Course-ID:	BA-INF 051
Course:	Wissensentdeckung und Maschinelles Lernen: Wissensentdeckung
Term:	Summer 2021
Supervisor(s):	Prof. Dr. Stefan Wrobel Dr. Tamas Horvath Dr. Pascal Welke Till Schulz Sebastian Müller

UNIVERSITÄT BONN ML AI lab

Prototype-based Learning Algorithms

Guo Chen Lennart Metz Linus T. Mallwitz

September 15, 2021

Abstract

Prototype-based Learning Algorithms are a category of artificial neural networks that take inspiration from Hebbian Learning. In this report, we will introduce and evaluate 4 different prototype-based algorithms. The first two algorithms are Kernel Generalized Learning Vector Quantization (KGLVQ) and Generalized Matrix Learning Vector Quantization (GMLVQ) from the LVQ algorithm family with a purpose of data pattern classification. The other two algorithms are the Self-organizing Map (SOM) and the Growing Neural Gas (GNG), which are designed for data representations and visualizations.

Guo

1 Introduction

The Hebbian Learning Theory proposed by Donald O. Hebb in 1949 is one of the earliest and most common learning rules for neural networks. The concept of this theory is to simulate the psychological and neurological learning process of the neurons in our brains. When we try to learn something new, for example, the neurons in our brains will be activated and begin to form connections with one another. These connections are usually weak at first, but as we get more experience with our learning process, the neurons are activated more frequently, resulting in stronger connections that will eventually form a neuron network[7]. Guo

Inspired by this theory, the concept of prototypes is introduced. In short, the prototypes behave like neurons, and the weights are induced by the spatial proximity principle (things that are close together are perceptually grouped together[19]). This concept is the basis of our main topic in this report: the Prototype-based Learning Algorithms.

In Prototype-based Learning Algorithms, every data sample from the input space has a class label, and the data samples that share the same class label are put into one class. A certain number of prototypes with the same class label are generated and then assigned to each class, and they can be seen as reference vectors that represent the main characteristics of the data samples in that class. In the learning process, one or more best matching units (BMU) for the current sample will be chosen, and the weights or distances between them will be updated following the rules.

In this report, we will discuss two types of algorithms derived from the Prototype-based Learning Algorithms. The first one is called Learning Vector Quantization (LVQ), which is a supervised classification algorithm designed to offer simple and straightforward solutions for pattern classification tasks. To be specific, we will introduce two new LVQ algorithms based on one commonly used LVQ variant: Generalized Learning Vector Quatization (GLVQ). These two algorithms are Kernel Generalized Learning Vector Quantization (KGLVQ) and Generalized Matrix Learning Vector Quantization (GMLVQ), respectively.

The second type is the Self-organizing Map (SOM) and its derivation: Growing Neural Gas (GNG). These two algorithms belong to the unsupervised machine learning algorithm category, and they focus on finding optimal data representations based on prototypes, which makes high-dimensional datasets easier to visualize and analyze.

2 Fundamentals of prototype-based learning

In this section we will present the basic principles of prototype-based learning including terminology, notation and the GLVQ-algorithm that two of our subtopics are build upon. Lennart

2.1 Basics

For the regular classifcation algorithms of prototype-based learning methods or, learning vector quantiziation, as introduced in by Kohonen in 1995 we define the following [10]. Let X be a set of M training samples $(x_n, y_n) \in \mathbb{R}^N \times \{1, ..., C\}$ consisting of N-dimensional data and an associated class. The data defines the input space, which consequently is N-dimensional as well. Central to the classification are the prototypes $m_k \in \mathbb{R}^N$ with $k \in \{1, ..., K\}$ which occupy the same input space as our training data. Every prototype is given a class label $c(m_k) \in \{1, ..., C\}$ at initialization that stays fixed for the entirety of the training process. Note that at least one prototype per class is needed. The values of each prototype, an N-dimensional vector, is subject to change during training and in fact the adjustment of the prototypes constitutes the learning.
Additionally we define a distance measure $d(x, m)$ denoting the distance between a data sample and a prototype. Here standard euclidean distance $d(x, m) = \sqrt{(x-m)^T(x-m)}$ is most commonly used when handling euclidean data and for now we will be referring to it as well when discussion distances. In theory it is possible to apply prototype-based learning methods to non-euclidean data under the condition one defines a sensible distance measure, however we will limit us to euclidean data in this project.
Using the distance measure we can classify unseen data samples. For a data sample x we calculate the distances $d(x, m_k)$ to each prototype and the sample assumes the class label of the closest prototype, i.e. the one with the smallest distance. This way the set of prototypes divide the input space into subspaces where every point in that subspace is closest to one prototype and therefore associated with the class of that prototype. In figure 1 this concept is visualized.
The objective during the training process is to repeatedly adjust the position of the prototypes so as to minimize the number of missclassifcations via the closest distance criterion or in other words, move the prototypes so that as few as possible training samples are situated in a subspace associated with a class label other than that of the sample. There are a multitude of training algorithms and in the next section we will present one specific method, the GLVQ algorithm.

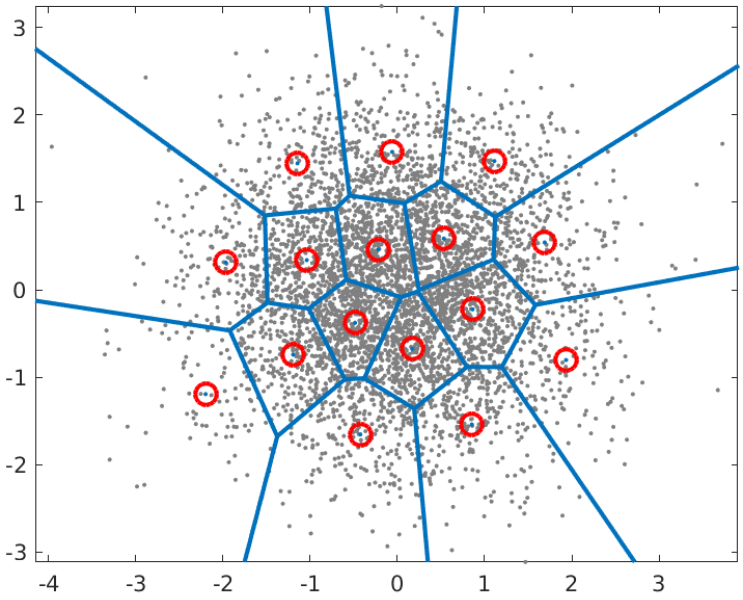


Figure 1: Prototypes dividing the input space into subspaces. The red circles mark the position of prototypes, the grey dots are data samples and the blue lines, called decision boundaries, indicate the borders of the subspaces. Image source: [2]

2.2 GLVQ

With the fundamentals defined we will now present the training process of generalized learning vector quantization [13]. Given a training sample (x_n, y_n) we calculate the distances $d(x_n, m_k)$ for each prototype $k \in \{1, ..., K\}$. Now we find the prototype m_i with the smallest distance d_i and the *same* class label as x_n and the prototype m_j with the smallest distance d_j and a *different* class label. Through this we can calculate a relative distance measure $\mu(x_t) = \dfrac{d_j - d_i}{d_j + d_i}$.
Additionally one needs to define an activation function $f(\mu(x_t))$ such as the identity function or $tanh(x)$, with the condition that it is monotonically increasing. Finally we can formulate the central part of the GLVQ algorithm, the cost function, which is a summation over all M data samples:

$$S = \sum_{t=1}^{M} f(\mu(x_t))$$

We now minimize S using gradient descent. For this we update the two prototypes m_i and m_j according to the partial derivatives $\dfrac{\partial S}{\partial m_i}$ and $\dfrac{\partial S}{\partial m_j}$:

$$m_i = m_i + \alpha \frac{\partial f}{\partial \mu} \frac{d_j}{(d_i + d_j)^2} (x - m_i)$$
$$m_j = m_j - \alpha \frac{\partial f}{\partial \mu} \frac{d_i}{(d_i + d_j)^2} (x - m_j)$$

Note that the effect on the two prototypes is essentially inversed, since we want to move the same-class prototype closer and the different-class prototype further away from the data sample. These update steps are performed for each data sample and applying this to all M data samples in succession constitutes 1 epoch of the training process. α is the learning rate and should be a small positive constant.

3 Kernel Generalized Learning Vector Quantization

3.1 Motivation

Guo

The Generalized Learning Vector Quantization (GLVQ) algorithm proposed by Sato[13] has been one of the most popular and stable variants among all the LVQ based algorithms for a long time and has shown excellent results in fields like character recognition. However, the GLVQ algorithm shows weaker compatibility when processing complex datasets with nonlinear class boundaries and usually gives an averagely low classification accuracy. Besides this, datasets in modern world applications are frequently represented in a non-vectorial form, for example, in node-based graphs. Thus, conventional pattern classification algorithms like GLVQ will face a significant obstacle in dealing with such complex datasets.

3.2 Solution

Kernel Generalized Learning Vector Quantization proposed by A. K. Qin and P. N. Suganthan[12] is an extension to GLVQ that can process datasets with complex data structures and non-vectorial datasets, while still giving outstanding performance with a high classification rate.

3.2.1 Mapping Function

In this kernel-inspired algorithm, a mapping function $\phi(\cdot)$ is introduced with the purpose of transforming non-linearly separable datasets into linearly separable datasets. To realize this transformation, the mapping function $\phi(\cdot)$ will map the data sample x_i from lower dimensional space into the higher dimensional feature space F as $\phi(x_i)$. Thus, linear data separation becomes possible in the higher dimension. Following this logic, a prototype in the feature space W_j^F can be defined using some linear combination of all mapped data samples with weights:

$$W_j^F = \sum_{m=1}^{N} \gamma_{jm} \phi(x_m) \tag{1}$$

Here, N is the size of the dataset, j is the index of the prototype, $\phi(x_m)$ is the mapped data sample x_m, and γ_j represents the coefficient vector of the j-indexed prototype. Each coefficient vector γ_j stores N weights in it, and γ_{jm} is the weight that corresponds to the m-indexed data sample. In this formula, we use all data samples from the input space to express a prototype, because the exact dimension of the feature space is unknown. However, in reality, due to the complexity of operations in a higher dimension, the mapping function is usually not implemented. To reduce the computational expense, some alternative implementations are proposed. We will introduce them later.

3.2.2 Kernel Function

As previously stated, because of the impracticability of the mapping function, we need to find a method to bypass this step. Hence, the kernel function is presented with a purpose of computing the dot product of two data samples implicitly and thus converting all computations in the feature space F into lower-dimensional operations. As a result, the complexity of the algorithm is heavily reduced and the implementation becomes possible. In this paper, we use the Gaussian Kernel Function for vectorial datasets and the Weisfeiler-Lehman Kernel Function for non-vectorial graph datasets.

