Bioinformatics

Graph Neural Networks

Genes and pathways embedding

Locicero Giorgio, 1000024196

Contents

Introduction

Biological pathways are a way to represent sequences of interactions among molecules in a cell that leads to a certain product or a change in a cell or system in an organism. Pathways could be of many types, for this project we are interested in genetic pathways (**Gene Regulatory Network**). A gene (or genetic) regulatory network (GRN) is a collection of molecular regulators that interact with each other and with other substances in the cell to govern the gene expression levels of mRNA and proteins which, in turn, determine the function of the cell. GRN also play a central role in morphogenesis, the creation of body structures, which in turn is central to evolutionary developmental biology (evo-devo) [1]. This project will treat meta-pathways built from common pathways and the integration of genes coding for microRNA molecules, the resulting meta-pathways will be graphs of interactions between genes(expression or inhibition). These networks of gene interactions hide some information that is not currently used in the most applications of clinical analysis and that information could be used as new **Biomarkers**.

The objectives of this project are: to understand the hidden relations of the pathways related to some illnesses and confront the results with some controls to get some significant knowledge; to get some meaningful genes embeddings that could lead(along the pathways embeddings) to good, stable and trustworthy classifiers capable of identifying the categories and the differences to a group of people given their gene expressions.

During this project, the library and API used are StellarGraph(for the implementation of the GNN, which uses Tensorflow)[2] and Tensorflow (for the training of models and the definition of the core definitions used to get the models). Another alternative to stellargraph is **spektral**[3] but the core algorithms implemented are more or less the same (because the code for GNN and GraphSage are taken from the original code[4][5]) In the end, the pathway embedding will be confronted with two types of patients(some of them will be sane and used as controls, and some with some disease, especially tumours of different types).

The embeddings generated will be used in some use cases, they will also be compared among themselves by seeing the resulting embeddings for different types of cancer to see if the structure defined by the embeddings is significant enough to be used in other instruments or to get further insight about the locality and importance of some pathways for a specific tumour case. The tumours analysed in this research are all of **breast cancer**.

We will also encounter some problems related both to the definitions used for the models and to the general theory of GNN, which also lacks some capabilities that could be useful, especially in the use cases defined that could be done to fulfil the objectives defined previously.

1 Objectives of the project

As already introduced before, the main objectives of this project are:

- to understand the hidden links and pathways related to some illness and confront the results with data that is from a known normal case (sane patients) to get some knowledge and obtain new and unknown facts from the generated data.
- to get some meaningful genes and pathways embeddings that could lead to trustworthy and explainable classifiers (with explainable knowledge and consequences/decisions) capable of identifying the categories and the differences to a group of people, given their gene expressions, by generating some feature that will be used for inference.

The pathway embedding will be the main focus of this project since the genes embedding do not provide much knowledge alone since the features generated are directly dependent on the patients that constitute the expression matrix, and GNN are fixed for inputs(the inputs should always be a fixed number of features for every gene), while something like predicting or creating some features for a single patient that arrives in the graph (like streaming data, single patients that need some type of embeddings that could be used to classify them as they arrive) is not possible for this type of models, this matter will be discussed in details in 4. To define a model to create gene features that encode the structure of genes based on a metapathway (a graph induced from the pathways) and on a user-defined loss that maps close or related nodes near each other(seen in 4), and to find the pathways embeddings from the genes embeddings. With these embeddings, the main focus is to find some spatial and structural relationships between different data (different patients with no diseases, patients with different diseases), like clusters or confronting the density in the spatial domain of the embeddings. The significance of the results and the knowledge that is acquired with the pathway embedding will be validated by confronting the findings with research that also have seen similar solutions and conclusions (like the association of some genes' expression or inhibition with the presence of some kind of tumour).

2 The whole model

To understand what is the aim and the strategies used in this research, the model is introduced in its most important features which will, in turn, be dissected in the next chapters.

The whole model firstly predicts genes embeddings with a GNN model that is trained with a user-defined loss function that will be seen and commented[4], secondly, it predicts **pathways** embeddings based on another loss function[8] and with a custom model[7](implemented in TensorFlow) that could be seen as an alteration of the GCN model (2.1.2).

The theoretical model to compute the embeddings for the genes with a GNN is the following(1,2,3):

$$h^0_{g_i} = g_i \tag{1}$$

Where g_i is the tensor of features of the gene i, that is the expressions of a gene to a group of patients. This is the base case of the embedding for the genes and could be seen as the final layer of the graph neural network of the paths from gene i to its neighbourhood genes.

$$h^k_{g_i} = \sigma(W_k \sum_{u \in N(g_i)} \frac{h^{k-1}_u}{|N(g_i)|} + B_k h^{k-1}_{g_i}) \tag{2}$$

Where the σ is a non-linear function(RELU or Sigmoid), W_k and B_k are matrices(rank-2 tensors, even though the relation between tensors and matrices is not bijective) and are the parameters learned by the model to minimize the loss function(the methods used are Gradient Descent or variants[6], in the implementation of the model, **Adam** is used for genes embedding), this formula is a recurrent formula but could be unrolled for every layer that is part of the GNN(especially because Tensorflow does not handle really well recursive models and usually unrolls them).

The final embedding for every node will be:

$$z_{g_i} = h^k_{g_i} \tag{3}$$

The **Loss** function used to create the model for genes embeddings with a GNN model(trained with this function and the genes graph) will be the following:

$$\mathcal{L}(E) = \sum_{(u,v) \in E} c(u,v) ||(z_u - z_v)||_2 \tag{4}$$

With this function, genes linked by an edge with similar embeddings will be mapped closer in the embedding space while genes with different local structure will be more distant, more details will be seen in 2.1.4.

The weight $c(u,v)$ is defined in the following ways:

$$c(u,v) = \frac{abs(|K_v| - |K_u|) + abs(|J_v| - |J_u|) + (|K_v \vee K_u| + |J_v \vee J_u|)}{max(\text{in-out degree})} \tag{5}$$

$$c(u,v) = \frac{(|K_v \wedge K_u| + |J_v \wedge J_u|)}{abs(|K_v| - |K_u|) + abs(|J_v| - |J_u|) + 0.01} \tag{6}$$

Where K_i is the set of the predecessors of i (in edges), while J_i is the successors set of i (out edges).

The weight defined in 5 gives more value to nodes that have not the same nodes for predecessors and successors along with giving more value to nodes that have not similar structures,

while the weight defined in 6 values more nodes that have similar predecessors and successors along with similar neighbourhood structure. It is logical to give more weight to genes that have similar predecessors(influenced by) and successors (consequences) because genes that are activated-expressed by the same predecessors should have similar embeddings, and genes that have a similar influence on other genes(inhibition or expression) should also have similar embeddings. For these reasons, the weight chosen to be implemented for the genes embedding models is the one defined in 6. The 0.01 is added to the divider to prevent division by zero from happening.

The embedding for **pathways** is defined by the following equation:

$$z_{P_i} = \sigma(W_P \sum_{u \in P_i} c(u) \frac{z_u}{|P_i|}) \tag{7}$$

Where $c(u)$ is the weight function that assigns to every gene a real number. Because the embedding created for the pathways should be related to the genes that play a major role both in the single pathway and in the metapathway, there are a lot of possible strategy to follow to get significant embeddings:

- genes that play a central role of information propagation through the pathway and metapathway should have more significance than genes that do not handle information as hubs.
- genes that are usually the start of pathways should have higher significance than genes that gets activated as a consequence.
- genes that are known to be important for the metabolism should have higher significance.

The strategy adopted is the first one, and the weight applied to every gene to get the pathway embedding is the **betweenness** for the whole metapathway. Additional consideration for a better approach are seen in 4.

The loss function used to create the model that predicts the pathways embeddings will be the following:

$$\mathcal{L}(P) = \sum_{P_i \in P} \sum_{P_j \in P} c(P_i, P_j) ||(z_{P_i} - z_{P_j})||_2 \tag{8}$$

Where P_i is the pathway i, and $c(P_i, P_j)$ is the weight assigned to the pair of pathways.

The weight is the following:

$$c(P_i, P_j) = \frac{|P_i \cap P_j|}{|P_i \cup P_j|} \tag{9}$$

The weight defined in 9 is the Jaccard similarity, further consideration on the weight of the loss should be similar to the loss defined for the genes embedding in 4.

This loss function needs to be minimized so the pathways embedding distances/similarities are minimized(Pathways with similar genes embeddings will be closer than pathways with little to no genes in common). The distance used will be a norm-2 but another norm should suffice and work the same way, in the implementation a stable version of the norm-2 is used to avoid unexpected behaviour.

2.1 Graph Neural Network

All Graph Neural Network algorithms are conceptually related to node embedding approaches, general supervised approaches to learning over graphs, and classification of nodes or graphs. In this research, there will not be an implementation of matrix factorization methods(related to spectral clustering, multidimensional scaling or approaches related to PageRank with random walks) because these methods were tested on the data and extremely unsatisfactory results were obtained from the models obtained, especially for the task at hand, because these type of models do not take into account any(or very little) graph structural information during training.

All models tested in this paper are the best(among the available ones) for the data given, that is **Graph Convolutional** models and inductive approaches like **GraphSage**.

2.1.1 General framework

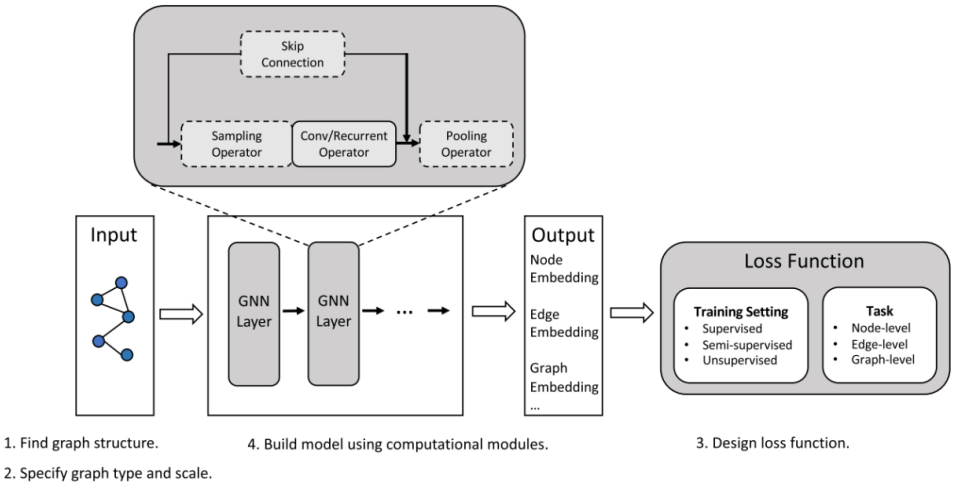


Figure 1: Pipeline for a GNN model

Graphs are a kind of data structure which models a set of objects(nodes) and their relationships (edges). As a unique non-Euclidean data structure for machine learning, graph analysis focuses on tasks such as node classification, link prediction, and clustering. Graph neural

networks (GNNs) are deep learning based methods that operate on graph domain. Due to its convincing performance, GNN has become a widely applied graph analysis method recently. Almost all of the work on GNN is born from the transposition of the CNN models to graph data, but the passage from Euclidean data to non-Euclidean data rises a lot of concerns and problems on the definitions from CNN and the theory behind. Extending deep neural models to non-Euclidean domains, which is generally referred to as geometric deep learning, has been an emerging research area.

