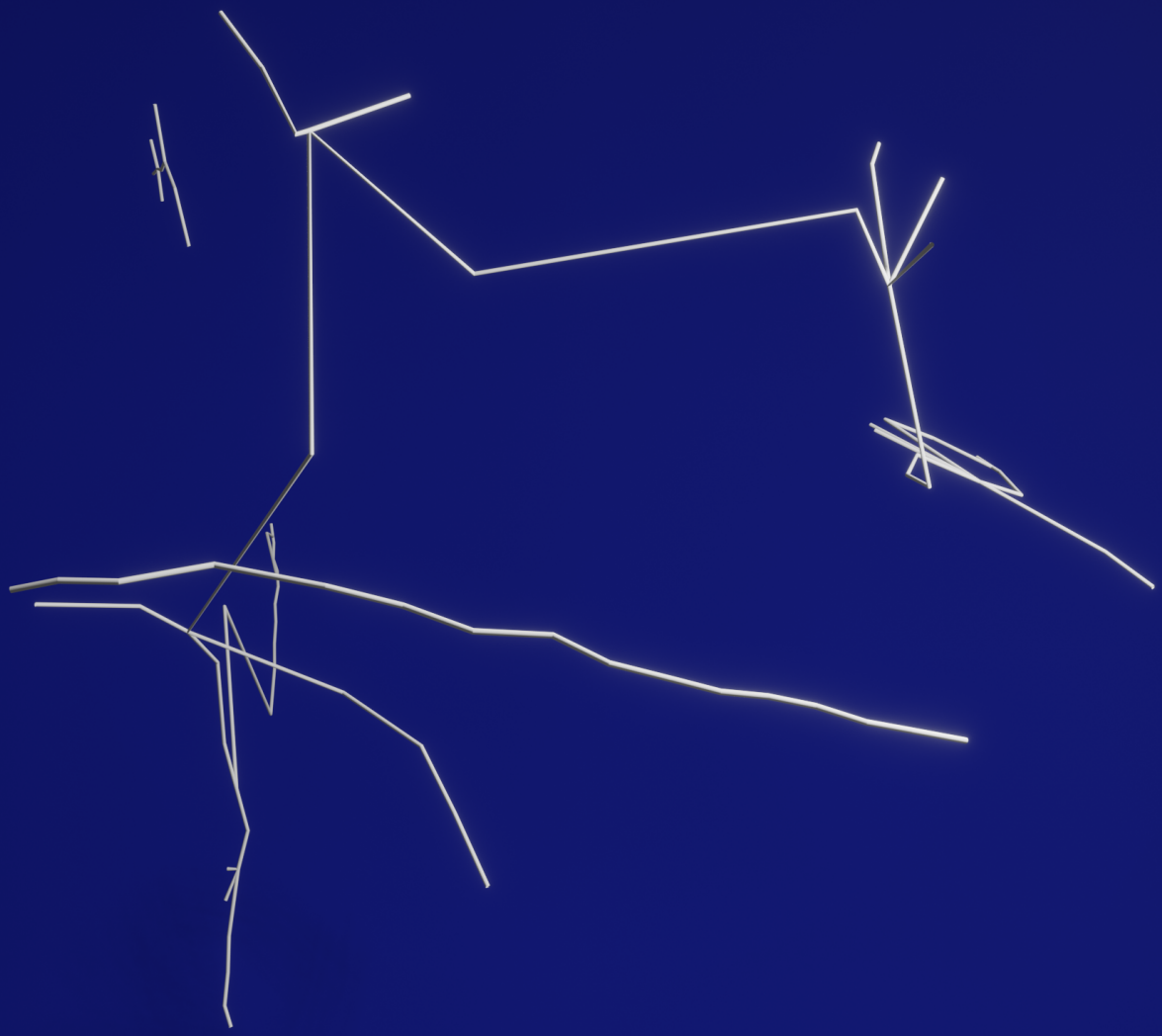
DTU Compute
Department of Applied Mathematics and Computer Science

Graph-Based Methods for Semi-Supervised Learning

Bachelor Thesis



Approval

This thesis has been prepared over three months at the Department of Applied Mathematics and Computer Science, at the Technical University of Denmark, DTU, in partial fulfilment for the degree Bachelor of Software Technology, BSc Eng.

It is assumed that the reader has a basic knowledge in the areas of graph theory and machine learning.

Gustav Lang Moesmand - s174169

.....

Signature

.....

Date

Abstract

Graph-based semi-supervised learning is a promising branch of semi-supervised learning with an already established value in the real world. Its use of the graph structure to infer label information has seen tremendous value. Skeletonization is the theory of compressing graphs to an extreme extent removing noise and keeping the features of the graph present in a minimal representation. Merging these two concepts might seem without merit, but skeletonizations' ability to highlight features and reduce the noise of a graph could help guide labels through the graph.

This thesis proposes and explores the integration of the local-separator skeletonization algorithm in a graph-based semi-supervised learning setting. In a sparsely labeled data set (99.9% unknown labels), the local separator-algorithm boosted the accuracy of the label propagating algorithm from an accuracy of 86.64% to 88.4%, compressing the original graph of 60.000 nodes to a size of 84 nodes. The integration, however, also adds a considerable amount of computing time, as the valency of the graph has to be high for the skeletonization algorithm to provide optimization, and might suggest that the value of this implementation is best suited when labeling is expensive.

Acknowledgements

Eva Rotenberg, Lektor, Lektor, Department of Applied Mathematics and Computer Science
Aasa Feragen, Professor, Department of Applied Mathematics and Computer Science
Jakob Andreas Bærentzen, Lektor, Department of Applied Mathematics and Computer Science

I want to express my gratitude to my above mentioned supervisors for introducing and guiding me through these interesting fields of science: graph theory and machine learning. Although difficult the input from each one of you made for a great introduction. Thank you for being so responsive and answering all my questions, and Eva thank you for introducing me to Aasa and Andreas.

Contents

1 Introduction

Semi-supervised learning has achieved excellent results in real-world applications, in cases where data sets are expensive to label and unlabelled data is easily obtained. The semi-supervised learning algorithms assume the data has a manifold structure, suggesting that data points located near each other on a low-dimensional embedding share the same labels. Within SSL, graph-based solutions dubbed graph-based semi-supervised learning have great promise, as the manifold structure of the semi-supervised learning domain translates to the natural structure of graphs. Graph-based semi-supervised learning utilizes the structure of nodes and edges to represent data points and their similarity. The primary questions for GSSL are then how to produce suitable graphs and how to use them to infer labels.

Skeletonization is the process of compressing the structure of a graph into its main feature. A good skeletonized graph will represent the features of the original graph at a minimum of nodes and edges. Recent work by Andreas Bærentzen and Eva Rotenberg in [BR20] introduced a skeletonization algorithm, local-separators, that excels in compressing the number of nodes in a graph while retaining its features.

Coupling these two concepts together, the aim is to utilize the local-separator algorithms' ability to highlight features of the graph to guide the inferring of labels.

To achieve this, several restraints were discovered, ultimately leading to creating the using knn-based graph construction and label propagation. As the local-separator algorithm requires the nodes of the graph to be embedded in a 2- or 3-dimensional space, scaling the high dimensional data set was required and achieved by utilizing Multi-Dimensional embedding.

Through the methods and implementation explored in this thesis, the aim is to test whether skeletonization boosts the accuracy of regular graph-based semi-supervised learning algorithms. This will be done through the following chapters. First, the MNIST data set and the various concepts and algorithms used in graph-based semi-supervised learning are introduced in Chapter 2. This aims to equip the reader with an overview of the different parts of the final framework. The implementation of the different methods previously introduced is then presented, as well as the optimizations and constraints the skeletonization algorithms turned out to produce, in Chapter 3. The data and algorithms are then used to test and discuss regular label propagation and the new proposed skeletonized label propagation in Chapter 4. Lastly, the various methods and results are reflected on, and possible future work is hypothesised in conclusion in Chapter 5

2 Methods and Materials

This chapter aims to equip the reader with knowledge of the data and methods later implemented and explain why skeletonization is promising in a Graph-Based Semi-Supervised Learning setting.

First, the data set is explored, and the measure to compare two data points in the data set. The purpose and use case of Semi-Supervised Learning (SSL) and the general method for SSL. Then graph-based semi-supervised learning is discussed, including the reason for using GSSL, the general method of GSSL, and the specific process of constructing a graph with k nearest neighbors and inferring labels with label propagation. Lastly, the skeletonization notion and local- and front-separator algorithms are explained.

2.1 MNIST Numbers

The MNIST data set consists of 70.000 handwritten numbers divided into a 60.000 sized training set and a 10.000 sized test set, all fully labeled. The data set was in this thesis chosen as a means to compare the different methods used in Chapter 4, partly for the visual nature of it, and partly because it is a common data set used in Machine Learning, providing plenty of other results to compare the methods to such as [Yan21].
Each number is a fixed size of 28×28 pixels and is encoded in the greyscale color code, where each pixel value ranges from 0 to 255. Figure 2.1 shows an example of each digit in the data set. The numbers are size-normalized and centered.

2.1.1 Convention of conversion

The digits are represented in one of two states:

1. 28×28-matrix: A matrix that translates to the viewed image; the pixel in row 0 and column 0 corresponds to row 0 column 0 in the matrix.
2. 764-vector: A vector, the flattened matrix, where each 29th index corresponds to an increment in the row and a reset in the column to index 0.

To reduce redundancy assume the digits are in vector shape unless visualised.

2.1.2 Measure of similarity

It becomes apparent in the following chapters, that finding a good measure of similarity between two digits is crucial. The requirements are a low computation time and a rea-

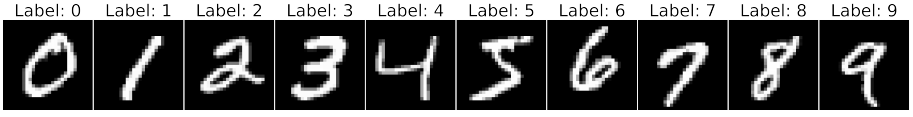


Figure 2.1: A subset of the MNSIT data set with an example of each label

sonable estimate of similarity where digits of the same label have a low value and digits of different labels have a high value. The Euclidean distance shown in Equation (2.1) provides both of the requirements but had a hurdle in that the comparison is pixel-to-pixel based.

$$\|u - v\|_2 = \left(\sum \left(w_i \left|(u_i - v_i)\right|^2\right)\right)^{1/2} \tag{2.1}$$

This means that if the shape of two digits is the same, but they are placed differently, the equation would rate them as being less similar than their shapes suggest, as is displayed in Figure 2.2, where the shape of the two images are the same, but they have been placed one square apart. What was not immediately obvious was the fact that the MNIST numbers are both ”size-normalized and centered in a fixed-size image”[Yan21], so any positional problems were naught, and it can be assumed that the Euclidean distance gives a reasonable estimate of similarity between the shapes of digits.

