Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für Wissens- und Sprachverarbeitung

Studienarbeit

Semi-Supervised Learning using Support Vector Machines and Growing Self-Organizing Maps

Verfasser:

Stephan Günther

August 26, 2009

Betreuer:

Prof. Dr. Rudolf Kruse, Georg Ruß

Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D–39016 Magdeburg
Germany

Günther, Stephan:
Semi-Supervised Learning using Support Vector Machines and Growing Self-Organizing Maps
Studienarbeit, Otto-von-Guericke-Universität Magdeburg, 2009.

Contents

List of Tables **iii**

List of Figures **iii**

1 Introduction **1**

1.1 Algorithm Idea 1

1.2 Motivation 3

1.3 Structure 3

2 Machine Learning **5**

2.1 Supervised Learning 5

2.2 Unsupervised Learning 7

2.3 Semi-Supervised Learning 8

3 Algorithms **9**

3.1 Growing Self-Organizing Maps 9

3.1.1 Metric Spaces and Quantization 10

3.1.2 The GSOM Specification 11

3.1.3 Implementation 15

3.2 Support Vector Machines 16

3.2.1 Inner Product Spaces and Kernel Methods 17

3.2.2 The concept behind SVMs 19

3.2.3 LibSVM: Some remarks about the implementation 25

4 Combining the GSOM and SVM Algorithms **27**

4.1 A detailed explanation of the combination approach 28
4.2 Potential Benefits and Issues 28
4.3 About the Implementation 29
4.4 Evaluating the algorithm 30

5 Conclusions and Future Work **33**

Bibliography **35**

List of Tables

4.1 The average accuracy results for each dataset and split size. 30

List of Figures

1.1 Combining GSOM and SVM. 2

3.1 Pseudo code for the GSOM algorithm 12
3.2 The GSOM map. 13
3.3 The `neighbourhood` $(node, size)$ function. 14
3.4 Multiple hyperplanes separating point sets. 19
3.5 The maximum margin hyperplane separating the point sets. 20
3.6 Two point sets which are not linearly separable. 23
3.7 Points from figure 3.6 made linear separable through the transformation ϕ. 24

Chapter 1

Introduction

> "Machine learning is a subfield of artificial intelligence (AI) concerned with algorithms that allow computers to *learn*." — Segaran [20]

What such a vague and general definition alludes to is the fact that machine learning is a vast field which is difficult to define concisely and encompasses many topics. Indeed, in its more than 50 year history[1] machine learning has used ideas and techniques from—and found applications in—fields such as statistics, optimization, neural networks, genetic algorithms, evolutionary programming and data mining. This work focuses on the aspects of machine learning closely related to data mining. These are concerned with the automatic processing of large amounts of data on the one hand and the generalization of known information about given data on the other. To be precise, the focus will be on a new algorithm which is a combination of two algorithms respectively taken from the fields of unsupervised and supervised learning. The algorithms in question are the *growing self-organizing maps* (GSOM) and the *support vector machines* (SVM) algorithms.

1.1 Algorithm Idea

What is meant by the terms *supervised* and *unsupervised* learning will be explained in more detail in the second chapter. Briefly, unsupervised learning refers to a setting in which there are a number of input points given and the only additional information is a certain metric or similarity measure.

In this work the GSOMis used to group the input points with respect to their degree of similarity. These groups are called *clusters*. The name unsupervised learning stems from the absence of additional information about each individual input point.

This setting is contrasted with the supervised learning setting, in which specific information is attached to each input point. This information is called a *label*. One

[1] If we begin with Rosenblatt [19].

example setting is a collection of gene sequences which are labeled with the micro-organisms to which they belong. A second example is a collection of attributes of certain flowers—color, blooming period, average height—given as inputs while the label for each item in the collection is the family the flower belongs to, e.g. the daisy family.

The task of the given supervised learning algorithm is to use the supplied data to find a way to predict the label that would be given to a new input. Since labels can be seen as classes for each input point, and assigning a label to a new input means classifying it, the task of an supervised learning algorithm is usually referred to as classification. Similarly, the task of the unsupervised learning algorithm used here is referred to as clustering, as it groups the data into clusters of certain similarity.

