scientific paper we study deals with GraphSAGE or inductive learning on large graphs. It was authored by William L.Hamilton, Rex Ying and Jure Leskovec from the department of computer science at Stanford University and it was published at the 31st conference in neural information processing systems in 2017.

Today, the structure of graphs has allowed us to make quantum leaps in the performance of data classification tasks, certainly because of the efficiency with which graphs seem to encode the set of data, in it's topology. Today, despite the many advantages of using graphs in the field of machine learning, there is still a problem. Indeed, there is an issue in the way of coding the data so that the weight matrices have a rather easy convergence time. This is how GraphSAGE was introduced, a solution that seeks to solve these problems.

This solution is still related to node embedding approaches, supervised and unsupervised learning and recent advances in the application of convolutional neural networks on graphs. The idea behind this method is to learn how to aggregate information about the characteristics of a node neighborhood. The theoretical analysis given in this article explains how this approach can understand the graph structure really well.

After the explanation of how GraphSAGE algorithm works, we can apply it to some use cases. Thus the performance of GraphSAGE has been evaluated on three reference tasks by making predictions on nodes that have not been seen during the model training. Therefore, for all the datasets, different models are implemented : from basic classifiers to GraphSAGE algorithms combined with various aggregators. To evaluate these methods, the prediction score, the inference time and the neighborhood size are compared.

2. Explanations

2.1 Introduction

The concept of node embedding approaches is to apply dimensional reduction techniques to transform high-degree data neighboring nodes into a compact embedding vector. These node embeddings are then used for machine learning programs and for operations such as classification, clustering and relationship prediction. To add a new node in a dynamic graph, several models exist, such as Graph Convolutional Network (GCN) or DeepWalk. However, these methods remain expensive in terms of computation time. To tackle this issue and to be able to obtain an embedding more easily, a new method is proposed called GraphSAGE (SAmple and aggreGatE). This method uses the characteristics of the nodes to learn an aggregation function. GraphSAGE is able to learn structural information about the role of a node in a graph, despite the fact that it is basically feature-based.

For GCNs, the problem is that they have some flaws that limit their practicality. Moreover the architecture does not really allow for a batch model to be trained or the network would converge extremely slowly or not at all. This means that there are real practical limits on the size of an input graph. DeepWalk also needs the Laplacian matrix for the entire graph at the time of training the model. This implies that all test and validation nodes must be present from the beginning which practically rules out the method. Graph sage is a solution that attempts to solve these problems while retaining the benefits of working with graphs in machine learning.

2.2 GraphSAGE motivation

GraphSAGE is an inductive version of GCNs, it does not need the entire graph structure for learning, and it can generalize well to unseen nodes. It is a branch of graph neural networks that learns node representations by sampling and aggregating neighbors from multiple search depths. This algorithm is capable of predicting embedding of a new node, without requiring a re-training procedure. To do so, GraphSAGE learns aggregator functions that can induce the embedding of a new node given its features and neighborhood. This is called inductive learning.

Besides, GraphSAGE has many advantages, including sampling the graph. Indeed, to train the model, the graph can be trained in small batches or small portions at a time and the representations formed can be generalized to new inputs. Indeed, after training, GraphSAGE can

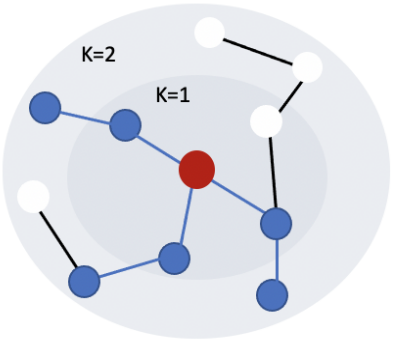
be used to embed new nodes without retraining. To do this, these new graphs must have the same attribute scheme as the example data that was used to train the graph.

One interesting advantage of GraphSAGE is that we can train our model on one subset of the graph and apply this model on another subset of this graph thanks to parameter sharing. So when a new architecture appears, we are able to borrow the parameters, do a forward pass, and consequently get our prediction.

2.3 Sample and aggregation

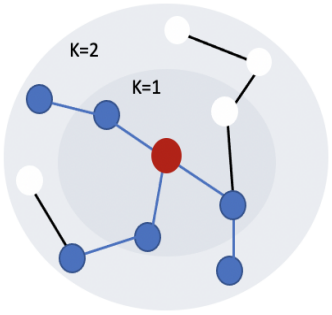
Let's take a closer look at how graph edges work. The basic idea of graph edges comes from the concept of sampling a set of neighborhoods of a node to learn its properties rather than working on the whole graph in a single effort.

What we call a neighborhood is defined by hyperparameters where K, a defined value, represents the number of steps or traversals along the edges of the node from a given target node, here marked in red.The first step of K equal to 1 allows us to have the nodes adjacent to the target node : here as we can see on the image it is the three blue nodes in the first layer K=1. Then we have a further step of K equal to 2 which gives the final neighborhood of the blue nodes in layer 1.

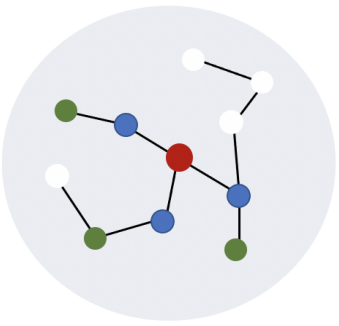


Now, once this neighborhood is defined we would like to gather all the data provided by the selected nodes and combine them in a useful way in a step called aggregation, working backwards from K equals 2.

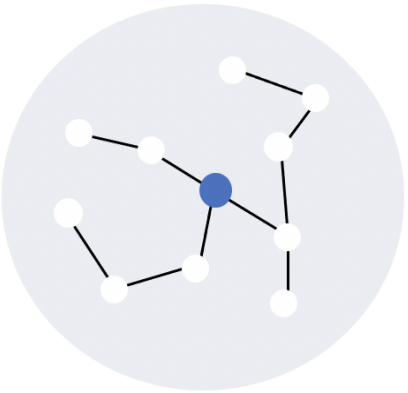
Sample



Aggregate

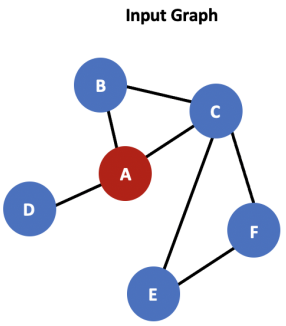


This time the feature vectors of the green nodes will be combined through some aggregation function.
In the next step we take these embeddings and perform an aggregation again by producing an embedding of the target node. With this a classification of its label or a prediction of its context can be done.



When we untangle such a neighborhood from a target node, we have a kind of tree structure. We can think of K-values as a type of layer in the model, but we should not confuse them with the notion of depth or layers in a neural network. The K values we talk about here refer only to the number of steps we want to propagate in the graph during the process of sampling a neighborhood.

Now that we have a first idea of the Graph Sage method, let's see how the aggregation of neighbors works with a simple case of linear graphs.



Here the node to predict is in red, it is the node A. We have 3 adjacent nodes (in its layer k=1) : B, C and D and 2 nodes with a step of 2 (is the layer k=2) : E and F. For each of the neighbors of the node to predict we will randomly assign them a weight. Let's assume that we have : A: 1.1; B: 2.6 ; C:4.7; D:3.6 ; E:1.8 and F:3.2.

At each step we will update the weights of the nodes using an aggregate function. Let's take for example the mean function. Here node A is linked to nodes B, C and D so we calculate the average of these 4 weights and the result will be assigned to node A. For node C, this one is linked to nodes A, B, E and F: we will then calculate the average between the weights assigned to the 5 nodes.

For node C we have : $(1.1 + 2.6 + 4.7 + 1.8 + 3.2)/5 = 2.68$. The weight of node C is now 2.68

On the same principle as the example above we update the weight of each node in a first step. Here are the new weights obtained for each node :

A: 3.0 ; B: 2.8 ; C: 2.68 ; D: 2.35 ; E: 3.23 et F: 3.23

After calculating the new weights, we get a new graph. We will repeat the same operation, calculating the average for each node. Indeed here we are interested in two layers: the neighbors and the neighbors of the neighbors of the node to predict. By computing the average for each node with its neighbors we have generated the integration of each node. Now we have to use these new integrations to generate the new integrations of the neighbors of neighbors. In the case of the average as an aggregation function we will simply recompute the average of each node with its neighbors but this time with the new weights.

Now for example for node C we have : $(3.0 + 2.8 + 2.68 + 3.23 + 3.23)/5 = 2.988$. The weight of node C is now 2.988.