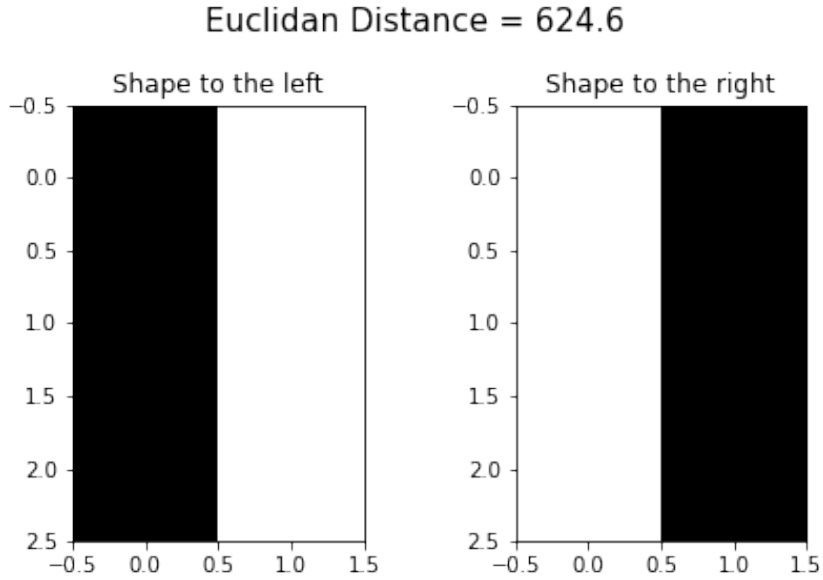


Figure 2.2: Same shape in different locations

2.2 Graph-based Semi-Supervised Learning

This section will introduce the reader to semi-supervised learning, a cornerstone in graph-based semi-supervised learning. Afterwards, graph-based semi-supervised learning is introduced, along with its taxonomy, and the particular method of constructing a graph with k-nearest-neighbors and label inferring with Label Propagation.

Graph construction and label propagation lean on the works of Zixing Song, Xiangli Yang, Zenglin Xu in [Son+21], providing a taxonomy for defining a graph-based semi-supervised learning problem as well as a general survey of the different gssl algorithms, and the works of Xiaojin Zhu and Zoubin Ghahramani in [ZG02] clearly defining the label propagation algorithm and implementation.

2.2.1 Semi-supervised learning introduction

Semi-supervised learning is an answer to a gap in data science. One would usually work with a feature-label pair in supervised learning or a fully unlabelled data set in unsupervised learning. Semi-supervised learning works with the spectrum in between with

a data set partly labeled, often sparsely. SSL assumes that data can be placed on a low-dimensional manifold, and samples near each other should have the same label. SSL can be divided into two types: transductive and inductive learning.

Transductive learning aims to predict labels within a given data set and not any unknown data points. Informally it can be described as reconstructing the data set. Formally transductive learning is defined as: Given a training set consisting of labelled and unlabelled data $D = \{\{x_i, y_i\}_{i=1}^{n_i}, x_{i_{i=1}}^{n_u}\}$ the goal of a transductive learning algorithm is to learn a function $f : X \to Y$ so that f is only able to predict the labels for the unlabelled data $\{x_i\}_{i=1}^{n_u}$[Son+21, Definition II.1].

Inductive learning aims to produce a function that returns a label for any input x. Formally inductive learning is defined as: Given a training set consisting of labeled and unlabeled data $D = \{\{x_i, y_i\}_{i=1}^{n_i}, x_{i_{i=1}}^{n_u}\}$, the goal of an inductive algorithm is to learn a function $f : X \to Y$ so that f is able to predict the output y of any input $x \in X$[Son+21, Definition II.2]

2.2.2 Overview

Graph-based semi-supervised learning extends the manifold assumption of SSL by using the structure of a graph. Each node in the graph corresponds to a data point in the data set, and an edge between nodes symbolises a similarity.

Based on the taxonomy proposed in [Son+21] the process of constructing a GSSL-problem involves two steps: 1) graph construction, 2) label inferring on the graph. There are multiple ways to achieve both steps. Because this thesis is rather broad, a decision was made early on to find algorithms for both steps without too much complexity. Constructing the graph with the k-nearest-neighbors algorithm and inferring the labels with label propagation met these criteria.

Definition of graph

Given a data set D, the graph G consists of a set of nodes V where each node represents a data point in the data set such that $|V| = |D|$, a set of edges E such that $E \in V \times V$ and a weight matrix $W \in R^{|V|\times|V|}$ to represent the weight on the edges if the graph is weighted. If the graph is not weighted the weight matrix will represent edges such that $W_{i,j} = 0$ if i, j does not share en edge and $W_{i,j} = 1$ if i, j does share an edge. This is akin to an adjacency matrix if it is unweighted. The graph is then formally described as $G = (V, E, W)$.

2.2.3 Graph construction

To perform graph-based semi-supervised learning, a graph is necessary. The nodes in the graph represent data points in the data set, and edges between nodes represent a similarity between the two nodes. Edges might also be weighted to reflect the similarity between two nodes. Edges in the graph can either be directed or undirected, which does not limit the construction of the graphs. In some cases, a graph network might already exist, such as a social network, and a graph will already be present. If that is not the case, a similarity function is used to identify similar nodes and connect them, such as the Euclidean distance.

KNN-based graph construction

The premise of KNN-based graph construction is to choose a number of neighbors K and construct a graph where each node is connected to the K closest nodes based on a similarity function $sim(x_i, x_j)$ that can quantify similarity between node pairs. These values will then be added to the weight matrix such that

$$W_{i,j} = \begin{cases} sim(x_i, x_j) & i \in neighbors(j) \\ 0 & otherwise \end{cases}$$

In the case of an unweighted graph, the $sim(x_i, x_j)$ can be exchanged with 1, which is the case of this thesis in the implementation of KNN.

The $neighbors(j)$-function is the cornerstone of the KNN algorithm. The function requires to choose the K lowest values of similarity between given digits x_i and all other digits $x_j \mid j \neq i$, which also translates to a graph with no loops. The no loop rule is essential as using the Euclidean distance on the same digit always yields 0, and if loops were allowed, every node would have a loop, as 0 will always be the lowest value.

The runtime of KNN is $O(n^2)$ as we have to check each index in the weight matrix, and the space complexity is $O(n^2)$ as the row and column have to exist for every data point in the data set.

Visualizing KNN-based graph construction
To aid the reader, a small example is displayed in Figure 2.3. Unlike the MNIST numbers, the data set in Figure 2.3 is located in a 2-dimensional space; the same rules, however, exist for MNIST. The Euclidean distance is used to measure similarity, and the only difference is that it is impossible to visualize all 764 dimensions of an MNIST data point in a single graph.

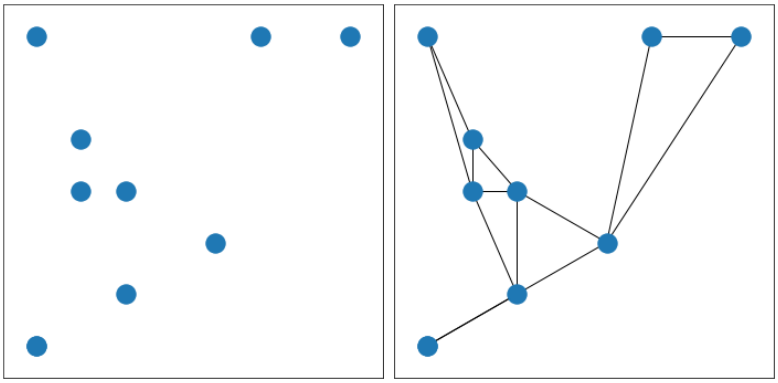


Figure 2.3: Example of KNN graph construction in 2 dimensions. Left is data points, right is graph constructed with 2 neighbors

2.2.4 Label inferring with label propagation

Label inferring is the process of spreading known labels throughout the graph. The algorithm is inductive, as it only is able to spread labels within the constructed graph, and does not produce a function to predict unseen data. Data points can be added to the graph, and label propagation can be applied again, which is time consuming, but possible. Using label propagation to infer requires three steps:

1. Spread the known labels step throughout the graph
2. Normalise the populated unlabelled nodes
3. Clamp known labels and repeat until the algorithm converges

This process can be done through matrix multiplication alone, as this thesis does, by multiplying the weight matrix with a one-hot encoded matrix of labels. It will later show that this approach can vastly improve performance utilizing sparsity matrices.

Example of label propagation
To get an intuition for the procedure of label propagation, a small example has been created shown in Equation (2.2) and visualized in Figure 2.4. Initially, the problem consists

of a weight matrix W representing the graph, and some known and unknown labels in a list of labels L, where each index corresponds to a specific node in G.

$$W = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \quad L = \begin{bmatrix} -1 \\ 0 \\ 1 \\ 1 \\ -1 \end{bmatrix} \tag{2.2}$$

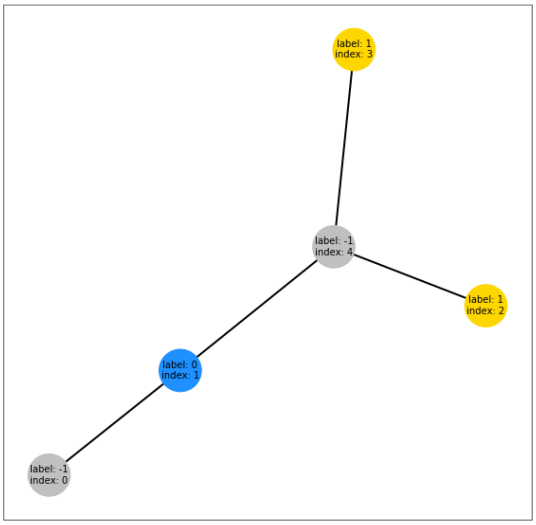


Figure 2.4: Visual representation of weight and label matrix in Equation (2.2)

By transforming the label matrix such that it has as many columns as unique labels in the data set and as many rows as data points in the data set, $L_{i,0} = 1$ indicates that a given data point with index i has a label 0, and an empty row indicates that a given index does not have a label at all. This is also known as one-hot encoding. The transformed label matrix named OL (One-hot encoded Labels) can be seen in Equation (2.3)

$$L \xrightarrow{\texttt{one-hot encode}} OL = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \tag{2.3}$$

Multiplying the weight matrix and transformed label matrix, the result will effectively be the same as counting the number of labels the neighbors of each node have, as can be seen in Equation (2.4). It will perform the first step of label propagation in spreading the labels one-time step. Notice how all nodes with labeled nodes - specifically node 0 and node 4 - have counted each label's type and occurrence in their labeled neighbors.

$$W \cdot OL = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 2 \end{bmatrix} \tag{2.4}$$

Step 2 of label propagation is row normalizing the label matrix. These normalized values can be interpreted as the probability that a given unknown node i has label x. The normalized rows can be seen in Equation (2.5)

$$W \cdot OL \xrightarrow{\texttt{row-normalise}} \begin{bmatrix} 1.00 & 0 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.33 & 0.67 \end{bmatrix} \tag{2.5}$$

This completes the second step of label propagation. Before checking for convergence, the already known labels are clamped back to the known labels, as can be seen in Equation (2.6). Intuitively from this matrix, it can now be seen that the algorithm converges and node $L_0 = 0$ and $L_4 = 1$.

$$\begin{bmatrix} 1.00 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0.33 & 0.67 \end{bmatrix} \tag{2.6}$$

A small note on converging
Interestingly, later in this project, I found the paper [ZG02], showing that the label propagation algorithm converges to a simple solution not requiring this iterative process. Unfortionately it was found at a late stage of the project, and as the run time of the final implementation of the algorithm was not very significant, this simpler solution was not implemented.