Another useful matter linked to Graph learning is **graph representation learning** which learns to represent graph nodes, edges or subgraphs by low-dimensional vectors. In the field of graph analysis, traditional machine learning approaches usually rely on hand engineered features and are limited by its inflexibility and high cost. Following the idea of representation learning and the success of word embedding, a lot of ways to do Graph Embeddings were born (Node2Vec is one of these embedding algorithms that uses random-walks and anonymous walks to create embeddings for nodes and graphs [7], along with other algorithms which will not be seen in this research).

Based on CNNs and graph embedding, variants of graph neural networks (GNNs) are proposed to collectively aggregate information from graph structure. Thus they can model input and/or output consisting of elements and their dependency (usually local dependency in the graph, especially for inductive models that build models based upon local neighborhoods of a defined depth, sampling is also really useful in these models, as seen in 2.1.3).

The graphs used in this research will be static(no dynamic edges during runtime or entering nodes) because the goal of this project is to find genes embeddings and pathways embeddings given the genes features and graph. The size of the graph used is not really large (around 5000 genes and 28500 edges).

The first step should be to design the loss function based on the task type and the training setting. For graph learning tasks, there are usually three kinds of tasks:

1. **Node-level** tasks focus on nodes, which include node classification, node regression, node clustering, etc. Node classification tries to categorize nodes into several classes, and node regression predicts a continuous value for each node. Node clustering aims to partition the nodes into several disjoint groups, where similar nodes should be in the same group.

2. **Edge-level** tasks are edge classification and link prediction, which require the model to classify edge types or predict whether there is an edge existing between two given nodes.

3. **Graph-level** tasks include graph classification, graph regression, and graph matching, all of which need the model to learn graph representations.

In this research, we have defined a **Node-level** loss function that uses nodes generated embeddings as well as edges between the embeddings to compute the final value, as seen in

4. We work in a semi-supervised/unsupervised environment because we have the feature to use to create the embeddings but not the embedding itself that minimizes the loss function.

The next step is **building the model** using the **computational modules**. Some commonly used computational modules are:

- **Propagation Module**. The propagation module is used to propagate information between nodes so that the aggregated information could capture both feature and topological information. In propagation modules, the **convolution operator** and **recurrent operator** are usually used to aggregate information from neighbors while the **skip connection** operation is used to gather information from historical representations of nodes and mitigate the over-smoothing problem.
- **Sampling Module**. When graphs are large, sampling modules are usually needed to conduct propagation on graphs. The sampling module is usually combined with the propagation module.
- **Pooling Module**. When we need the representations of high-level subgraphs or graphs, pooling modules are needed to extract information from nodes.

Both the GCN model and GraphSage use **Propagation modules** to propagate information between nodes, even though they use different strategies to create the whole model(the GCN propagation is aimed toward spectral and trasductive approaches, while the GraphSage model is inductive and structural based upon the neighborhood of nodes).

Other Modules and techniques could be used to optimize and upgrade GNN models as seen in figure 2(like sampling, used in GraphSage and in other methods), but the literature is too much and, in this research, we will see in depth two models in particular that use different techniques with the same goal.

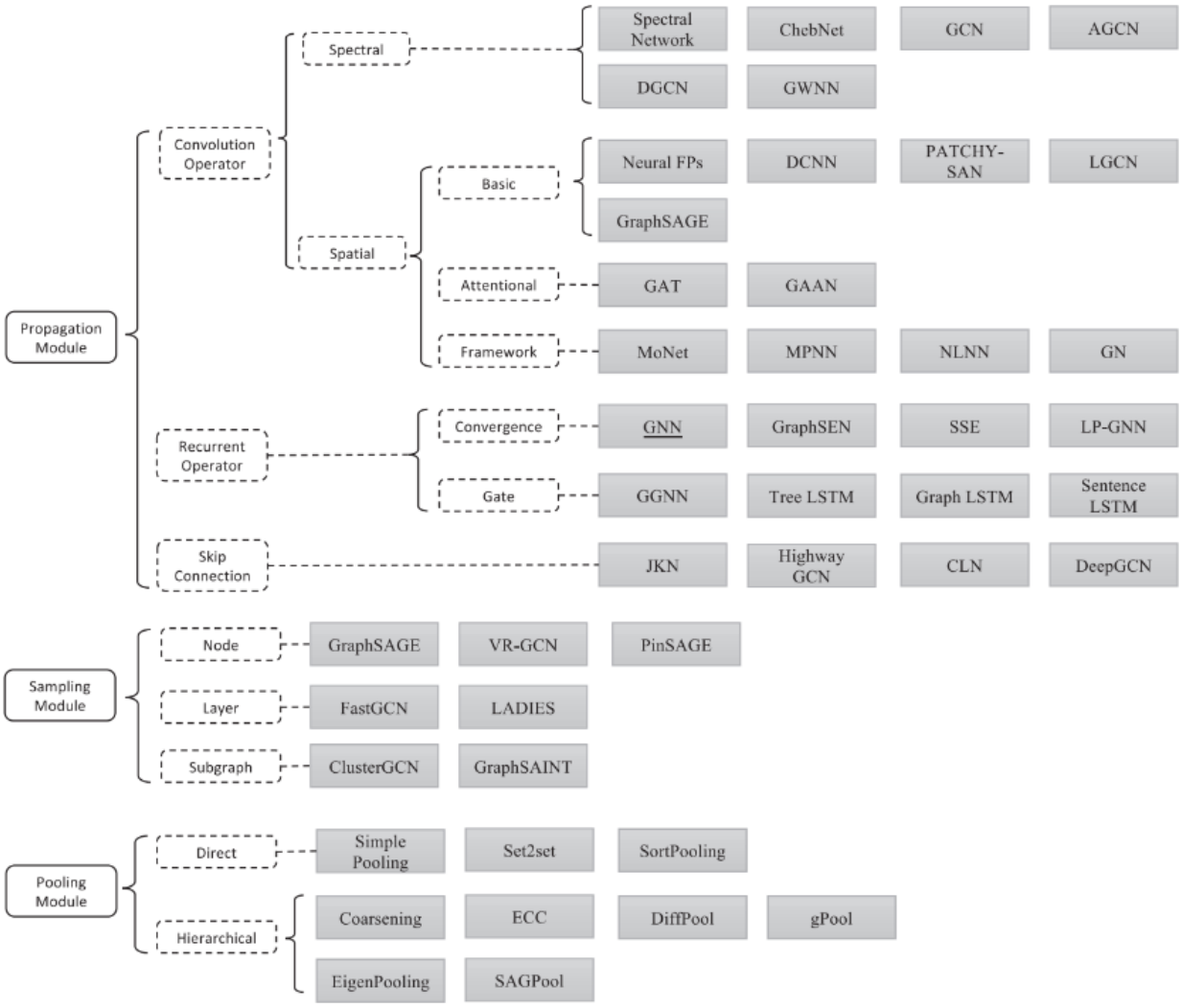


Figure 2: Different Modules and techniques used to create GNN models

For additional information about GNN and the general framework behind it, see the specifics models for GCN,GraphSage and so on, and give a look at [8] review about GNN.

2.1.2 Graph convolutional network

Graph Convolutional Networks are a transposition of Convolutional Neural Network on images and bear resemblance to spectral approaches for graph clustering and classification. GCN are called convolutional because filter parameters are typically shared over all locations in the graph(or a subset).

For these models, the goal is to learn a function of signals/features on a graph $G = (V, E)$ which takes as input:

- A feature description x_i for every node i summarized in a $N \times D$ feature matrix X (N: number of nodes, D: number of input features)
- A representative description of the graph structure in matrix form; typically in the form of an adjacency matrix A (or some function thereof)

The model produces a node-level output Z (an $N \times F$ feature matrix, where F is the number of output features per node). Every neural network layer can then be written as a non-linear function as defined in the following pattern formula:

$$H^{(l+1)} = f(H^{(l)}, A) \tag{10}$$

with $H^{(0)} = X$ and $H^{(L)} = Z$ (or z for graph-level outputs), L being the number of layers. The specific models then differ only in how $f(\_, \_)$ is chosen and parameterized.

An example of a very simple form of a layer-wise propagation rule is the following:

$$f(H^{(l)}, A) = \sigma(AH^{(l)}W^{(l)}) \tag{11}$$

where $W^{(l)}$ is a weight matrix for the l-th neural network layer and $\sigma(\_)$ is a non-linear activation function like the ReLU.

There are two limitations about the model previously described:

1. Multiplication with A means that, for every node, we sum up all the feature vectors of all neighboring nodes but not the node itself (unless there are self-loops in the graph). We can "fix" this by enforcing self-loops in the graph: we simply add the identity matrix to A.
2. A is typically not normalized and therefore the multiplication with A will completely change the scale of the feature vectors (understandable by looking at the eigenvalues of A). Normalizing A such that all rows sum to one, i.e. $D^{-1}A$, where D is the diagonal node degree matrix, gets rid of this problem. Multiplying with $D^{-1}A$ now corresponds to taking the average of neighboring node features. In practice, dynamics get more interesting when we use a symmetric normalization, i.e. $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ (as this no longer amounts to mere averaging of neighboring nodes).

The final formula after these considerations is :

$$f(H^{(l)}, A) = \sigma(D^{-\frac{1}{2}}(A + I)D^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{12}$$

This example is a way of using graph convolutional network to predict and compute features, this type of computation will be modified and used for the **path embedding** problem 2.2

The original GCN algorithm [9] is designed for semi-supervised learning in a transductive setting, and the exact algorithm requires that the full graph Laplacian is known during training. A variant of the GraphSage algorithm can be viewed as an extension of the GCN framework to the inductive setting.

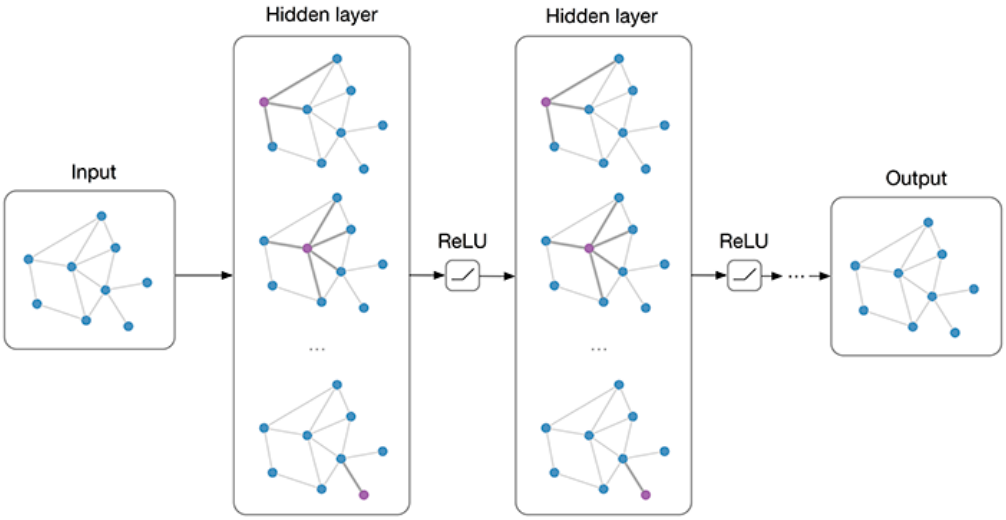


Figure 3: Multi-layer GCN with first order filters

2.1.3 Graphsage

The approach followed by GraphSage(SAmple and aggreGatE) is an **inductive** approach that takes into consideration unseen nodes during the training, in contrast of **trasductive** approaches that need all nodes to be present during the training of the model to predict embeddings. This is done by taking a subset of the neighborhood during the training of the model and aggregate the embeddings in this subset to find the node embedding. Unlike embedding approaches that are based on matrix factorization, by incorporating node features in the learning algorithm, the algorithm simultaneously learn the topological structure of each node's neighborhood as well as the distribution of node features in the neighborhood.