After updating the weight of each node we have :

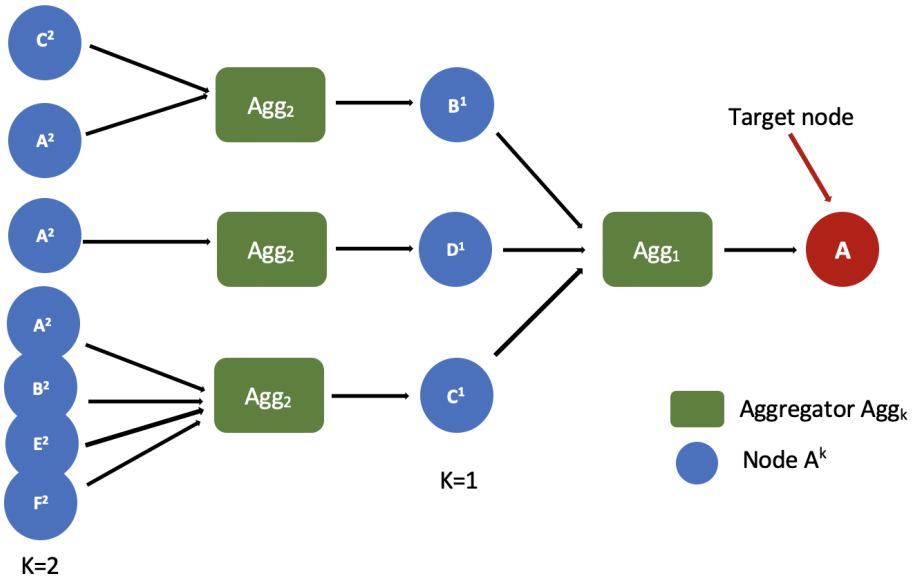
A: 2.71 ; B: 2.83 ; C: 2.988 ; D: 2.35 ; E: 3.05 et F: 3.05

Indirectly in the calculation of the new weight of A for layer 2, we use the nodes E and F which are not adjacent nodes to node A. Indeed, to calculate the new weight of A we use the weight of node C which itself was calculated from the weight of E and F. Thus, even if its nodes are not directly adjacent to node A they have an influence on its weight.

To summarize, in this example we have repeated the operation twice. All the neighbors of the node to be predicted (nodes B, C and D) and the neighbors of these neighbors (nodes E and F) have an influence on the node A to be predicted. In this case the farthest neighbors of the node to be predicted have a distance of 2 edges. So more generally, if we take a distance of X edges from the node to be predicted, then we will repeat the operation X times to generate new integrations. Here all the nodes that will have a distance inferior or equal to X will have an influence on the node to predict.

This method allows embeddings of nodes, but is not sufficient to generate new embeddings for new nodes added to the graph. Therefore the aggregation function used to generate new integrations will be replaced by several functions: each one for a layer. So if we have K layers of neighbors, we have K aggregation function.

To summarize this method, let's make a diagram :



In this case, we want to predict node A. We are going to connect these neighboring nodes by aggregators: the aggregation functions schematized by the green rectangles.

Here we have two layers, so the farthest neighbors of the node to predict have a distance of 2 edges. For the nodes adjacent to the node to predict we note them with an index 1, corresponding to the layer k=1, for the neighbors of these new nodes, they will have an index 2 (for k=2) . Then as explained above, we have an aggregation function for each layer: So we have here two aggregation functions that we note Agg1 and Agg 2.

So more generally, for k layers, we have A(k) the value of the embedding of A after (k-1) layers and after computing (k-1) aggregator function. If we use for example the mean again as an aggregator function we get the following formula : A(k) = F(A(k-1),B(k-1),C(k-1),D(k-1))
To describe the set of neighbors of a node v let us denote by N(v) the set of neighbors of v. For example if we take node A, it has for neighbor nodes B; C and D. Thus N(A)=B,C,D for the layer k=1 and consequently N(A)(k)=B(k),C(k),D(k) for the layer k.

Using the above equation : A(k) = F(A(k-1),B(k-1),C(k-1),D(k-1)), we obtain with the new notation N(v) :

$$A(k) = F(\ A(k-1), N(A)(k-1))$$

That is the average of all the neighbors of A : so the nodes B,C and D which are N(A) and himself (that's why we have A(k-1) (himself) and N(A)(k-1) for his other adjacent neighbors). So with this equation we can compute the embedding value of A after generating (k-1) integrations with (k-1) aggregator function.

2.4 Pseudo code

Now that we understand how Graph SAGE works, let's take a closer look at how the "*GraphSage embedding generation algorithm*" works. But before that we are going to formalize the notations used above to explain them in order to facilitate the understanding of the algorithm.

Up to now we have been talking about layer k, we will keep this notation. We have mentioned above A(k): the embedding value of A after generating (k-1) integrations. From now on, we will use the notation h^k_a(A corresponds to the node and K the layer). For the k aggregation functions (Agg_k on the diagram), we will use the notation $AGGREGATE_K$. Our graph will be represented by the following notation: G($\mathcal{V}$,$\mathcal{E}$), with $\mathcal{V}$ the set of nodes as above, and $\mathcal{E}$ the set of edges. Now let's look at the pseudo code of the algorithm :

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
for $k = 1...K$ **do**
  for $v \in \mathcal{V}$ **do**
    $\mathbf{h}^k_{\mathcal{N}(v)} \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}^{k-1}_u, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}^k_v \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}^{k-1}_v, \mathbf{h}^k_{\mathcal{N}(v)})\right)$
  end
  $\mathbf{h}^k_v \leftarrow \mathbf{h}^k_v / \|\mathbf{h}^k_v\|_2, \forall v \in \mathcal{V}$
end
$\mathbf{z}_v \leftarrow \mathbf{h}^K_v, \forall v \in \mathcal{V}$

Let's have a look at the first lines of the code which correspond to the input and output :

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$
Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

Here we find in input the graph G($\mathcal{V}$,Ɛ) with $\mathcal{V}$ the number of nodes and Ɛ the number of edges. Then we have the initial node features for each node in $\mathcal{V}$, the depth K which corresponds to the maximum distance of neighbors from the node to predict. We have then the weight matrix W^k which contains the weights of each node, which will be trained and updated during the training, we have a non-linearity sigma that we will see after. We also find our aggregation functions $AGGREGATE_K$ for each layer k and finally the Neighborhood function N(v)=2^$\mathcal{V}$ (for exemple if we have 4 nodes the whole power cell would have 16 elements).

Output : The output of this algorithm will be the vector representation of the node

Now let's see what the first line of code does :

1 $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;

We start the algorithm by initialize initialize the embeddings of all the nodes with the node features

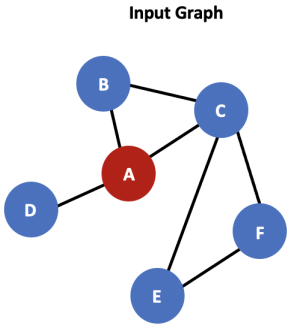
Next, we go through each of the k layers and for each one we create a neighborhood integration. To do this in each layer we use the aggregation function on each node and then we concatenate it with the existing integration of the node.

2 **for** $k = 1...K$ **do**
3 **for** $v \in \mathcal{V}$ **do**
4 $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
5 $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
6 **end**
7 $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
8 **end**

We have not yet mentioned concatenation in the explanation above, but it is part of the process. Let's quickly go back to the example seen before when we calculated the averages of each node with its adjacent neighbors. After averaging each node a second time, on the k=2 layer, we obtained the following weights:

A: 2.71 ; B: 2.83 ; C: 2.988 ; D: 2.35 ; E: 3.05 et F: 3.05

We notice that the nodes E and F have an equal weight in spite of their different weight in the initial graph.



In reality if we look at the graph, it is normal to have similar weights for its two nodes: E has for neighbor C and F and F has for neighbor E and C: The calculation of average is thus the same for these two nodes . However, this is disturbing in the learning of our model, we want to have differences in their embeddings, that's why we use concatenation.

In the case of this example, the aggregation function will not take the two nodes in its calculation but the neighbor and we will concatenate this vector with the vector of this node . Then the vectors will not have the same embedding after the aggregation.

After having obtained the aggregated representation we can go to the next step. This one consists in contacting or combining the obtained representation with the representation of the previous layer. The output obtained will then be transformed by multiplying it by a weight matrix Wk. And then, a non-linear activation function is used in order to learn and perform more complex tasks. This step is represented by the following line c $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$

The last step of the algorithm is the normalization which is applied on node representation. Indeed, it helps the algorithm to keep the general distribution of node embeddings. This step is computed as : $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$

Finally, the result is stocked in the variable of the vector representation $\mathbf{z}_v$.

Once we have these representations, one may wonder how to train the model.This question will be answered in the next part.

2.5 Learning the parameters

As explained previously, the strength of learning aggregator features to generate node embeddings, instead of learning the embeddings themselves, is inductivity. When the aggregator weights are learned, the embedding of an unseen node can be generated from its features and neighborhood.

As a result, aggregators avoid the need to retrain when new nodes are included in the graph.

In order to learn the weights of the aggregators and the embeddings, a metric is needed. The goal is to obtain a graph in which the neighboring nodes have similar embeddings and the unrelated ones have distant embedding vectors. To emphasize this, we have defined the following loss function:

$$J_{\mathcal{G}}(\mathbf{z}_u) = -\log\left(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)\right) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log\left(\sigma(-\mathbf{z}_u^\top \mathbf{z}_{v_n})\right)$$

In this case, u and v are two neighboring nodes. The loss function is therefore computed in an unsupervised way, for node u. First, the blue term maximizes the similarity of the 2 node embeddings. Because u and v are connected, we want their representations to be as close to each other. In fact, once they are really close, the product $\mathbf{z}_u^\top \mathbf{z}_v$ gets really big, thus the sigmoid function σ equals 1 and the log part equals 0. In sum, we are minimizing the loss function. Similarly, in the red zone, because of the minus, the log function gets also pulled up to 0. In a nutshell, on the one hand, the nodes that are close to each other are connected and have similar embeddings and representations and on the other hand the ones which are not connected in the graph have very different representations.

Furthermore, GraphSAGE could also be trained in a supervised manner. For example if we have a classification test, the representations are chosen, then we use the cross entropy and train the aggregation functions.

