1 Implementation

This chapter will discuss the implementation of GSSL and the incorporation of skeletonization using the front-separator algorithm using the MNIST data set. First, the graph is constructed using KNN as discussed in **??** and the test for choosing an optimal number of neighbors will be shown. The inferring of labels is done using label propagation as discussed in **??** and optimized using sparsity matrices explained in **??**. Then incorporation of skeletonization. Lastly, the implementation of testing the test data set is shown.

1.1 Implementation specification

The implementation is done in python. Python is very high level, and therefore it is straightforward to implement complex structures in less code. A downside of python is performance, where languages such as `c` are much faster. To up performance, an effort has been made to use c libraries such as `NumPy` which runs c-code to do extensive computations. In general, if something in python can be run in c, it will be faster. As such, the graph structure will be implemented in `NumPy`- and `Scipy`-arrays, where vector and matrix multiplications are done through the library will allow for a faster compute time.

1.2 Graph Construction

The graph construction is done in two steps. First, the similarity between data points are calculated into a weight matrix such that,

$$W_{i,j} = \|i - j\| \tag{1.1}$$

In this case, the euclidian is chosen as the heuristic. Afterward, the graph is constructed using the KNN method discussed in **??**.

1.2.1 Weight matrix

To construct the graph, we first need some heuristics for the similarity between the MNIST numbers. In [Son+21] we read, that there might be some underlying implicit graph-structure. Unfortunately, we do not have that luxury and instead will have to look at the similarity between images. As discussed in **??** there are several ways to approach this. For this implementation to we choose to base the similarity between images on the euclidian norm seen in eq. (1.2).

$$\|u - v\|_2 = \left(\sum \left(w_i \left|(u_i - v_i)\right|^2\right)\right)^{1/2} \tag{1.2}$$

One problem that might arise with the euclidean is that it indicates how similar the two vectors are. Looking at Figure 1.1. As we see, even though the shape is identical, the distance suggests that we are looking at not so similar shapes because the euclidian describes the similarity in the image, not in shape. To our fortune, ”The digits have been size-normalized and centered in a fixed-size image”[Yan21]. This transformation means that the position should not be a problem, and the euclidian should give a good estimate of how similar the shapes are.

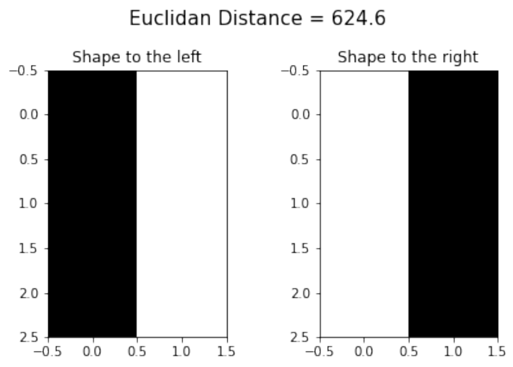


Figure 1.1: Same shape in different locations

Because the MNIST data set is normalized and repositioned, there is no danger of similar numbers being in different places, the center of each number in the image is the same.

The algorithm for calculating the weight matrix W can be seen in Listing 1.1.

```
% Weight matrix calculation
def constrcut_weight_matrix(data):
    l = len(data)
    W = [l][l]
    for i in range(l):
        for j in range(l):
            W[i][j] = sim(data[i], data[j])
    return W
```

Listing 1.1: Weight matrix calculation where sim funciton is the euclidian

The runtime of this algorithm given `x` data points is $O(n^2)$, and the space complexity is $O(n^2)$.

Optimization
To fasten the computation, threading is used, and some tweaking of the storing size is done.