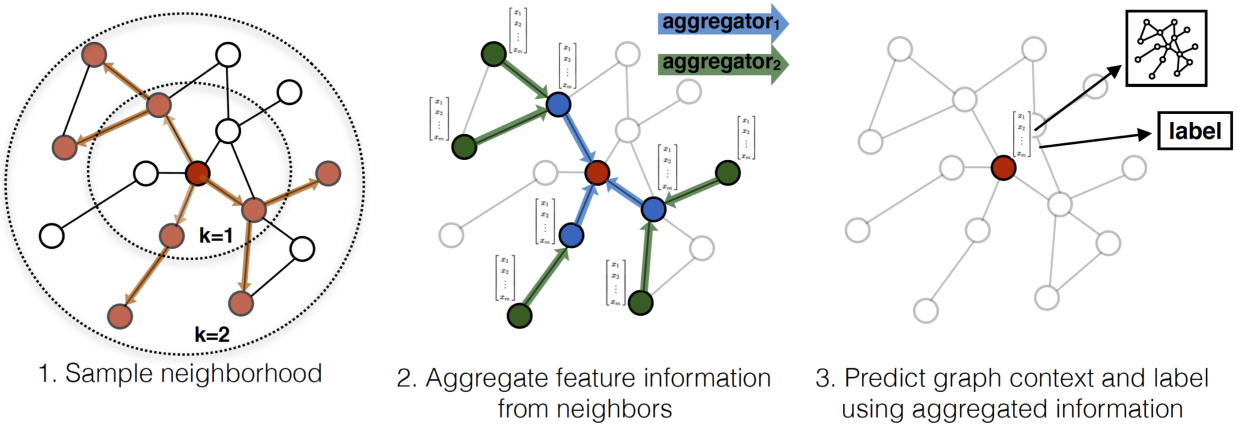


Figure 4: Sample and aggregation visualization of Graph Sage

However, in this research, a trasductive approach is considered where all nodes need to be present for the model to work, this is because the loss function needs all the edges to be present to find the minimum(or local minimum in a certain range).

Instead of training a distinct embedding vector for each node, the model firstly trains a set of aggregator functions that learn to aggregate feature information from a node's local neighborhood. Each aggregator function aggregates information from a different number of hops, or search depth, away from a given node.

The key idea behind this model is that it learns how to aggregate feature information from a node's local neighborhood. The first part of the GraphSAGE model is the training of the model with a loss function(known loss function or user defined) to find the parameters that will be used to generate the embeddings, while the second part is embedding generation (forward propagation) algorithm, which generates embeddings for nodes assuming that the GraphSAGE model parameters are learned by the first step.

The embedding generation, or forward propagation algorithm(algorithm 1), assumes that the model has already been trained and that the parameters are fixed. In particular, the model assumes that we have the parameters of K aggregator functions(denoted $\text{AGGREGATE}_k, \forall k \in$

$1, ..., K$, which aggregate information from node neighbors, as well as a set of weight matrices W_k,$\forall k \in 1, ..., K$) are learned, which are used to propagate information between different layers of the model or "search depths".

Algorithm 1: Embedding generation (forward propagation)

Input: Graph $G(V, E)$; input features $x_v, \forall v \in V$; depth K; weight matrices W_k,$\forall k \in 1, ..., K$; non-linearity σ; differentiable aggregator functions AGGREGATE_k,$\forall k \in 1, ..., K$; neighborhood function $N : v \longrightarrow 2^V$

Output: Vector representations $z_v \forall v \in V$

$h_v^0 \longleftarrow x_v, \forall v \in V$;

for k = *1,...,K* **do**

  for $v \in V$ **do**

    $h_{N(v)}^k \longleftarrow \text{AGGREGATE}_k(h_u^{k-1}, \forall u \in N(v))$;

    $h_v^k \longleftarrow \sigma(W^k \cdot \text{CONCAT}(h_v^{k-1}, h_{N(v)}^k))$;

  end

  $h_v^k \longleftarrow \frac{h_v^k}{||h_v^k||_2}, \forall v \in V$

end

The intuition behind the algorithm is that at each iteration, or search depth, nodes aggregate information from their local neighbors, and as this process iterates, nodes incrementally gain more and more information from further reaches of the graph. Further considerations on the algorithm could be seen in [10]

The neighborhood is taken by a sample of k-distant nodes from the node considered.

Ideally, an aggregator function would be symmetric (i.e., invariant to permutations of its inputs) while still being trainable and maintaining high representational capacity. The symmetry property of the aggregation function ensures that the model neural network model can be trained and applied to arbitrarily ordered node neighborhood feature sets. The aggregation function used could be user-defined or taken from different aggregation architectures, the one that will be used for this research is the mean aggregator that take the element-wise mean of the vectors in the neighborhood(It is also called convolutional because it is the generalization of the filters seen in 2.1.2, with an inductive approach, so it is a linear approximation of a localized spectral convolution), that is:

$$h_v^k \longleftarrow \sigma(W \cdot \text{MEAN}(h_v^{k-1} \cup h_u^{k-1}, \forall u \in N(v))) \tag{13}$$

That is what is seen in 2(even though the old value of the node is directly multiplied and summed, but that is a detail, the formula leads to similar, almost equal, results).

An important distinction between this convolutional aggregator and the others aggregators(that will not be used in this research) is that it does not perform the concatenation operation in line 5 of Algorithm 1 i.e., the convolutional aggregator does concatenate the node's previous layer representation h_v^{k-1} with the aggregated neighborhood vector $h_{N(v)}^k$. This concatenation can be viewed as a simple form of a "skip connection" [8](this paper is

really good and a must read because it is a review of most of the strategies used for GNN and the differences/similarities between them) between the different "search depths", or "layers" of the GraphSAGE algorithm, and it leads to significant gains in performance.

The model learns a function that generates embeddings by sampling and aggregating features from a node's local neighborhood. In the main paper[10],[5] the algorithm could also be used to predict embeddings in dynamic graphs, but in the **stellargraph** implementation, the graphs used needs to be static and defined at runtime (no nodes or edges could be added to a graph at runtime to be embedded) or this is what the documentation seems to be implying(for GraphSage, the given demo takes a static graph and train the model on the graph).

2.1.4 Genes embeddings

The data at hand is highly structured, graph edges were generated from the meta-pathways of genes, edge weights are the interactions between genes in pathways(if a gene inhibits or stimulates another genes based on the value of the edge weight) and genes expressions for patients as features. Every one of these structured characteristics will be used to generate the genes embeddings, and the model used will be the one defined by the formulas seen in the beginning of this chapter 2.

Genes embeddings will be created from the group of patients' genes expression (the whole expression matrix), and the expressions are normalized. For different types of tumours, the group of patients that has been marked with the illness will be sampled to match the number of patients in the controls (113 in this case).

The loss function used to compute the model is the one defined in previous sections 4, the loss function is modified a little in the code to account for instability(of the norm computation) and for the requirements of tensorflow and keras models(for method definitions and signatures).

Algorithm 1: Loss function used for the generation of the GNN model for genes embeddings in GraphSage

```
indexOfGenes = dict()
for i in range(0,len(controlsDK.index)):
    indexOfGenes[controlsDK.index[i]] = i

tensorEdges = tf.constant(geneEdges,tf.int32)
tensorEdgesTest = tf.map_fn(lambda row: tf.
            map_fn(lambda element:indexOfGenes[element.numpy()],tf.
                    gather(row, [0, 1]).numpy())  ,tensorEdges)

@tf.function
def difference_genesGRAPHSAGE(edges,y_pred):
    gene1 = y_pred[edges[0]]
    gene2 = y_pred[edges[1]]
    return tf.sqrt(tf.reduce_sum(tf.square(gene1 - gene2)) + 1.0e-12)
```

```
def optimized_lossGNN(tensorEdgesTest,constantTensor):
    @tf.function
    def loss_out(y_true, y_pred):
        return tf.reduce_sum(
         tf.math.multiply(
          tf.map_fn(lambda edge:difference_genesGRAPHSAGE(edge,y_pred),
            tensorEdgesTest,
            fn_output_signature=tf.float32),
          constantTensor)
         )
    return loss_out
```

This loss is specific to the GraphSage because the models use different representations for layers in the keras models. The loss for the GCN uses a different function to compute the difference of the genes embeddings.

The loss is optimized for TensorFlow and uses the function introduced in 4. The constant tensor is a static tensor that represents the weights for every gene as defined in 6.

An important consideration about this loss function is that it implies that the underlying model is not divided into batches because different batches could contain different edge embeddings that need to be in the same batch, this slows down the computation and make it impossible to divide the execution into batches to use less memory. For this reason, there were some problems when executing in the GPU, since it has less memory than the device, and the tumour model with all the patients was done using CPU and saving checkpoints. The final model for all the patients with tumours was finished after 5 days of training.

Algorithm 2: StellarGraph graph creation

```
graphWithFeatures = sg.StellarGraph(controlsDK ,geneEdges )
directedGraph = sg.StellarDiGraph(controlsDK ,geneEdges)
```

For the GCN construction, an undirected graph is used, while ,for GraphSage, a directed graph is used. Between these two strategies the directed graph seems to be the better choice because the original graph is directed and weighted to represents inhibition or stimulation of a gene to another in a pathway, this consideration will find additional proofs when the model is built and confronted with the GCN results(how the loss function behave when creating the models through the epochs) in figure 5.

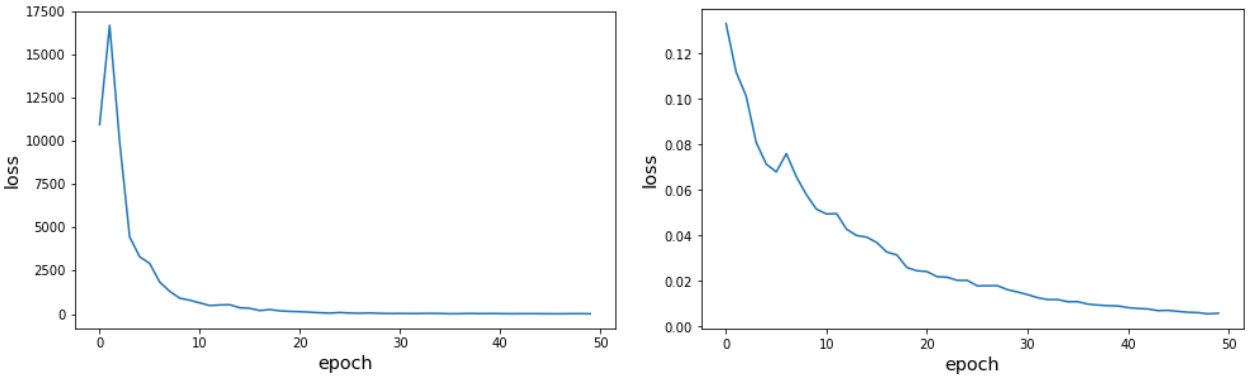
Algorithm 3: GNN model

```
generator = FullBatchNodeGenerator(graphWithFeatures, method="gcn")
gcn = GCN(
    layer_sizes=[16, 16],
    activations=["relu", "relu"],
    generator=generator,
    dropout=0.5
)
x_inp, x_out = gcn.in_out_tensors()
predictions = layers.Dense(controlsDK.shape[1],
```

```
                                    kernel_initializer='normal')(x_out)
model = Model(inputs=x_inp , outputs=predictions)

model.compile(
    optimizer=optimizers.Adam(learning_rate=0.0001),
    loss=genes_loss_with_closure(edges=tensorEdgesTest),
    metrics=["acc"])

history = model.fit(
    train_gen,
    epochs=50,
    batch_size=controlsDK.shape[0],
    verbose=2,
    shuffle=False
)
```



(a) GCN

(b) GraphSage

Figure 5: Loss progress for GCN and GraphSage during the epochs for controls

As could be seen in figure 5, the loss function for GCN decreases overall but maintains higher values and fluctuates too much, it also does not summarize the structure information of the meta-pathways graph so this model is not very good for the task of genes embeddings with the structure defined in this research. The GraphSage model, on the other hand, maintains low values and will produce better results that preserve the graph structure information of the meta-pathways.