3.2.3 Kernel Matrix

A kernel matrix is also introduced to lighten the computational expense. It is defined as:

$$K = (k_{st})_{N \times N} \tag{2}$$

The square matrix K stores kernel function results of all possible combinations of two data samples from the dataset. In practice, we use these kernel matrix results to express the formula for feature space distance. The feature space distance d_{ij} from sample i to prototype j is defined as below:

$$\begin{aligned} d_{ij} &= \left\| \phi(x_i) - W_j^F \right\| = \sqrt{\left\| \phi(x_i) - \sum_m^N \gamma_{jm} \phi(x_m) \right\|^2} \\ &= k(x_i, x_i) - 2 \cdot \sum_{m=1}^{N} \gamma_{jm} k(x_i, x_m) + \sum_{s,t=1}^{N} \gamma_{js} \gamma_{jt} k(x_s, x_t) \end{aligned} \tag{3}$$

The equation is expanded from a conventional euclidean distance form, where $\phi(x_i)$ is the data sample and $W_{\overset{}{j}}^F$ represents the prototype. As we can see from the equation, whenever we need to perform computations in the feature space, we can simply extract the required corresponding kernel function results of the two data samples (for example, $k(x_s, x_t)$ is the result from sample x_s and sample x_t) from the kernel matrix and obtain the results instantly. Since we can not implement the mapping function in reality, we are also not able to express the feature space prototypes in the mapping function form. Hence, we let the coefficient vectors γ_j from Equation 1 represent the prototypes. After each training, the weights inside each coefficient vector are modified, therefore the changes in prototypes are updated as well.

3.2.4 New Update Rules

The new update rules can be defined as:

$$\begin{aligned} \gamma_{ks}(t+1) &= \begin{cases} [1 - \text{coeff} \cdot \frac{4 \cdot d_l^F}{d_k^F + d_l^F}] \cdot \gamma_{ks}(t) & \text{if } x_s \neq x_i \\ [1 - \text{coeff} \cdot \frac{4 \cdot d_l^F}{d_k^F + d_l^F}] \cdot \gamma_{ks}(t) + \text{coeff} \cdot \frac{4 \cdot d_l^F}{d_k^F + d_l^F} & \text{if } x_s = x_i \end{cases} \\ \gamma_{ls}(t+1) &= \begin{cases} [1 + \text{coeff} \cdot \frac{4 \cdot d_k^F}{d_k^F + d_l^F}] \cdot \gamma_{ls}(t) & \text{if } x_s \neq x_i \\ [1 + \text{coeff} \cdot \frac{4 \cdot d_k^F}{d_k^F + d_l^F}] \cdot \gamma_{ls}(t) - \text{coeff} \cdot \frac{4 \cdot d_k^F}{d_k^F + d_l^F} & \text{if } x_s = x_i \end{cases} \end{aligned} \tag{4}$$

coeff $= \epsilon \cdot \frac{\partial l(\mu_k(\phi(x_i);W_F))}{\partial(\mu_k(\phi(x_i);W_F))}$, ϵ is the learning rate, and $(\cdot)$ is the cost function, which is defined as $l(\mu) = \frac{1}{1+e^{-\epsilon(t)\cdot\mu}}$. $\epsilon(t)$ is the time coefficient that increases after each sample iteration, $\epsilon(t) = 1.0001 \cdot \epsilon$ and $\epsilon(0) = 1$. $\mu(\cdot)$ is the misclassification measure, which is redefined as $\mu_k(x_i; W) = \frac{d_k^F - d_l^F}{d_k^F + d_l^F}$ for computations in the feature space. A classification is correct when the result of μ is negative. $\frac{\partial l}{\partial \mu_k}$ is the gradient descent of the cost function, which is simplified to $\partial l \cdot (1 - \partial l)$ in the actual implementation. As we have mentioned before, because we can not use a mapping function to represent the prototypes in the feature space, the indexed coefficient vectors in the new update rules are now used to represent the prototypes. Here, γ_k represents the coefficient vector of the closest prototype with the same label as the input data sample, and γ_l is the other way around.

The algorithm will iterate through all the data samples from the dataset as t increases. x_s represent all the data samples from the dataset, because each weight in the coefficient vector matches one data sample. In each sample iteration, the weight of sample $x_s = x_i$ will be updated, and the rest of the weights, where $x_s \neq x_i$, will be normalized. We repeat this update process for many epochs until the cost function curve becomes flat and the prototypes converge.

3.3 Experiments

3.3.1 Preliminary Experiments

Before comparing the performance of the KGLVQ and GLVQ models, we will do a few test runs to examine the new KGLVQ algorithm. In our tests, we use the 2D "make blobs" dataset from [17] for a better visualization, so we can interpret the results intuitively.

The dataset is set to have 300 samples divided into 6 classes, and each class has 1 prototype. The standard deviation of the clusters is set to 0.3, which is relatively low, so that the data structure is simple and clean. The learning rate ϵ is set to 0.01 for all, and every test will run for 100 epochs. Since the data samples are in vector form, the Gaussian Kernel Function is chosen for the tests.

The results are shown in Fig.2 and Fig.3. As seen in both figures, the KGLVQ model gives 100% accuracy under both kernel parameter values. The prototypes in each class are accurately located in the convex hull defined by the data samples, which means that they are in convergence under both conditions. Thus, we can say that the KGLVQ model can give perfect results under an optimal condition.

However, when we raise the standard deviation of the cluster to 0.9, a few prototypes will start to move out of the convex hull when under a higher kernel parameter value. In Fig.5 we can see that, when the Bayesian boundaries of each class are overlapping and not piece-wise linear anymore, kernel parameter $\sigma = 1.1$ will lead to such result. But it is worth noting that the prototypes will not move further as the cost function curve starts to flatten, which means that they are still in convergence, and the resulted accuracy also dose not deteriorate either. This means that the KGLVQ model can still perform well when dealing with complex datasets, but finding a matching σ value for the current dataset is very important.

3.3.2 Comparison with GLVQ

In this section, we will compare the performance of the KGLVQ and GLVQ model on different datasets and under different parameter settings.

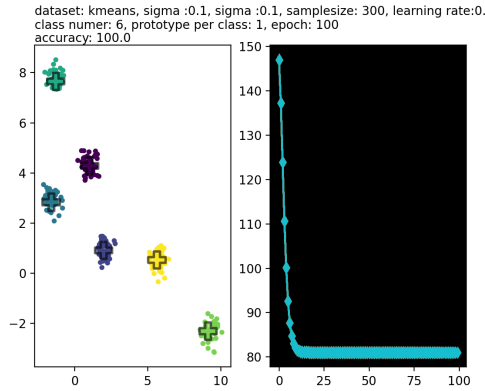


Figure 2: 2D Visualization of test run on generated "make blobs" dataset, $\sigma = 0.1$

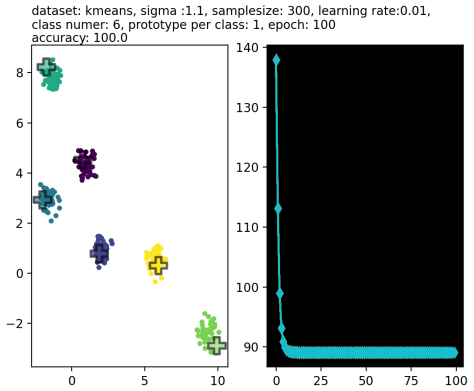


Figure 3: 2D Visualization of test run on generated "make blobs" dataset, $\sigma = 1.1$

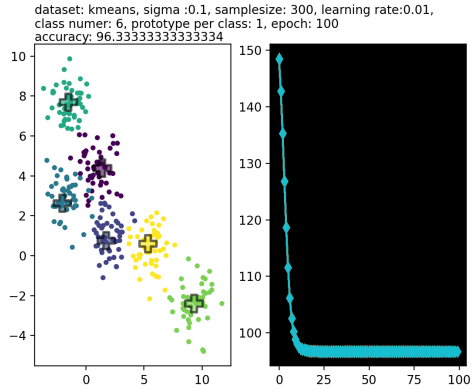


Figure 4: 2D Visualization of test run on sparsely-clustered generated "make blobs" dataset, $\sigma = 0.1$

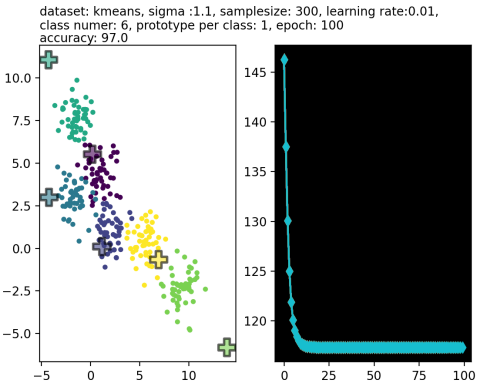


Figure 5: 2D Visualization of test run on sparsely-clustered generated "make blobs" dataset, $\sigma = 1.1$

Generated Datasets We first choose 2 datasets generated with the methods from Sklearn[17]. The first dataset (see Fig.6) has 4 classes, each class has 200 data samples and overlapping class boundaries. The second dataset (see Fig.7) has 2 classes, with 300 data samples in each class and a curve shape class boundary.

In both datasets, the data samples in every class are still relatively tightly clustered, which means that one class has only one substructure, so we set the prototype number per class also to one. In order to evaluate and compare the performance of both algorithms, we applied 10-folder cross validation method for each algorithm. Each dataset is evenly split into 10 folds and the final result is averaged over 10 runs. We set the learning rate ϵ to 0.01 for all experiments, and let each model run for 200 epochs until the cost function curves become flat. For the KGLVQ model, the kernel parameter value σ has a range from 0.1 to 1.1.

The results are shown in Table 1. We can see that both models have good performance on these two datasets, and the average classification rates (CR) of the KGLVQ algorithm are sightly better, even for the whole value range of the kernel parameter σ. However, only in the "make blobs" dataset, the standard deviation is smaller than the result from GLVQ.

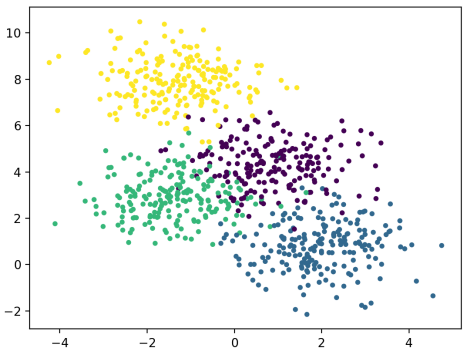


Figure 6: 2D Visualization of "make blobs" Dataset

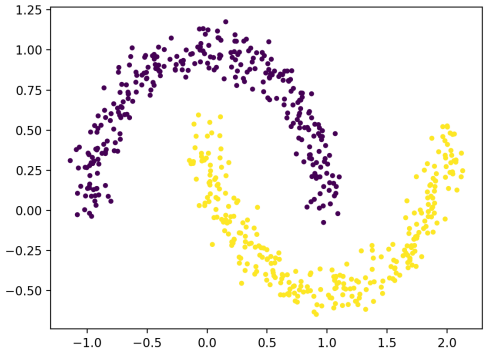


Figure 7: 2D Visualization of "make moons" Dataset

Table 1: Results of "make blobs" and "make moons" datasets in GLVQ and KGLVQ

Dataset	GLVQ	KGLVQ (σ: 0.1 ~1.1)
make blobs	0.9350 ± 0.0305	0.9450 ± 0.0214
make moons	0.8167 ± 0.0553	0.8209 ± 0.0878

We then set the σ in the KGLVQ to 1.1 as it shows the best CR for both datasets, and compare the performance differences with more prototypes in each class. The results in Table 2 and Table 3 display a better CR and a smaller standard deviation in the KGLVQ. Thus, we can say that, when the best matching σ value is chosen, the KGLVQ algorithm can result a better accuracy on average. It is also worth noting that the increased prototype numbers in each class have also contributed to a better classification rate here.