2.6 Type of Aggregators

2.6.1 Mean Aggregator

Mean aggregator computes the mean of the vectors. In other words, we can average embeddings of all nodes in the neighbourhood to construct the neighbourhood embedding.

$$\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W} \cdot \text{MEAN}(\{\mathbf{h}_v^{k-1}\} \cup \{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\}).$$

2.6.2 Pool Aggregator

Pool aggregator employs min or max functions

$$\text{AGGREGATE}_k^{\text{pool}} = \max(\{\sigma\left(\mathbf{W}_{\text{pool}}\mathbf{h}_{u_i}^k + \mathbf{b}\right), \forall u_i \in \mathcal{N}(v)\}),$$

2.6.3 LSTM Aggregator

LSTM stands for Long Short Term Memory which is a deep neural network. It is a cell composed of three calculation areas (“3 gates”), that regulate the flow of information (by performing specific actions) :

- A forget gate which will filter the information contained in the previous memory cell and decide by mathematical calculations the relevant values that can be transmitted to the next gate.
- An input gate that will decide which new incoming values from the LSTM will be allowed to pass. These values will be stored in the memory cell
- An output gate that will decide whether the content of the cell should influence the output of the neuron.

These operations allow the LSTM to retain or delete information that it has in memory. It retains important information and deletes that which can be forgotten by the network. To allow the LSTM network to learn, the inputs of each computation area, the gates, are weighted by weights linked to the gates and by a bias.

The gates of an LSTM are analog, in the form of sigmoids, which means that they go from 0 to 1. The fact that they are analog allows them to do a backpropagation with them. If the data multiplies by one, the value stays the same. On the other hand, if the data multiply by zero, the value becomes zero and disappears

2.7 Experiment

To better understand the working process of GraphSage in more detail, we will study its implementation on real datasets. Thus, in order to evaluate the performance of GraphSAGE, the 3 following databases are studied :

- **Citation** : classifying academic papers into different subjects from Web of Science citation using more than 300k nodes with an average degree of 9.15
- **Reddit** : classifying posts as belonging to different communities using more than 230k nodes with an average degree of 492
- **PPI** : classifying protein functions across various biological protein-protein interaction

In all these databases, we perform predictions on nodes that are not seen during training, and, in the case of the PPI dataset, we test on entirely unseen graphs. For doing so, different models have been used:

- Random classifier (Random)
- Logistic regression feature-based classifier (Raw features)
- DeepWalk algorithm (DeepWalk / DeepWalk + features)
- GraphSAGE (using various aggregator function : GCN/ mean/ LSTM/ pool)

As parameters of these methods, we set the weight k=2, the neighborhoods for the first layer S1= 25 and for the second layer S2 =10. Besides, the metric to sort prediction results is F1 score. Moreover, all models were implemented in TensorFlow with the Adam optimizer (except DeepWalk).

The goal of the experiment is to compare the performance of each model through supervised and unsupervised functions.

2.8 Result

2.8.1 F1 prediction scores

predict paper subject categories on a large citation dataset

predict which community different Reddit posts belong to

predict protein roles from gene ontology in various protein-protein interaction

	Citation		Reddit		PPI	
Name	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1
Random	0.206	0.206	0.043	0.042	0.396	0.396
Raw features	0.575	0.575	0.585	0.585	0.422	0.422
DeepWalk	0.565	0.565	0.324	0.324	—	—
DeepWalk + features	0.701	0.701	0.691	0.691	—	—
GraphSAGE-GCN	0.742	0.772	**0.908**	0.930	0.465	0.500
GraphSAGE-mean	0.778	0.820	0.897	0.950	0.486	0.598
GraphSAGE-LSTM	0.788	0.832	**0.907**	**0.954**	0.482	**0.612**
GraphSAGE-pool	**0.798**	**0.839**	0.892	0.948	**0.502**	0.600
% gain over feat.	39%	46%	55%	63%	19%	45%

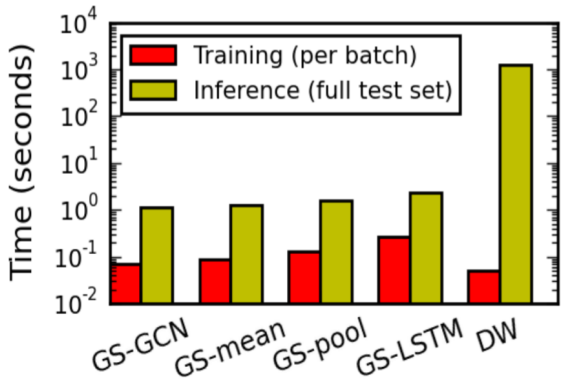
As an outcome, we can observe that GraphSAGE outperforms all the baselines by a significant margin, and the trainable, neural network aggregators provide significant gains compared to the GCN approach. For both supervised and unsupervised scores, GraphSAGE is obviously competitive with the other models, witnessing that this method combined with a suitable

aggregator function can achieve strong performance. Naturally, supervised scores are higher than unsupervised ones.

Furthermore, we can note that the LSTM- and pool-based aggregators performed the best, in terms of both average performance and number of experimental settings where they were the top-performing method. Besides, we remark that we don't have any result for PPI on DeepWalk models. It is explained by the fact that in the multi graph setting DeepWalk can't be applied due to its orthogonal invariance.

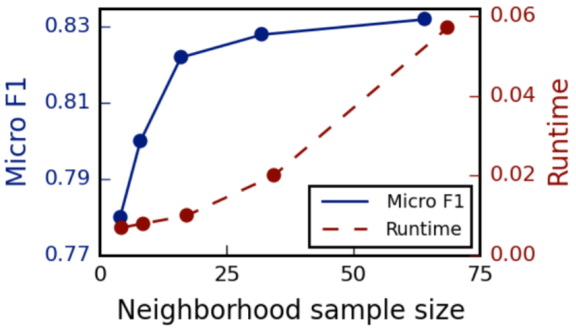
Now, let's take a look at the runtime impact

2.8.2 Timing experiment on Reddit data



The graph displays the training and test runtimes for the different models. For the training time, we observe that it is comparable for them all, but we can note that GraphSAGE using LSTM aggregator is slightly slower. But the main point of this chart is that DeepWalk has a high cost to do the inference on the full test set because they have to recalculate and retrain those embeddings. It's 1000 seconds more than what GraphSAGE takes.

This plot shows how the runtime and the score increase as the sample size grows. Performance improves as well as the runtime when the neighborhood size expands. Some small sizes such as 10 or 25 are the best compromise. Thus, despite the higher variance induced by sub-sampling neighborhoods, GraphSAGE is still able to maintain strong predictive accuracy, while significantly improving the runtime.



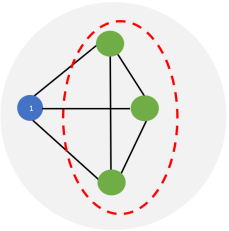
2.8 GraphSAGE theorem

One more interesting capital gain of GraphSAGE is the following theorem :

$$|z_v - c_v| < \epsilon, \forall v \in \mathcal{V}$$

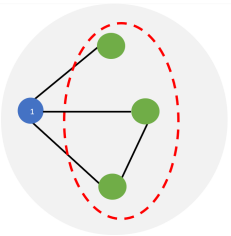
This expression proves that the model can deeply understand the graph structure. Indeed, the final representation z_v can be made arbitrarily close to the c_v appoints the clustering coefficient.
To better understand the meaning of clustering coefficient, we use the example below.

c_v =1



For the blue node, if all nodes are connected together then the clustering coefficient equals 1. They form a clique because the neighborhood nodes have the highest connectivity possible and result in a fully connected graph.

c_v =⅓



If we disconnect 2 edges for example, the clustering coefficient drops to ⅓. So the GraphSAGE can learn all of these structural info to a chosen precision

3. GDS Integration

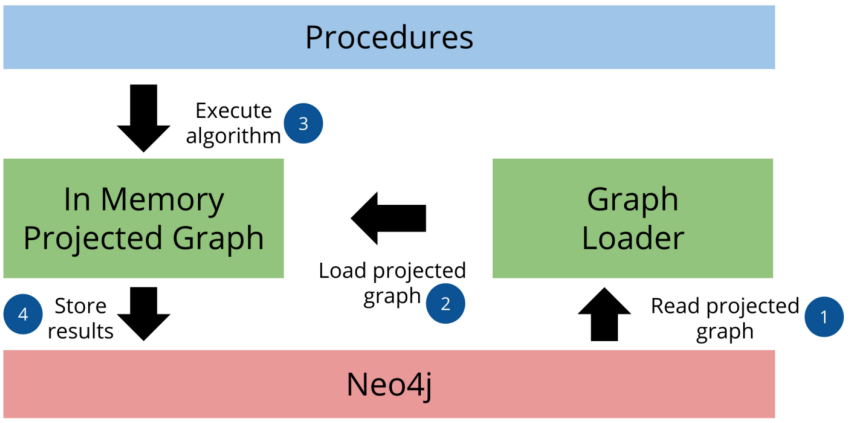
This section is dedicated to explain how the GraphSAGE approach can be integrated in GDS (Graph Data Science).

3.1. Theoretical aspect

But, what is GDS (Graph Data Science) ? The literature gives us many definitions. For example, in an interessant book in the domain we can find the following:

"*GDS is a science-driven approach to gain knowledge from relationships and structures in data, typically to power predictions; it's very interesting when relationships matter*"[1].