After fitting the model to create the embeddings that minimize the loss(for the defined epochs), we have a model that could be used to predict the genes embeddings, but for optimization and best choice, the next chapter 2.2 will use the embeddings created by the **GraphSage** model that creates better embeddings.

The GNN model that uses GraphSage takes 10 nodes at depth 1(5 in-edges, 5 out-edges) and 4 nodes at depth 2(2 in-edges,2 out-edges), results on the model could differ if the number of

nodes or layers changes, right now this is not the main task of this research and the results do not seem to change for different layers for the GraphSage model.

Algorithm 4: GNN model for GraphSage and fitting with previous loss function

```
batch_size = controlsDK.shape[0]
in_samples = [5, 2]
out_samples = [5, 2]
trainGen = generator.flow(controlsDK.index, controlsDK)

graphsage_model = DirectedGraphSAGE(
    layer_sizes=[32, 32],
    generator=generator,
    bias=False,
    dropout=0.5,
)

x_inp, x_out = graphsage_model.in_out_tensors()
prediction = layers.Dense(units=controlsDK.shape[1],
    activation="softmax")(x_out)

model = Model(inputs=x_inp, outputs=prediction)
model.compile(
    optimizer=optimizers.Adam(learning_rate=0.005),
    loss=optimized_lossGNN(tensorEdgesTest=tensorEdgesTest,
        constantTensor=constantTensor),
    metrics=["acc"]
)
history = model.fit(
    trainGen, epochs=50,
    batch_size=batch_size,
    verbose=2,
    shuffle=False,
    callbacks=[cp_callback]
)
```

The *in-samples* and *out_samples* lists are the number of nodes to consider in the GraphSage model for the first and second layer, that is 10 nodes in the first layer(5 in and 5 out) and 4 nodes in the second layer(2 in and 2 out). Different models could be obtained by changing these parameters to get more information about the structure for every node.

These parameters were maintained in all the models for coherence and consistence between the models.

The training is done in a keras model and will save a checkpoint after every epoch since, for some models, the computation of the current epoch execution could take a lot of time depending on the hardware.

The accuracy metric will not be reported here but was used to see how the final embeddings were different from the original genes expressions.

As could be seen in figure 5, the loss function for GraphSage decreases overall and in a better way than the loss for the GCN model.

After the creation of the model, the prediction of the genes embedding can be done with the following lines of code:

Algorithm 5: Genes embedding prediction with the GraphSage model

```
all_mapper = generator.flow(controlsDK.index)
all_predictions = model.predict(all_mapper)
all_predictions = all_predictions.astype("float64")

genesEmbeddings = pd.DataFrame(all_predictions,
                               index =controlsDK.index,
                               columns =controlsDK.columns)
```

Where controlsDK are the genes with their identifiers as indexes and model is the GraphSage model(or the GCN).

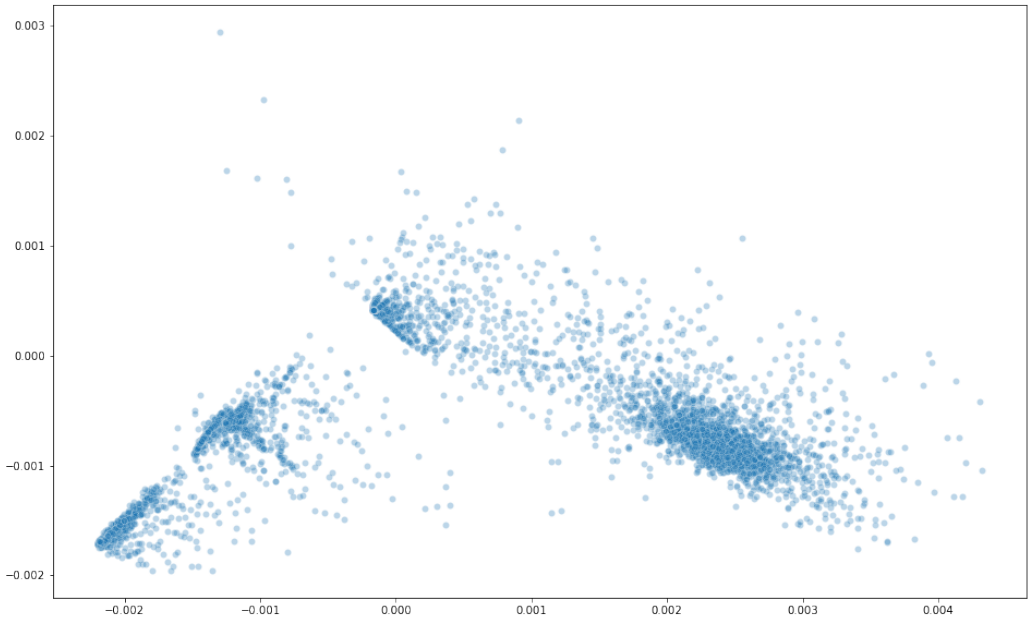


Figure 6: Visualization of genes embeddings in 2D

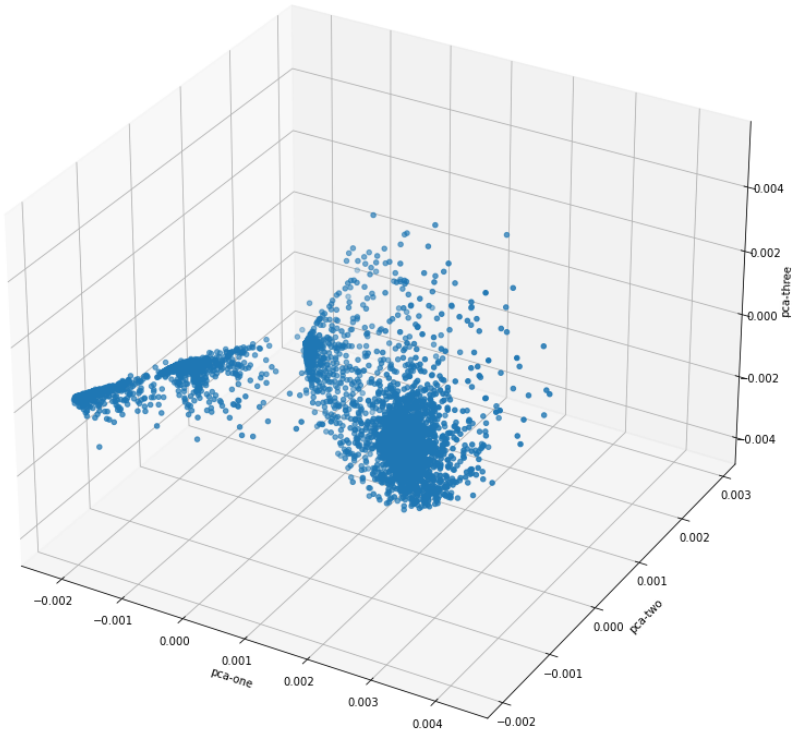


Figure 7: Visualization of genes embeddings in 3D

	0	1	2	3	4	5	6	7	8	9	...	103	104	105	106	107	108	109	110	111	112
2	0.007786	0.008302	0.010081	0.009973	0.009758	0.007783	0.009583	0.008450	0.008097	0.008481	...	0.008452	0.008252	0.009922	0.009459	0.008165	0.009406	0.007620	0.008498	0.010242	0.009860
3	0.007874	0.008274	0.010160	0.009925	0.009824	0.007807	0.009602	0.008594	0.008329	0.008302	...	0.008350	0.008152	0.009959	0.009577	0.008199	0.009360	0.007467	0.008464	0.010236	0.009646
7	0.007804	0.008252	0.010091	0.009956	0.009855	0.007742	0.009620	0.008464	0.008166	0.008371	...	0.008491	0.008189	0.009796	0.009546	0.008178	0.009366	0.007491	0.008508	0.010152	0.009663
8	0.007824	0.008215	0.010088	0.009966	0.009843	0.007775	0.009621	0.008511	0.008175	0.008403	...	0.008425	0.008175	0.009820	0.009540	0.008202	0.009365	0.007558	0.008447	0.010207	0.009686
10	0.007822	0.008201	0.010088	0.009961	0.009843	0.007764	0.009618	0.008508	0.008151	0.008405	...	0.008436	0.008180	0.009809	0.009538	0.008219	0.009373	0.007578	0.008437	0.010205	0.009702
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
20526	0.007838	0.008162	0.010587	0.010033	0.009856	0.007605	0.009502	0.008600	0.008129	0.008243	...	0.008537	0.008085	0.009777	0.009570	0.008351	0.009394	0.007616	0.008649	0.010194	0.009538
20527	0.007836	0.008210	0.010095	0.009954	0.009835	0.007783	0.009617	0.008529	0.008186	0.008393	...	0.008408	0.008178	0.009843	0.009546	0.008215	0.009366	0.007567	0.008426	0.010220	0.009701
20528	0.007839	0.008212	0.010074	0.009958	0.009831	0.007808	0.009609	0.008522	0.008203	0.008410	...	0.008400	0.008189	0.009849	0.009519	0.008189	0.009365	0.007541	0.008441	0.010211	0.009692
20529	0.007825	0.008249	0.010127	0.009961	0.009849	0.007753	0.009626	0.008523	0.008193	0.008362	...	0.008423	0.008158	0.009847	0.009583	0.008204	0.009371	0.007535	0.008448	0.010210	0.009681
20530	0.007823	0.008240	0.010125	0.009955	0.009855	0.007733	0.009644	0.008528	0.008191	0.008344	...	0.008433	0.008138	0.009827	0.009599	0.008230	0.009359	0.007560	0.008446	0.010212	0.009668

Figure 8: Sample genes embeddings

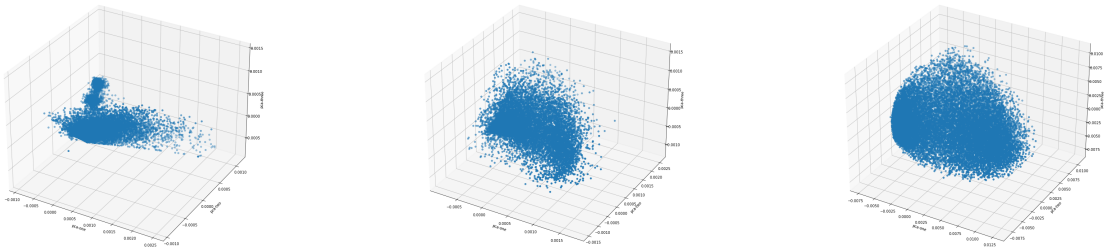
An example of genes embeddings computed after the model fitting and prediction for all nodes-genes could be seen in 8, a visualization of the genes embeddings(with dimensionality reduction on the embeddings to be 2-3 dimensional data, PCA was used for this task) is provided in figure 6 and 7.