2.2.5 Bridges in graphs

A quite interesting aspect of the graph-based semi-supervised learning algorithms, is their ability to classify two dissimilar images of the same label correctly. This becomes more prevelant as the size of the data set grows. This is due to a bridging effect, where two data points with the same label are far from eachother, but in betweeen them exists data points, sort of similar to both of them. An example of this can be seen in Figure 2.5, where the left and right image are score the highest value with the Euclidean distance, but the image in between are close to both of them. If the data set has a small size, the lack of variate needed to connect dissimilar digits could limit the accuracy of the label propagation algorithm. , especially when testing with the large data set, as there might be unseen shapes not similar to any in the training data set. Skeletonization might be able to recognize features even in the small-sized training data set. To include this in the experiments, the size of the data set will range from 500 to 60.000 data points (the entire training set), with some interval. The low range is to test if skeletonization has a hidden potential in small data sets.

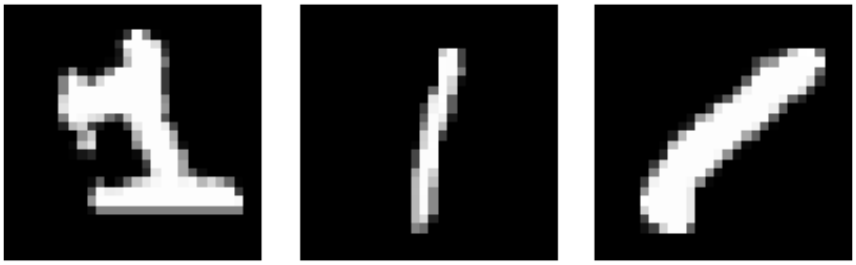


Figure 2.5: Example of bridge between digits. The left and right ones are the two ones most dissimilar in the MNSIT data set, but the one in the middle has a close distance to both of them ”bridging” the left and right digit.

2.3 Skeletonization

This section will introduce the reader to the concept of skeletonization and introduce the two algorithms: front-separator and local-separator devised and implemented by Andreas Bærentzen, and Eva Rotenberg in [BR20]. The section will briefly touch on the process of packing and extracting the skeleton. Lastly, it will reflect on skeletonization in a graph-based semi-supervised learning setting.

2.3.1 Overview

Skeletonization is the process of simplifying graphs with a focus on highlighting features and reducing noise. The premise of skeletonization is to take the nodes from the original graph and shrink them until only the overall shape of the graph is back. One can imagine using it on a 3D model of a human and only having the path of the skeleton left when the algorithm is done.

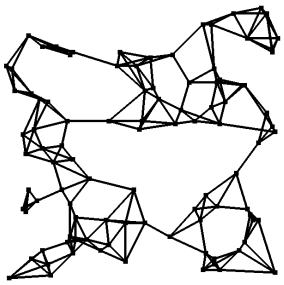
The algorithms work on the premise of separators. A separator is defined as a subset of vertices V where starting from one side of the separator and ending on the other is not possible without passing through. This has some interesting properties.

Suppose a graph is cyclical, and passing through all nodes can only be done by visiting each node. In this case, each node in the cycle is effectively a separator, and using either of the algorithms would yield the same output graph as the input graph. An opposite example of this is a graph G with nodes spread around, all connected to a single node in the middle. The valency of each node around the center node would be 1, and the center node would have a valency of $|V| - 1$, as it is connected to each node except itself. This center node would effectively be a single separator connecting all nodes, as one can not pass from an outside node to another outside node without passing through the center node. Applying either of the skeletonization algorithms would yield a graph with a single node placed where the center node is located. The last example to give a broader picture is a graph where all nodes are connected. Here all nodes effectively create a separator together, and the final graph would again be a single node in the skeleton.

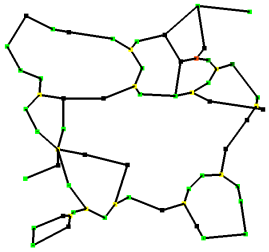
This can be extrapolated and shown that an increase in valency in the graph computes a simpler skeleton. This is shown in Figure 2.6, where we see that the graph with fewer neighbors provides a more complicated skeleton and vice versa.

2.3.2 Local separator algorithm

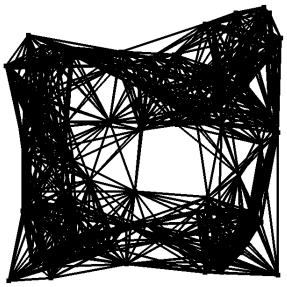
The informal approach of the local separator algorithm is as follows. From the input graph, choose a selection of random nodes. From each of these random nodes, now look at all neighbors and add them to a queue of new nodes to explore as a front (F). From F, we



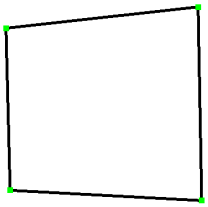
(a) Graph with $k = 4$



(b) Skeleton from graph with $k = 4$



(c) Graph with $k = 20$



(d) Skeleton from graph with $k = 20$

Figure 2.6: Example of correlation between valency in graph and magnitude of simplification in skeleton. Dataset is a 100 random datapoints, and edges are created by finding k closest nodes for each node

now choose the closest neighbor using a bounding sphere[1]. At some point, the separator will have grown so much that it separates the graph, meaning there is no path from one side of F to the other without passing through the separator. This is shown in Figure 2.7 G. Lastly, the separator is shrunk, as can be seen in Figure 2.7 H. The shrinking works on a few principles. First, the separator is smoothed using Laplacian smoothing. Then the set of nodes in the separator is removed regarding decreasing distance (the furthest away from the center of the separator is removed first). If the separator breaks into two components, that is, two subgraphs of G not connected by an edge, in this stage, the separator is discarded. If not, the separator can now be added to a set of separators and be further handled in the packing stage of the algorithm.

A significant property of the local separator algorithm is the reliance on the bounding sphere, specifically because the implementation only allows for nodes embedded in 2 or 3 dimensions. This is a constraint as the MNIST numbers only have an embedding in their original 764 dimensional space. This will be addressed further in Section 3.5.

2.3.3 Front separator algorithm

The front separator algorithm also works with a front F. It is, however, defined a little differently. Instead of choosing the front based on a bounding sphere from random nodes, the front is devised as a breadth-first search. Informally the algorithm works by choosing a selection of random nodes from the graph. From these nodes, a distance is calculated to all other nodes. This could be the Euclidean distance selected in this thesis.

[1] A bounding sphere works by finding the center of ”mass” of the nodes already explored and uses this center to calculate the distance to neighbors. This is done to find the closest point to the collective of the explored nodes

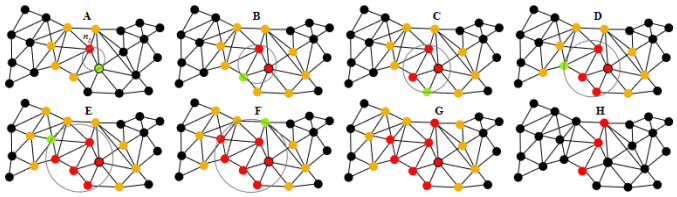


Figure 2.7: Picture from [BR20] showing the growing of a local separator

A breadth-first search algorithm is used to find the shortest path from the seed node to all other nodes from each of these nodes. This is saved for later reference. From here, the algorithm chooses random starting nodes within the graph. For each node in the graph in a random order, a node is hypothesized to start a new separator. If the node has already been explored and added to a separator, the attempt is terminated, and a new node is chosen. If not, the node finds all the closest neighbors and adds them to the front. For each node in the front, the algorithm checks if the neighbor has been explored. If another separator has explored it, the attempt is terminated, and a new node is chosen. Suppose the neighbor has not been explored, and the path from the current node to the neighbor is the optimal route. In that case, the neighbor is added to the separator, the neighbors are added to the front, and the node is indicated to have been visited by the separator. This continues until the front is empty and adds a separator to the set of separators. If the separator at any time finds one of the reasons mentioned above to terminate, the indicated nodes visited by the separator are reset as if not visited yet.

The clear advantage of using the front-separator algorithm is that, unlike the local-separator algorithm, it does not need embedded positions as it does not rely on the bounding sphere but instead only needs a pre-calculated distance from one node to all other nodes.

2.3.4 Packing

With both separator algorithms, it is possible to get a node placed in multiple separators, as separators can overlap. Packing ensures that each vertex on the graph only is covered by a single separator. Vertices without a separator might also exist. These are assigned to the closest separator.

2.3.5 Skeleton extraction

From packing to extracting the skeleton includes two steps. First, the vertices of the skeleton are calculated. This is done by calculating the average position of each separator. However, this step might lead to vertices in the graph with a valency $V \geq 3$, which is undesirable. The final step consists of removing these vertices by merging them with neighboring vertices of valency 2.

2.3.6 Motivation for skeletonization in a graph-based semi-supervised learning setting

The initial reasoning for exploring skeletonization coupled with graph-based semi-supervised learning was its formerly described ability to shrink graphs and keep the features present. If the graph is constructed such that edges connect similar nodes and dissimilar nodes are not connected at all, compressing the graph could lead to a skeleton expressing the core structure of the data. This could be a valuable asset when label inferring on very sparsely labeled data, as the skeletonization process would guide the label inferring algorithm.

3 Implementation

This chapter will discuss the implementation of GSSL and the incorporation of skeletonization using the local-separator algorithm using the MNIST data set. The graph is constructed using KNN as discussed in Section 2.2.3. The inferring of labels is done using label propagation as discussed in Section 2.2.4 and optimized using sparsity matrices. Lastly the integration with skeletonization and the graph-based semi-supervised learning algorithms earlier described is described.