First of all, it could be relevant to explain how Graph Data Science operates for graph topology. Actually, GDS is running on heap memory and contains only significant weights, nodes and relationships. It gives a very simple and minimalist view for a graph. The graphs algorithms are executed on an in-memory projected graph model, separated from Neo4j stored one; this permitted by the Graph Loader component, as modeled below :



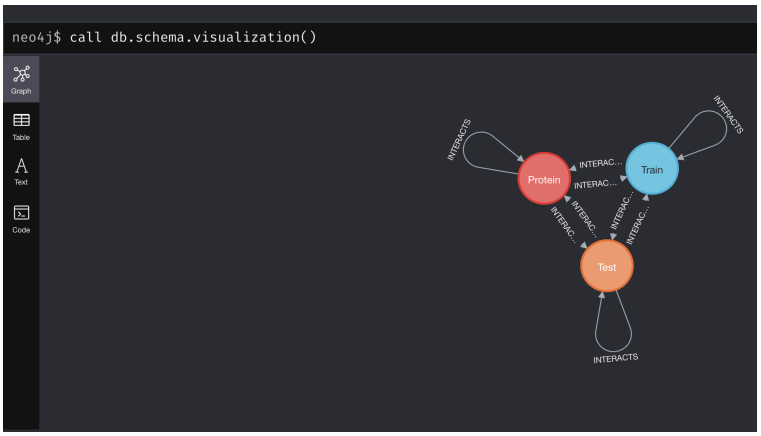
Source : Neo4j GDS library documentation[2]

[1] https://techresearchonline.com/wp-content/uploads/white-papers/Graph_Data_Science_For_Dummies_Book.pdf
[2] https://neo4j.com/docs/graph-data-science/current/common-usage/

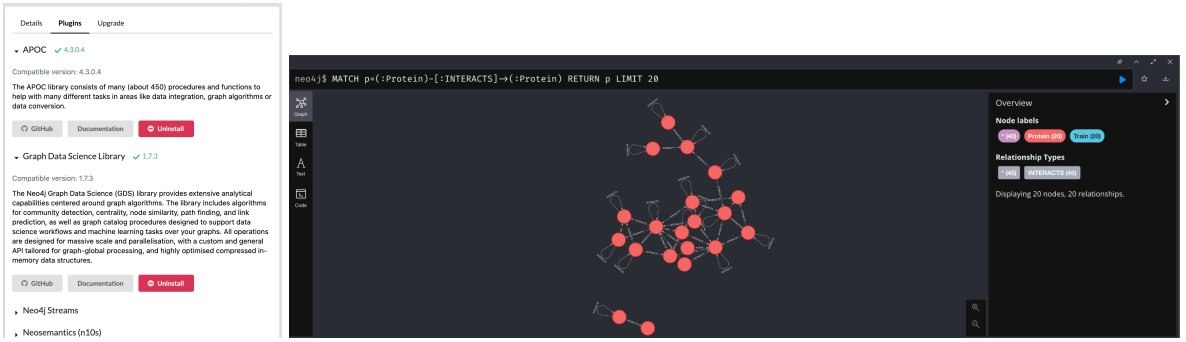
3.2. Practical aspect

3.2.1. Protein-protein dataset

To be more explicit, we can illustrate the approach through the integration of the protein role classification dataset used by the researchers in the article. This dataset represent a protein-protein monopartite (i.e. a class of genome) network, where a node is equal to a protein and relationships symbolize the interaction between them, like this :



Dataset model



We obtained this output after installing the GDS, APOC library (on the left) and uploading the data, then executing the query above (on the right)

Now, we are going to explain in detail how we could integrate the GraphSAGE approach in GDS by using the dataset just described above. Please note that this represents a proposal and is not the only way to integrate GraphSAGE algorithm.

1. To begin the integration, we project the train set in order to train the model and we stock it in the Graphs Catalog. As GraphSAGE input must be numerical values, it's required to convert the features used by our model. By default, the graph has the DIRECTED orientation, but we switch it into UNDIRECTED to simplify the computation because it's not very relevant.

```
UNWIND range(0,49) as i
WITH collect('embedding_' + toString(i)) as embeddings
CALL gds.graph.create('train_graph','Train',
    {INTERACTS:{orientation:'UNDIRECTED'}},
    {nodeProperties:embeddings})
YIELD graphName, nodeCount, relationshipCount
RETURN graphName, nodeCount, relationshipCount
```

	graphName	nodeCount	relationshipCount
1	"train_graph"	44906	1266396

Started streaming 1 records after 1 ms and completed after 293 ms.

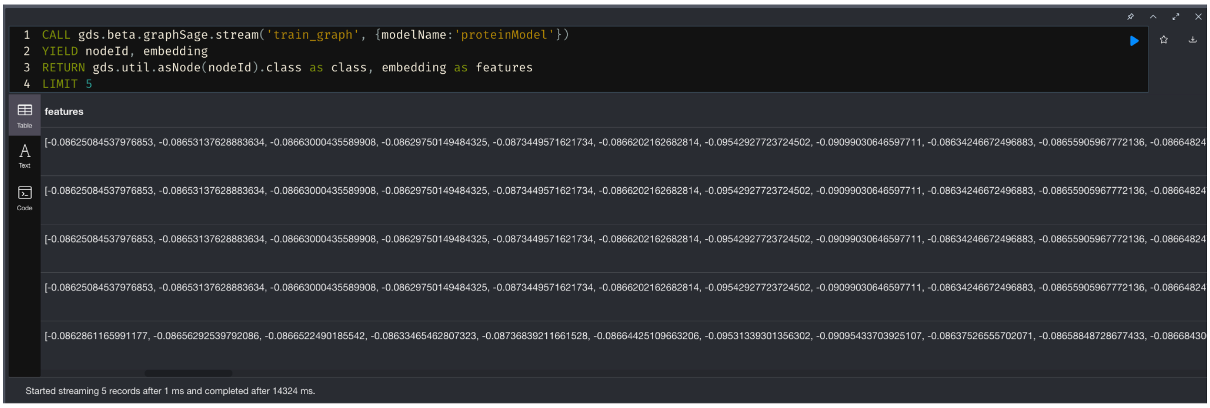
2. Then, we train the GraphSAGE model. We vary the different parameters among those used by the researchers in the article. We find that it's interesting to modify the size of the list SampleSize, which corresponds to the number of layers (k parameter in the algorithm explained above). The process is quite time consuming as you can see.

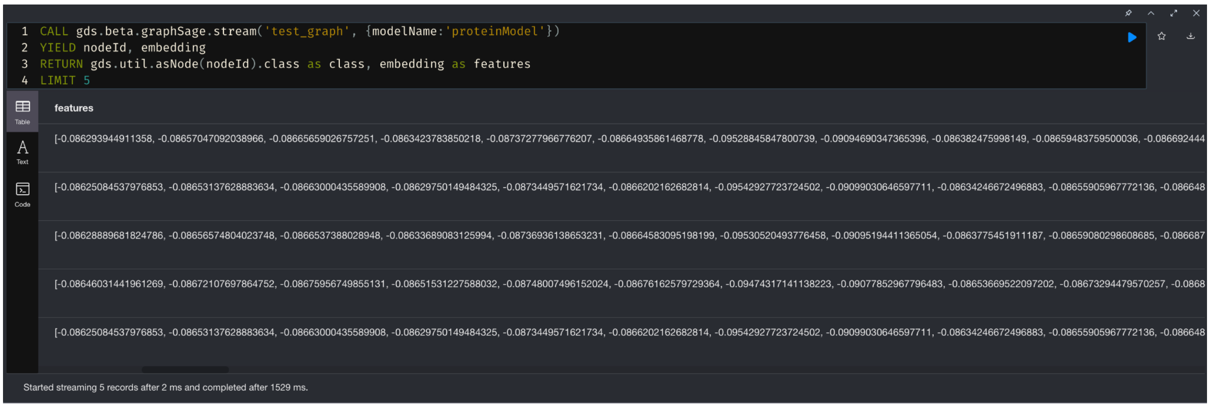
```
UNWIND range(0,49) as i
WITH collect('embedding_' + toString(i)) as embeddings
CALL gds.beta.graphSage.train('train_graph',{
  modelName:'proteinModel',
  aggregator:'pool',
  batchSize:256,
  activationFunction:'relu',
  epochs:5,
  sampleSizes:[15,10],
  learningRate:0.01,
  embeddingDimension:128,
  featureProperties:embeddings})
```

	modelInfo
1	{ "name": "proteinModel", "type": "graphSage" }

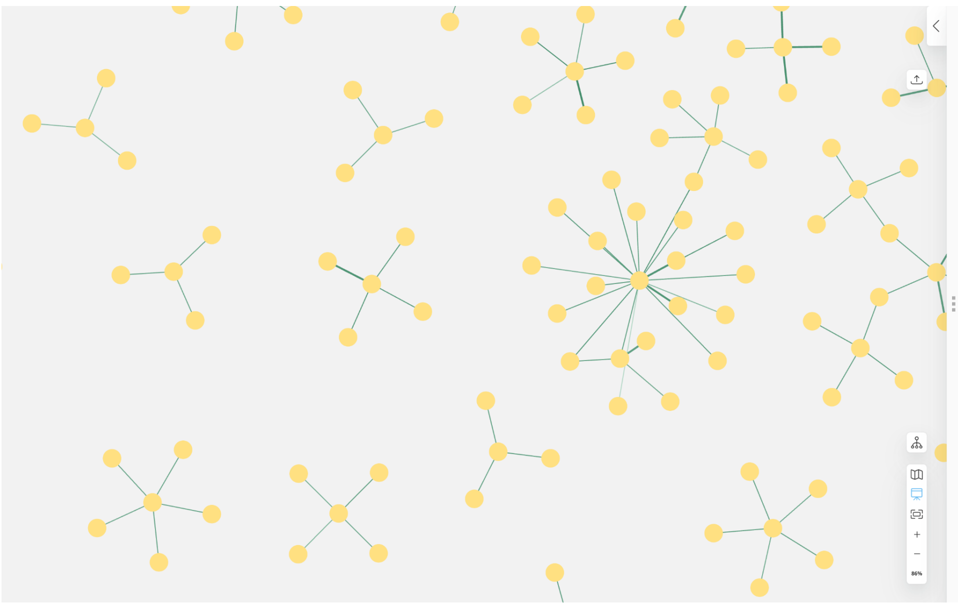
Started streaming 1 records after 2 ms and completed after 387282 ms.