In the visualization in figure 7, four clear clusters seem to form, so there seems to be some kind of structural clustering of the genes that have the same functions or are often expressed together in people that have no tumour.

2.1.5 Final genes embeddings for tumours

The final embeddings are predicted from models that have been trained with 113 patients (the same number of patients as controls) sampled from different groups grouped by the type of tumour or by the stage of the tumour. Even after the sampling, the different models were coherent for most of the types of tumours (LumA tumours models were really sparse in the embeddings that they created).

The code used to create these models is really similar to the one presented in 2.1.4. The final embeddings are not presented but will be visualized in 3D (with dimensionality reduction) to see if there are some resulting clusters or some other relation in the final predictions.

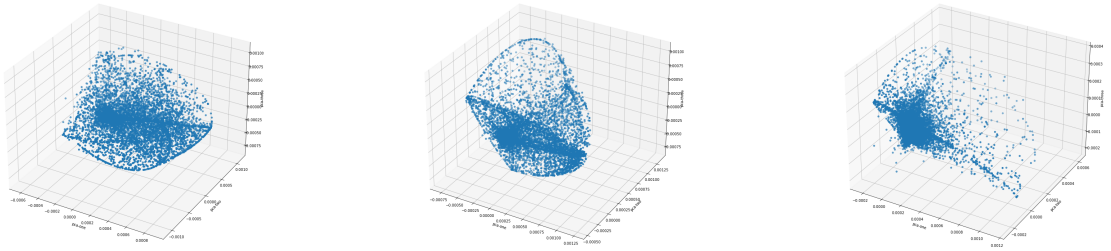


(a) Basal (b) LumA (c) LumB

Figure 9: Visualization of genes embeddings: type

The embeddings for the basal type tumours are quite different for 3 structures in the 3D space after the dimensionality reduction. The embeddings for LumA are similar to the embeddings of LumB since they both form a kind of dense cloud that is fundamentally different from the basal case.

These differences in the embeddings will influence the pathways embedding in the section 2.2 and the final comparison of these embeddings will shed some light on the different influences of every pathway in the overall behaviour of a tumour.



(a) Stage I (b) Stage II (c) Stage III

Figure 10: Visualization of genes embeddings: stage

It can be seen visually that the embeddings grouped by types form some clouds while the embeddings grouped by stage form some structures because the expressions of patients in the same stage were not related that much and the final results show that the structures resulting from the PCA (dimensionality reduction) are at the limits of a quarter of a sphere, since the final genes from the patients in the stage were not much related. From these results, it can be seen that the embeddings based on the stage of a tumour are not that useful.

The type or stage of tumours that were not populated enough were not considered in the final analysis but different models were created and can be seen in the repository in the notebook **AdditionalModels**, these models are not coherent since the number of patients is not the same as the models described in this research. Tumour types and stages that fall into these additional models are:

- **Normal** tumour type
- **Her2** tumour type
- **Stage IV** tumour stage
- **Stage X** tumour stage

These additional models are called **inconsistent**.

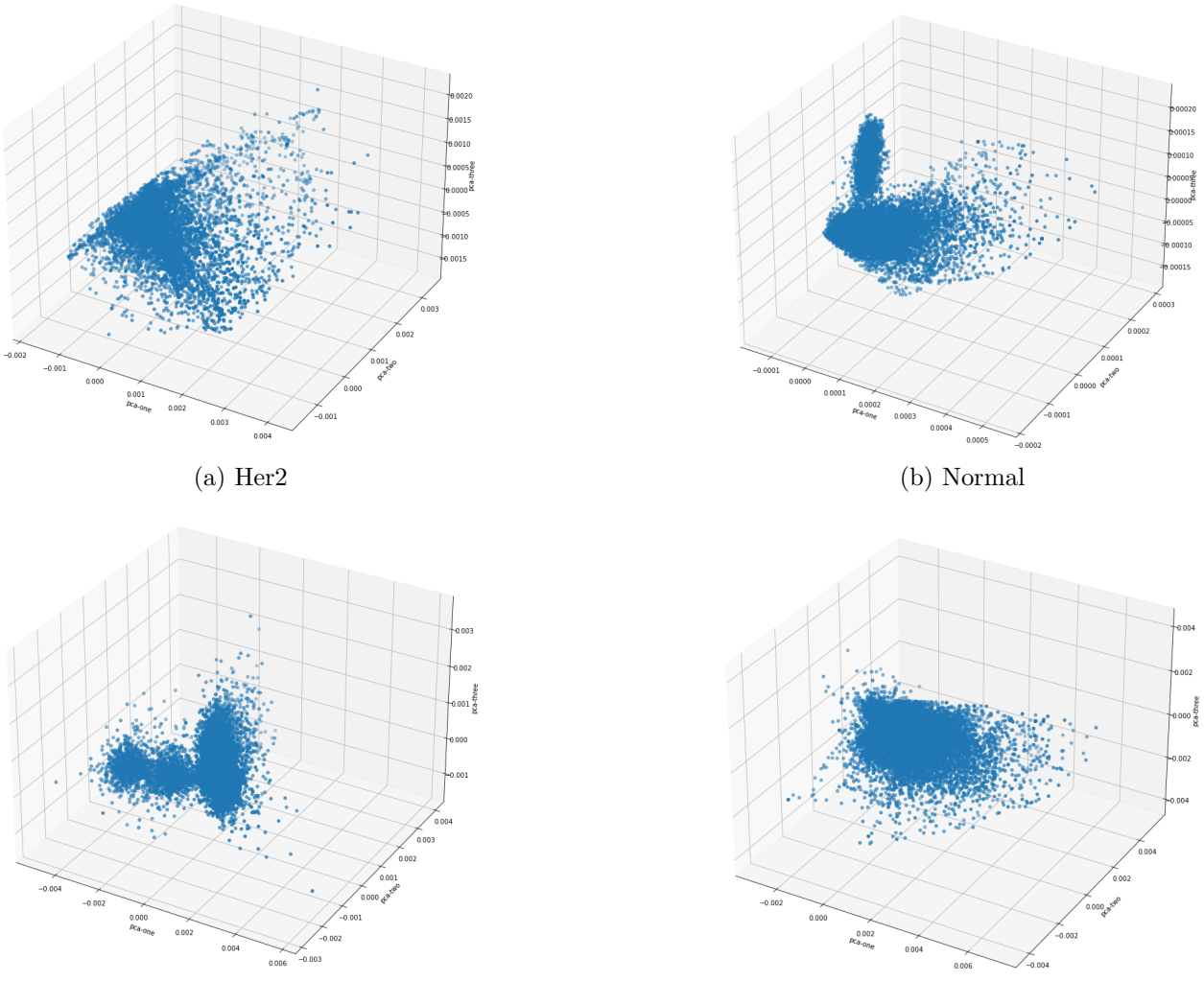
Definition 2.1. (inconsistency of a GNN model): a model is said to be **inconsistent** with other models if the dimension of the column space or the dimension of the final embeddings are different than the ones of the other models. A set of GNN models is **inconsistent** if it contains an inconsistent model.

Definition 2.2. (partial incoherence of a GNN model): a model is said to be **partially incoherent** if the model was created from a heterogeneous group of patients, which means

that one or more patients were not of the same category. If a group is completely incoherent, that means that all the patients are of different groups among each other. The coherence of a group can be measured by the entropy of the group similar to decision trees. Coherence is a term similar to **purity** in decision trees.

In the *AdditionalModels* notebook, there are also the models for the gene embeddings that consider all of the patients in a group, vice-versa there are some models that expand the embeddings by having more dimensions than the number of patients available. These embeddings could be used to get more information about the structure but the problem of the dependence of these embeddings on the number of patients in the original data remains as described in 2.1.7.

A visualization of these additional embeddings can be seen in figure 11



(a) Her2

(b) Normal

(c) Stage IV

(d) Stage X

Figure 11: Visualization of genes embeddings: inconsistent models

2.1.6 Further optimization and ideas

To further improve the model, **probability calibration** could be done to calibrate parameters and get a more flexible GNN to predict embeddings for other measures that shares the same structure behind.

The data at hand was normalized, other experiments were done with not normalized data (with different techniques) but, overall, the results were almost the same, only the scale of the final embeddings and the density were different.

In case of dynamic graphs (where new genes are inserted in the graph, and these genes have some embeddings that could be used to inference some connections with other genes), there could be a problem with the overall model because the prediction of the edges

There are some problems with this kind of model because the overall strategy aims at the minimization of the loss function defined in 4, but the models will probably underestimate the values because it will minimize the loss, so almost all genes embeddings will be close to 0 with little difference between them. This problem will be even more present in the pathway embeddings that will be commented on in section 2.2.

Another problem with the overall structure is that the relation between these genes embeddings reflects the relation in the original data(even with the GNN model), so if there is no relation(or linear/sub-linear relation within the data), the model will not work really well to estimate embeddings, because there is no way of knowing if two genes are really close or not. This problem will be seen in 2.2 and a discussion in 4 will be seen that takes into account all the possible problems with this research and with the data at hand.

2.1.7 Criticism of the model

GNN are not suitable for some of the use case that could take advantage of the theory behind them, especially for one instance that could be really important to this project: The prediction of patients embeddings dynamically. This instance is not doable for GNN since the network needs a set of patients to build the features and the feature prediction will be directly dependent on these preliminary features(the number of patients and the expression of the gene for that patient). In this setting, other than the number of features (patients), even the order of the features will influence the final results. The model of genes embedding right now will take into account and will be strongly dependent on the following characteristics:

- **The number of features of every gene**, since the features are patients, the final results will be directly influenced by the number of patients. To compare the embeddings for two different expression matrices, the column space (patients) should have the same dimension among the expression matrices, which means that a **subset** or **data augmentation** should be done beforehand, and that leads to choosing which patients to take for the embeddings and on what criterion the sampling of these patients should be done(the patients in the same group that are near each other and cannot be con-

sidered outliners, the patients that are closer to the controls, based on some statistics, etc.).

- **The values of the expression set (for the set of patients decided)**, like the number of patients, the expression of the patients will influence the final product obviously. The consideration about sampling the patients are the same for this consideration and are also expanded on the fact that some genes could be not significant (co-expression of these genes could be insignificant and will lead to no additional information and worsening of the model) enough.
- **The order of the patients in the expression matrix** is really important, if the order is changed the final results will change too. Considerations about the comparison with other embeddings are the same as before, that means that the order could change the significance of the research(some order of the patients of the first expression matrix could lead to a better model with comparable embeddings).
- **The final layer of the GNN model** is the one that will establish the dimension of the final embeddings, in stellargraph it is treated as a neural layer. The number of feature in output should be the same for different groups of patients, it also should have the same size as the patients to mantain coherence in the information that could be encoded in the embeddings. Different dimensions for different embeddings of two or more groups of patients will lead to incomparable vectors that have different information.
- **A predefined structure for the graph (metapathways) and the GNN that cannot be changed during execution**, this is also related to the co-expression of a group of patients, since different groups with different pathologies could have different significant genes and also genes significant to all the groups. To make a distinction between these considerations, the graph should be dynamic and should be able to not consider some nodes when predicting the embedding(after the training phase). This is not possible with the implementation of stellargraph and is also not considered in the literature.
- **The loss function** defined that will define the final results and performance, right now they are not that great both for the genes embedding and the pathway embedding that will be analysed in the following section.