3.1 Language

The implementation is done in `Python`, as it was most familiar to me and provides a broad selection of packages. A common approach to optimize Python algorithms is to use libraries implemented in `c` such as `numpy`. This is emphasized in algorithms with a runtime of $\geq O(n^2)$.

3.2 Similarity matrix

The similarity function described in Section 2.2.3 works by comparing a data point to all other data points. This is a costly computation, as the data set is comprised of 60.000 data points. Because of the exploratory nature of this thesis, one can expect the frequent computation of the graph, and here calculating the similarity between each digit would cause a considerable bottleneck. To save time, the similarity is calculated between each data point and stored in an N^2-sized matrix, where $sim_{i,j}$ returns the distance between data point i and j. The matrix can then be saved and loaded, drastically reducing time. Semantically, the similarity function has changed from a function call to a table lookup. The algorithm for computing the similarity matrix can be seen in Listing 3.1. The runtime of this algorithm given `x` data points is $O(n^2)$, and the space complexity is $O(n^2)$.

```
% similarity matrix calculation
def construct_similarity_matrix(data):
    l = len(data)
    W = [l][l]
    for i in range(l):
        for j in range(l):
            W[i][j] = euclidean_distance(data[i], data[j])
    return W
```

Listing 3.1: similarity matrix calculation where sim function is the euclidian

3.2.1 Memory optimization

By default, `Python` stored variables as either a 64 bit float or integer. If the whole training data set of 60.000 points is used with the previously mentioned space complexity, we end up having to store $60.000^2 * 64\texttt{bit} = 230.4\texttt{gigabit} = 26.8\texttt{gigabyte}$, which might not be possible on local machines. Using a cluster is a possibility, but a small goal I set for the project was to be able to run the code locally; therefore, the minimum bit size of a variable was calculated.

To choose the correct bit size, we look at the maximum distance between two images using the Euclidean distance, which is the distance between a vector filled with zeros and a vector filled with the value 255.

$$A = \langle 0, 0, \cdots \rangle \; B = \langle 0, 0, \cdots \rangle \; ; |A - B| = 7140.0 \tag{3.1}$$

The selection ranges from 8 to 128 bits. Even with an unsigned integer, the highest value possible for 8 bits is 255. 16 bits is another story, where the maximum value is 65500, well within limits. Therefore, the bit size is changed from 64 to 16 giving our similarity matrix a total of 7.2 gigabytes, almost a fourth of the previously needed memory allocation.

3.2.2 Threading

Even though the computation is preferably only done once, as the similarity matrix can be saved and loaded, the computation time was still significant. Threading was therefore introduced to fasten the computation.

With threading, complications are unfortunately also introduced, as the algorithm no longer runs sequentially. Using [And00] to familiarize with the pitfalls of parallel code, we see that there are some key factors necessary to consider in regards to reading and writing.

Generally, reading is always thread-safe, unless the read values can change. In this case, fortunately, the values (MNIST numbers) do not change, as that would mean the images change too. Reading is therefore safe.

When writing to the similarity matrix array, some possible problems needs to be considered. Most importantly, we need to know if we can write to separate variables in the matrix simultaneously. As `numpy` is written in `c`, where arrays are not dynamically changing size, we see that the array is instantiated as actual variables at every index. This is confirmed by the article[Kat]. This means that the different indices can be treated as individual variables. Each variable is only written to once, while the distance between each node only requires one computation. This means that things like mutex's or locks do not need to be implemented again, improving time.

One common approach to multithreading with k threads is to divide the dataset into k "chunks." Python, however, also supports using a pool, adding jobs to be executed when a thread is available. Using the latter approach, each job added to the queue is a new column in the similarity matrix, meaning that a job is added for every image in the collection.

The optimal way of proceeding would then be to store the calculated values in the array from the thread. Unfortunately, some limitations using threads on windows were encountered, denying the possibility of storing data in a shared similarity-matrix. Instead, we return the result from the function every time, leaving the first thread to solely storing the result in a final array. This does mean that there are no global variables, but is also highly probable to be a bottleneck. The new threaded algorithm can be seen in Listing 3.2

```
% Threaded similarity matrix calculation
def thread_calc(i, data):
    l = len(data)
    row = [l]
    for j in range(len):
        row[j] = sim(data[i], data[j])
    return i, row

def thread_similarity_matrix(data):
    l = len(data)
    wm = [l][l]

    pool = Pool(threads)
```

```
    x = [i for i in range(l)]
    pool.execute(thread_calc, x)
    for i, row in pool.get_result():
        wm[i] = row

    return wm
```

Listing 3.2: Threaded similarity matrix algorithm

3.3 Graph Construction

The implementation of graph construction is done by finding the $k+1$ lowest values, as the distance from each node to itself will always be zero. The current index of the row is discarded from the selection, and the remaining indices are added as ones in the weight matrix row- and column-wise to adhere to the requirement of an undirected graph. The algorithm can be seen in Listing 3.3.

```
def construct_knn_graph(similarity_matrix, k):
    l = len(similarity_matrix)
    G = [l][l]
    for i in range(l):
        indices = k_smallest(similarity_matrix[i], k)
        G[k, i] = 1
        G[i, k] = 1
    return G
```

Listing 3.3: Construction of graph

The runtime of the algorithm is $O(n * k\_smallest)$, as we have to find the k smallest values for each row n. $k\_smallest$ can be implemented in a variety of ways which is why the specific runtime of the algorithm is not included.

3.3.1 Sparsity

The end result of the graph construction algorithm will be a weight matrix with more zeros than ones, unless k is set unreasonable high. That is a lot of wasted space. To combat this, and to optimize the matrix multiplication in label propagation, the 2-D array is converted to a sparsity matrix.

The particular structure of the one used in this thesis is called a ”compressed sparse row matrix” (csr matrix), implemented in `scipy`, and has a structure very similar to the graph structure edge list. Instead of allocating space for all variables as the 2D does, the csr matrix stores a coordinate-value pair for each value in the matrix not equal to zero. A conversion from a 2D array to a csr matrix can be seen in Equation (3.2)

$$\texttt{2D Matrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \texttt{CSR Matrix} = [((0,1),1),((1,0),1),((2,1),1)] \tag{3.2}$$

Scaling this to 60.000 rows and columns and using a csr matrix significantly lowers the required space.

The other benefit is faster arithmetic operations on the matrices. Unfortunately the implementation of the sparsity matrix in `scipy` does not refer to any documentation of improved runtime, but the paper [YZ05] suggests that the upside is quite significant compared to both naive solutions and optimized solutions on regular arrays.

3.4 Label propagation

The conversion from method to implementation is straightforward in the case of label propagation. As the graph is already constructed, the only other requirement is the one-hot encoding of the labels. The encoding is rather trivial and is not included, and the algorithm assumes it receives a graph and one-hot encoded labels. The step of multiplying the matrices is possible with between two normal matrices or two sparsity matrices. As the internal structure of the code is similar, no particular mention of using sparsity matrices is shown in the algorithm in Listing 3.4.

```
# Label propagation algorithm
def propagate_labels(graph, labels, max_itter):
    known_rows, known_cols = labels.nonzero()
    for _ in range(max_itter):
        # Propgated labels
        p = dot(graph, labels)
        # Normalise rows
        n = normalise(p, norm="euclidian", axis=1) # Axis=1: Along rows

        # Reinsert clamped values
        n[org_row] = 0
        n[org_row, org_col] = 1

        labels = n
    # Return list of max indices for row-wise
    return indice_max(labels, axis=1) # Axis=1: along rows
```

Listing 3.4: Label propagation

3.5 Skeletonization

The implementation of skeletonization can only be described as a journey. Already before experimenting with the implementation, complications appeared. The complications might imply that building a suitable setting for skeletonization is rather difficult, and are therefore included.

To use skeletonization, the implementation of [BR20] was used. This implementation does not accept an array as input, but instead requires a process of building the graph within the package, by adding nodes and edges separately through function calls.

To make integration with the package seamless, two functions were needed: conversion from matrix to package, and conversion from package to matrix. In addition to this, after skeletonization, a map between skeleton nodes and nodes in the original graph is returned. This map can be used to remap propagated labels in the skeleton to the original data set. A third function is therefore needed named `remap`. These three functions are valid for both the front- and local-separator integration.

3.5.1 Initial plan of skeletonization

The initial idea for the implementation of skeletonization was to use the front-separator algorithm, as it had the benefit of not requiring the data points to be embedded in a 2- or 3-dimensional space. Even though this is true for the front-separator algorithm, the package required an instantiating of its graph object. This object requires an embedding of the nodes. Three options were devised: create random positions for the nodes, generate optimal node placement based on nodes and edges, or embed the MNIST numbers in a 3-dimensional space.

A combination of option 1 and 2 were chosen. Initially, the nodes were placed in a sequential line, which, although practical, did not offer any visual guide for what was going

on. To visualise the graph an additional implementation was done arbitrerely placing the nodes in regards to their edges.

To place the nodes, the Fruchterman-Reingold force-directed algorithm[SP19] was used, implemented in the package `networkx`. The algorithm takes an adjecency matrix as input, and outputs optimal placements of nodes based on both nodes and edges. The running time of the algorithm is $O(|V| + |E|)$.

Using the algorithm to pseudo-embed each node in the graph, the initial implementation of skeletonization could be implemented. This however quickly showed issues, as can be seen in Figure 3.1

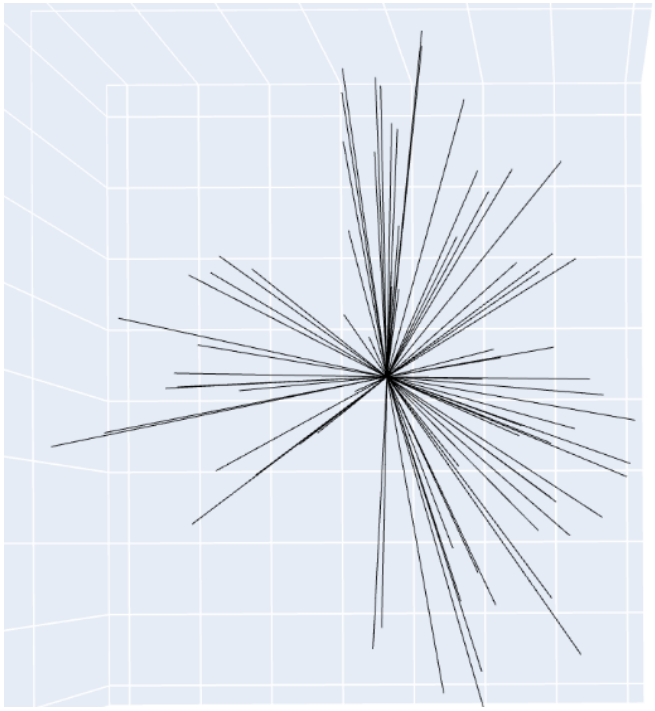


Figure 3.1: Resulting graph from using the front-separator algorithm on a graph consisting of 10.000 nodes, built with the k-nearest-neighbors graph construction algorithm, where k=10

As the position of the embedded vertices does not account for the outcome of the local-separator algorithm, the issue most likely has to do with the particular type of graph the KNN-based graph construction algorithm produces. The reason for this star shape most likely has to do with a heavily connected cluster of nodes. As with the example in Section 2.3, a dense collection of nodes would produce the same effect, and lead to this particular shape. In a label propagation-oriented sense, the structure is less than helpful, as the center might represent a label, and propagating from one node to another would have to pass through the center node, halting all communication between all outer nodes.