The algorithm presented in this paper will use an unsupervised learning algorithm to enhance the dataset used to train a supervised learning algorithm. Given a sparsely labeled set of inputs the idea is to use an unsupervised learning algorithm to cluster the given data. After clustering is done, the new algorithm will test which cluster contains certain labels and label all input points in the cluster according to those. Thus more labeled points than in the original input are generated which are then used to train the classifier of an supervised learning algorithm. The resulting classifier can then be used to classify those points which have not yet been labeled. These may be new ones or points from the input which did not belong to a cluster with a known labeled point in it. This idea is illustrated in figure 1.1 and a more thorough explanation of the actual implementation along with an outline of the possible benefits and issues is given in chapter 4.

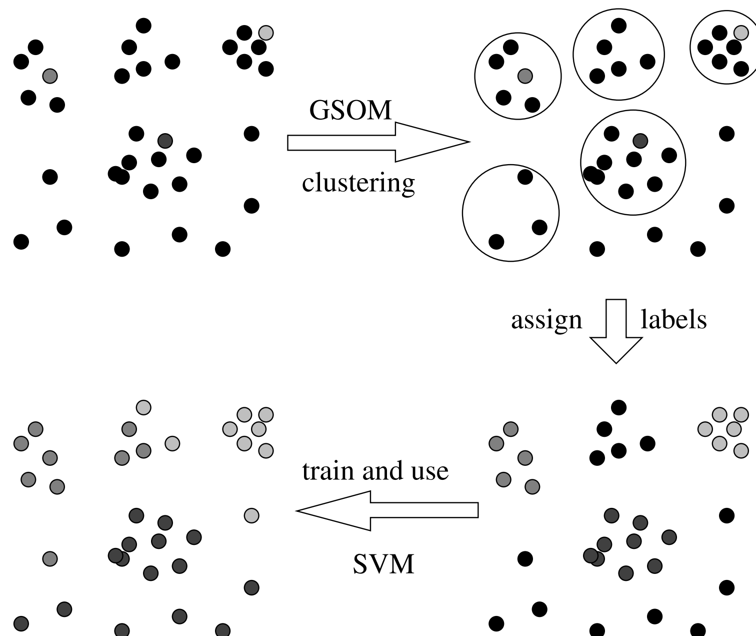


Figure 1.1: Combining GSOM and SVM.

1.2 Motivation

Using a combination of labeled and unlabeled data is referred to as *semi-supervised learning*. The field of semi-supervised learning is relatively young and only recently has an extensive overview of semi-supervised learning algorithms and results been published—Chapelle et al. [7]. The growing interest in this field is due to the exponential increase in data being retrieved, stored and processed. This growth has in turn strengthened the role of computers as an aid to experts for data processing and emphasised the need for computers to operate with increasing independence. Minimizing the need for human intervention in the act of data processing is one of the goals of the field of machine learning. The problem is that the performance of supervised learning algorithms depends on the amount of training data. Unfortunately, the labels are normally obtained by presenting unlabeled data to experts. As such, the labeling of unlabeled data has become a bottleneck.

This dilemma is the motivation for better utilising labeled data, and the algorithm proposed in this work does this by using unsupervised learning techniques to generalize label information.

1.3 Structure

The second chapter establishes some Machine Learning basic and a brief history. It also defines explicitly what is meant by the terms supervised and unsupervised learning. The chapter closes with a short overview over the field of semi-supervised learning, to which the algorithm developed in this work belongs.

The third chapter explains the two algorithms which are combined to yield the algorithm proposed in this paper. For each of the two algorithms a short history will be outlined and their main attributes will be presented. This is followed by an explanation of the basic mathematical concepts used in the respective algorithms after which the algorithms themselves will be explained. The section for each algorithm will close with remarks on the implementation used in this work.

The fourth chapter outlines how the algorithms explained in the third chapter have been combined. The motivation for the new algorithm will be revisited and the algorithm itself will be explained extensively. This will be followed by an explanation of the possibilities created along with the design choices made and the problems inherent in the algorithm design. The chapter will close with an evaluation of the algorithm's performance on a set of benchmark data taken from Chapelle et al. [7].