Realworld Datasets In order to analyze the performance differences of both models in real life, we also choose two benchmark datasets from UCI[21].The first one is called "Hepatitis"[8], which samples the characteristics of different groups of people with hepatitis and their medical status. It consists of 154 data samples witch 19 attributes, and these samples are divided into 2 classes by the life status. The second dataset is called "Arrhythmia"[1], which aims is to distinguish between the presence and absence of cardiac arrhythmia and to classify it in one of the 16 classes. This datasets has 451 data samples with 279 attributes. The missing values in both datasets are replaced by "0" for a more convenient computation. Since both datasets are from the real world, the class distribution is more likely to be complicated, which provides us with a perfect condition to test the ability to classify datasets with complex data structures in both algorithms.

We set the learning rate ϵ to 0.01 and run the algorithms for 200 epochs in σ range from

Table 2: Classification rates of "make blobs" dataset under different prototype number per class

Num of Proto.	GLVQ	KGLVQ ($\sigma = 1.1$)
1	0.9350 ± 0.0305	0.9525 ± 0.0175
2	0.9388 ± 0.0276	0.9588 ± 0.0186
3	0.9388 ± 0.0293	0.9638 ± 0.0153

Table 3: Classification rates of "make moons" dataset under different prototype number per class

Num of Proto.	GLVQ	KGLVQ ($\sigma = 1.1$)
1	0.8167 ± 0.0553	0.8733 ± 0.0318
2	0.8817 ± 0.0311	0.9483 ± 0.0157
3	0.8967 ± 0.0245	0.9607 ± 0.0149

Table 4: Results from "Hepatitis" and "Arrhythmia" datasets in GLVQ and KGLVQ, σ range from 0.1 to 1.1

Dataset	GLVQ	KGLVQ (σ: 0.1 $\sim$1.1)
Hepatitis	0.7063 $\pm$ 0.1188	0.8182 $\pm$ 0.0894
Arrhythmia	0.8130 $\pm$ 0.0577	0.9891 $\pm$ 0.0326

Table 5: Classification rates of "Hepatitis" dataset in GLVQ and KGLVQ under σ=0.6

Num of Proto.	GLVQ	KGLVQ ($\sigma = 0.6$)
1	0.7063 $\pm$ 0.1188	0.8186 $\pm$ 0.0904
2	0.7063 $\pm$ 0.1188	0.8375 $\pm$ 0.0800
3	0.7063 $\pm$ 0.1188	0.9938 $\pm$ 0.0186

Table 6: Classification rates of "Arrhythmia" dataset in GLVQ and KGLVQ under σ=0.6

Num of Proto.	GLVQ	KGLVQ ($\sigma = 0.6$)
1	0.8130 $\pm$ 0.0577	0.9891 $\pm$ 0.0326
2	0.8130 $\pm$ 0.0577	0.9957 $\pm$ 0.0088
3	0.8107 $\pm$ 0.0608	0.9935 $\pm$ 0.0100

0.1 to 1.1, and the results can be seen in Table 4. For both datasets, we can observe that the KGLVQ model has a higher classification rate and a lower standard deviation.

Then we choose the best matching σ values for both datasets, and compare the performance when more prototypes are introduced in each class. Here, because the results are mostly the same from the range 0.2 to 1.1 for both datasets, we simply choose σ as 0.6, and other parameters stay unchanged.

From Table 5 and Table 6 we can easily see that both datasets also have averagely higher classification rates in the KGLVQ model than the GLVQ model, while the standard deviations remain smaller. However, the increased prototype number per class does not always improve the performance (see Table 6), which is different from the results of the generated datasets. Therefore, we can conclude from the experiment results above that the KGLVQ algorithm can give better performance when dealing with real-life datasets that have non-linear class boundaries, while classification rates in the GLVQ model can deteriorate.

3.4 Experiments with non-vectorial datasets

Because of the kernel function, the KGLVQ algorithm is also able to interpret non-vectorial datasets. Thus, we can now classify graph datasets. For the experiments, we pick 3 node-based graph datasets from TU Dortmund[20]. The first one is the "MSRC_9" dataset that contains 221 graphs and 8 classes with a single number as the label for each node. The second one is the "Cuneiform" dataset that contains 267 graphs and 30 classes. For each node, there are 2 integers as the label, and each node has 3 floating point numbers as the attributes. Every edge also has a label that is expressed as a single integer, and each edge has 2 floating point numbers as the attributes. The third one is called "Enzyme", which consists of 600 graphs and 6 classes. It has single integer node labels and node attributes expressed as 18 floating point numbers for each. Since the Gaussian Kernel Function is not capable of computing graph-based data samples, we will use the Weisfeiler Lehman Kernel Function here instead. The GraKel[6] API is used in the implementation to compute the Weisfeiler Lehman kernel matrix. The number of iterations inside the kernel will be referred as the kernel parameter "n_iter" here. We first perform a test run with different kernel parameters on the "MSRC_9" dataset for 200 epochs, learning rate ϵ = 0.01, and 1 prototype per class.

The results can be found in Fig.8 and Fig.9. We can see that under both kernel parameter values, the KGLVQ model yield good performance. However, when the kernel

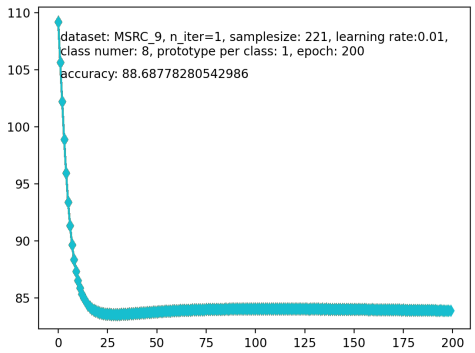


Figure 8: Results from "MSRC_9" under kernel parameter = 1

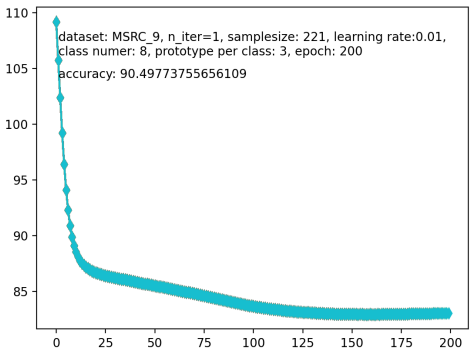


Figure 9: Results from "MSRC_9" under kernel parameter = 30

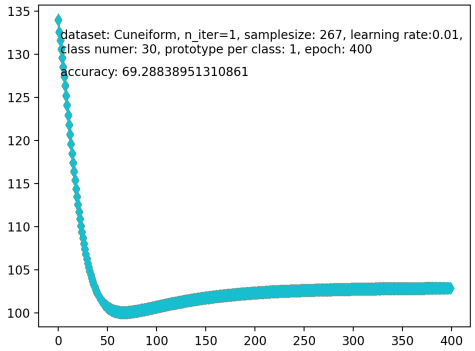


Figure 10: Results from "Enzyme" under kernel parameter = 1

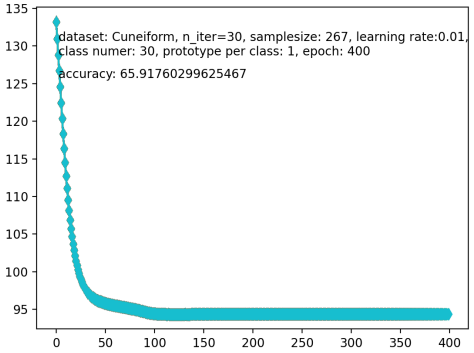


Figure 11: Results from "Enzyme" under kernel parameter = 30

parameter is higher (see Fig.8), the accuracy is slightly better, while it takes more epochs for the prototypes to converge.

This does not necessarily mean that a higher kernel parameter value can guarantee a better result. Let us do another test run on the "Cuneiform" dataset for 400 epochs, while the other conditions remain unchanged. From the results in Fig.10 and Fig.11 we can see that, although a higher kernel parameter value results a better convergence status (No dips in the cost function curve), the final accuracy is actually worse. Apparently, choosing a matching kernel parameter for the current dataset in the Weisfeiler Lehman Kernel is as important as it is in the Gaussian Kernel.

For the actual performance evaluation, the 10-folder cross validation method is applied. We first choose the best matching kernel parameter value for each dataset. In

Table 7: Classification rates from all 3 graph datasets under different prototype number per class

Num of Proto.	MSRC_9 (n_iter= 20)	Cuneiform (n_iter=4)	Enzyme(n_iter=15)
1	0.9978 ± 0.0667	0.7500 ± 0.0769	0.9483 ± 0.0263
2	1.0000 ± 0.0000	0.7944 ± 0.0617	1.0000 ± 0.0000
3	1.0000 ± 0.0000	0.8167 ± 0.0417	1.0000 ± 0.0000

"MSRC_9", the results are the best and stay unchanged when the kernel parameter is equal or above 20, thus we choose "n_iter"=20, and the "Enzyme" dataset has also similar results, thus we choose "n_iter"=15 for it. In "Cuneiform", the results are the best at "n_iter"=4. Then we let the algorithm run for 400 epochs under the same conditions, but with more prototypes per class this time. The results in Table 7 all indicate good performance from all 3 datasets, only 'Cuneiform' is slightly worse. The classification rates are also higher when we increase the number of prototypes per class, while the standard deviation decreases as well. As can be seen, the KGLVQ model has strong compatibility with non-vectorial datasets.

3.5 Conclusion

In this section, we have reviewed and evaluated the Kernel Generalized Learning Vector Quantization (KGLVQ) algorithm, which is an extension of the Generalized Learning Vector Quantization (GLVQ) algorithm. In this algorithm, the data samples and prototypes are mapped into the feature space, which is in a higher dimension. And the concept of the kernel function and the kernel matrix is also introduced to make computations in the feature space possible. As a result, the KGLVQ algorithm can averagely give better classification rates than the GLVQ algorithm when dealing with complex datasets with non-linear class boundaries. Particularly on real-world datasets, the KGLVQ model has demonstrated its superior compatibility with complicated data structures. With the help of the kernel function and kernel matrix, the KGLVQ is now also capable of classifying non-vectorial graph datasets, while giving excellent classification results. This is one feature that the original GLVQ algorithm lacks.