The issue might have been benign, but instead investiaating the issue it was decided to prioritise getting a result with another method, local-separators.

3.5.2 Integration with the local-separator algorithm

To integrate the local separator algorithm, the three same functions as described in the introduction, is needed. In addition to this, a new requirement needs to be met. Because the local-separator algorithm relies on a somewhat correct geometry of the embedding of the nodes, the embedding is now important.

It might have been a possibility to use the Fruchterman-Reingold force-directed algorithm, but from prior experience, an already tried alternative was found in multi-dimensional scaling (MDS). There are a variety of implementations of this method, but the general approach is the same, and can quite easily be compared to principal component analysis. The informal procedure of MDS is to reduce the number of dimensions and keep as much information about the placement of data points in relation to each other.

Recent work by Akshay Agrawa, Alnur Ali and Stephen Boyd in [AAB21a] provides an implementation in the package `pymde` of an alternatively phrased solution called Minimum-Distortion Embedding (mde). In this setting, the Minimum-Distortion Embedding is an optimization problem, where the solution is an embedding.

Using this, the placement of nodes seen in Figure 3.2 is achieved. In this, defined clouds of similarity can clearly be seen. Because the change in dimensions is a compression of information, using the vectors of the original digits, might produce a graph slightly less suited for the local-separator algorithm. As the local-separator algorithm relies on the position of nodes, using the 3-dimensional Euclidean distance will most probably create shorter edges in the embedded space, creating better conditions for the algorithm. The similarity function in the graph construction step is therefore changed to be based on the newly computed embedding.

Because the dimensions of the embedding are so low, a kd-tree can now be used to find closest neighbors instead of calculating the distance for similarity for each node. This was not previously possible as using a kd-tree in a high-dimensional space (> 20) does not run ”significantly faster than brute force”[1].

With these changes to the construction of the graph and the embedding of data points, the integration can finally be implemented.

From matrix to pygel

To construct the graph within pygel, nodes, position of nodes, and edges are needed. The nodes and their embedding is already known from mde. To find the edges, the convenient structure of the sparsity matrix is used. As mentioned earlier, the structure corresponds to an edge list, where each entry in the sparsity list includes a row-column pair, which corresponds to an edge between the given index for the row and column. The algorithm can be seen in Listing 3.5

```
def matrix_to_pygel(g, embedding):
    edges = g.row_column_pairs()
    g = Graph()
    for i in range(len(embedding)):
        g.add_node(embedding[i])
    for edge in edges:
        g.connect_nodes(edge)

    return g
```

Listing 3.5: Matrix to pygel algorithm

From pygel to matrix

The conversion from pygel will only be done after the local-separator algorithm has skeletonized the graph. The goal of the conversion from `pygel` to matrix form is therefore twofold. A symmetric weight matrix is first created to adhere to requirement of having an undirected graph, and a one-hot encoded matrix of labels is thereafter created. As

[1] https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.KDTree.html

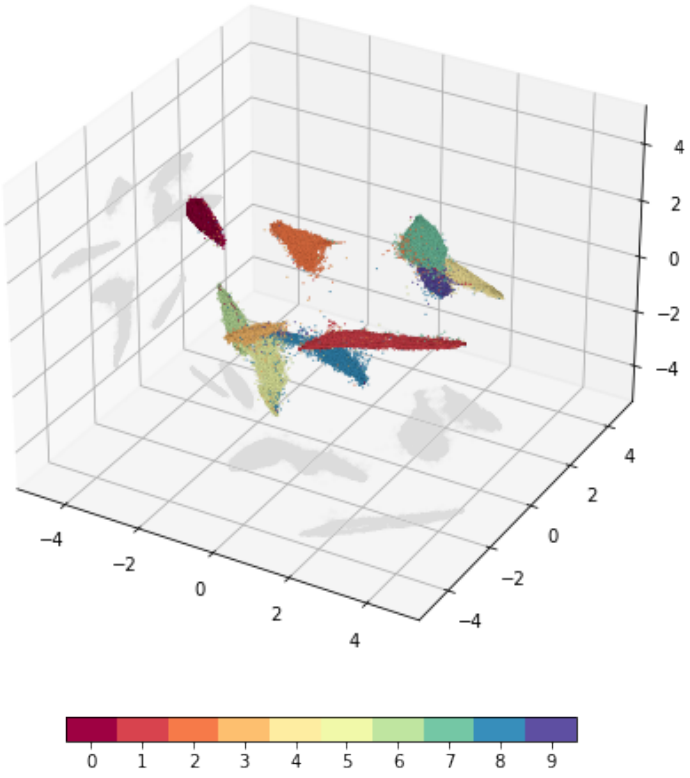


Figure 3.2: Embedding of MNIST numbers into 3 dimensions using Multi-Distortion Embedding

the skeletonization algorithm compresses multiple nodes together it is possible that multiple labelled nodes will be put into the same separator. The compression might lead to different labels within the same skeleton node. To address this two approaches were hypothesised. Either a count of each label in the node is put into the corresponding row of the one-hot-encoded label matrix, and the row is normalised. This step is very similar to the second step of label propagation. The other solution is to just choose the majority of a label in the node and assign that as the label for the skeleton node. For simplicity the latter approach was chosen. In a heavily labelled data set, this approach breaks the third step of label propagation, as the true values of labels are not clamped. If the sparsity is very low however, the occurrence of multiple labels within a skeleton node becomes more unlikely. The algorithm can be seen in Listing 3.6. The algorithm takes three inputs: the graph object created with pygel, the map from graph to skeleton returned from the skeletonization algorithm, and the labels of the graph. The map has as many indices as the original graph, and each index in the map returns the new node id in the skeletonized graph.

```
1 def pygel_to_matrix(skeleton_graph, skeleton_map, labels):
2     l = pygel_graph.num_of_nodes()
3
```

```
    # matrix is instantiated with size l^2 filled with 0
    for node in skeleton_graph.nodes(): # node is an integer ranging from 0..
        number of nodes - 1
    skeleton_matrix = [l][l]
        neighbor_indices = skeleton_graph.neighbors(node)
        skeleton_matrix[node][neighbor_indices] = 1
        skeleton_matrix[neighbor_indices][node] = 1

    # As many row as nodes, and as many columns as unique labels
    skeleton_labels = [l][10]
    index = 0
    for skeleton_index in skeleton_map:
        if labels[index] != -1:
            skeleton_labels[skeleton_index][labels[index]] += 1
        index += 1

    for row in skeleton_labels:
        choose_majority(row) # This sets majority column to 1 and rest of
            columns in row to 0.

    return skeleton_matrix, skeleton_labels
```

Listing 3.6: Pygel to matrix algorithm

Remapping nodes

To remap the nodes, the same map between original graph and skeletonized graph as used in Listing 3.6 is used. The remap algorithm walks through the map where each index corrosponds to a node in the original graph and each value corrosponds to a node in the skeleton graph, and maps the label of the skeleton node to the original graph node. As the step of remapping the labels to the original graph is the last step of label propagation, the algorithm returns a list of labels instead of the one-hot-encoded labels. The algorithm can be seen in Listing 3.7

```
def remap_labels(skeleton_labels, skeleton_map):
    l = len(skeleton_map)
    remapped_labels = [l]
    index = 0
    for skeleton_index in skeleton_map:
        remapped_labels[index] = skeleton_labels[skeleton_index]
        index += 1

    return remapped_labels
```

Listing 3.7: Remap algorithm

4 Experimentation

This chapter will use the implementation of knn-based graph construction, label propagation and skeletonization to test how skeletonized label propagation compares to regular label propagation.

From early experiments, it became apparent that the training data set would have to be very sparsely labeled for skeletonization to improve accuracy. The results from the early experiments can be seen in appendix Table A.1, Table A.2, Figure A.1 and Figure A.2, where the change in accuracy of the label propagation algorithm is marginal.

Therefore, the label sparsity of the training data set is narrowed to a range of $90\% \rightarrow 99.9\%$ removed labels. The narrowing is halted at 99.9% as a 99.99% removal of labels in the 60.000 sized training set would yield 6 nodes with labels, which does not cover all ten unique labels of the MNIST data set.

This chapter will experiment with the different levels of label sparsity in the training data set to compare the two models and reflect on the experiment results.

4.1 Procedure of experiments

Both algorithms will use a graph created with knn-based graph construction. This graph is then skeletonized. The standard label propagation algorithm will use the original graph to propagate labels, and the skeletonized label propagation algorithm will use the skeletonized graph.

The procedure for each algorithm is as follows:

1. Delete a percentage p of all labels in the train data set.
2. Apply the algorithm to the training set, which will return the relabelled data points in the train set.
3. Compare the true labels and the reconstructed labels. The clamped labels are not used in the comparison as these will always be equal.
4. Forget all labels in the test data set.
5. Use the reconstructed train data set to predict labels in the test set using the majority of labels in k nearest neighbors to determine each unknown label.
6. Compare the true labels with the predicted labels, that is, the accuracy of predicting the tested labels.

This approach outputs two different test metrics, reconstruction accuracy and testing accuracy. The reconstruction accuracy defines how well the labels within the graph have been assigned. Because the labels are embedded, this metric might be skewed as the `mde`-algorithm embeds nodes close together. To provide an independent metric, a second test is therefore made. Here the test set finds the K closest digits based on the Euclidean distance of two MNIST numbers in their 764-dimensional space. This is probably not the best implementation of making label propagation inductive, but as the test is the same for label propagation and skeletonized label propagation, making a comparison between the test results seems fair. Only the test data will be extrapolated and shown, as it reflects the reconstruction accuracy and is unbiased.

Because the local-separator algorithm is not fully deterministic[BR20, page. 12] and because the time to compute a skeleton for a highly connected graph is high, the skeletons are initially created in a range of $k = 2$ to $k = 40$ from the entire training data set and reused for the different levels of label sparsity. Although the difference between skeletons is marginal, this does make the comparisons between different experiments fairer.