Threading With threading, complications are unfortunately also introduced, as our algorithm no longer runs sequentially. Using [And00] to familiarize with the concepts, there are some key factors we must look at in regards to reading and writing. Generally, reading is always thread-safe, as no values change unless the values we are reading change. In this case, fortunately, the values (mnist numbers) do not change, as that would mean the images change too, so reading is safe. When writing to the weight matrix array, we then look at writing problems. Firstly we need to know if we can write to separate variables in the matrix simultaneously, introducing a problem. As NumPy is written in c where arrays are not dynamically changing size and by the article [1], we see that the array is instantiated as actual variables in every index. This means that the different indices can be treated as individual variables. We do not need to write to the same variable more than once, as each entry is one calculation between two image vectors. 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 weight matrix, meaning that we add a job for every image in our 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 weight-matrix, and there was not more time to spend. 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 that might be more correct but could also lead to worse performance. The new threaded algorithm can be seen in Listing 1.2

```
 % Threaded weight 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_weight_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 1.2: Weight matrix calculation where sim function is the euclidian (threaded)

[1]https://blog.paperspace.com/numpy-optimization-internals-strides-reshape-transpose/

Memory By default, values are stored with 64 bit either float or integers. If we use the whole training data set of 60.000 points with the before seen 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 if the capacity for high values is not necessary, some further adjustments might improve performance nonetheless. Having the ability to run the algorithms locally makes it less complex to develop and has been a small goal of this project; therefore, a small investigation followed.
To choose the correct bit size, we look at the maximum distance between two images: a matrix filled with 0 and an array filled with 255 with size 28×28.

$$\left\{ A = \begin{bmatrix} 0 & \dots \\ \vdots & \ddots \end{bmatrix} B = \begin{bmatrix} 255 & \dots \\ \vdots & \ddots \end{bmatrix} \right\}; |A - B| = 7140.0 \tag{1.3}$$

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 weight matrix a total of 7.2 gigabytes, almost a fourth of the previously needed memory allocation.

1.2.2 KNN Graph Construction

Graph Structure

To implement the graph, a discrete graph structure is first needed. Generally, there are three approaches to storing graphs: Adjacency list, Adjacency Matrix, and an Incidence matrix, and the choosing of the structure depends on the use case. For this work, the adjacency matrix was chosen for two reasons. Firstly if the number of K is reasonably low compared to the overall number of nodes, the matrix can be represented in a sparsity matrix as discussed in **??**. This provides a considerable benefit in computing time later when propagating the labels. Secondly, the functionality we miss out on, such as the possibility of multiple edges between nodes, is not relevant here, as the graph is undirected and the similarity between nodes are symmetric such that $W_{i,j} = W_{j,i}$.

Construction

The construction of the graph is done by looking at each row in the weight matrix and finding the K smallest values without including the current node, as we do not allow self-loops. Finding the K smallest values yields the indices of the given nodes, and the corresponding nodes are set to 1 in the graph matrix on both the row and the column. This is because we are dealing with an undirected graph, where the n'th row and m'th column indicate that n has an edge directed towards m. If both are not set to 1, the graph would suddenly be directed. An example of the operation can be seen in Equation (1.4) where $k = 1$.

$$W = \begin{bmatrix} 0 & 2388 & 2774 & 2554 \\ 2388 & 0 & 2898 & 2766 \\ 2774 & 2898 & 0 & 2572 \\ 2554 & 2766 & 2572 & 0 \end{bmatrix} \rightarrow G = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \tag{1.4}$$

The algorithm for graph construction can be seen in Listing 1.3

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

Listing 1.3: Construction of graph

The runtime of the algorithm is $O(n^2)$ as we have to

1.3 Label Propogation

The goal of label propagation is to spread known labels to unknown labels. This can be done in several ways. The basic approach is to:

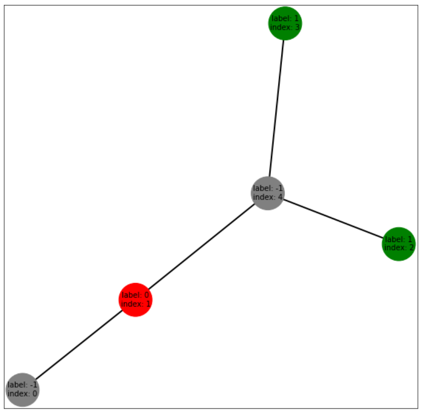
1. Propagate all labels one step such that $Y \leftarrow TY$
2. Row-Normalise Y
3. Clamp the already labeled data and continue until Y converges.

In this thesis, this is implemented as multiplying a graph matrix and a label matrix. The procedure is first to encode the labels. Then the graph and label matrices are converted to sparsity matrices to 1) save space 2) fasten matrix multiplications. Lastly, the algorithm for label propagation is done by following the above steps.

As graphs are a visual field to accompany the reader, a random graph with labels has been constructed in Figure 1.2. It will provide a visual representation of the different steps of label propagation.

1.3.1 Encoding labels

The labels are encoded to ensure that propagated labels are not confused. The encoding takes shape in a $n \times m$ matrix, where n is the number of unique labels - 10 in case of the MNIST numbers - and m is the number of data points. To continue the example from Figure 1.2 the labels encoded can be seen in Equation (1.5), where $L =$ `labels` and $OL =$ `One-hot encoded Labels`.



$$G = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

$$L = [-1, 0, 1, 1, -1]$$

Figure 1.2: Graph example, where G is the graph, and L are the labels. Labels with value -1 are unknown

$$L = \begin{bmatrix} -1 & 0 & 1 & 1 & -1 \end{bmatrix} \xrightarrow{\texttt{encode}} OL = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{1.5}$$

To represent unknown labels, all cells in a row are set to zero, as is seen in the blue rows in Equation (1.5). The space used might seem excessive considering that we use ten cells to store one label. This, however, is mostly removed when converting the matrix to a sparsity matrix. Using the sparsity matrix a list of a (coordinate, value)-pair is produced as seen in Equation (1.6) where $SOL = \texttt{Sparse One-hot encoded Labels}$, using a compressed sparse row matrix (csr_matrix).

$$\texttt{sparse(OL)} \rightarrow SOL = [\{(0,9),1\},\{(1,0),1\},\{(2,1),1\},\{(3,1),1\},\{(4,9),1\}] \tag{1.6}$$

1.3.2 Propagating the labels

The algorithm can be seen in Listing 1.4