The fifth chapter offers some concluding remarks and suggestions for future work.

Chapter 2

Machine Learning

In this chapter some machine learning basics will be explained. A sketch of the history of machine learning will be presented and some of the many machine learning formulations will be mentioned. The bulk of this chapter is devoted to defining and explaining the machine learning settings which are of interest to this work and the terminology used. This is necessary to avoid ambiguities and misunderstandings since some of the formulations of machine learning can mean or be used to accomplish different things depending on the context.

The history of machine learning can be traced back to 1958 where Rosenblatt's construction of the *perceptron* in [19] marks a generally accepted starting point for the field. Combining such a long history—for a branch of computer science—with the broadness of its definition, it is unsurprising that over time many sub-fields of machine learning have attracted the interest of researchers. Examples include Unsupervised Learning, Supervised Learning, Inductive Learning, Symbolic Learning, Statistical Learning Theory, Clustering and Discovery, Case-Based Learning, Clustering, and Classification.

For further reading into these topics and the topic of machine learning the interested reader is referred to [18] and [22]. Some of those approaches are overlapping or have different meanings in different contexts. Since this work is confined to three sub-fields of machine learning, in order to avoid ambiguities the following sections precisely define the terminology used in this work.

2.1 Supervised Learning

Generally, *supervised learning* refers to a setting in which the input dataset is given along with additional information about each input. The supervised learning algorithm's task is to find a way of generalizing the given information to new inputs. More specifically, the setting which will be referred to as supervised learning for the remainder of this work is such that the input for the respective supervised learning algorithm is a set of points with each point having a *label* attached to it. The algorithm's output will be a *classifier*—a mapping from the input space to the 'label space'.

2.1 Definition. Let $X \subsetneq \mathbb{X}$ and $Y = \{y_1 \dots y_n | n \in \mathbb{N}\}$ a finite set with $|Y| \geq 1$. Let $f\colon X \to Y$. A **classification problem** is a triple $(X \subsetneq \mathbb{X}, Y, l)$ where:

- the set X is called the **input dataset** or **set of inputs**,
- the elements of X are called **input points**,
- the set $\mathbb{X}$ is called the **input space**,
- the set Y is called the set of **labels** or **classes** and
- the function l is called the **labeling** of the input points.

A **classifier** is an algorithm which takes a classification problem $(X \subsetneq \mathbb{X}, Y, l)$ as input and whose output is a **classification function** $l'\colon \mathbb{V} \to Y$.

A few things should be noted along with this definition. Firstly, for the remainder of this work, the term 'supervised learning' will exclusively refer to classification. This is emphasised here to differentiate it from other supervised learning formulations, the most frequent example being regression. For regression the set of given labels is just a subset $Y \subseteq \mathbb{Y}$ of the continuous set of possible labels. For example, the field of real numbers. In this case the algorithms task is to compute a continuous function $l'\colon \mathbb{X} \to \mathbb{Y}$. The restriction of supervised learning to classification for this work is a conscious choice made for reasons of convenience because other formulations are not important in the context of this work. Additionally, making the restriction explicit avoids ambiguities and confusion.

The next thing to note should be that in this work the following phrases will be used synonymously in reference to a point x:

- x has the label y,
- x belongs to class y and
- x is labeled y.

In all cases the phrase refers to the fact that $l(x) = y$. This notation is stated explicitly because it is contrasted by the fact that the phrase "x is classified as y" is used to denote that $l'(x) = y$.

Thirdly, it should be noted that a classification problem may also be be given as a tuple (X, l) that is, the specification may only consist of the set of inputs and the labeling. In this case the set of labels is given implicitly as the codomain $Y = \{l(x) | x \in X\}$ of f. Furthermore, the input space does not have to be an explicit parameter of the algorithm, because in most cases it is defined by an implicit assumption within the respective algorithm, and the context in which it runs.

The last thing to point out is the fact that the set of labels is restricted to have cardinality greater than one. While there are classification formulations with only one

label, these are mostly concerned with density estimation. As this formulation is not of interest here, it has been removed from what is covered in the definition above and is mentioned here only for the sake of completeness.