However, the performance of KGLVQ can vary greatly between datasets. When dealing with simple data structures, the KGLVQ model does not necessarily enhance classification rates significantly, but its computation time is much longer than the GLVQ model. Furthermore, since a $N \times N$ kernel matrix is always required as the initialization of the algorithm, the efficiency of the KGLVQ model can deteriorate intensively when dealing with datasets that have an enormous number of data samples and attributes. The value of kernel parameter does influence the final results as well, choosing an unsuited kernel parameter value can lead to worse performance. Thus, more test runs are needed when we are trying to obtain the optimal results from the KGLVQ model.

To summarize, the new KGLVQ algorithm produces superior classification results on real-world datasets with complex data structures and has good compatibility with graph-based datasets, but it is less efficient than the GLVQ model. The choice between the KGLVQ and the GLVQ is largely influenced by the data structure complexity of a dataset.

4 GMLVQ

In this section we will discuss another augmentation of the GLVQ algorithm as presented in 2.2. The so called generalized matrix learning vector quantization (GMLVQ), as first introduced by Sato and Yamada in 1995, falls in the realm of relevance learning vector quantization [13]. In simple terms this area extends the regular learning vector quantization by weighting the different input dimensions, or features, according to their relevance for the given task. The relevance of the dimensions is also learned during training.

4.1 The GMLVQ algorithm

Now we will examine the GMLVQ algorithm in detail. As aforementioned the GMLVQ is directly based on GLVQ and only modifies a few aspects of the latter in order to achieve the relevance learning. Firstly the distance measure $d(x, m)$ between a data sample and a prototype as introduced in section 2.1 is modified to:

$$d^{\Lambda}(x, m) = (x - m)^T \Lambda (x - m), \quad \text{with } \Lambda = \Omega^T \Omega$$

Here Λ is a $N \times N$ matrix for $x \in \mathbb{R}^N$ and $\Lambda = \Omega^T \Omega$ ensures that Λ is positive semi-definite and symmetric. This is necessary to guarantee the transformation into a euclidean space of the same dimension. Note that the n-th column and row correspond to the n-th input dimension and since Λ is part of our new distance measure, they determine how much the n-th dimension influences the final value of the distance. Ideally after successful training each element $\Lambda_{g,h}$ should represent the importance for the classification of the input dimension g and h in correlation with each other. Consquently the diagonal of Λ represents the relevance of the individual dimensions. It can be considered a weight matrix for the input dimension. Another way to interpret its function, is that it scales and rotates the data samples according to how well each dimension separates the data.
Since we modified our distance measure, we also have an altered cost function and thus receive a different result when calculating the update rules for the prototypes m_i and m_j by minimizing the cost function:

$$m_i = m_i + \alpha \frac{\partial f}{\partial \mu} \frac{d_j}{(d_i + d_j)^2} \Lambda (x - m_i)$$
$$m_j = m_j - \alpha \frac{\partial f}{\partial \mu} \frac{d_i}{(d_i + d_j)^2} \Lambda (x - m_j)$$

Furthermore we also need to update the weight matrix Λ. We do this indirectly by updating its decomposition matrix Ω:

$$\Omega_{l,k} = \Omega_{l,k} - \alpha \cdot \frac{\partial f}{\partial \mu} \cdot \left(\frac{d_i^{\Lambda}}{(d_j^{\Lambda} + d_i^{\Lambda})^2} \cdot \left((x_k - m_{\mathbf{j},k}) [\Omega (x - m_{\mathbf{j}})]_l \right) - \frac{d_j^{\Lambda}}{(d_j^{\Lambda} + d_i^{\Lambda})^2} \cdot \left((x_k - m_{\mathbf{i},k}) [\Omega (x - m_{\mathbf{i}})]_l \right) \right)$$

This update rule essentially increases the weight of the dimensions in which the data sample was close to the same class prototype and decreases the weight of dimensions in which the different class prototype was close. Similarly to the prototype update it is also applied after every data sample. Finally to ensure stability of the algorithm we have to normalize Λ after every update step, so that

$$\sum_i \Lambda_{ii} = 1$$

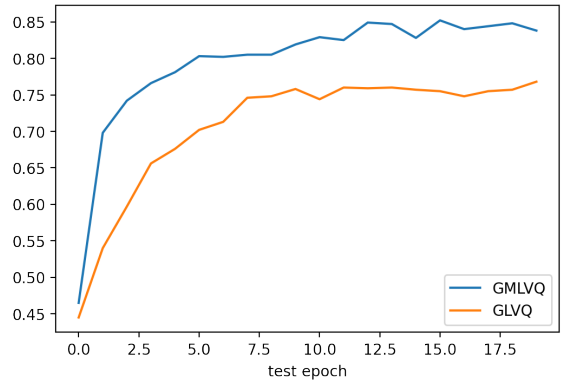
We achieve this dividing all elements of Ω by $\sqrt{\sum_i (\Omega^T \Omega)_{ii}}$.

4.2 Experiments

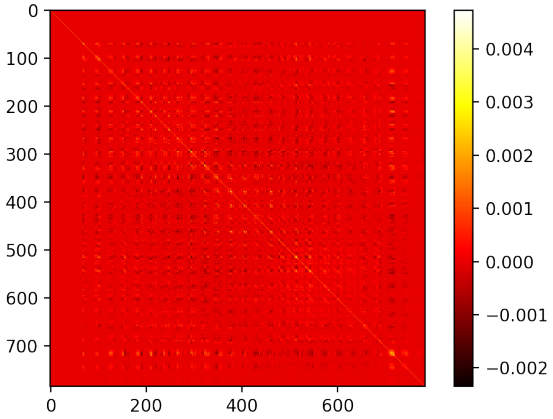
We conduct multiple experiments using the GMLVQ algorithm. Our goal is to test the classification ability of the GMLVQ and especially its ability to identify the relevant input dimensions. In our implementation of the algorithm we followed the methodology as presented in 2.2 and 4.1. For the activation function f we use the identity function and the learning rates are set between 0.001 and 1. We set a different learning rate for the update of the elements of Ω, one order of magnitude lower than the learning rate of prototype updates. The prototypes were initialized by taking random data samples of the required class and the matrix Λ was initially set as the identity matrix.

4.2.1 MNIST

For the first experiment we choose the popular MNIST dataset of handwritten digits. It consists of 28x28 pixel images with one of the numbers 0 through 9 handwritten and centered in the middle . They are 8-bit grayscale images, meaning that the pixel values range from 0 (black) to 255 (white). Since the GMLVQ algorithm cannot handle the 2-dimensional grid structure, the images have to be converted into a single 784 long vector. Additionally all values are normalized to be within 0 and 1 by dividing them by 255. Early tests proved a very long runtime for the algorithm. This is caused primarily by updating the matrix Λ, since it is a 784×784 matrix where each element has to be updated individually, which in turn is performed for every data sample. Therefore we limit us to 10000 images for training and 1000 for testing in order to reduce the runtime of the algorithm. Similarly the experiment is conducted with only one prototype per class, so in this case 10 prototypes, one for each digit. We train the GMLVQ algorithm on the data for one epoch and apply the classifier to the test samples every 500 data samples. We run the same experiment using the regular GLVQ algorithm for comparison.
The results are shown in figure 12a. We observe that the algorithm reaches an accuracy of about 0.85. Considering we used only one single prototype per digit, we find this to be a strong result. Furthermore we can see that the GMLVQ achieves a significantly higher accuracy than the GLVQ being executed with the same hyperparameters. This shows that the usage of a weight matrix in the distance measure can improve the classification ability of the algorithm. Figure 12b showcases the matrix Λ of learned relevances of the 784 input dimensions in correlation with one another. We can clearly see a distinct diagonal line with little other bright spots. This leads to the interpretation that the algorithm found the individual input dimensions to be important and very little correlated relevance.



(a) Accuracy of the GMLVQ model on the test data over 1 epochs



(b) Heatmap of the relevance matrix Λ after the training

Figure 12: MNIST GMLVQ

One strong suit of prototype based learning over other machine learning algorithms is interpretability of the classification process. In this case we can extract the prototypes which are 784 long vectors and reshape them back into 28 x 28 pixel images. Additionally all prototypes were normalized so that they values lie between 0 and 255. The prototypes of all ten digits can be seen in figure 13. They are clearly defined structures and are easily identifiable as the digit they are supposed to represent. Interestingly the individual digits are drawn in a non uniform manner. Some prototypes show thick lines while others construct their digits using very thin lines. Taking a look at likely the most similar digits, 1 and 7, we see that the vertical line of the 1 is around twice as wide as that of the 7. A possible explanation for this pattern is that this serves to better distinguish prototypes of similar digits, thus being a beneficial position of the prototypes in our input space and being found as a minimum of the loss function using gradient descent.

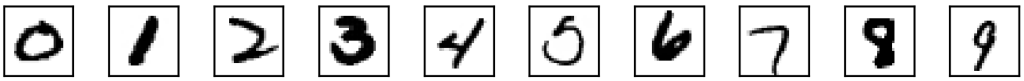
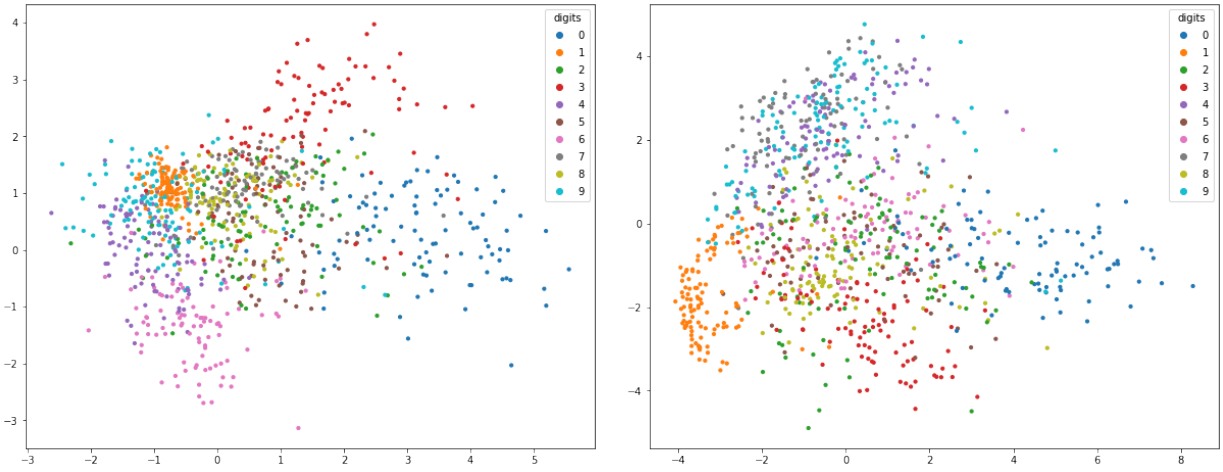


Figure 13: The prototype of each digit calculated by the GMLVQ algorithm on the MNIST dataset