Then, we do a request in order to print the embedding matrix for the first 5 nodes to be fast. We found this interesting because it gives us a more precise idea of the values calculated by the model. We apply it on both the train and test graph.



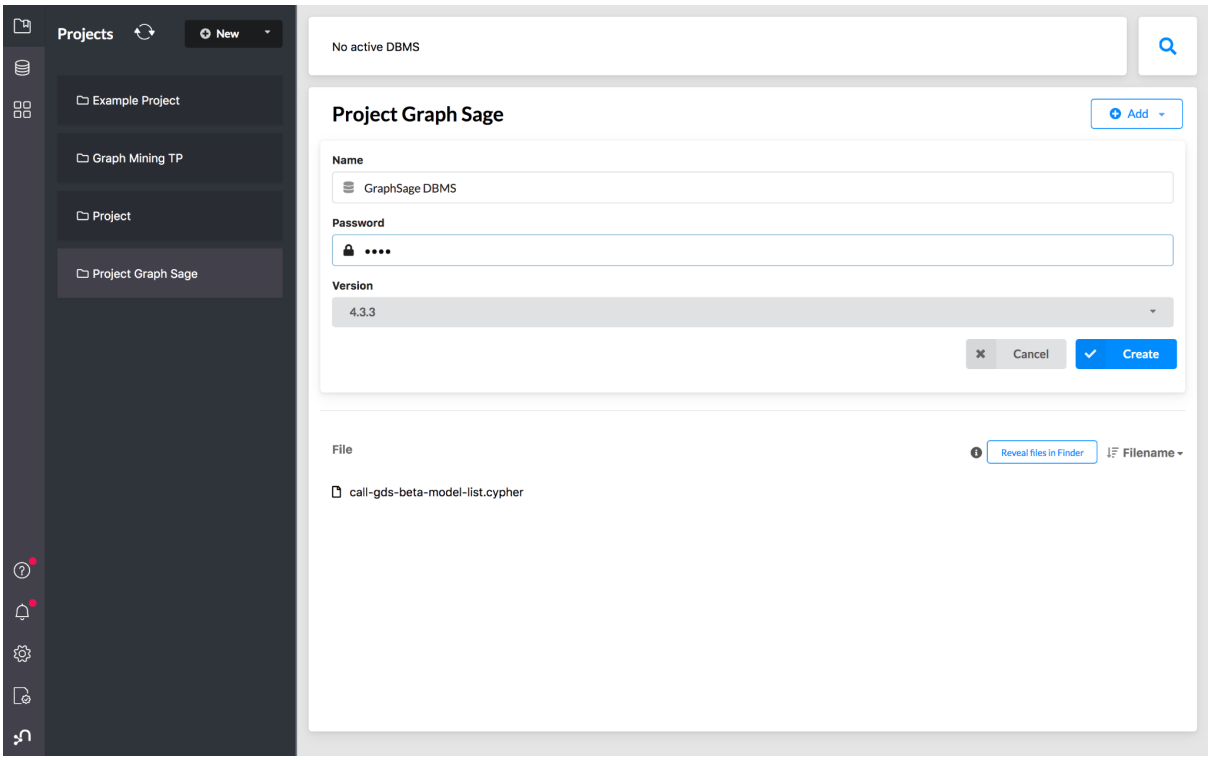


3. Once the model is trained, and stocked in Neo4j, we can use the model to induce node embeddings on the graph having the node properties in order to test it. But before we test it, we have to load the test set as an in-memory graph in the Graph Catalog.
4. Then, we used the knn algorithm to regroup the 3 neighbors which are the most similar based on the embedding for each node. The more the relationship color is darkened, more the node is very similar.