2.2 Unsupervised Learning

The setting in which one speaks of unsupervised learning is slightly more general than the setting for supervised learning. In the unsupervised case the algorithm is only given the input dataset and no explicit additional information. The algorithm's task then is to infer some desired knowledge from the structural properties of the inputs. While it is not necessary for explicit additional information to be attached to the inputs, there may be implicit assumptions being utilised by the algorithm. These assumptions may be in the form of a certain distance or similarity measure, or a certain way of measuring probabilities. They are generally the result of specific knowledge about the domain from which the input data is taken or they are taken from basic mathematical properties of the input data. For example, if the input dataset is given as a set of customer records, the unsupervised learning algorithm may implicitly employ specific marketing knowledge. In the field of bioinformatics, the inputs may be gene sequences and the algorithm may use distance measures specifically designed for these inputs. Alternatively, if the inputs are vectors from a Euclidean vector space the assumption may simply manifest itself in the usage of the Euclidean distance measure.

Owing to such a general formulation the actual task of an unsupervised learning algorithm may come in many forms, e.g. density estimation or clustering. For density estimation the algorithms task is to compute the probability distribution which generated the input dataset. As mentioned in section 2.1, some supervised learning algorithms may also be used for density estimation. This alludes to the fact that there can very well be a connection made between supervised and unsupervised learning.

The unsupervised learning formulation used in this work is called *clustering*. Here the algorithms task is to find a decomposition of the input dataset into disjoint subsets, called clusters. Most of the terminology, like the notion of input dataset and input points is shared with the the terminology used for classification. The only terms which have to be defined separately are *clustering*, *cluster* and *cluster center*.

2.2 Definition. Let X be any set. A **clustering** of X is defined as a subset $X_c \subseteq 2^X$ of the power set of X having the following properties:

(i) $\forall X_1, X_2 \in X_c \colon X_1 \cap X_2 = \emptyset$, and

(ii) $\bigcup_{\bar{X} \in X_c} \bar{X} = X$.

That is, a clustering is a disjoint decomposition of the input dataset. The elements of X_c are called **clusters**. A **clustering algorithm** is an algorithm which takes a set X as input and whose output is a clustering X_c of X.

The definition above does not put any restrictions on the set of input values and the final clustering has to be defined by explicitly specifying the set of clusters. In fact, this definition is still too general. To define a few more phrases needed during the course of this document, the input dataset has to be a subset of a certain input space.

2.3 Definition. Let $X, C \subsetneq \mathbb{X}$. Furthermore let $f\colon C \to 2^X$ be defined in such a way that $f(C) = \{f(c) \,|\, c \in C\}$ is a clustering of X. Then the elements of C are called the **cluster centers** of the clustering $f(C)$, which is called the clustering **induced** by C.

This establishes the terminology necessary for discussing supervised learning and the supervised learning algorithm used in this work.

2.3 Semi-Supervised Learning

As the name suggests, semi-supervised learning refers to a setting which may neither be clearly defined as supervised nor as unsupervised learning. It does not belong to either of them because the input dataset is partially labeled—there exists both labeled and unlabeled points. That is, the set of inputs X is decomposed into two sets X_l and X_u where $X_l \cap X_u = X$ and $X_l \cup X_u = \emptyset$. The set X_l is called the labeled subset of X while X_u is called the unlabeled subset of X. Furthermore, in the semi-supervised learning setting the labeling function f is only defined on the labeled subset of X, $f\colon X_l \to Y$.

The field of semi-supervised learning is a relatively recent development leading to the release of the first extensive overview on the subject with Chapelle et al. [7]. During this short period of time there have been multiple approaches to semi-supervised learning. However, most have focussed on how supervised learning algorithm algorithms can be made more accurate through the use of additional unlabeled inputs. But this approach can only be used if the number of labeled input points is not too small in comparison to the number of unlabeled input points. The algorithm used in this work combines an unsupervised learning algorithm with a supervised learning algorithm to handle such sparsely labeled data. The clustering computed by the unsupervised learning algorithm is used to artificially inflate the amount of labeled data available to train the supervised learning algorithm.