4.2 Expectations

Before discussing expectations, an intuition is needed to translate how much the local-separator algorithm affects the results. From Section 2.3 a clear link between the connectivity of the graph and the amount of compression skeletonization performs is established. If k is set to 1 in the graph-construction algorithm, the output graph of skeletonization will be the same as the input graph, and as k increases, so will the amount of compression. This can be seen in Figure 4.1, showing the size of the skeletonized graph as k increases. In short, skeletonization results will be more affected as a higher amount of neighbors are used to construct the graph.

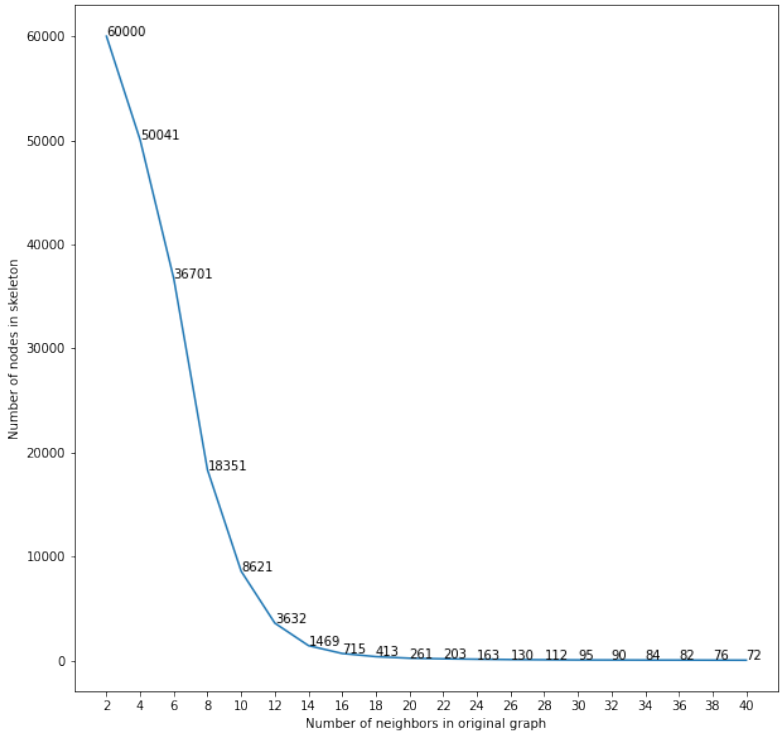


Figure 4.1: Size of skeletonized graph consisting of 60.000 nodes. Graph is constructed with the k-nearest neighbors algorithm at an increment of 2 between $k = 2$ and $k = 40$

The expectation is that under the circumstances of a large, sparsely labeled training set and a high amount of neighbors, the MDS algorithm will be able to define clear clouds of similar points. The local-separator algorithm can use these to create a skeleton with separator nodes consisting of very similar digits and an edge-set consisting of the most similar nodes to each other. With a combination of high label sparsity and a high amount

of neighbors we expect the skeletonized label propagation to equal or outperform regular label propagation.

It is also possible that if the skeletonized graph gets very compressed, the implementation of the majority vote of the label in a node, when converting from pygel-graph to matrix-graph, could worsen results.

4.3 Parameters

The experiments will revolve around two parameters: the number of neighbors K and the number of forgotten labels FP[1].

Choosing parameters is a case of qualified guesses as well as points of interest. Experiments and parameters for the specific hypothesis described in Section 4.2 will be tested, but to be more encompassing, a wider range of parameters are chosen. Through early testing, a high number of neighbors increases the time spent on skeletonizing quite significantly. Therefore, a ceiling of 40 neighbors is set, as the amount of compression seems to flatten out at around this value.

A quick summary of the ranges of parameters can be seen in Table 4.1.

Parameter	**Min**	**Max**
Number of neighbors K	1	40
Percentage of forgotten labels FP	90%	99.9%

Table 4.1: Summary of ranges of parameters

4.4 Varying number of neighbors

Increasing the number of neighbors paired with skeletonization will show whether a compressed skeleton graph decreases, increases, or does not influence the result of label propagation. If the accuracy decreases, skeletonization may not be helpful at all. If it increases, skeletonization has a positive effect on label propagation. If it remains unchanged, the result signifies that the skeletonization algorithm is good at keeping the features of the graph intact, but unfortunately, does not improve the label propagation algorithm.

4.4.1 Results

At an earlier stage of experimentation, the skeletonization algorithm was run and tested on a subset of the test set. These results showed that there was almost no difference between including and excluding the MNIST data set. The graphs for the results can be seen in appendix Figure A.3. The results of running the two algorithms using the whole test set show a more positive outcome and can be seen in Figure 4.2, and Table 4.2. Both tests ran on the same graphs and skeletons; the only difference is the size of the test data set.

Looking at the results, the skeletonized label propagation algorithm did - as expected - outperform the label propagation algorithm. The graph that outperformed the label propagation can be seen on the front page of this thesis. The results, however, also show some interesting and surprising properties. The label propagation algorithm needs more than two neighbors to produce a good result. However, when this is saturated, the amount of neighbors affects the accuracy minimally, although there is a down-going trend. This is also visible as the shape of the three different settings is similar with an offset. The results

[1] Forgotten Percentage. If FP=80%, 80% of labels are removed

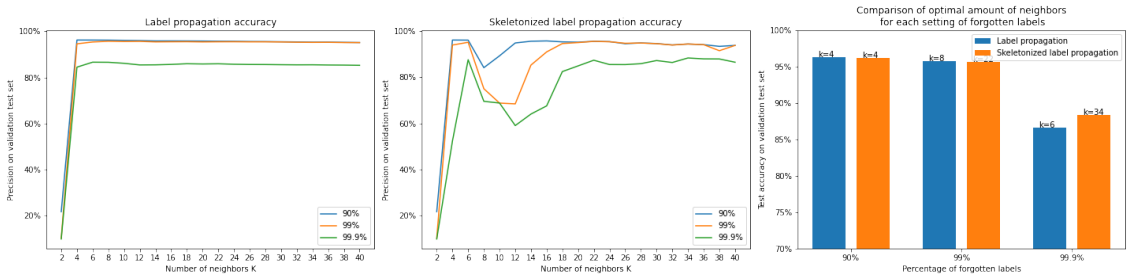


Figure 4.2: Validation set result. The first two figures show the result of an increased number of neighbors used to construct the graph, and right image shows a comparison of the best result for the two algorithms in each setting of sparsety of labels.

	Forgotten percentage	90%	99%	99.9%
Label propagation	Accuracy	96.26%	95.73%	86.64%
	Number of neighbors	4	8	6
Skeletonized label propagation	Accuracy	96.18%	95.61%	88.4%
	Number of neighbors	4	22	34

Table 4.2: Highest accuracy achieved and paired value of k. Values match the bar chart in Figure 4.2

also suggest that the label propagation algorithm can deal with quite a significant absence of labels (99%) without affecting the algorithm's result. This is backed up by the result of 90% and 99% unknown labels practically being identical when the number of neighbors is high.

The result from skeletonized label propagation is a bit surprising. Before conducting the experiments, we had a suspicion that an increase in compression would worsen results, as the probability of squeezing different labels into the skeleton nodes would affect the result, especially at a high amount of labels. The results, however, suggest otherwise. As with label propagation, the accuracy trends downwards as the number of neighbors is increased from 12 to 40. The biggest surprise, however, was the valley of low accuracy seen at all three levels of label sparsity. Initially, the suspicion is that different labels are somehow put into the same nodes in this interval. To further explore this and to estimate how the implementation of the majority node in each skeleton node affects the result, we look at the percentage of nodes with multiple labels in them. This can be seen in Figure 4.3.

To digest this, let us pick the information apart. The first thing to notice is that there are no multiple labels in nodes when $k = 2$. This is not very surprising as Figure 4.1 shows that the skeleton has the same number of nodes as the original graph, and so no nodes have been compressed together. The next and most exciting aspect of this result is that the number of nodes with multiple labels affects the result more than the percentage of nodes. This can be seen as the spike in the number of nodes with multiple labels align with the drop inaccuracy of the skeletonized label propagation algorithm in the interval between $k = 8$ and $k = 12$. This effect can be seen as ripples in all three levels of label sparsity, where the decline of accuracy starts at around six neighbors, and with some delay, stabilizing at around 20 neighbors.

The culprit of this behavior is likely the majority vote. What is probably happening is that the skeletonization algorithm has not been able to compress the graph enough to guide

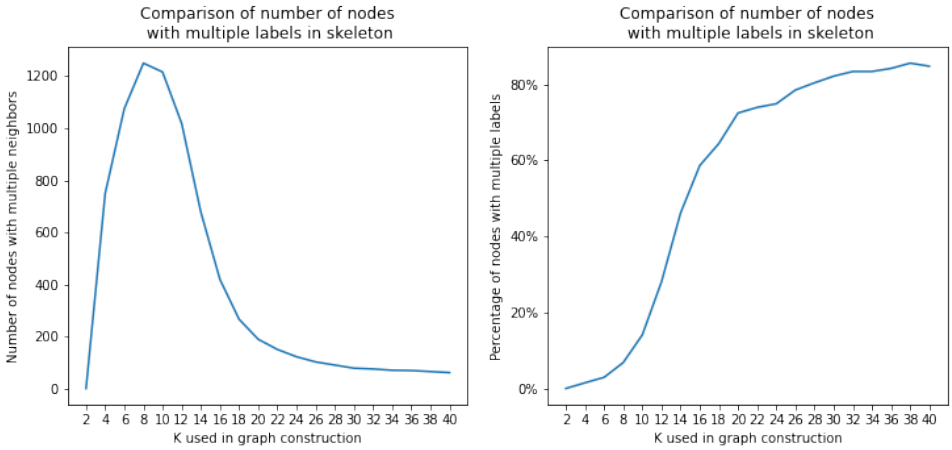
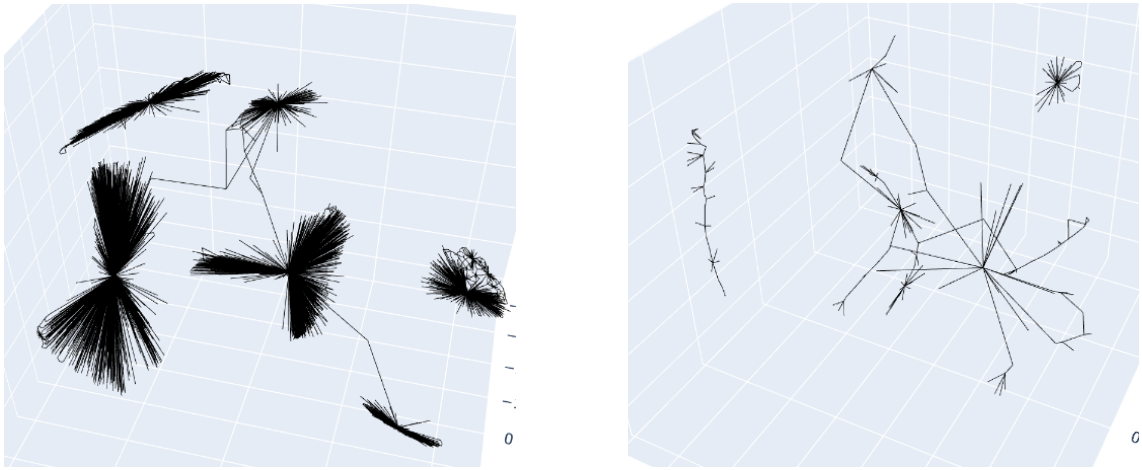


Figure 4.3: Percentage of nodes with multiple labels in them

the direction of the label propagation algorithm. Instead, the connectivity from the original graph is intact, and within this, true labels are changed. To explore this further, let us look at the structure of the skeletons. To keep the data manageable, we look at the skeletons created at a high number of neighbors as they have fewer nodes and are therefore easier to look at and understand. With label sparsity set to 99.9%, we see two skeletons reaching the lowest point and returning to a somewhat stable accuracy at $k = 12$ and $k = 22$. We expect the less compressed skeleton to be similar to the original graph, with edges between different regions of labels, and the more compressed skeleton to have removed a lot of these edges. Results can be seen in Figure 4.4.