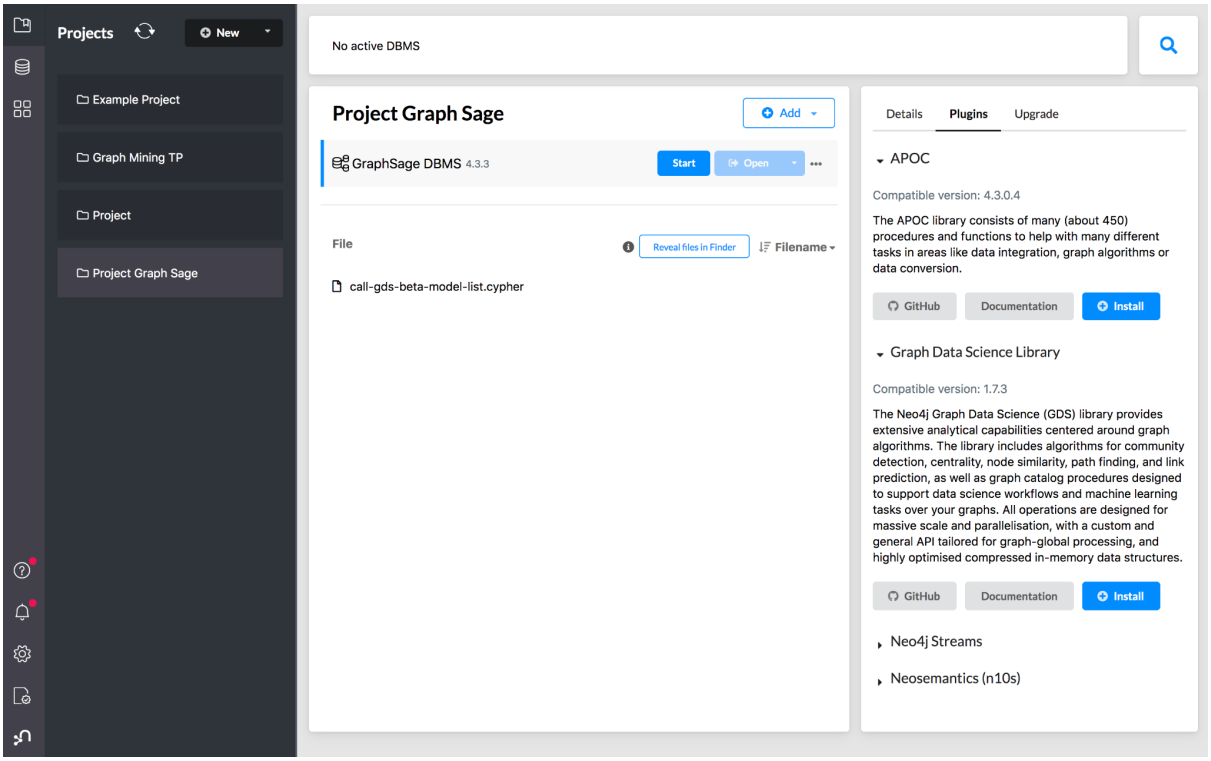


3.2.2. Another example

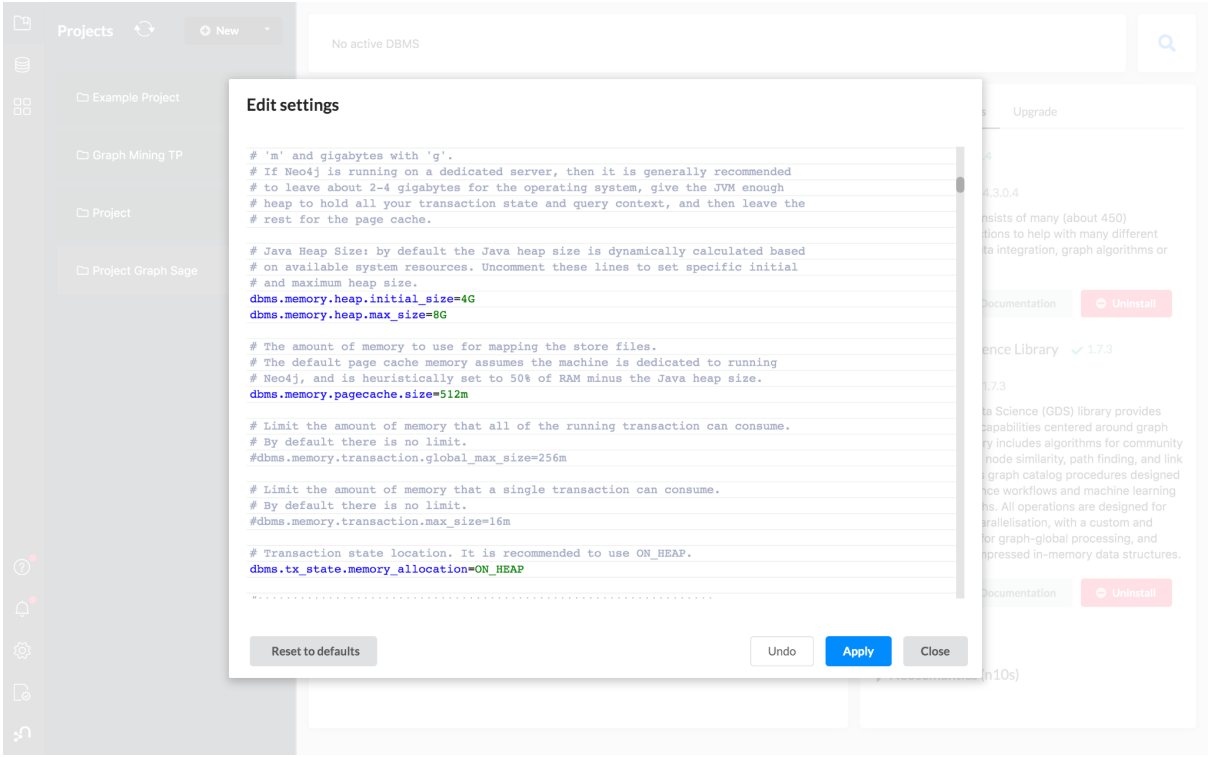
Step 1: create a local DBMS



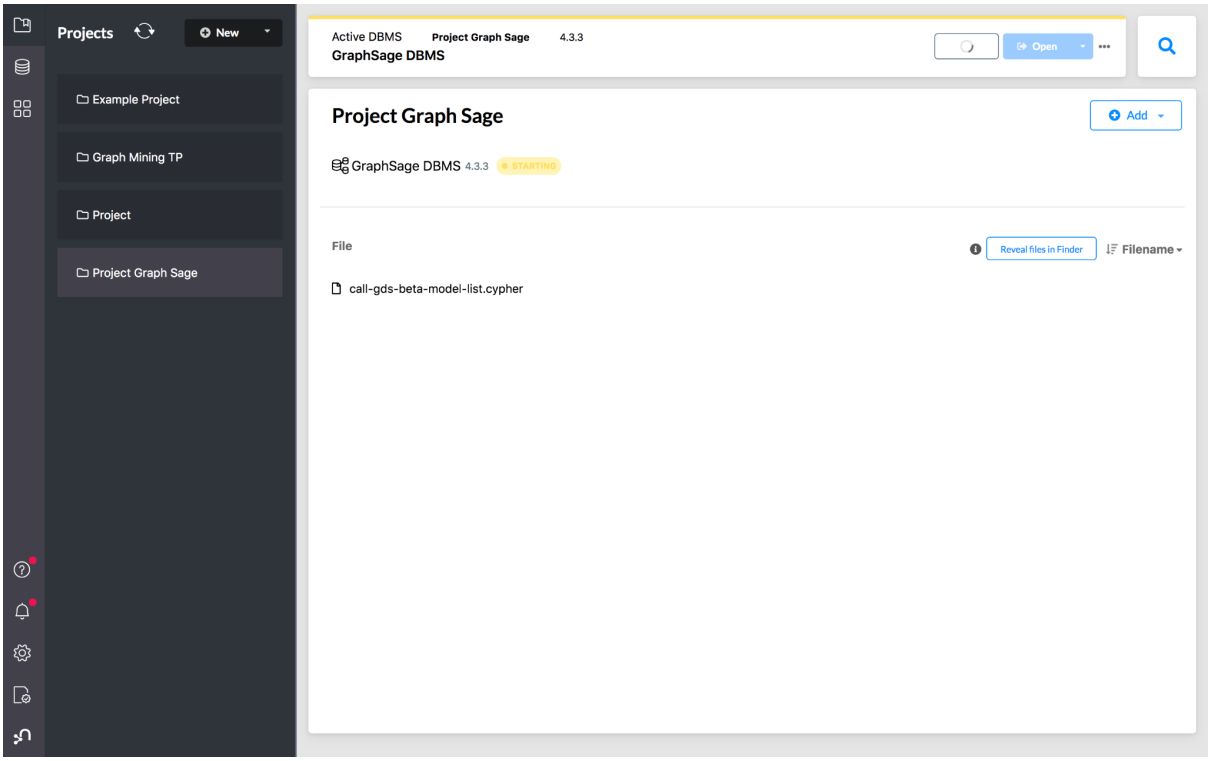
Step 2: install plugins APOC and Graph Data Science Library



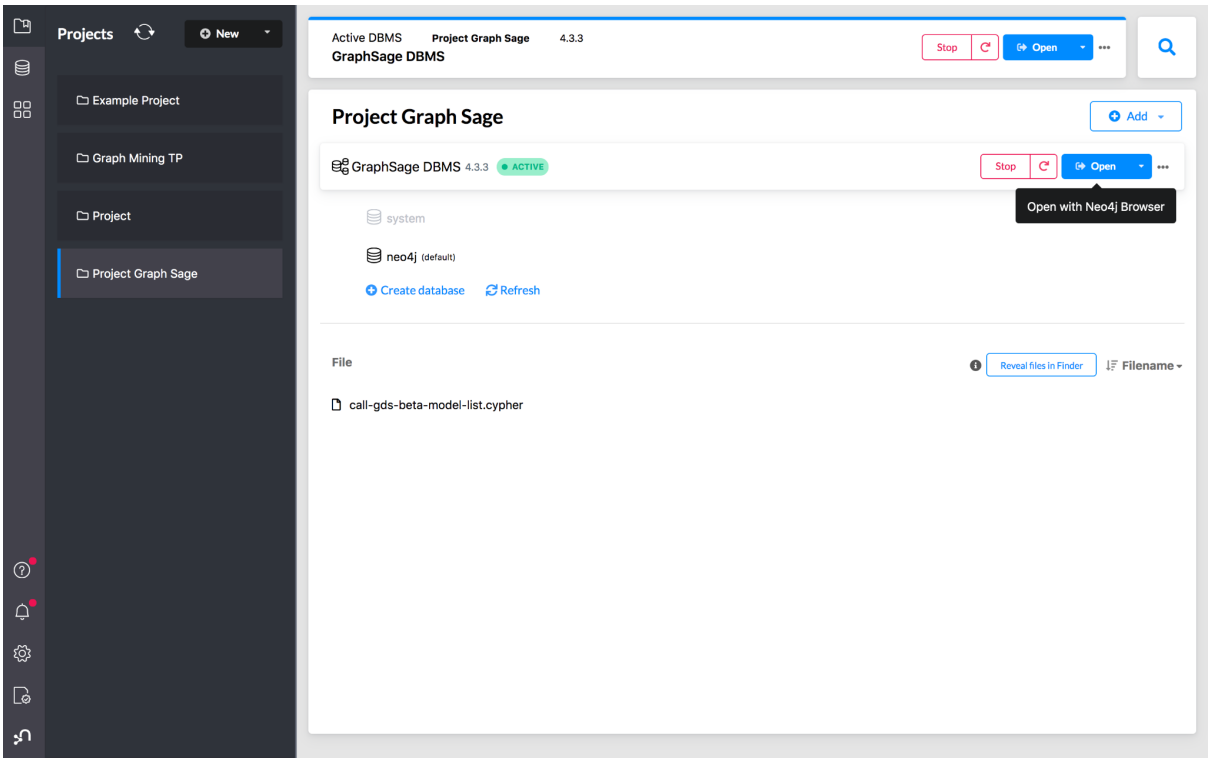
Step 3: change the dbms heap memory to have a better computation time



Step 4: start the dbms that we have just create



Step 5: open the dbms with Neo4j Browser to execute the query



First query:

```
CREATE
( dan:Person {name: 'Dan', age: 20, heightAndWeight: [185, 75]}),
(annie:Person {name: 'Annie', age: 12, heightAndWeight: [124, 42]}),
( matt:Person {name: 'Matt', age: 67, heightAndWeight: [170, 80]}),
( jeff:Person {name: 'Jeff', age: 45, heightAndWeight: [192, 85]}),
( brie:Person {name: 'Brie', age: 27, heightAndWeight: [176, 57]}),
( elsa:Person {name: 'Elsa', age: 32, heightAndWeight: [158, 55]}),
( john:Person {name: 'John', age: 35, heightAndWeight: [172, 76]}),

(dan)-[:KNOWS {relWeight: 1.0}]->(annie),
(dan)-[:KNOWS {relWeight: 1.6}]->(matt),
(annie)-[:KNOWS {relWeight: 0.1}]->(matt),
(annie)-[:KNOWS {relWeight: 3.0}]->(jeff),
(annie)-[:KNOWS {relWeight: 1.2}]->(brie),
(matt)-[:KNOWS {relWeight: 10.0}]->(brie),
(brie)-[:KNOWS {relWeight: 1.0}]->(elsa),
(brie)-[:KNOWS {relWeight: 2.2}]->(jeff),
(john)-[:KNOWS {relWeight: 5.0}]->(jeff)
```

It manually creates a graph to Neo4j Database, with 7 different nodes connected to other nodes with the relationship **KNOWS.**

With the following data structure:

	nodeType	nodeLabels	propertyName	propertyTypes	mandatory
1	":`Person`"	["Person"]	"age"	["Long"]	true
2	":`Person`"	["Person"]	"heightAndWeight"	["LongArray"]	true
3	":`Person`"	["Person"]	"name"	["String"]	true

Second query:

```
CALL gds.graph.create(
 'persons_graph',
 {
  Person: {
    label: 'Person',
    properties: ['age', 'heightAndWeight']
  }
 },
 {
   KNOWS: {
     type: 'KNOWS',
     orientation: 'UNDIRECTED',
     properties: ['relWeight']
    }
});
```

Then, we create a graph using the function create of the GDS library, to train the Graph SAGE algorithm on it.