Next we conduct another experiment to study the ability of the GMLVQ algorithm to facilitate a reduction in dimensionality. A standard method to achieve this, is by performing a principle component analysis (PCA). A PCA is done by calculating a covariance matrix on a given dataset. Next the eigenvectors and their respective eigenvalues of the covariance matrix are obtained and ordered in descending order according to their eigenvalues. For a reduction in dimensionality to a target dimension n we simply choose the n largest eigenvalues and construct a transformation matrix with their eigenvectors. Afterwards every data sample is transformed into the n-dimensional space spanned by the transformation matrix.
Note that the matrix Λ of learned relevances, that is part of the modified distance measure and which is updated along with the prototypes, can be interpreted as a covariance matrix. Therefore we perform the same routine as is done for the PCA on the matrix Λ

and similarly perform a reduction to n dimensions.
We apply this method to the MNIST dataset with our calculated matrix Λ with a target dimension of two and choose 1000 random data samples to transform. The results can be seen compared to that of a standard PCA in figure 14. Both images show similar results. Digits 0 and 3 have distinct clusters which suggests their images are easier to distinguish from the rest. Also in both visualizations the digit 1 has the most compact clustering, potentially indicating their training samples to be the most similar. Looking at the images themselves, this trend should be expected since a 1 is written by a single line and therefore has a low possibility for variations within the class. In general we can see the clusters being more tight when using the matrix Λ, as compared to the PCA. In total we conclude that using the relevance matrix Λ provides a similar result as a PCA when aiming to reduce the dimensionality of a given dataset and can be used as an alternative in exploratory data analysis. As for computational complexity, the regular covariance matrix can be computed in $O(M \cdot N^2)$ for M data samples of dimension N [11]. The complexity of one update step in the GMLVQ for one sample is $O(N_m \cdot N^2)$ with N_m being the number of prototypes [14]. For one epoch of M samples this comes to $O(M \cdot N_m \cdot N^2)$. The addiontal steps performed on the covariance matrix and Λ for the dimensionality reduction are identical. Therefore if we are dealing with a limited number of prototypes and epochs, like in this experiment, a PCA and our GMLVQ based approach have comparable computational complexity.



(a) tranformation into 2d-space using Λ (b) transformation into 2d-space using PCA

Figure 14: MNIST in 2d-space; GMLVQ vs. PCA

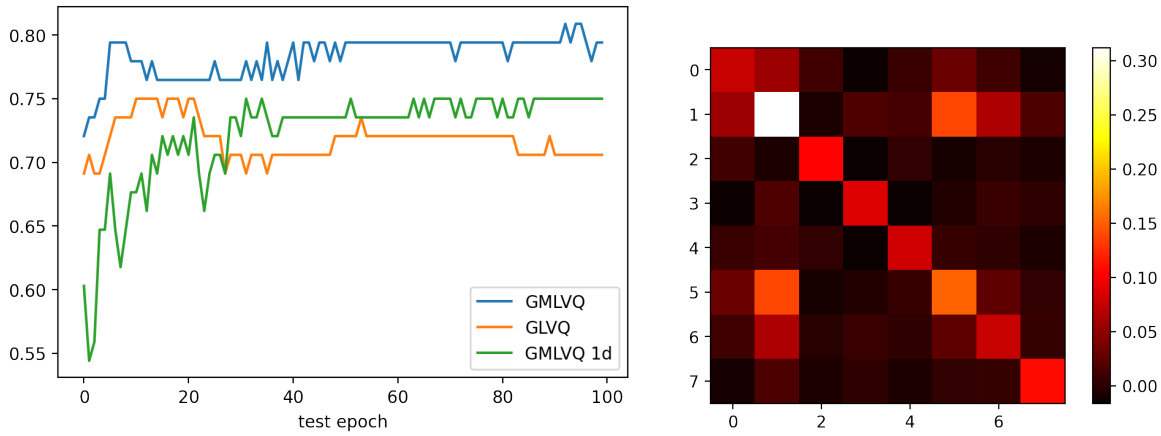
4.2.2 Diabetes prediction

In this experiment we choose a dataset consisting of medical data with the goal of predicting whether an individual suffers from diabetes [16]. The data is a selection from a larger database and only includes female patients over the age of 21 and of Prima Indian herritage. The dataset consists of 768 instances with 9 attributes each, including a binary class attribute on whether the patient did suffer from diabetes or not. The remaining 8 attributes are used for training and are as follows:

1. Number of times pregnant
2. Plasma glucose concentration after 2 hours in an oral glucose tolerance test

3. Diastolic blood pressure (mm Hg)
4. Triceps skin fold thickness (mm)
5. 2-Hours serum insulin (mu U/ml)
6. Body mass index (weight in kg/(height in m)^2)
7. Diabetes pedigree function
8. Age (years)

All attributes are normalized to values between 0 and 1. The training is conducted by using the first 700 data samples for 100 epochs, with their order being randomized each time. The remaining 68 instances are used during the testing after the completion of every epoch. We used five prototypes per class.
The results can be seen in figure 15a once again compared to the standard GLVQ algorithm with identical properties and hyperparameters. The accuracy rises quite quickly as the model learns and settles slightly under 0.8. Since we are dealing with a binary classification, meaning only two possible classes, the baseline is of course an accuracy of 0.5. We also see that similar to the classification of the MNIST dataset in 4.2.1 the accuracy of the standard GLVQ is lower than that of the GMLVQ.



(a) Accuracy of the model on the test data during the training process

(b) Heatmap of the Lambda matrix after the training

Figure 15: Diabetes prediction

In figure 15b the matrix Λ of learned relevance is visualized as a heatmap. The darker and more red an element $h_{i,j}$ is the *less* relevant the input dimensions i and j in correlation are to the classification. The lighter and more yellow they are the *more* relevant they are. We clearly see one input dimension with a very high weight. Feature two, the plasma glucose concentration.
The conclusion to draw from this pattern is that merely using the plasma glucose concentration one could reach similar results. To confirm this we test the same classification algorithm on the diabetes dataset using only feature two. The results are also plotted in figure 15a. It shows that reducing the number of input dimensions from eight to one only leads to a decrease in accuracy of around 5%. Even more it also managed to still beat the

GLVQ algorithm which was executed on the full-scale dataset. This clearly showcases the ability of the GMLVQ algorithm to find the most relevant input dimensions and the possibility to vastly reduce the complexity of the data while maintaining similar results. However judging the results from the accuracy alone is not sufficient in this case. Additionally we calculate the sensitivity and specificity. They are defined as:

$$\text{sensitivity} = \frac{|\text{true positives}|}{|\text{true positives}| + |\text{false negatives}|} \qquad \text{specificity} = \frac{|\text{true negatives}|}{|\text{true negatives}| + |\text{false positives}|}$$

In figure 16 we can see these measures over the course of the training process. What we observe is that the specificity is high nearing almost 1, while the sensitivity falls over the course of the training, settling around 0.5. This is very problematic for data of this kind. In a medical setting a false negative, failing to diagnose a present disease, can be dangerous and much more detrimental for the usability of a classifier than a false positive.

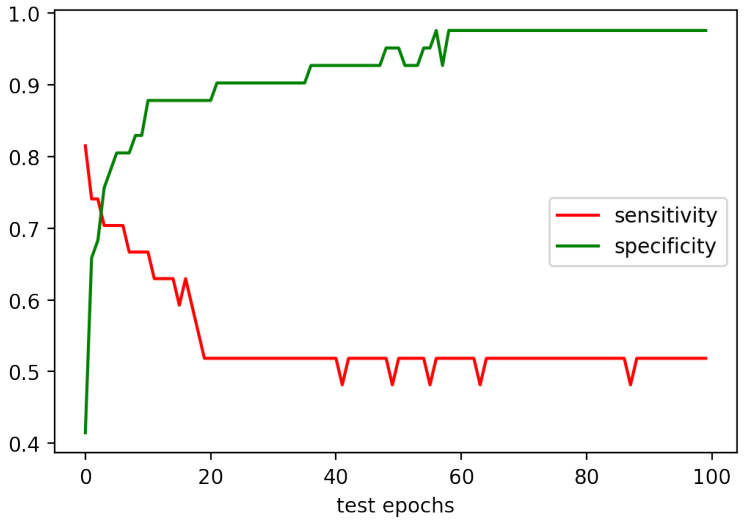


Figure 16: Sensitivity and specificity of the GMLVQ algorithm for diabetes prediction

		Actual diagnosis		
		Positive	Negative	Total
Prediction	Positive	14	1	15
	Negative	13	40	53
	Total	27	41	68

Table 8: Confusion matrix of GMLVQ on diabetes dataset

In table 8 we can see the full results of the final classification of the test samples. It shows that our algorithm in fact favors to predict a negative diagnosis. Consequently it identifies almost all negatives, but fails to correctly label half of all diabetes cases. One influencing factor for this could be the class imbalance is the dataset, which features 500 negatives and only 268 positives.

4.2.3 Conclusion

In this section we have investigated the properties of the generalized matrix learning vector quantization algorithm. Our experiments show that the transformation of the input dimensions via the weight matrix Λ in the distance measure d^{Λ} provides a better separation of the data and significantly increases the accuracy in both our datasets. It surpasses the performance of the regular generalized learning vector quantization in both cases. Furthermore we have proven the GMLVQs ability to find the most relevant dimensions and how important they are by performing a classification with nearly equal results with only one eighth of the dimensions as input. Finally we have shown that using the learned relevance matrix we can perform an operation for dimensionality reduction with a similar performance to that of a PCA. This indicates that the GMLVQ algorithm is a helpful tool in exploratory data analysis.

5 Self-organizing Maps

5.1 Introduction

Mallwitz

The **S**elf-**O**rganizing-**M**ap(**SOM**) was introduced by professor Kohonen in the 1980s. The algorithm is based on the same neural phenomena, that inspired the Hebbian Learning Theory. Although, this algotihm has a revered order. At the beginning of the algorithm are the "strong connections" already established and the prototypes position them self accordingly.