The desired behaviour would be to train the network with a similar approach described in the GNN literature(aggregation of the nodes embeddings) to get structural information from the nodes of the graph, but the predictions should take some patients as an input(not necessarily the same number of patients as the training) and will give the embedding for the genes for the patients described, and these embeddings should not have a fixed amount of features in the output but should be programmable in the number of features for a gene to compute.

With this behaviour, the classification of patients could be done quickly since the genes will have some numbers related that will define the role of the patients in the GNN network for

the metapathway. Pathway embeddings will also be much more quick and flexible since the model that will be described in the following sections will be directly implemented (with other techniques but with the same objectives) in the tool provided.

2.2 Pathway embedding

As already introduced in 2.1.2 section and at the start of 2, the pathway embedding is done with a GCN-like formula that is 7, and the resulting model to predict this pathways embeddings are trained with the loss 8 that minimizes the distance between pathways with a big intersection of genes.

2.2.1 Gradient descent

To find the parameters of the model, Gradient Descent is used among all data and it is implemented in tensorflow [11].

To get the betweenness of the nodes in the graph used in 8, the *networkx* package is used.

The first thing to do is to prepare the data to compute the current pathway embedding, for this task firstly the genes are filtered through a hash table that takes the genes embeddings that belong to a pathway:

Algorithm 6: Pre-processing of the data

```
betweennessList = nx.betweenness_centrality(Gcontrols,
    normalized = True,
    endpoints = False)

def gene_weight(geneIndex,dictBetw):
    if(geneIndex in dictBetw):
        return dictBetw[geneIndex] + 1.0
    else:
        return 1.0


@tf.function( experimental_relax_shapes=True)
def getPathwayEmbeddingEnhancedFinal(partialsVec_tensor,Wmat):
    W_tensor = Wmat
    trasformedPartialVec = W_tensor @ partialsVec_tensor
    finalVec = tf.map_fn(lambda values: tf.map_fn(tf.nn.leaky_relu,values),
        trasformedPartialVec)
    return finalVec

@tf.function
def combine(x, y):
  xx, yy = tf.meshgrid(x, y, indexing='ij')
  return tf.stack([tf.reshape(xx, [-1]), tf.reshape(yy, [-1])], axis=1)
```

```
def compute_not_equal_path(pathways,coup):
    if(coup[0]!=coup[1]):
        return len(set(pathways["nodes_set"][coup[0]]) &
                   set(pathways["nodes_set"][coup[1]]))/
               len(set(pathways["nodes_set"][coup[0]]) |
                   set(pathways["nodes_set"][coup[1]]))*10 +1
    else:
        return 0.0
def precompute_pathways_weight(pathways,coupPath):
    return list(map(lambda coup: compute_not_equal_path(pathways
            ,coup)
        ,coupPath))


@tf.function
def pathway_embedding_norm(pathwayEmbedding, couple):
    path1 = pathwayEmbedding[couple[0]]
    path2 = pathwayEmbedding[couple[1]]
    return tf.sqrt(tf.reduce_sum(tf.square(path1 - path2)) + 1.0e-12)

@tf.function
def loss_opt(pathwaysEmbeddingTensor,pathwayCoup, weightTensor):
    test = tf.map_fn(
        lambda couple:
            pathway_embedding_norm(pathwaysEmbeddingTensor,couple),
            pathwayCoup,
            fn_output_signature=tf.float64)
    return tf.reduce_sum(tf.math.multiply(test,weightTensor))
```

The next step is to get the partial vector that will be used for every computation of the final pathway embedding to speed up the process, this vector is the sum of all the genes in the pathway divided by the cardinality of the pathway, that is $\sum_{u \in P_i} \frac{z_u}{|P_i|}$ as seen in 7, this partial vector is shared among all steps of the gradient descent.

Algorithm 7: filtering and preparing the constant tensor to speed up computation

```
def getPartialVec(Nodes):
    listatest =  Nodes.split(";")
    listatest = list(map(int, listatest))

    keys_tensor = tf.constant(listatest,dtype=tf.int64)
    vals_tensor = tf.ones_like(keys_tensor)  # Ones will be casted to True.

    table = tf.lookup.StaticHashTable(
        tf.lookup.KeyValueTensorInitializer(keys_tensor, vals_tensor),
        default_value=0)  # If index not in table, return 0.

    def hash_table_filter(index, value):
        table_value = table.lookup(index)  # 1 if index in arr, else 0.
        index_in_arr =  tf.cast(table_value, tf.bool) # 1 -> True, 0 -> False
```

```
        return index_in_arr

    tfFiltered = tfExprMatrixWeighted.filter(hash_table_filter)
    tfFiltered = tfFiltered.map(lambda index,value : value).
        reduce(np.float64(0.0),lambda x,y: x+y)/
        (len(listatest) * normalizer_constant)
    return tfFiltered.numpy()



geneWeights = list(map(lambda value: gene_weight(value,betweennessList),
    list(genesEmbeddings.index)))
exprMatrixWeighted = genesEmbeddings.mul(geneWeights,axis=0)
tfExprMatrixWeighted = tf.data.Dataset.from_tensor_slices((exprMatrixWeighted.index,
    exprMatrixWeighted.values))

pathways["partialsVectors"] = pathways["nodes"].map(getPartialVec)
pathways["partialsVectors"] = pathways["partialsVectors"].
    where(pathways["partialsVectors"].
            map(lambda value: not not value.shape))
```

To compute the final embedding, the W tensor(firstly a diagonal matrix, after what is estimated by the gradient descent) is multiplied by the partial vector and the result is passed to a non-linear function(leaky-relu in this case) that transforms values of the vector point-wise. This function will also be used to compute the gradient descent to estimate the W matrix.

Algorithm 8: Code to compute final embeddings

```
W = np.diag(np.full(len(pathways["partialsVectors"][0]),0.001))
W_tensor = tf.Variable(W,dtype=tf.float64)
npPartial = np.stack(pathways["partialsVectors"]).transpose()
partialsVectors_tensor = tf.constant(npPartial)
finalVectors_tensors = getPathwayEmbeddingEnhancedFinal(partialsVectors_tensor
    ,W_tensor)
pathways["pathwayEmbeddings"]= finalVectors_tensors.numpy().
    transpose().
    tolist()

pathways["nodes_set"] = list(map(lambda nodes: nodes.split(";") ,
    pathways["nodes"]))
pathwayPairs = combine(tf.constant(list(pathways.index)),
                       tf.constant(list(pathways.index)))

indexOfPathways = dict()

for i in range(0,len(pathways.index)):
    indexOfPathways[pathways.index[i]] = i

tensorPathways = tf.map_fn(
    lambda row: tf.map_fn(
        lambda element:indexOfPathways[element.numpy()],
```

```
            tf.gather(row, [0, 1]).numpy()
        ), pathwayPairs)

    precomputed_weight_pathways = tf.constant(precompute_pathways_weight(pathways,
                    pathwayPairs.numpy()
                ),dtype=tf.float64)
```

Lastly, the computation for gradient descent is done using the tensorflow framework as defined in [11] and with the previous auxiliary function defined.

Algorithm 9: Gradient descent computation

```
W_numpy = np.diag(np.full(len(pathways["partialsVectors"][0]),0.01))
W = tf.Variable(W_numpy,dtype=tf.float64,name = "W")
npPartial = np.stack(pathways["partialsVectors"]).transpose()
partialsVectors_tensor = tf.constant(npPartial)
loss = tf.Variable(0.0,dtype="float64")


loss_progression = np.array([loss])
loss = tf.Variable(0.0,dtype="float64")

learning_rate = 0.000001
for i in range(0,20):
    with tf.GradientTape(persistent=True) as tape:
        tape.watch(W)
        finalVectors_tensors =
            getPathwayEmbeddingEnhancedFinal(partialsVectors_tensor,W)
        loss = loss_opt(tf.transpose(finalVectors_tensors),
                tensorPathways,
                precomputed_weight_pathways)
    dl_dw = tape.gradient(loss, W)
    W.assign_sub(learning_rate * dl_dw)
    loss_progression = np.append(loss_progression,loss.numpy())
```

A the end of the gradient descent algorithm, the model is trained and ready to predict pathways embeddings.

(a) controls pathway embeddings 2D

(b) controls pathway embeddings 3D

Figure 12: pathway embeddings for controls visualized

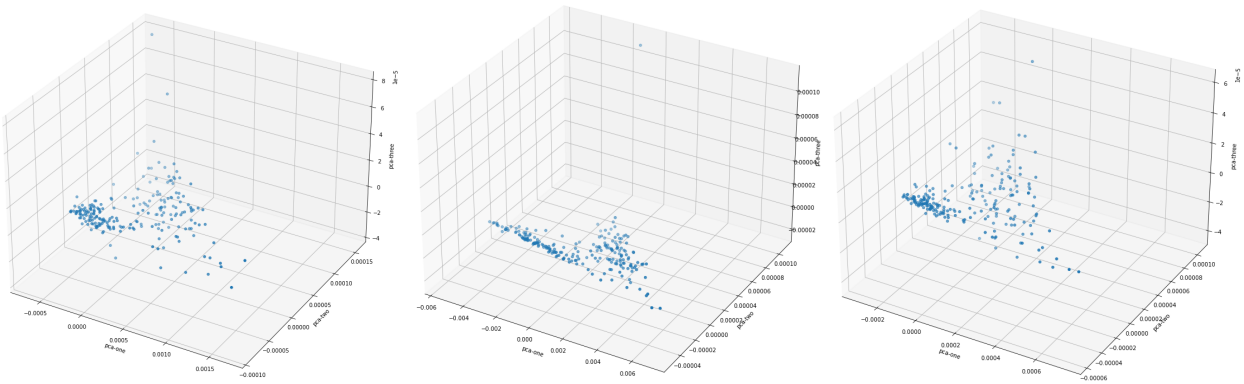
The embeddings are really small and seem to be close to one another. This consequence is related to what was introduced previously about the consequence of non-linearity of the data and the minimization of the 4 and especially 8, since the loss is linearly dependent of a scalar multiplied by W matrix used in the model to transform the pathway embeddings.

In figure 12, the pathways embeddings were treated with dimensionality reduction to help visualize them(PCA was used, also T-SNE but the results were really similar). Two clusters could be seen among the pathways along with a lot of outliers. For the non-normalized data, the clusters are in the same places as the one for the normalized data, but the density seems lower in the left cluster for the non-normalized data, while it seems lower in the second cluster for the normalized data.

2.2.2 Final pathways embeddings and comparison

The code used to get the pathway embeddings for the tumours is the almost the same as the one described in 2.2.1, the only thing that changes is the meta-pathway (since the genes for tumours are not all the same as genes for controls), so the betweenness is recomputed.