Third query:

```
CALL gds.beta.graphSage.train(
  'persons_graph',
  {
    modelName:'personModel',
```

```
    aggregator:'mean',
    featureProperties: ['age', 'heightAndWeight'],
    batchSize:512,
    activationFunction:'relu',
    epochs:10,//how many times to traverse the graph during the training
    sampleSizes:[25,10]
  }
) YIELD modelInfo as info
RETURN
info.name as modelName,
info.metrics.didConverge as didConverge,
info.metrics.ranEpochs as ranEpochs,
info.metrics.epochLosses as epochLosses
```

Now, we train the model Graph Sage on the graph ***'persons_graph'***, we use the mean as an aggregator and the relu function as an activation function and the features *age* and heightAndWeight and we don't use not the name as features because it's a string value, we can only use numerical value.

We obtain an epoch losses during the training :

	didConverge	ranEpochs	epochLosses
1	true	2	[186.04947619820788, 186.04947619820788]

After training the model, we can see the embedding for each node.

Fourth query:

```
CALL
gds.beta.graphSage.stream('persons_graph',{modelName:'personModel'});
```

	nodeId	embedding
1	0	[-0.09549030222497214, -0.10938523001798224, -0.11613232167058493, -0.1197400371867788, -0.15141203194905342, -0.0801767
2	1	[-0.09526705139660749, -0.10952687477906847, -0.11620801091092985, -0.11963057448505442, -0.1511903365975257, -0.0800656
3	2	[-0.09520495139621861, -0.10960325654092831, -0.11579073000592512, -0.11989764189173135, -0.15114403076261007, -0.079543
4	3	[-0.09574550936622413, -0.10934539969278141, -0.11545550603567607, -0.12021907944794699, -0.1515718392266131, -0.0798322
5	4	[-0.09446507295892588, -0.10989227141718133, -0.11708665534346334, -0.11888559027384801, -0.15044460905283955, -0.080247
6	5	[-0.09315487754705211, -0.11063464172041912, -0.11726194304291716, -0.11849558631123035, -0.14921262540796687, -0.079226

Fifth query:

```
CALL gds.beta.graphSage.mutate(
  'persons_graph',
  {modelName: 'personModel', mutateProperty: 'embedding'}
);
```

We add embeddings to the graph '*persons_graph*', to use them for specific tasks like regrouping each node which are similar.

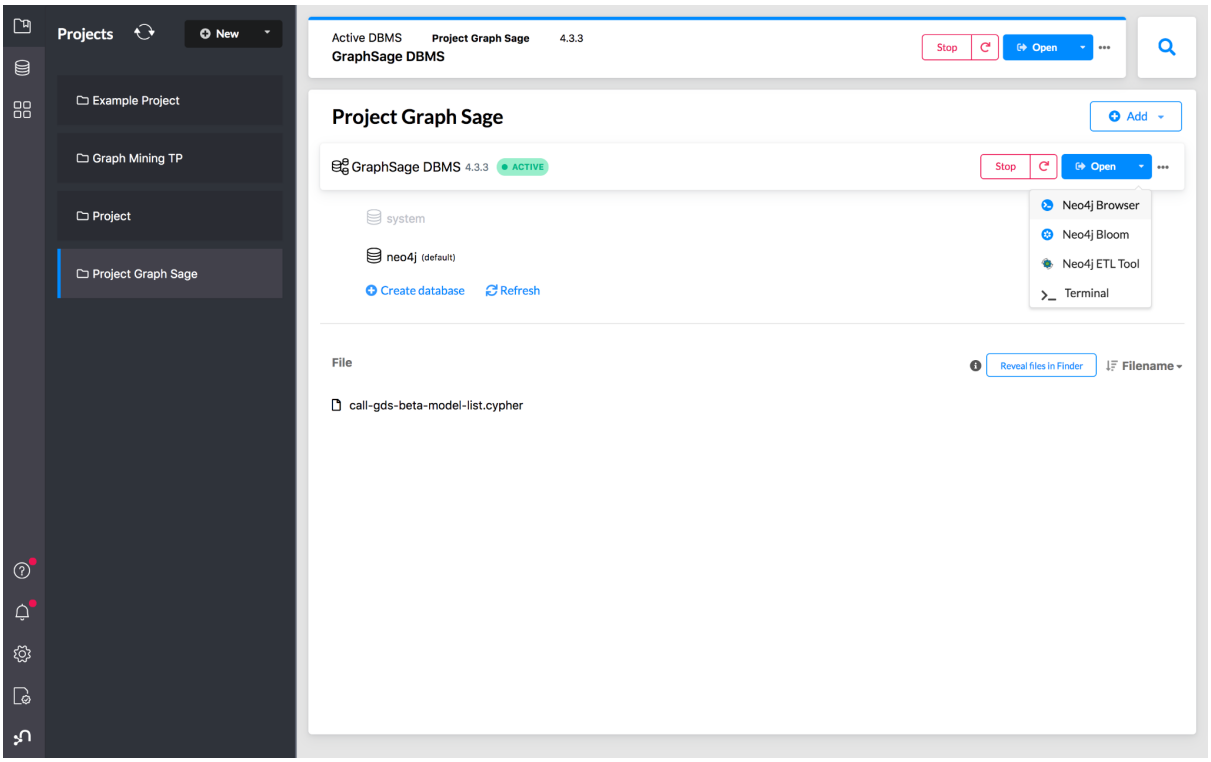
Sixth query:

```
CALL gds.beta.knn.write('persons_graph',
 {
   writeRelationshipType:'Similar Embedding',
   writeProperty: 'score',
   topK: 3,
   randomSeed: 42,
   nodeWeightProperty: "embedding"
 });
```

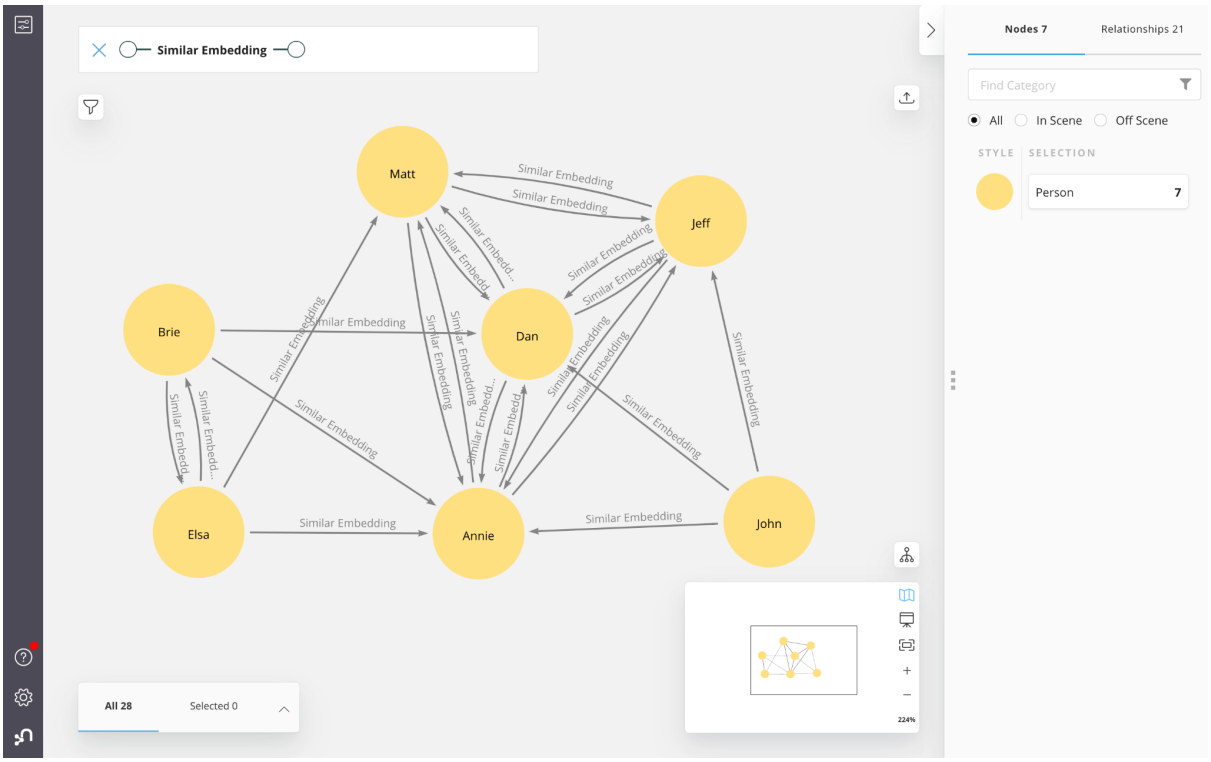
We use the Knn algorithm to add the property '*Similar Embedding*' to each node on the graph.