5.1.1 The SOM algotihm

The SOM is a connected net of nodes. In most cases this net is square shaped and two dimensional (one-dimensional and three-dimensional structures are possible as well). Each node has a representing prototypes m with a vector v_m. During the training phase of the SOM algorithm, the prototypes spread out in the sample space to represent the samples x as well as possible. The unique trait of the SOM algotihm is that the connections from the SOM, affect the training process of the prototypes. Every time a prototype m_i is pulled closer towards a sample x_i its topological neighbours, from the SOM, is pulled closer as well. Unlike the previous algorithms, the SOM does not incorporate labels in its training phases.
In the *initialization phase* phase of the algorithm the SOM structure is created (a simple matrix storing all v_m values) with a node count of K. The v_m vectors are internalized with random sample vectors. In addition a distance matrix will be initialized. The columns represent the samples and the rows represent the prototypes. The elements of the matrix will then be filled with the distances. The distance metric used varies from dataset to dataset (the euclidean distance was employed for the following experiments).
In order to train the prototypes, the following three phases will be repeated for several epochs. In the *sample selection phase* a sample x_i will be selected in a random fashion (in the following experiments an array containing the indexes was shuffled and then iterated on). Then the **B**est **M**atching **U**nit(**BMU**) vector m_{BMU}, the prototype with the smallest distance in the sample x_i column, will be selected.
The *shifting phase* will "pull" the m_{BMU} towards the sample $x_i$with the vector v_{x_i} as well as its topological neighbours from the SOM. The pulling force is based on a function p that considers topological distance defined by the SOM and the learning rate hyperparameter

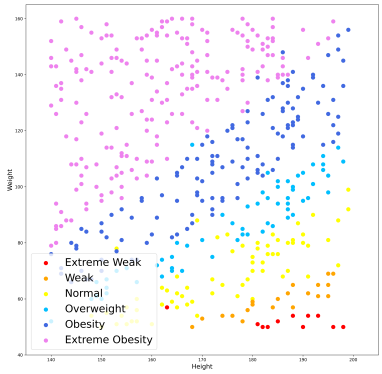
α that will decrease linearly towards zero during the duration of the training phase.

$$v_m = v_m + ((v_{x_i} - v_m) \cdot p \cdot \alpha)$$

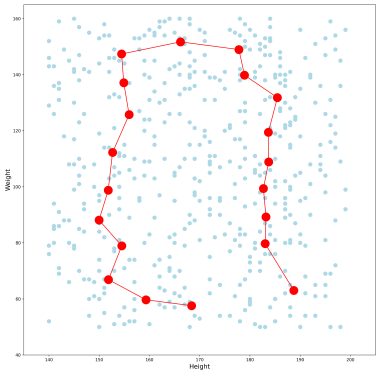
$\alpha = 0$ result in no changes to the prototypes vector, while $\alpha = 1$ replaces the v_m position with the sample vector v_{x_i}. The changes in the prototype positions are recorded in the *updating phase*. Only the the rows of the moved prototypes will be updated in the distance matrix. In the final *termination phase* the algorithms will either jump back to phase two or terminate. The termination criteria can be a set amount of iterations, e.g. after I iterations have been reached or the sum of the prototype distances moved in the past several iterations (convergence criteria) [9].

5.1.2 Example

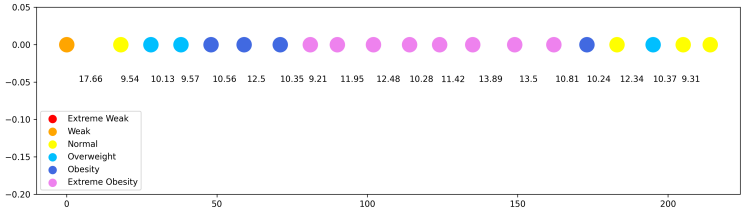
In the following, a small dataset visualizes the unfolding of the SOM in a higher dimensional sample space and showcases how one can interpret the SOM.
The "Obesity" dataset is made up of $M = 500$ uniformly distributed weight and height ($N = 2$) vectors. The five labels of the dataset "Extremely Weak", "Weak", "Normal", "Overweight", "Obesity" and "Extreme Obesity" are arranged, in the order they were named, from the upper left to the lower right corners of the sample space. The overrepresentation of the "Extreme Obesity" and "Obesity" labels as well as the underrepresentation of the "Extremely weak" label is already very apparent [4]. The goal for this example is to show how one, who does not have access to figure 17a, can discover the two traits (disparity in the representation and uniform distribution) of the dataset.
The SOM's hyperparameter, K will be set to twenty. The exact number of prototypes is arbitrary, but choosing too few would not represent all sample labels and choosing too many would lead to more redundancies (There are only 5 labels after all). The α is set to .7 to ensure a quick unfolding of the map, I was chosen to be $10 \cdot |x|$ to ensure that the prototypes have enough stimulation.
Figure 17b shows the SOM's prototypes in the sample space. The SOM unfolded into an open square to cover as many samples as possible. Also visible, the distances between the prototypes are roughly the same.
The finished SOM can be seen in Figure 17c (the closest sample to each prototype determines the label of the nodes). The distances between the nodes are roughly the same and one can see that only one is labeled "Weak", zero are labeled as "Extremely Weak" and the majority is labeled as "Extreme Obesity".
The SOM nodes reflect the two traits mentioned above. The samples are not positioned in distinct clusters, since the node distances are roughly the same and the labels go between "Obesity" and "Normal" two times. The "Extreme Obesity" label is overrepresented, because roughly half of the prototypes are violet. The opposite can be said for the "Weak" and "Extremely Weak" labels. The latter is not represented at all.
It is important to mention, that the underrepresentation of the "Extremely Weak" label is only partly caused by the underrepresentation. A flaw of the SOM algorithm is that it cannot represent edges well. The closer a prototype gets to an edge, the fewer samples pull it towards it and the more are pulling it back [9]. All in all, the obesity dataset example was successful at showcasing the unfolding of the map in the sample space and visualising key information about the dataset (disparity in the representation and uniform distribution).



(a) obesity distribution



(b) unfolded SOM in the sample space



(c) the SOM with distances

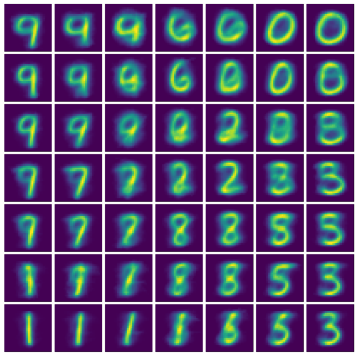
Figure 17: obesity dataset: $K = 20, I = 5000, \alpha = .7$

5.2 Experiments

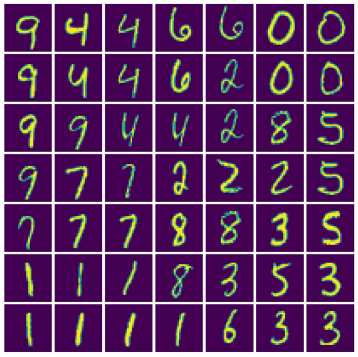
The SOM will now be applied to two different datasets the MNIST dataset and the German house price dataset. The two dataset highlight different aspects of the SOM. The section will then be concluded with an overall evaluation of the SOM algorithm.

5.2.1 MNIST dataset

The SOM is not a classifier, but it can support the development of classifiers by pointing out flaws(trends) in the dataset. In this experiment, the SOM will generate a new visual order for the MNIST dataset samples.
The MNIST (Modified National Institute of Standards and Technology) dataset is made up of $M = 60,000$ samples of the handwritten numbers $0 - 9$. The numbers were written by $6,000$ individuals and are stored in 28x28 pixel square images.
The value of each pixel, from the images, is stored in one dimension of the sample vectors. Therefore, the sample dimension of one image is $N = 784$ (28x28), equal to the resolution of the image. The map dimension is two to show more interesting relations between the numbers, while still being easy to visualize. The pulling force function p is changed from a linear one to the inverse function (the reasons will be discussed below)[3]. I is set to $|x| = 60,000$, because $60,000$ thousand epochs are more than enough to train $K = 49$ prototypes (all samples will be visited once).The α value was set to .7. The prototype amount K was chosen to be large enough to showcase all numbers and some transitional frames, without increasing the run time too much.
The goal of the SOM for this experiment is to create a visual hierarchy of the numbers and point out classification problems, e.g. very similar numbers that need extra attention.



(a) The SOM with the actual prototype values one can see transitional frames as well as distinct digits



(b) the closest sample to the prototype, thin and thick numbers are clustered together

Figure 18: MNIST dataset: $K = 49, I = 60.000, \alpha = .7$

The finished SOM, as seen in figure 18a, shows four very clear numbers in each corner that are visually very different from one another. Between those numbers are either other numbers or morphed shapes, half of one number, half of another. Three occurrences on the map are especially interesting. In the upper right corner are the "9" and "4" images, nearly indistinguishable. On the other side is a slight mix up between the numbers "3", "5" and "8". Those situation are less chaotic in the sample figure 18b, which shows the closest sample to each prototype. The last interesting occurrence, the number "6" appears twice on the map.

The SOM highlights two trends of the MNIST dataset. On the one hand are similar numbers, e.g. the "4" and "9", after closer inspection, one can see that only the top horizontal stroke differentiates the two digits. Another example are the digits "3", "5" and "8". One can see that they share three similar horizontal strokes and only differ in how these strokes are connected. The misplaced six, on the other hand, highlight a writing habit. The number "6", in the bottom case, is written as intended, with a big circle so it becomes similar to the "3", and in the top case it was written with a quick tiny circle so it becomes similar to the "0".

The algorithm showcased similar numbers as expected. But the "6" was a very interesting detail that should also be thought of when designing a digit classifier.

For this experiment, it was difficult to balance the two different scopes of the SOM. The global scope, a clear overall structure (clustering of the same numbers), and the local scope, clearly defined number shapes. This scope is affected by the α value, but increasing the alpha value above .7 is rather pointless because the prototypes will no longer "remember" any previous simulations and will only "remember" the most recent one until the α value falls below the .7 threshold. In contrast, a small stating value, would lead to trivial changes from the starting position, which is undesirable. After all, the starting position are entirely random. In conclusion, the α value impact on the SOM's unfolding is very significant, but it cannot be utilized to balance the local and global scope.

The function p mentioned in the *shifting phase*, which calculates how many neighbors are being pulled and how strong this pulling force is, is the most effective way to adjust these scopes. In the beginning, a linear force function was employed, that pulled the closest

neighbours stronger than the more distant neighbours. In addition, the function only affects a certain radius around the BMU.

$$p_{linear}(a) = \frac{a_{max} - a}{a_{max}}$$

a denotes the distance from the neighbor and a_{max} denotes the furthest neighbor for the current radius. Choosing a small radius, only pulling close neighbors, resulted in very defined numbers, but lacked global sorting. In contrast, a large radius resulted in very blurry shapes (that are trying to represent all the numbers at once). In the Kohens paper, an inverse function was proposed [9]:

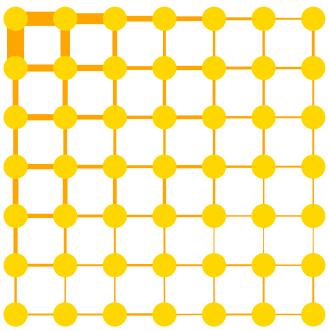
$$p_{inverse}(a) = \frac{1}{a + 1}$$

The one was added so that BMU is pulled with α as a factor and not infinity. The inverse force function pulls the BMU and its direct neighbours strongly to a sample creating more distinct shapes(local scope). Meanwhile, by always pulling all prototypes a bit too, an overall more unified order is created(global scope).
The MNIST dataset experiment was successful in revealing new insights into the MNIST dataset (different writing styles, e.g. "6"). It is also noteworthy how the SOM visualized 60.000 samples with 49 samples, so it became easy to grasp any trends and get an overall picture of the dataset.