Only the pathway embeddings grouped by types will be used, since the results for the genes embeddings for patients grouped by stages were inconsistent and unusable.



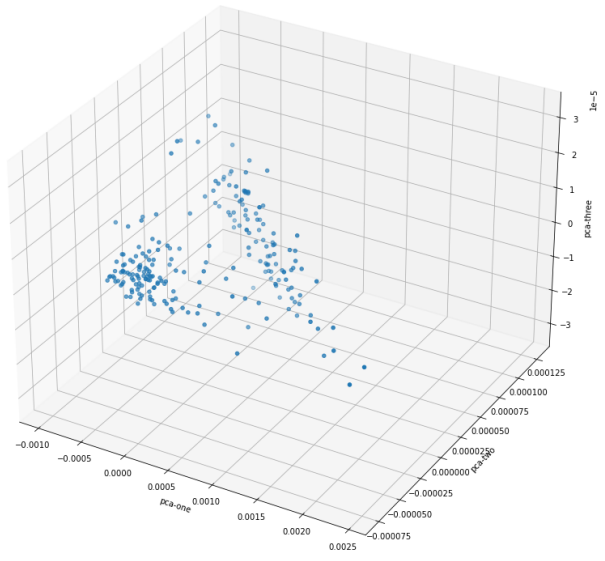
(a) Basal (b) LumA (c) LumB

Figure 13: Visualization of pathway embeddings

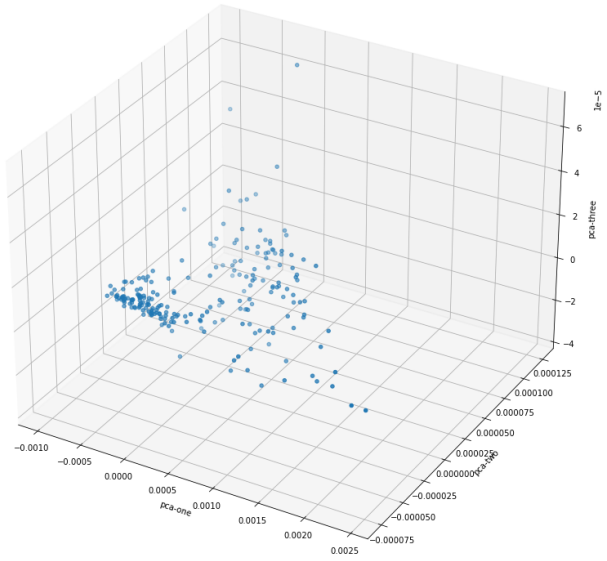
The embeddings form some visual patterns that are similar between them, as can be seen in figure 13, where the pathways form some clusters and some sparse clouds in different parts of the 3D space but they all have similar structures.

The controls embeddings form one dense cluster and one sparse cloud, and these are aligned to each other. Some pathway embeddings are far from these two clouds, this means that that pathway is probably more important in the biological process since the genes embeddings (and the expression as well) have high values. The embeddings for tumours groups result in similar structures (one dense cluster and a sparse cloud with some pathways embeddings outside of these two structures), but results are shifted, more sparse, distributed in other forms(like the LumA, where the dense cluster is distributed in a sort of line) and rotated.

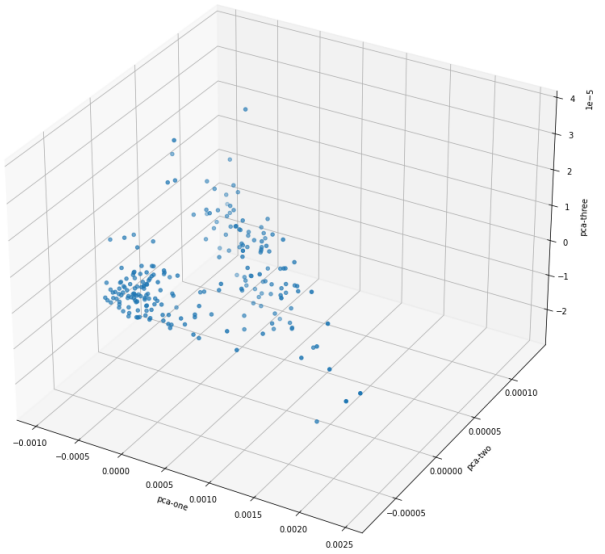
Pathway embeddings were also computed for most of the patients of LumA to see if the final results were similar, this can be seen in figure 14



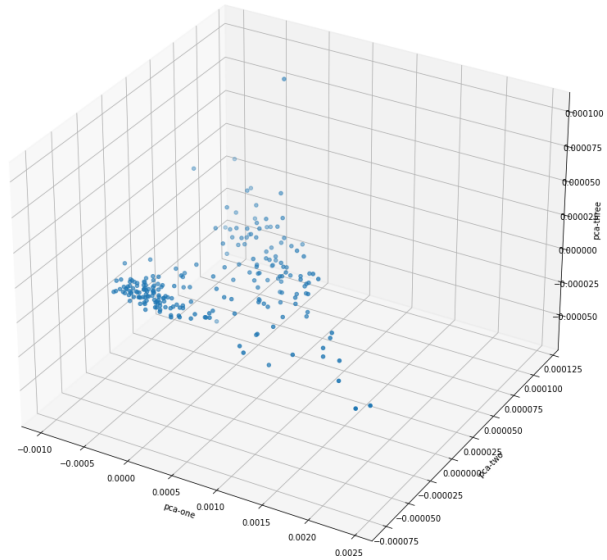
(a) 1-113



(b) 113-226



(c) 226-339



(d) 339-452

Figure 14: Visualization of genes embeddings: stage

The embeddings for LumA groups seem to have the same kind of structures, which means that the models and the overall methodology are significant and patients with the same type of tumour will probably produce similar embeddings.

3 Experimental analysis

The final embeddings (for the consistent models, that is controls, basal, LumA, LumB) comparison can be seen in figure 15, figure 16 and figure 17. The other embeddings showed previously were not used since they were inconsistent.

The comparison is done by taking the Euclidean distance from the pathway embeddings of two groups and plotting this distance in a scatter-plot to see if the results are distant enough (in the case of two different groups) or close enough (in the case of embeddings of subgroups of the same tumour type, that is LumA for this research). This kind of plot is also used to see which pathway is more important for the classification of a group than another pathway, distances that are high enough imply that the pathways involved are significant for some biological process and, vice-versa, distances that are really low means that the pathway reacts and is involved in similar ways to different pathologies.

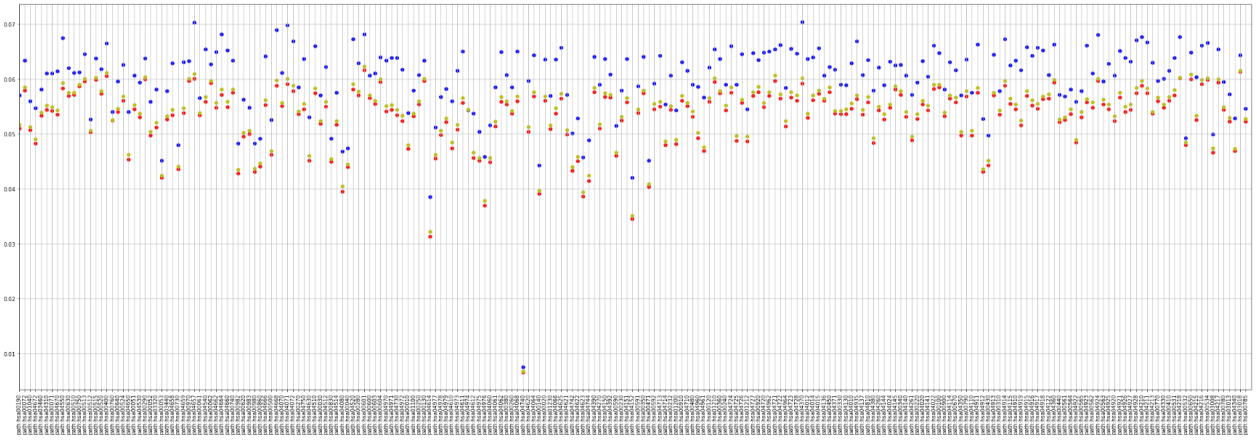


Figure 15: Euclidean distance of pathway embeddings for controls : **blue** points are distances control-basal; **yellow** points are distances control-LumA; **red** points are distances control-LumB

The distances from the controls pathway embeddings from the different types of tumors speak for themselves, the results are high enough for most pathways (e.g. *hsa04740* is a pathway that is not involved in the process of tumour probably since the pathway embeddings are not that far from each other).

The distance from the basal pathway embeddings is the greatest, that could mean that the tumour type is more active than the other tumours and the overall meta-pathway is more stimulated. Other conclusion could be inferenced, but a comparative analysis with research on pathways and other types of models could be done, but this is not the scope of this project.

Distances from LumA and LumB pathways embeddings seem to be similar, that could be explained by the types of breast cancer that the two tumours are part of, that is luminal A and luminal B. Since these two types of breast cancer share some common characteristics,

the consequence of this similarity could be that the two embeddings are quite similar between each other and that is what is observed with the final results.

Another consideration could be that the distances seem to have some pattern, that is:

- **basal** basal breast cancer pathway embeddings are always further from the controls embeddings than the LumA-LumB pathways embeddings. That could probably mean basal's type of breast cancer is more active in the meta-pathway than the other two.
- the **trend** of the distances from every group is almost the same for every pathway, e.g. pathway *hsa04610* distance is less than pathway *hsa:04973* and *hsa:04979* for every distance group. There are some cases when this is not true (e.g. *hsa:04974* or other pathways) but the overall behaviour is similar to what was described.

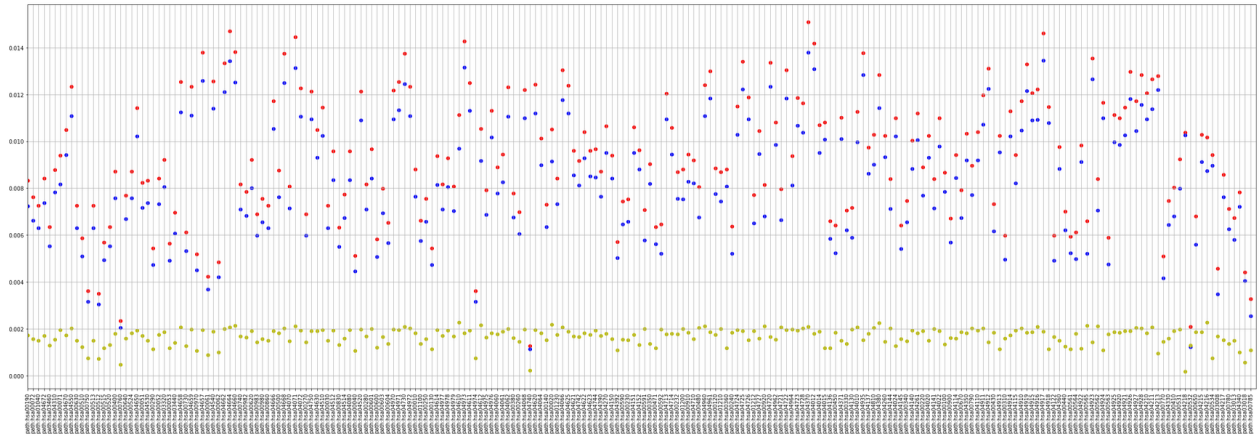


Figure 16: Euclidean distance of pathway embeddings for tumours : **blue** points are distances basal-LumA; **yellow** points are distances LumA-LumB; **red** points are distances basal-LumB

The comparison of different types of breast cancer with Euclidean distances reported in figure 16 shares some behaviour with the previous comparison. The distances of LumA-LumB are at a different scale than the basal distances and seem to be low, adding to the pieces of evidence of what was stated before about the similarity of the two behaviours of the luminal breast cancers.