```python
# 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
        if converge:
            break

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

Listing 1.4: Label propagation

Propagate all labels such that $Y \leftarrow TY$
To propagate the labels 1 step, the graph matrix and the label matrix are multiplied. What this effectively does is count the number of labels for all neighbors. Continuing the example from Figure 1.2 the propagated values can be seen in Equation (1.7). The first and last row is seen to have counted the number of neighbored labels, as we see that the first row with $index = 0$ has one neighbor with label 0, and the last row with $index = 4$ has one neighbor with label 0 and two neighbors with label 1. Also, notice the rows filled with zeroes. As the labeled nodes only neighbor non-labeled nodes and the non-labeled nodes are represented as rows filled with zero, the counted labels are none.

$$G \cdot OL \rightarrow p = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{1.7}$$

Row-Normalise Y
The rows are now normalized row-wise. This can be done by taking the total of each row and dividing each cell value by that total amount. Continuing from Equation (1.7) the result can be seen in Equation (1.9)

$$\texttt{normalize}(p) \rightarrow n = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.3 & 0.7 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \tag{1.8}$$

Clamp the already labeled data and continue until Y converges
To retain the already known labels, they are now reinserted or ”clamped.” To remove all information, each known node row is first set to 0 to eliminate all propagated information.

Then each known label is reinserted into the corresponding column indices. The result can be seen in Equation (1.9)

$$\texttt{clamp}(n) \rightarrow OL = \begin{bmatrix} 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.3 & 0.7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{1.9}$$

This matrix is now effectively the new labels, where unknown labels are the probability of known neighboring labels. Thus we can see that the first row in Equation (1.9) has a 100% probability of being of label 0, and the last row has a 70% probability of being a 1.

1.3.3 Optimization

To optimize the calculations, sparsity matrices are used implemented in `scipy`. The operations such as matrix multiplications performed in previous sections are supported in this framework. The main concern is to optimally construct the graph after creation in the graph construction phase and optimally edit values when we need to clamp the values.

Construction

When constructing the graph, the correct approach is to use a row-based list of lists sparse matrix (`lil_matrix`) where we add one row at a time. At the time of implementation, the graph, however, was already constructed in a conventional matrix, and there existed a direct function for converting from standard matrix to sparsity matrix, which was less efficient. As the time to convert the standard matrix to a sparsity matrix using this method was not significant however, this was not changed.

Editing the matrix

The clamping of values did, however, see some performance benefit from changing matrix form, as seen in Table 1.1

Without conversion	`to_lil`	`to_dok`
...s	...s	...s

Table 1.1: Runtime for algorithm with 60.000 entries

As the `to_lil` was faster with almost an order 10, the matrices are converted to a row-based list of lists sparse matrix before changing values.

The optimized part of Listing 1.4 can be seen in Listing 1.5

```python
# Optmizied label propogation
def propagate_labels(graph, labels, max_itter):
    labels = csr(labels)
    known_rows, known_cols = labels.nonzero()
    for _ in range(max_itter):
        #...
        # Reinsert clamped values
        n.tolil()
        n[org_row] = 0
        n[org_row, org_col] = 1
        n.tocsr()

        #...
```

Listing 1.5: Label propagation

1.4 Skeletonization

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 **??**. 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 [Son+21].

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.

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 [SP19] 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 **??**, 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 [AAB21], 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 Section 1.3.3 already acts as a list of edge pairs. As seen in Equation (1.10), the sparsity matrix already stores the edge pairs as coordinates.

$$G \xrightarrow{matrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, G \xrightarrow{sparsity} [((0,1),1),((1,0),1)] \tag{1.10}$$

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 [SP19] 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

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

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 Section 1.3.3, 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.

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 **??**, 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.