5.2.2 German house price dataset

The German house price dataset provides a parametric descriptions of German houses. The objective for this experiment is to first use the SOM to find a general hierarchy in the parametric descriptions of German houses and find a correlation between the new hierarchy and the prices and geographic locations of the houses.
Until this point, the SOM was mainly employed to analyze correlations between the dimensions that were used for forming the SOM. This time, the SOM algorithm will be used to examine the relationship between the sample space dimensions and two other dimensions, that were omitted during the map formation phase.
The German house price dataset was taken from the German online real estate platform "Immo Scout 24". Only villa-style housing were taken into account to allow comparisons. In this experiment, $M = 1,590$ samples were analysed. The $N = 5$ sample Dimensions used to describe the villa's parameters are living space and lot size, as well as the numbers of rooms, bathrooms, floors, and garages[15].
For this experiment, a two-dimensional map with $K = 49$ prototypes was chosen, simply because of its intuitive visuals. The α value remains the same as in the experiment before. I was set to $10 \cdot M$, because the individual sample vectors in this experiment differed more than the digits in the previous experiment and thus should be visited multiple times. The p function will be the inverse function introduced above. The average price of all houses in one decision boundary will be reflected in the radius of the prototype. Finally, the place with the largest quantity of samples in a given prototype decision boundary will be chosen as the representative.
In general, house prices and the size of the villas correlate, so this should be reflected in the upcoming map. Regarding the representative geographic locations, a west-east divide is expected.
The SOM in figure 19a shows the average price distribution. The highest average house price is in the upper left corner and the lowest is in the lower right corner. The nodes in the

(a) SOM: average prices are reflected in the radius and locations are shown in text



(b) SOM: Map of prototype distances. Distances are expressed through line thickness (thicker represents a greater distance)

Figure 19: German house price dataset: $K = 49, I = 15,900, \alpha = .7$

middle of the SOM do not represent any houses. Overall, the circle sizes in the lower diagonal half of the map are the same size. In regards to geography, "Starnberg", "Bodensee" and "München" (upper left) are located in the south of Germany, while "Schaumburg", "Hamelin-Pyrmont" and "Hannover" are all in the same area (north of Germany). In general, the higher priced ones are closer to large cities and the lower priced ones are in more rural areas, except for "Hannover". In figure 19b one can see the prototype distances. They have the same distribution as the prices. The longest prototype distances are in the upper left corner(thickest lines) and the shortest are in the lower right half (thinnest lines).
The SOM reveals a clear correlation between the sample parameters and housing prices. In addition, there seems to be a minority of very expensive (and large) villas. This minority has very different parameters, since the prototype distances between the most expensive houses and their neighbours are the largest. The rest of the villas seem to be of similar sizes, because of their short prototype distances. There is also a correlation between geography and the demand for large houses. All prototypes that represent high-cost houses have representative locations that are either in large cities, e.g. Berlin and Munich, or next to large lakes (Lake Constance and Lake Starnberg). The just mentioned cities are known for their economic importance. That explains the high demand that leads to high prices. The lakes, especially Lake Starnberg, are well known expensive vacation locations. An additional geographic trend from the locations suggests that urban houses are in higher demand and generally bigger than rural ones. Cities have higher average incomes than rural areas, and thus can afford bigger and more expensive houses. Additionally, villas in the city of Hanover and its neighbouring districts seem to be in rather low demand.
Most of the correlations the SOM revealed were expected, but it is noteworthy that despite the expensive housing prices that exist in the capital and Munich, they still have the largest houses too. The original intent was to show the disparities between the western and eastern real estate markets, but these assumptions did not hold up with the actual results. Instead, the rural-urban divide is very dominated in this dataset. A north-south divide is also noticeable, but not as strongly. It is important to mention that the 1590 Villas did not represent some of the places well enough. Some had only one sample. This made

it impossible for them to be featured as prototype representatives on the map since those were chosen based on quantity.
All in all, the experiment successfully showed how the SOM can be applied to non-visual data and also revealed correlations between dimensions not used for the forming phases of the algorithm.

5.2.3 evaluation

The SOM algorithm is an effective tool to showcase a given dataset in a way that is intuitive, even for non-data scientists. This has been demonstrated in the experiments. In the obesity dataset example, the algorithm reflected both the over representation of the obesity label as well as the lack of a distinct segregation between the labels in a simple to understand way. The second experiment, the MINIST dataset, presented a very intuitive visual-order between the digits, that highlighted similar numbers (e.g. "3","8","5") and individual writing trends (writing "6" with small or large loop). In the last experiment, the correlation between house size and price, as well as location and price, were introduced in a simple map that underlined these trends.
The SOM does have the flaw, as one can see in experiment one, that it has difficulty with representing edges and under represented samples. But those are acceptable as long as the specialist that uses the SOM is aware of this. It is also noteworthy that the SOM is not deterministic, e.g. the MNIST SOM can have completely different numbers in each corner. One should run the algorithm several times before analysing the SOM.
In regards to the hyperparameters, I should be chosen fairly large to ensure a good result. The α value should be somewhere between $1-, 7$. Although for $alpha = 1$, the first thirty percent of the epochs are squandered because the prototypes ignore their previous locations. In regards to K, a small number will not adequately represent the dataset, while a large number will make reading the SOM inconvenient. Thus, a balance should be struck (mostly through trial and error).

6 growing neural gas

6.1 Introduction

Mallwitz

The **G**rowing **N**eural **G**as(GNG) algorithm is, like the SOM, a nature-inspired algorithm. It imitates the nature of gas, which spreads out to fill up a given shape, e.g., a room. The prototypes take over the role of gas "molecules" that are "chemically" linked only to their neighboring "molecules".
The GNG algorithm's prototypes are connect too, but unlike the SOM algorithm's prototypes, is the GNG algorithm able to add and delete prototypes as well as edges over time. This leads to a very adaptable prototype cloud that can grow (by adding prototypes) or split (by removing edges) into any number of differently sized shapes. The product of the algorithm is a mesh. The prototypes will be aligned in induced Delaunay triangles. Each prototype will have similar long edges, connecting it to its neighbours. Edges outside the sample structure will be removed over time.[5]

6.1.1 The GNG algorithm

The GNG algorithm can be roughly divided into nine distinct phases. Two new variables are introduced to determine how well the prototypes reflect the samples: the *prototype-*

error m_{error} and the *edge-age* e_{age}. The *prototype-error* evaluates how well the prototype represents the samples that are closest to it. While the *edge-age* is in charge of removing unnecessary edges e. An edge is unnecessary when it connects two prototypes, that are further apart than their other neighbors. This way Delaunay triangles are created and edges outside the sample shape are deleted. The GNG algorithm also adapts the the Hebbian Learning Theory. Old edges are weak connections, and edges with an e_{age} close to zero are strong and reflect that two prototypes are close to one another.

- *initialization phase*: The first two prototypes m will be created with $m_{error} = 0$. In addition, a connecting edge with $e_{age} = 0$ will be inserted too.
- *sample phase*: A sample x_i will be chosen at random, and the BMU as well as the **S**econd best **M**atching **U**nit(**SMU**) will be chosen based on the Euclidean Metric.
- *shifting phase*: The BMU and all its direct topological neighbors will be "pulled" towards the selected sample. This time is α a constant. Therefore, no pulling function is required. The neighbor prototypes will simply be pulled with a different smaller value β.

  $$v_m = v_m + (v_m - v_{x_i}) \cdot \alpha$$

  (v_m and v_{x_i} are the vectors of m and x_i)
- *evaluating phase*: The edge-age e_{age} for all edges connected to the BMU is increased by one, and the prototype error of the BMU is set to the square distance to the sample (m_{error} is not calculated based on all samples in the decision boundary of the BMU due to unnecessary run times. This is acceptable, because all samples will be eventually selected throughout the epochs).
- *binding phase*: If there is already an edge between the BMU and the SMU, the edge-age e_{age} will be set to zero. If none exist, one will be created (with $e_{age} = 0$).
- *aging phase*: All edges that have reached the threshold age_{max} with their e_{age} value, will now be removed.
- *removal phase*: Some prototypes have now become *edge-less* and will also be removed.
- *insertion phase*: Since the prototype with the *prototype error* is not representing its assigned samples well enough, a new prototype will be added between it and its neighbor with the highest prototype error. The old connecting edge will be replaced with two new ones connected to the new prototype. This phase will only be executed once every z epoch or will not be executed if there are already a certain number of prototypes, m_{cap}.
- *updating phase*: All *prototype-errors* will be reduced by multiplying them with γ. The algorithm will then either go back to the *sample phase* or terminate. An alternative to the normal set number of epochs criteria is the changing rate of the prototype number. After a certain amount of time, the algorithm will add as many new prototypes as it deletes, but this has the disadvantage that the prototype amount will become very large. The set number of epochs run is I [5].

6.1.2 Example

For this example, the image of one zero-digit image was extracted from the MNIST dataset. Every sample point is a pixel that had a non-zero color value[3].
In this example, $\alpha = .7$, $\beta = .4$, $z = 20$, $age_{max} = 5$, and $\gamma = .95$ were used due to the small number of sample points.The position of the prototypes will be plotted every 45 epochs until $I = 400$ epochs are reached (the prototypes did not change significantly after that threshold). The example will be considered successful if the GNG created mesh forms a zero and reflects the "hole" in the sample space.

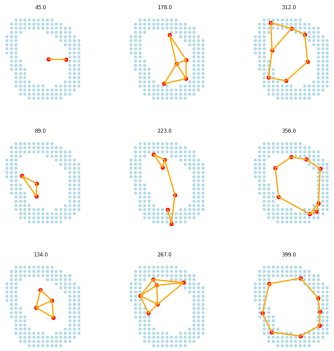


Figure 20: The prototypes form a zero ($I = 400, \alpha = .7$, $\beta = .4$, $z = 20$, $age_{max} = 5$ and $\gamma = .5$)

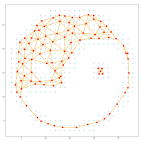
In figure 20 one can see how the prototypes start out in the middle and then slowly move outwards. One can also see how in the beginning there are several edges connecting all the prototypes, while in the end, each prototype is only connected by two edges. In the final position, the prototypes create a circle with roughly the same distances.
The prototypes start out in the middle, since in the beginning they will all be individually pulled from "all sides". After several prototypes have been added, they will focus on fewer samples and will be pulled apart. The edges will be reduced since they are no longer clustered and the distances between them will all be larger than the distances between their direct neighbours. This results in the two edges for each prototype.
All in all, this small example was a success. The sample space is well represented and the prototypes are all connected to an actual zero.

6.2 Experiments

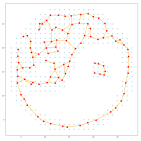
In the next section were two datasets analysed: The ying and yang dataset and the mammoth dataset. In conclusion will the GNG algorithm be evaluated.

6.2.1 ying and yang dataset

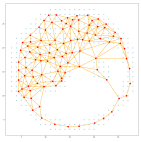
In this experiment, the GNG algorithm will be applied several times to the ying and yang dataset. One hyperparameter will be modified for every iteration. The goal is to learn about the hyperparameter's influence on the GNG algorithm.