Distances from the basal pathway embeddings were expected to be far enough from the luminal types so the outcomes are the same as the ones already said before.

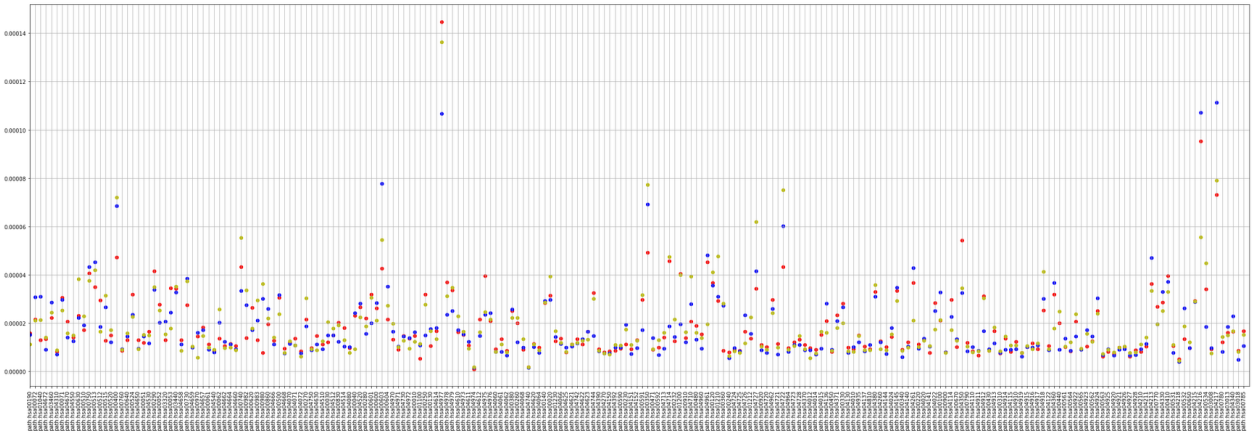


Figure 17: Euclidean distance of pathway embeddings for LumA subgroups

The distances of the subgroups of LumA pathway embeddings are close to 0 (except for some pathways that could have different behaviour for different patients, the final distance is very close to zero nonetheless). As already introduced before, this means that the embeddings generated by the GNN models are consistent for these groups of patients and the results will almost be the same independently of the sampling of subgroups of the patients.

3.1 Classification of controls or tumours types

There are a lot of use cases that could be implemented with the pathway embeddings obtained from the GNN models, classification could be one field of interests but there are some technical quibbles that need to be specified and considered when using the embeddings for statistical prediction and classification. It will be seen that classification for these topic is not feasible with what is presented here, but some applications and possibilities will be presented in 4.

The model used in this section is **Random forest** from the R library *randomForest*. The final model is trained with 75% of the data and the rest of the pathway embeddings is used as a test set, the features used to train the model are the embedding and the pathway name. The predicted feature is the type of the pathway(control or a type of breast cancer).

The final model performance can be seen in figure 18

```
 randomForest(formula = type ~ ., data = training, importance = TRUE,      proximity = TRUE)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 10

        OOB estimate of  error rate: 0.17%
Confusion matrix:
        Control Basal LumB LumA class.error
Control     172     0    0    1 0.005780347
Basal         0   166    1    0 0.005988024
LumB          0     0  174    0 0.000000000
LumA          0     0    0  681 0.000000000
```

Figure 18: Random forest model

The test pathway embeddings have the following confusion matrix in figure 19

predictedType	Control	Basal	LumB	LumA
Control	53	0	0	0
Basal	0	59	0	0
LumB	0	0	52	2
LumA	0	0	0	221

Figure 19: Test results confusion matrix

To be more significant, the number of patients and types of breast cancer should be even more wide to get better models and to have a lot of results that will be used to train better models usable in

3.2 Biomarkers

The embeddings generated for a group of patients could be used as a biomarker for the group, that is a measurable indicator of some biological state or condition. The possibility of using the embeddings as measurable indicators comes from the start of this section with the distances from different embeddings, the use cases are the following:

- create a model from the group of patients and use it to create the embeddings. Feed the embeddings to a classifier similar to what was done in the previous section 3.1.
- create the embeddings from the group of patients with a GNN model trained with the group expression data. Compare the generated embeddings with other embeddings generated by a model that was trained with a coherent group of patients and from the coherent group of patients as well, establish a threshold and recommend the types that have the lowest mean distance or another measure of comparison between groups.

The problems with the first approach are that: the embeddings could be inconsistent 2.1 (the number of patients or the number of features in the embeddings are not the same as the ones used to train the classifier); the group that was used to create the embeddings is not homogeneous (not enough patients that are in the same category) and the final results are partially incoherent 2.2. The second approach is the best approach to use for biomarkers.

The approaches discussed and further research will be discussed in the following and final section 4.

4 Conclusions and further research on the topic

The objective of this research was to find embeddings for genes and pathways, both of these tasks were done and analyzed during this research with the help of some external libraries

(stellargraph) and python frameworks to compute gradients and models for neural networks (Keras and tensorflow).

For the embeddings of genes, the GraphSage model was used with really good results with the loss used, also a GCN model was used but with poor results, so the model that has done the embedding was the GraphSage model.

For the embeddings of pathways, a GCN-like model was used to compute the embeddings, and the model was trained with Gradient Descent (the possibility of using a Stochastic Gradient Descent or batch-based optimization techniques should not be possible because the loss function requires that all the nodes for a specific set, that is genes in a pathway, should be in the same batch, if another framework is used to resolve the optimization problem, like a user defined procedure in Spark or Map-Reduce, these problems will not be present).

The research has demonstrated the power of the GNN and embeddings for non-euclidean data, further improvements in the field are welcome, especially in the field of the bio-sciences.

A future work will see how different model created from different genes features (obtained from the expressions of different patients affected by some pathology) will behave in the embedding space by confronting the density of the different embeddings space locations or the clusters that they create.

The weight of the gene used to generate the pathway embeddings could be the betweenness of the single pathway and not of the whole net, because the information of every pathway should be considered for only the nodes that are in the pathway, and the whole information for the whole meta-pathway should be already considered by the genes embeddings since the information that the embeddings carry is also structural of the whole graph.

A different approach could be to map the full expression matrix of features (all patients for every gene) to get an output with a defined dimension of features, but that approach could lead to big differences in the embeddings from expression matrix to expression matrix since the mapping is done on a neural network inside(that does the aggregation operation to get the embeddings and could be seen as a matrix multiplication to get more or less features in the output embedding).

Another problem with the **Graph neural network** approach for this type of problems is that the model is inherently linear in its parts (aggregators and the mapping of embeddings in the recursive steps) so if there are some non-linear relations in the data presented(e.g. in the information gathering of the nodes of the graph), the model will not be able to find them and the embedding will not represent the full information for the structure of the graph.

A thing to take into account when looking at this research is that the genes involved in the two expression matrices were not the same, a more significant research that could be done would be to take the same genes expressions for both the controls and the group that will be compared.

As stated in 1 and also at the end of 2, there are some inherent problems with some use cases and hidden objectives that could be implemented and done with the same logic used in GNN. The structural approach used in GNN could be used to predict new features(embeddings)

that are generated for a single patient and that could be used to do other useful things like classification, decisions, inference and also be a new way of getting useful and significant data from the expressions of a single patient. With GNN, the task of creating a model that takes as an input a single vector(of single elements for every node in the graph) and builds some embeddings for the single vector(a constant or a vector for every gene) is quite difficult since the theory and the various paper that constitutes the GNN is aimed toward training in a completely supervised(where the feature generated and used for training are fixed) or a completely unsupervised(where the embeddings are generated directly on top of the features of the nodes) context, so the possibility of having a model that is trained with a whole set of patients, takes as inputs single patients genes expressions and returns as output some constants or vectors for every gene is not possible for GNN. For this reason, the next step should be to create a model framework that also uses some ideas of GNN (getting the structural features of the graph from a local neighbourhood of the node) and gives what is needed to get something useful for the problem seen in the objectives(especially for the usefulness of genes embedding since they are useless aside from the creation of pathway embeddings since they need a group of features/patients that is fixed for the input and the training). This framework should address the criticism of the GNN seen in 2.1.7, and the new model should be more flexible and adaptable to the situation since dynamic networks with variable parameters are important in every field nowadays.

Another burden of these types of AI models is that the model is built with a group of patients that must belong to the same type since the results contain the information about the expressions of the group and some patients that are not in the same group will lead to worse model. This consideration also leads to the fact that these models could only be used with patients with the same type that was used to build the model, and that is a really strict consideration if one of the objective of the project is to build a classifier. Further research could be done with an AI theory that could do the following things:

- build a model independent of the groups of people used to train that uses the GNN theory and is usable for every type of patient;
- build a **differentiator** that will choose the best model for the group of patients by comparing the final embeddings and see what embeddings are the most similar to the embeddings of the original data or does some other computation to differentiate and recommend the best model to chose.

The problem with the first approach is that the whole idea is complex to implement so it is probably not feasible and could lead to unusable results. The second approach would be the better option to get some concrete theory on the use of GNN and such in the field of classification for biology and annexes.

As already mentioned in 2.2.2, a comparative analysis with research on pathways and other types of models could be done and more significant conclusions could be obtained, and that could be a future project that also uses the new model described before to get better results,

better performance, more flexibility and that could also be used as a **Biomarker** to help a lot of fields related to the topic.

Since expressions matrices are **snapshots** of the activity of a cell, the possibility of integrating the research with **temporal networks**[12][13][14] is the next step to get better performance and information about the meta-pathways and is the future of research and innovation since integrating the temporal part is fundamental to model and explain the behaviour of a system.

References

[1] Wikipedia. Gene regulatory pathways. "https://en.wikipedia.org/wiki/Gene_regulatory_network".

[2] CSIRO's Data61. Stellargraph machine learning library. https://github.com/stellargraph/stellargraph, 2018.

[3] Spektral. Spectral documentation. "https://graphneural.network".

[4] Thomas Kipf. Graph convolutional neural network. "https://tkipf.github.io/graph-convolutional-networks/".

[5] William L. Hamilton, Rex Ying, and Jure Leskovec. Graph sage snap. "https://snap.stanford.edu/graphsage/".

[6] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[7] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[8] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[9] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

[10] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.

[11] Tensorflow. basic training loops. "https://www.tensorflow.org/guide/basic_training_loops".

[12] GIORGIO LOCICERO. Temporal networks and applications.

[13] Giorgio Locicero, Giovanni Micale, Alfredo Pulvirenti, and Alfredo Ferro. Temporalri: a subgraph isomorphism algorithm for temporal networks. In *International Conference on Complex Networks and Their Applications*, pages 675–687. Springer, 2020.

[14] Giovanni Micale, Giorgio Locicero, Alfredo Pulvirenti, and Alfredo Ferro. Temporalri: subgraph isomorphism in temporal networks with multiple contacts. *Applied Network Science*, 6(1):1–22, 2021.