Chapter 3

Algorithms

This chapter will explain the GSOM and SVM algorithms in detail. Each of these algorithms will be treated in a separate section which will begin with a short overview of the algorithm's history, its main attributes and the settings in which it operates together with a sketch of the algorithm itself. This will be followed by a more thorough explanation of the algorithm including a definition of the mathematical concepts needed, the formulation of the algorithm itself and some notes about the respective algorithm implementations used in this work.

3.1 Growing Self-Organizing Maps

The GSOM algorithm is an extension of the self-organizing maps algorithm, which will be denoted with the abbreviation SOM for the remainder of this document. It was introduced by Kohonen in [16] and is extensively described in [17]. The GSOM algorithm was introduced by Alahakoon et al. in [1] and has been applied in the field of bioinformatics ([13], [5]) and for knowledge discovery in multi agent systems ([24]). Underlying both is the principle of vector quantization which will be formally defined later. Informally, both algorithms approximate an input dataset I by computing a set of points P from the same metric space as the elements of I. The goal is to have $|P| < |I|$, that is P should have lower cardinality than I. In this setting the points in P are then the cluster centers of the clustering induced by P. Both SOM and GSOM maintain the set P of cluster centers as a grid with a two dimensional layout, hence the term map in the name of both algorithms. In the original SOM algorithm the number of points in the grid and the dimensions have to be known *a priori*. This is one of the extensions which GSOM provides over SOM. For the former, the number of points in P does not have to be known in advance because GSOM starts with a fixed number of points and increases the size of P dynamically while it consumes the input. In their original formulations, the grid maintained by both algorithms was rectangular, while GSOM has later been extended to allow for hexagonal grids. The details of this will be outlined in 3.1.2 but they can also be found in Hsu [12], for example.

3.1.1 Metric Spaces and Quantization

As has been said earlier, both algorithms try to approximate the input space. For this purpose there has to be some way of measuring the quality of an approximation, which is usually done through some kind of distance measure. A set which is endowed with a distance measure is exactly what is defined by the notion of a metric space:

3.1 Definition (Metric and Metric Space)**.** Let X be a set and let $\mathbb{R}$ be the field of real numbers. A function $d\colon X \times X \to \mathbb{R}$ is called a **metric** on X if and only if it satisfies the following conditions for any $x_1, x_2, x_3 \in X$:

(i) $d(x_1, x_2) \geq 0 \wedge d(x_1, x_1) = 0$ (positivity)

(ii) $d(x_1, x_2) = d(x_2, x_1)$ (symmetry)

(iii) $d(x_1, x_3) \leq d(x_1, x_2) + d(x_2, x_3)$ (triangle inequality)

If d is a metric on X, the pair (X, d) is called a **metric space**.

The three axioms which a metric has to obey are the mathematical formulation of what one normally expects from a distance measure. For the SOM algorithm any metric on the input space is sufficient but for GSOM the situation differs slightly. The fact that the number of cluster centers does not have to be *a priori* specified leads to the restriction that the dimension of the input space is a parameter in the GSOM algorithm. This means that it can only handle finite dimensional input spaces. The reason for this restriction will be pointed out in 3.1.2.

Now that we have defined the notion of a metric, the notion of input quantization can be defined.

3.2 Definition (Vector Quantization)**.** Let $(\mathbb{X}, d)$ be a metric space and let $X \subseteq \mathbb{X}$. A **quantization** of X is a pair (X_c, f), such that $X_c \subsetneq X$ and:

$$f\colon X \to X_c, f(x) \mapsto \operatorname*{argmin}_{x_i \in X_c} (d(x, x_i)) .$$

The set X_c is then called the **set of code-book vectors** while the function f is called the **induced quantization function** of the quantization (X_c, f).

As one can see a quantization is a way of approximating the input space through a smaller set, because given an input vector v, the quantization function calculates the nearest code-book vector v_c to the input vector v. This concept is closely related to the concept of voronoi diagrams, because if a quantization (X_c, f) of the input dataset X is given, then for any code-book vector $x_c \in X_c$ the preimage $f^{-1}(x_c) = \{x \in X | f(x) = x_c\}$ is the voronoi cell of x_c. Further reading on voronoi diagrams may be found in de Berg et al. [10], for example.

