1 Overview of Graph-Based Semi-Supervised Learning

1.1 General overview of Graph-Based Semi-Supervised Learning based on

One of the main goals of this thesis is to test and experiment with modifying an already established GSSL method with skeletonization. This will be done using the front-separator algorithm discussed in sec:skeletonization. The two points that seem interesting to test are whether skeletonization makes a difference before propagating labels or after labels have been propagated and tested. To test the library `PyGel` will be used developed with the paper [**DBLP:journals/corr/abs-2102-13303**].

The integration with skeletonization is developed such that a graph at any time can be converted into a `PyGel` graph, skeletonized, and then converted back to matrix form again.

The implementation includes three main features: 1) convert the matrix-based graph to a graph in `PyGel`, 2) perform skeletonization 3) revert to a matrix-based graph.

1.1.1 From matrix graph to `PyGel` graph

To add a node in `PyGel`, the framework only allows for single nodes and then edges to be added at a time. The task then boils down to three steps: 1) adding all nodes, 2) finding all edges, 3) adding all edges. There does, however, arise a slight complication, as `PyGel` requires positions of nodes. To solve this complication, the graph is first implemented in `Networkx`, here the Fruchterman-Reingold force-directed algorithm [**graphpositionalgo**] is used to create arbitrary positions, and the graph is then converted to `PyGel`

To place nodes in the `PyGel` framework requires the use of actual positions, which are not readily available as the data points used to operate in a 28×28 sized space. Two options placing the nodes then exist. Either the original positions of nodes can be downscaled to two or three dimensions (the dimensions `PyGel` supports), or arbitrary positions can be used. As discussed in sec:skeletonization, the front-separator algorithm does not rely on the geometry of the graph but rather the distances between an originator node and the rest of the nodes specified when using the algorithm. Therefore the position of the nodes does not need to be accurate, nor do they need to convey the actual geometry. Downscaling dimensions could be achieved with algorithms such as multi-dimensional scaling recently implemented in python following [**pymde**], but this algorithm takes some computing time and might complicate the procedure. Instead, the Fruchterman-Reingold force-directed algorithm, implemented in the library `Networkx` is used.
To utilize the algorithm, we do, however, first need to establish a graph in `Networkx`, then get the positions, and then convert the graph to `PyGel`.

The nodes can be added as just an id. To follow the index-convention from the matrices, the indices range from $0..n-1$ where n is the number of data points. This is done in $O(|N|)$ time.

The sparsity matrix used to store the graph seen in sec:impoptimize already

acts as a list of edge pairs. As seen in eq:sparseex1, the sparsity matrix already stores the edge pairs as coordinates.

$$G[]matrix0110, G[]sparsity[((0,1),1),((1,0),1)] \tag{1}$$

To represent the undirected structure of the graph, edge pairs are only added once. The task is then to find all unique edge pairs. Traversing through the sparsity matrix, each pair is transformed such that the first coordinate is the lowest value such that $(x,y)|x << y$. The pairs are then inserted into a set. The time to check if a value already exists in a set in python is $O(1)$. The time used is then the time to check all elements in the sparsity matrix list $O(|s|)$ where $|s|$ are items in the sparsity matrix.

Adding the edge pairs to the graph in `Networkx` is done by iterating through the set and used $O(|E|)$ time, where e is the number of edges.

The Fruchterman-Reingold force-directed algorithm is then applied. From [**graphpositionalgo**] we see the running time to be $O(|V| + |E|$ time.

With the arbitrary, the positions can be added to `PyGel`. The time to add nodes is $O(|N|)$, and the time to add edges are $O(|E|)$.

In total, the conversion from a matrix-based graph to a `PyGel` graph is

$$O(|N| + |s| + |V| + |E| + |N| + |E|) \approx O(|N| + |s| + |E| + |V|)$$

1.1.2 From `PyGel` graph to matrix graph

To return to a matrix graph, we simply need the edges in the `PyGel` graph, as each edge is a pair of nodes with ids corresponding to indices in the matrix. To find each pair, we have to loop through each node and find all neighbors. This yields a node-neighbor pair. The runtime for this is $O(|N||n \in N|$ where $|n \in N|$ is the number of neighbors for each node.

Deviating from the convention of converting to `PyGel` the pair is added in both variations with the smallest value as first and last coordinate and inserted into a sorted set. Using sparsity matrices as discussed in sec:impoptimize, the aim is to return a sparsity type matrix again. To build the matrix, we use a row-based list of lists sparse matrix, as this is implemented with the intention of building sparse matrices to then convert to other formats. The optimal way of building it is to presort the pairs row-wise and then adding them. By inserting the edge list all with values one into the sparsity matrix, the graph is rebuild in matrix form.

1.1.3 Skeletonizing

To skeletonize, we first convert our graph into a `PyGel`-graph. Thereafter the front-separator algorithm can be used. The algorithm requires precalculated distances between one node to all other nodes. Luckily this is precisely what the weight-matrix is; a matrix filled with all distances from one node to all

others. The optimal number of distances used will be experimented with in chap:experiments, as although there are some guides for creating good skeletons, there is not necessarily a link between good skeletons and good GSSL results.

When the skeleton is determined, we get a map linking the new skeleton nodes with what nodes have been used to contract the skeleton. This map is essential as that is also the link between the labels of the old nodes and the new nodes. Using the label propagation convention of using the highest count of a label in a node to determine the label, the same is done in skeletonization. If three nodes of label 0 and one node of label 1 have been contracted into a skeleton node, that skeleton node is now a 0. This immediately shows some problems, as information might and probably will be lost. A solution might be to separate the separator into two distinct nodes, one with all labels 0 and one with all labels 1. Because these nodes all have a labeled or unlabelled image tagged to them, the average of the image could then be calculated, and the weight matrix might be recalculated for all nodes. This, however, is pure speculation as it is outside the scope of what could be done in this thesis.

With the graph and labels, the graph is transformed into sparse matrix form again, and the labels are returned.

2 Testing

3 Final algorithm

The final algorithm is implemented in a framework in hopes of getting the best possible result. The crux of this project is the hunch that high dimensional data when downscaled to 2 or 3 dimensions gets "clouds of similarity", which skeletonization is good at dissecting into nodes of importance. The pipeline can be seen in eq:imppipe

`High dimensional data` mds `2-/3-dimensional data` Create graph `Graph` Skeletonize
`Skeleton graph` Label propogate and remap labels to data points `Relabelled data set`