(a) Skeleton produced from graph with k=12

(b) Skeleton produced from graph with k=22

Figure 4.4: Skeletons from graph construction

Unexpectedly the skeletons seem to tell a different story. The grouping of labels together is still a problem, but what seems to be the biggest problem is the reappearance of the star shape from Section 3.5.1. As explained in Section 3.5.1, the problem seems to be with centers of the graph being heavily connected. The densely connected center of embedded clusters creates several separators close together, that either in the packing or skeleton-extraction stage, merges them. This issue is a choice in the packing/skeleton-extraction algorithm, chosen and optimized for 3D models, which seems to produce a

less compelling output for this arbitrary graph of embedded data points. Looking at b in Figure 4.4, the star shapes are more or less dispersed, and the few stars that are left are within the region of a specific label. To further check if the stars make the results worse, let us look at exactly what types of digits are guessed wrong. Looking at the embedding, the graphs are based on in Figure 3.2 it is seen that labels 0 and 1 are placed in distinct, separated clusters. This is reflected in both skeletons, as there are no edges between the two outside clusters. However, there is a star in the middle of the cluster consisting of 3s, 5s, and 8s, and the cluster consisting of 4s, 7s, and 9s. If the stars are the reason for the dropped accuracy, this will be reflected, as these two clusters will have a significant disadvantage in propagating information. As the skeleton created with $k = 12$ seems to suffer most from stars, we will look at its distribution of true and predicted labels. Looking at the heatmap in Figure 4.5, as expected, the main contributor to the low accuracy score stems from the two clusters previously mentioned.

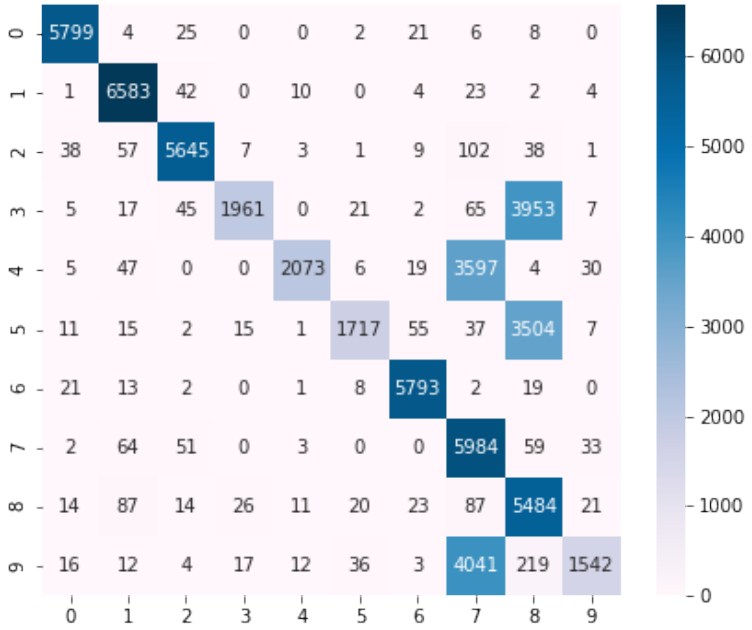


Figure 4.5: Heatmap of true labels and predicted labels. Rows are the true label of nodes, and columns are the predicted label.

4.5 Discussion

Despite its shortcomings, the skeletonized label propagation algorithm improved results at a high sparsity level, outperforming the regular label propagation algorithm at almost two percentage points. We also see that the number of neighbors for the optimal result was relatively high at 34 neighbors. This suggests that the skeletonization algorithm played a big role in boosting the label propagation. It would be interesting to see if resolving the issues at the lower amount of compression would further boost label propagation, both at high and low amounts of compression.

The shortcomings of the skeletonized label propagation algorithm involve two factors, and the interaction between them, that seems to limit the accuracy: the majority vote in nodes with multiple labels and the star-shaped clusters. The star structure limits the paths for information to pass through the graph structure. Numerous spikes are observed going from the center of the star, and the nodes at the ends of these spikes will have to pass

through the center. Because the center of nodes is created from an extensive collection of nodes, at least one label will most certainly be placed in the separator. Because label propagation reinserts the values of all known labels (clamping) given at initialization of the algorithm if the center is assigned a label, the information from the different spikes cannot propagate through the graph as it is reset at every iteration of the algorithm. In addition to this, the information network of edges between nodes is removed entirely and replaced with stars instead, limiting the label propagation algorithm further.

Earlier, we hypothesized a way to solve clamping by instead of deciding the label by majority a probability of each label being inserted as the label of the row, like the second step of label propagation. Whether this row of probabilities should be clamped or allowed to change is not immediately apparent and would need some experimentation. The implementation can also be tricky as labeled and unlabelled nodes would be a part of the separator, and assigning labels to these data points could not be done through label propagation.

The biggest problem, however, seems to be the star structure reflected by the results in Figure 4.5. If the skeletonization algorithm produced a small but connected cluster instead of a star structure, the probability of skeleton nodes with a single collection of identical labels might be higher. The most intuitive way to limit the creation of stars would be to make the center less connected, forcing the skeletonization algorithm to spread out separators and making the number of compressed nodes in the separators more equal. This could be done by limiting the max distance two nodes can have to create an edge in the graph construction phase.

A popular method of constraining distance between nodes is ϵ-neighbor-based graph construction algorithms. This was not implemented in this thesis as KNN-based graphs outperform ϵ-neighbors-based ones with better scalability according to [Son+21, page. 5]. The primary reason for disregarding this algorithm was that a poor choice of ϵ can be misleading and might produce worse results, and given the time constraint of the project, it thus seemed like a KNN-based approach was better suited. Looking at the result of this implementation, it becomes pretty apparent that an ϵ-neighborhood-based construction approach might be necessary to get consistently good results.

5 Conclusion and future work

The implementation of skeletonized label propagation on a knn-constructed graph has been presented. The algorithm takes high dimensional data and embeds the data points in 3 dimensions, where it constructs a graph. This graph is then skeletonized, producing a compressed graph used to guide the label propagation algorithm.

Through the implementation phase, it became apparent that the skeletonization algorithm introduces some constraints. A lot of the work in the thesis has been to navigate through the different methods of graph-based semi-supervised learning and skeletonization to find a common point merge them. Through experiments, it was found that in a data set with a high lack of labels and a high number of neighbors, the local-separator algorithm boosted label propagation from 86.64% to 88.4% with $k = 32$.

The algorithm's accuracy dropped when the number of neighbors used to construct the graph was comparatively low. Through an exploration of the results, the exact reason for the valley of low accuracy was found in two major contributors: the process of merging labels in skeleton nodes and the construction of the graph leading the local-separator algorithm to produce graphs with stars, limiting the flow of information in the label propagation stage.

Solutions for future work were hypothesized. The majority vote of labels contracted in the skeleton nodes seems intuitively and by the results as a bad idea. Using the number of contracted labels to spread probabilities through the graph might see better results. The star-shaped clusters were hypothesized to be caused by nodes close together and highly connected. To resolve this, either a limitation on the valency of each node must be imposed, or a limitation has to be set on the distance between nodes to be eligible to create edges between them.

In addition to this, it would be interesting to test whether a higher number of neighbors in the graph, resulting in a more compressed skeleton, would heighten or lower the accuracy.

Bibliography

[BR20] Andreas Bærentzen and Eva Rotenberg. “Skeletonization via Local Separators”. In: *CoRR* abs/2007.03483 (2020). arXiv: 2007.03483. URL: https://arxiv.org/abs/2007.03483.

[Yan21] Christopher J.C. Burges Yann LeCun Corinna Cortes. *THE MNIST DATABASE of handwritten digits*. Tech. rep. Courant Institute, NYU, Google Labs, New York, Microsoft Research, Redmond, 2021. URL: http://yann.lecun.com/exdb/mnist/.

[Son+21] Zixing Song et al. “Graph-based Semi-supervised Learning: A Comprehensive Review”. In: *CoRR* abs/2102.13303 (2021). arXiv: 2102.13303. URL: https://arxiv.org/abs/2102.13303.

[ZG02] Xiaojin Zhu and Zoubin Ghahramani. *Learning from Labeled and Unlabeled Data with Label Propagation*. Tech. rep. 2002.

[And00] Gregory R. Andrews. *Foundations of Multithreaded Parallel and Distributed Programming*. Pearson, 2000.

[Kat] Ayoosh Kathuria. *Nuts and Bolts of NumPy Optimization Part 3: Understanding NumPy Internals, Strides, Reshape and Transpose*. Tech. rep. URL: https://blog.paperspace.com/numpy-optimization-internals-strides-reshape-transpose/.

[YZ05] Raphael Yuster and Uri Zwick. “Fast sparse matrix multiplication”. In: *ACM Transactions On Algorithms (TALG)* 1.1 (2005), pp. 2–13.

[SP19] Mirco Schönfeld and Jürgen Pfeffer. “Fruchterman/Reingold (1991): Graph Drawing by Force-Directed Placement”. In: *Schlüsselwerke der Netzwerkforschung*. Ed. by Boris Holzer and Christian Stegbauer. Wiesbaden: Springer Fachmedien Wiesbaden, 2019, pp. 217–220. ISBN: 978-3-658-21742-6. DOI: 10.1007/978-3-658-21742-6_49. URL: https://doi.org/10.1007/978-3-658-21742-6_49.