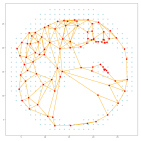
(a) no changes



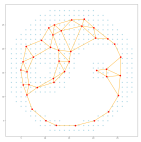
(b) $age_{max} = 5$



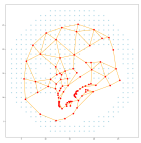
(c) $age_{max} = 150$



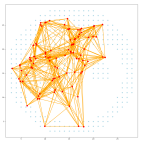
(d) $z = 5, m_{cap} = 200$



(e) $z = 500$



(f) $\alpha = .002$, $\beta = .001$



(g) $\alpha = .9$, $\beta = .5$

Figure 21: ying and yang dataset: $I = 15,000, \alpha = .2$, $\beta = .1$, $z = 100$, $age_{max} = 50$, $m_{cap} = 100$, $\gamma = .95$

The ying and yang dataset are $M = 387$ two-dimensional$(N = 2)$ vectors and was created by drawing the symbol and then converting the pixels into coordinate positions. The GNG algorithm will be initiated with the following hyperparameters: $\alpha = .2$, $\beta = .1$, $z = 100$, $age_{max} = 50$. For every experiment, one hyperparameter will be modified. The number of epochs will be limited to $I = 15,000$ and the number of prototypes will be capped at $m_{cap} = 100$ to make the comparison between the experiments more straightforward.

The experiment with the hyperparameters unchanged is depicted in figure 21a. The prototype structure represents the circle as well as the two dots. One can also see that the prototypes are in a triangle structure. The next figure 21b is a lot more disconnected, but the ying and yang shape is still recognizable. In contrast, the figure 21c is very connected. The small dots are no longer distinguishable. The next figure 21d is similarly connected, but unlike the one before, the prototypes are clustered and not evenly spread out. In this figure 21e are only a few prototypes, but the ying and yang are already there, although the dot is no longer separated. The second last figure 21f has a lot of edges, but one can not recognize any structure. The last figure 21g has a very unclear shape, but here one can see a few evenly spread prototypes and a big cluster of prototypes in the middle.

The figure 21a shows the ideal distribution of the prototypes. As one can see, the prototypes are evenly spread and they are connected in a Delaunay triangulation. The prototypes are similarly spread in the second example, but this time the maximum edge-age of age_{max} is so low that the prototypes have very few neighbors. A larger maximum edge-age would allow the prototypes to switch closest neighbors to reset the age in the *binding phase*. However, if age_{max} is only five, the edges will be removed far too quickly. The next example in figure 21c shows the opposite. It takes too long to remove the edges, so the prototypes are connected to too many neighbors. (The effect is not as drastic as expected,

because $I = 15.000$ and $m_{cap} = 100$ insure that most edges will still be removed after some iterations without new ones being added). The insertion rate of z was examined in figure 21d and 21e. It is first set to five. The algorithm adds the prototypes in the first 500 iterations, but it ends up being close to the ideal version. Therefore, the m_{cap} was increased. The result is now biased, to show how the algorithm struggles with moving too many prototypes. The takeaway is that if the algorithm has enough recovery time, it can undo the "damage" caused by adding too many prototypes in a short amount of time. In the second iteration seen in figure 21e each prototype has a longer time to settle into the sample space, but the I is too small to reach the prototype number of $m_{cap} = 100$. The result is still very good at representing the ying and yang, considering the low amount of prototypes.

The last two figures (figure 21f and figure 21g) show the effect of the modified α and β values. The first result is as expected. The first two prototypes start in the middle and all new prototypes were added there too, after all, they do not represent their samples well. The pulling force is so small that only a few prototypes are able to be pulled out (again a long recovery time would fix this problem). The last figure is, at first glance, a bit confusing. After all, the edge parameters were unchanged. The reason for the numerous connections is that all prototypes move so far, every epoch, that their edges are frequently reset (under normal circumstances, the prototypes would need to be pulled several times closer together before they become each other's closest neighbors).
In conclusion, the GNG algorithm is incredibly durable. Almost all problems that occurred can be fixed by giving the algorithm a long enough recovery time (not adding new prototypes). The only exception seems to be a too low maximum age_{max}.

6.2.2 mammoth dataset

The task for the last dataset is the accurate representation of the skeleton of a mammoth from the Smithsonian museum. The skeleton has a very large and complex structure and will require dynamic adjustments to the hyper parameters.
The dataset contains $M = 49,593$ three-dimensional ($N = 3$) sample vectors representing the mammoth [18]. The mammoth can be seen in figure 22a and figure 22b.
For the experiment, the insertion rate z was changed dynamically. Although the previous experiment results showed that the GNG algorithm can recover quite well from poorly chosen parameters, one should choose them appropriately to reduce the algorithm's runtime. The requirements for a mesh with 100 prototypes differ greatly from the one with $5,000$. For example, the number of epochs required until the prototypes of an edge in the latter example are visited enough times to reach the age_{max} differs greatly from the former. To give the prototypes enough time to adjust their positions, the insertion rate z is now dependent on the amount of prototypes m: $z = 2 \cdot |m|$. To consider this experiment a success, the GNG algorithm should generate a mesh that represents the skeleton of the mammoth well with distinct bones.
The $|m| = 6870$ prototypes shown in figure 22c and figure 22d show the mammoth skeleton. The shape of the bones, especially the rip cage, is well defined. The prototypes are connected with triangles, which wrap around the mass of the bones. After further inspection, some mistakes can be found, e.g. the large spine bones on top of the mammoth are connected, despite the tiny gabs visible in figure 22b.
First of all, the dynamical scaling of the insertion rate worked as intended. Since the Delaunay triangles were formed, the prototypes were not "overly" connected. There are also no clusters. In regards to the unintended gab-fills in figure 22d, for the prototypes and edges to not fill up the gabs between two structures, the distances between the proto-

(a) The mammoth seen from the front

(b) The mammoth seen from the side

(c) The GNG generated mammoth from the front

(d) The GNG generated mammoth from the side

Figure 22: Mammoth dataset: $I \geq 1,000,000, \alpha = .2, \beta = .1, z = 2 \cdot |m|, age_{max} = 40, \gamma = .9995$

types inside the structure must become significantly smaller than the gab. Only then will the edges that cross the gab be removed. Another element important for this experiment was the overall run time. The edges and prototypes were stored in a list. This made the algorithm extremely slow, especially with the large number of prototypes. One should consider implementing the algorithm with an other data structure, that speeds up the inserting and removing of the prototypes and edges.
The mammoth experiment showed the GNG applied to a three-dimensional input space and revealed that one should consider the number of prototypes when choosing the hyperparameters. In addition, the prototype amount is the single factor that dictates how detailed the representation of the sample structure is. The experiment also showed how the prototypes cloud can grow into larger sample spaces.

6.3 evaluation

As one could see in the experiments is the GNG algorithm an incredibly versatile algorithm. It is able to form the zero in the first example by becoming essential a line. In the ying and yang experiment are the prototypes able to represent the disconnected dot in the ying part. At last, in the mammoth experiment it was able to grow over 7000 new prototypes to fill up the skeleton of the mammoth properly. Another interesting benefit of the GNG algorithm are the surfaces. The original sample points of the mammoth had all edges removed and was only a point cloud, the GNG then added new surfaces.

References

[1] *Arrhythmia*. URL: https://archive-beta.ics.uci.edu/ml/datasets/Arrhythmia. (accessed: 07.09.2021).

[2] Tom Bäckström. *LVQ decision boundaries*. accessed: 02.09.2021. URL: wiki.aalto.fi/pages/viewpage.action?pageId=149883153.

[3] Dariel Dato-on. *MNIST in CSV*. URL: www.kaggle.com/oddrationale/mnist-in-csv.

[4] Yasin Ersever. *500 person gender height weight bodymassindex*. accessed: 27.08.2021. URL: www.kaggle.com/yersever/500-person-gender-height-weight-bodymassindex/code.

[5] Bernd Fritzke. "A Growing Neural Gas Network Learns Topologies". In: *Neural Information Processing Systems 7* (Mar. 1995).

[6] *GraKel*. URL: ysig.github.io/GraKeL/0.1a8/generated/grakel.WeisfeilerLehman.html#grakel.WeisfeilerLehman. (accessed: 07.09.2021).

[7] *Hebbian Learning*. URL: https://thedecisionlab.com/reference-guide/neuroscience/hebbian-learning/. (accessed: 07.09.2021).

[8] *Hepatitis*. URL: https://archive-beta.ics.uci.edu/ml/datasets/Hepatitis. (accessed: 07.09.2021).

[9] T Kohonen. "Self-organised formation of topologically correct feature map". In: *Biological Cybernetics* 43 (Jan. 1982), pp. 59 –69.

[10] Teuvo Kohonen. "Learning Vector Quantization". In: *Self-Organizing Maps*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 175–189. ISBN: 978-3-642-97610-0. DOI: 10.1007/978-3-642-97610-0_6. URL: https://doi.org/10.1007/978-3-642-97610-0_6.

[11] Vivek Kwatra and Mei Han. "Fast Covariance Computation and Dimensionality Reduction for Sub-window Features in Images". In: Sept. 2010, pp. 156–169. DOI: 10.1007/978-3-642-15552-9_12.

[12] A.K. Qin and P.N. Suganthan. "A novel kernel prototype-based learning algorithm". In: *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.* Vol. 4. 2004, 621–624 Vol.4. DOI: 10.1109/ICPR.2004.1333849.

[13] Atsushi Sato and Keiji Yamada. "Generalized Learning Vector Quantization". In: *Proceedings of the 8th International Conference on Neural Information Processing Systems*. NIPS'95. Denver, Colorado: MIT Press, 1995, 423–429.

[14] Petra Schneider, Michael Biehl, and Barbara Hammer. "Adaptive Relevance Matrices in Learning Vector Quantization". In: *Neural Computation* 21.12 (Dec. 2009), pp. 3532–3561. ISSN: 0899-7667. DOI: 10.1162/neco.2009.11-08-908. eprint: https://direct.mit.edu/neco/article-pdf/21/12/3532/830359/neco.2009.11-08-908.pdf. URL: https://doi.org/10.1162/neco.2009.11-08-908.

[15] Erdogan Seref. *german house prices*. accessed: 27.08.2021. URL: www.kaggle.com/scriptsultan/german-house-prices.

[16] Vincent Sigillito. *Diabetes*. accessed: 05.09.2021. URL: https://www.openml.org/d/37.

[17] *sklearn*. URL: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.datasets. (accessed: 07.09.2021).

[18] 3D Digitalization Smithsonian. *Mammuthus primigenius (Blumbach)*. accessed: 25.08.2021. URL: 3d.si.edu/object/3d/mammuthus-primigenius-blumbach:341c96cd-f967-4540-8ed1-d3fc56d31f12.

[19] *Spatial Proximity*. URL: https://www.sciencedirect.com/topics/computer-science/spatial-proximity. (accessed: 07.09.2021).

[20] *TU Dortmund*. URL: https://ls11-www.cs.tu-dortmund.de/staff/morris/graphkerneldatasets. (accessed: 07.09.2021).

[21] *UCI*. URL: https://archive-beta.ics.uci.edu/. (accessed: 07.09.2021).