Now, we are going to use Bloom :

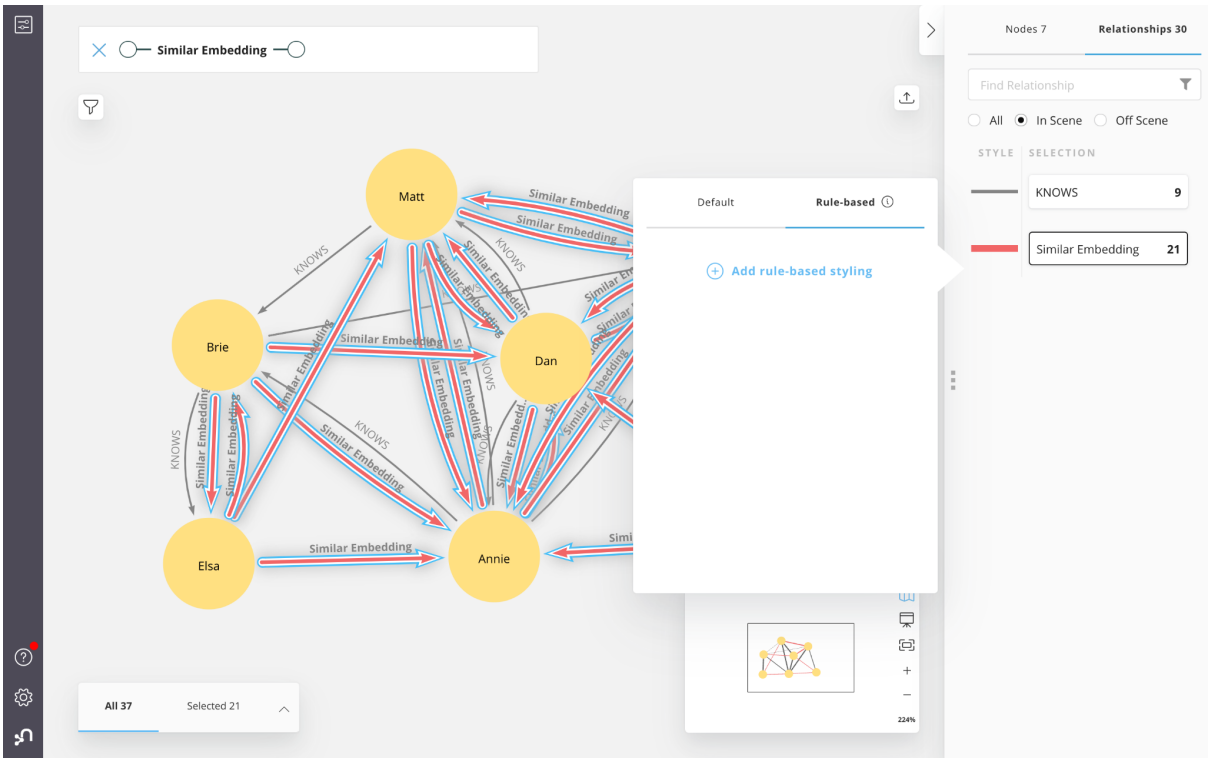
Step 6: open the dbms with Neo4j Bloom to visualize the result

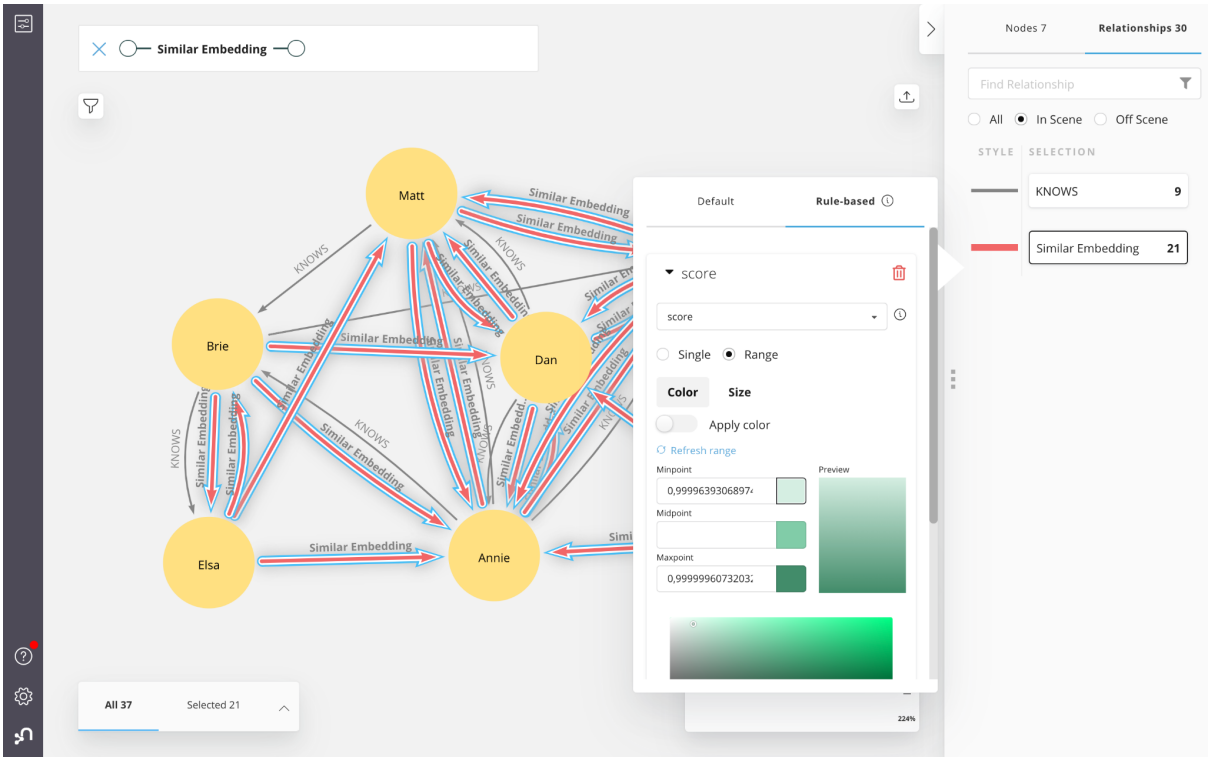


Step 7: we execute the query to display the Person based on the similar embedding

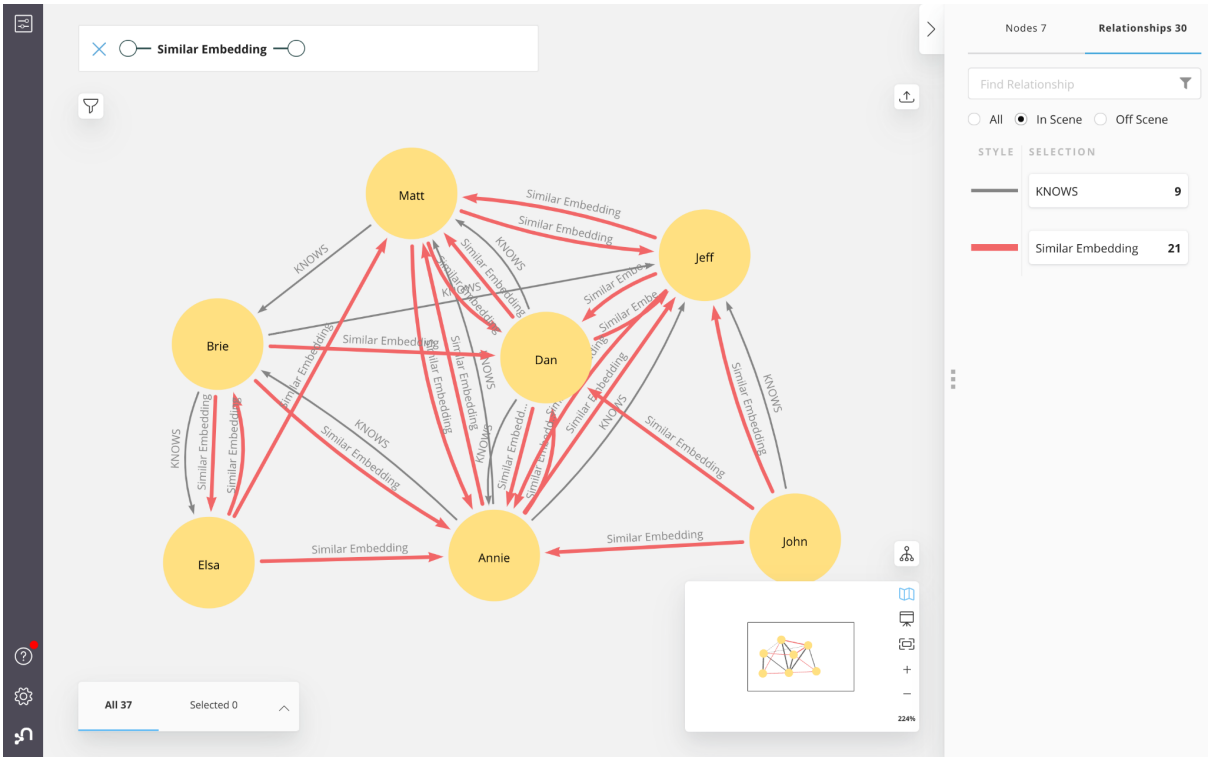


Step 8: we can add rules based on the relationship to highlight some of them





Step 9: we obtained this result



This ends this proposal of integration of GraphSAGE approach by using the GDS library in Neo4j.