[AAB21a] Akshay Agrawal, Alnur Ali, and Stephen Boyd. “Minimum-Distortion Embedding”. In: *arXiv* (2021).

[Yar95] David Yarowsky. “Unsupervised Word Sense Disambiguation Rivaling Supervised Methods”. In: *Proceedings of the 33rd Annual Meeting on Association for Computational Linguistics*. ACL ’95. Cambridge, Massachusetts: Association for Computational Linguistics, 1995, pp. 189–196. DOI: 10.3115/981658.981684. URL: https://doi.org/10.3115/981658.981684.

[GCY92] William Gale, Kenneth Church, and David Yarowsky. “A method for disambiguating word senses in a corpus”. In: *Computers and Humanities* 26 (Dec. 1992), pp. 415–439. DOI: 10.1007/BF00136984.

[CZ10] B. Schlkopf . Chapelle and A. Zien. *Semi-Supervised Learning*. THe MIT Press, 2010, pp. 191–249.

[Che+21] Changjian Chen et al. “Interactive Graph Construction for Graph-Based Semi-Supervised Learning”. In: *IEEE Transactions on Visualization and Computer Graphics* (2021), pp. 1–1. DOI: 10.1109/TVCG.2021.3084694.

[Van16] Jake VanderPlas. *Python Data Science Handbook*. 2016. URL: https://jakevdp.github.io/PythonDataScienceHandbook/05.12-gaussian-mixtures.html.

[AAB21b] Akshay Agrawal, Alnur Ali, and Stephen Boyd. “Minimum-Distortion Embedding”. In: *arXiv* (2021).

[Jer05] Xiaojin Zhu (Jerry). “Semi-Supervised Learning Literature Survey”. In: (2005). URL: http://digital.library.wisc.edu/1793/60444.

[TMll] Mikkel N. Schmidt Tue Herlau and Morten Mørup. *Introduction to Machine Learning and Data Mining*. Technical University of Denmark, 2019 Fall.

[ZHS05] Dengyong Zhou, Jiayuan Huang, and Bernhard Schölkopf. “Learning from labeled and unlabeled data on a directed graph”. In: *Proceedings of the 22nd international conference on Machine learning*. 2005, pp. 1036–1043.

[CFS09] Jie Chen, Haw-ren Fang, and Yousef Saad. “Fast Approximate kNN Graph Construction for High Dimensional Data via Recursive Lanczos Bisection.” In: *Journal of Machine Learning Research* 10.9 (2009).

A Appendix

A.1 Tables

K	FP	0.91	0.92	0.93	0.94	0.95	0.96	0.97	0.98	0.99
4	Reconstruction	95.13	95.08	94.88	94.66	94.40	94.53	94.68	94.18	93.87
	Test	95.85	95.87	95.74	95.51	95.27	95.29	95.31	95.12	94.66
16	Reconstruction	96.51	96.45	96.44	96.44	96.45	96.47	96.46	96.40	96.12
	Test	96.33	96.29	96.30	96.28	96.29	96.31	96.31	96.26	96.06
32	Reconstruction	96.48	96.51	96.51	96.52	96.52	96.51	96.50	96.51	96.42
	Test	96.38	96.38	96.36	96.34	96.34	96.33	96.33	96.35	96.23

Table A.1: Label propagation results with a training data set at a size S of 60.000 varying the number of neighbors K and forgotten percentage FP of the labels in the training data set

K	FP	0.91	0.92	0.93	0.94	0.95	0.96	0.97	0.98	0.99
4	Reconstruction	94.96	94.91	94.80	94.49	94.16	94.31	94.41	93.67	92.90
	Test	96.00	95.92	95.78	95.58	95.31	95.32	95.34	94.97	94.07
16	Reconstruction	95.91	95.93	95.92	95.91	95.54	95.33	95.41	95.12	93.96
	Test	95.71	95.75	95.75	95.69	95.62	95.60	95.53	95.38	94.29
32	Reconstruction	94.37	94.38	94.38	94.38	94.39	94.39	94.38	94.39	94.40
	Test	93.03	93.03	93.03	93.03	93.03	93.03	93.03	93.03	93.03

Table A.2: Skeletonized label propagation results with a training data set at a size S of 60.000 with varying number of neighbors K and varying number of forgotten labels FP in the data set

A.2 Visualised results

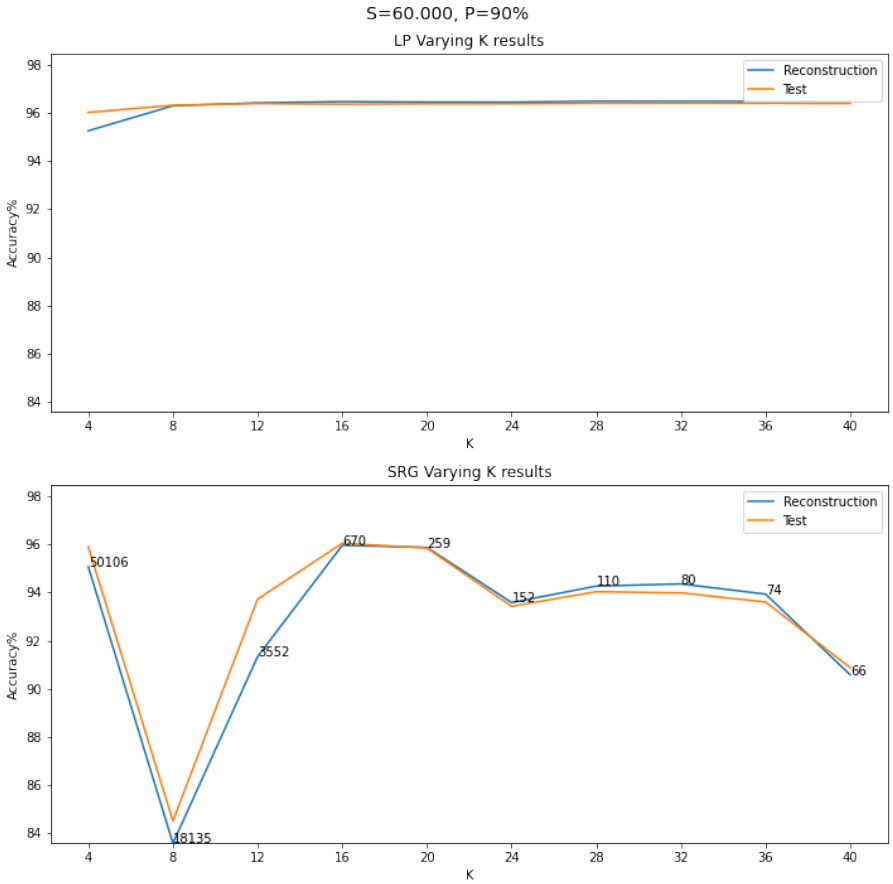


Figure A.1: Results of varying neighbors. The numbers on the plot in SRG is the number of nodes in the skeleton.

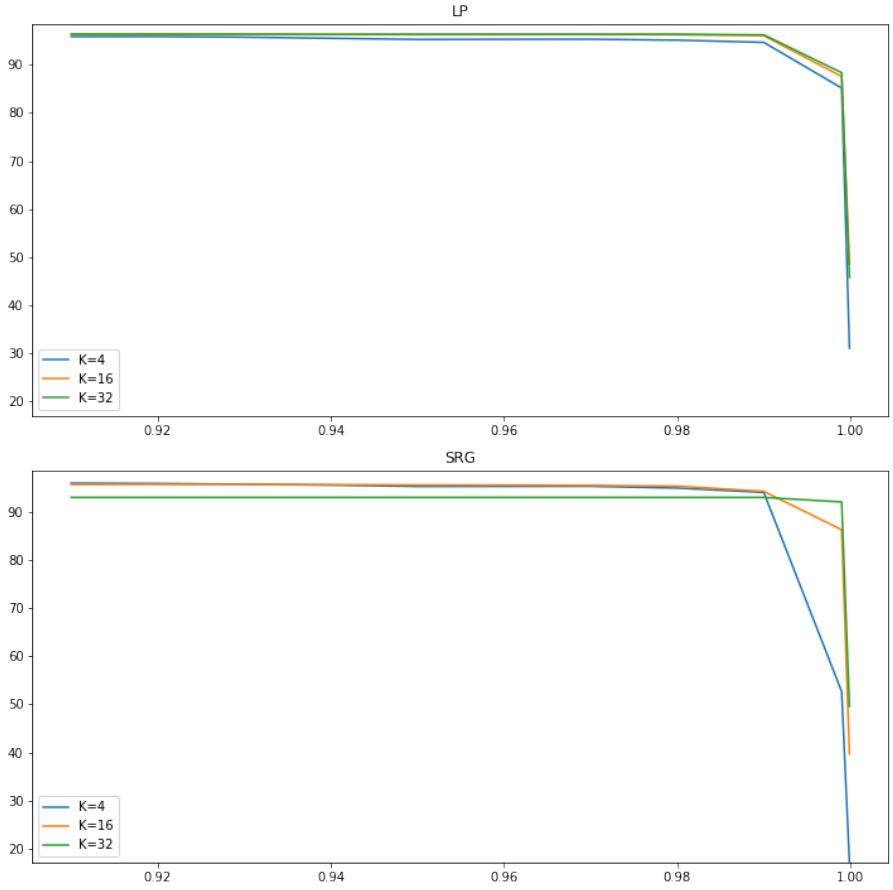


Figure A.2

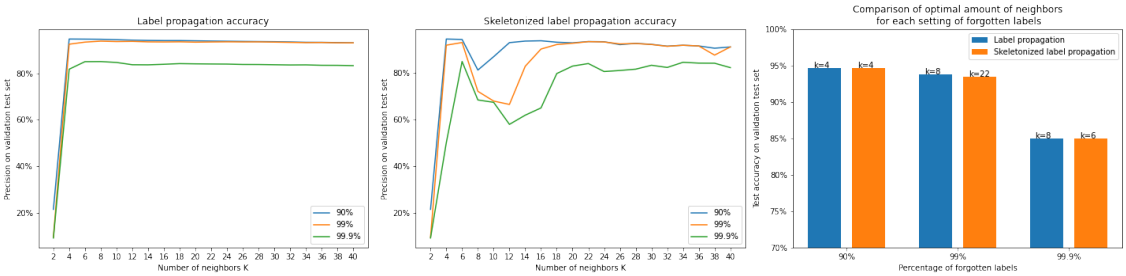


Figure A.3: Test result of running label propagation and skeletonized label propagation on a subset of the test set consisting of 5.000 points