As said earlier, the vector quantization is the underlying principle for both the SOM and the GSOM algorithm. This is because the set of nodes in the maps that both

algorithms compute act as the set of code-book vectors for a quantization of the input dataset. With this knowledge established, a specification of the GSOM algorithm can now be given.

3.1.2 The GSOM Specification

The GSOM algorithm in its original formulation from [1] has been improved several times and the specification given here most closely resembles the one given in [5]. The only difference is that the specification here is restricted to a hexagonal map topology. The pseudo code to the algorithm is given in figure 3.1 and what follows will be an in depth explanation of the different parts of the algorithm.

As can be seen from the input specification the algorithm has one more input apart from the input dataset and the metric to use, namely *phases*, which is a list of phase specifications. This is because the algorithm is divided into phases which are run sequentially. The phases to run and their parameters are thus specified by the user as a list of phase specifications. A phase specification is a quintuple of the parameters for one phase. These parameters are:

p: The number of passes the phase should make over the input. Each phase consumes the input and this parameter defines how the input set should be iterated over.

sf: The spread factor. It should be a number from the interval $(0, 1]$ and is used to influence the number of cluster centers. A higher spread factor will result in more and smaller clusters.

ns: The neighbourhood size. Each time a point of the input set is consumed, the weights of specific nodes in the map are changed. This parameter influences how many nodes have their weights changed on such an occasion.

lr: The learning rate is a measure for how strong the influence of a consumed point on the weights of the nodes in the map is.

g: The grow flag g determines whether new nodes should be grown during this phase or not. Phases where no new nodes are grown are needed to smooth out the approximation error after the map has grown sufficiently.

Thus a phase is specified with a quintuple (p, sf, ns, lr, g). The precise semantics of these parameters will be explained along with the part of the algorithm for which they are important.

While the algorithm runs it maintains a map of nodes. This map is essentially a planar graph with each node n of the map having two attributes assigned to it, the first being a weight w_n and the second being the node's accumulated error value e_n. The weight is an element from the same metric space as the input points, while the error value is a real number. The error of each node serves as the value which controls the

Input:

- Input dataset X having dimension D
- metric d: $X \times X \to \mathbb{R}$
- List of phase specifications *phases*

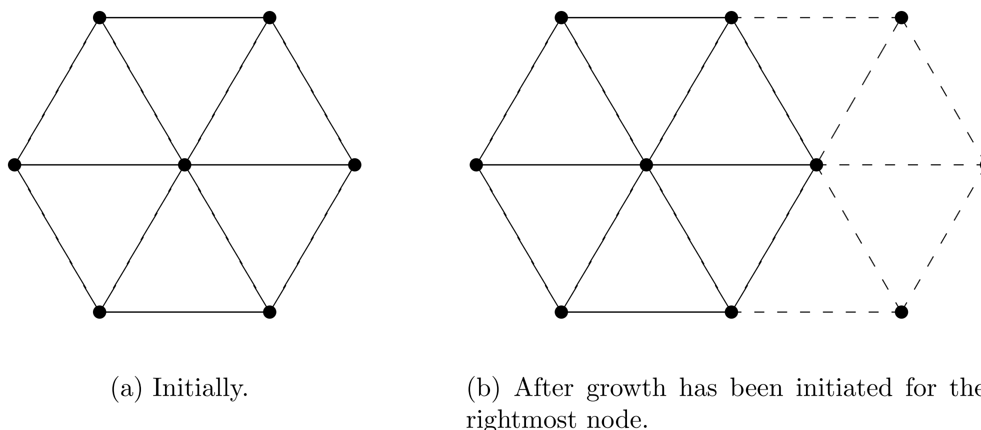
Output: A planar map of nodes with weights acting as a quantization of X

```
1  map ← initial map (figure 3.2a)
2  foreach (p, sf, ns, lr, g) in phases do
3      t ← 0
4      t_max ← p · |X|
5      for p_c ← 1 to p do
6          foreach x ∈ X do
7              t ← t + 1
8              n_min ← argmin_{n in map} (d(x, w_n))
9              d_min ← min_{n in map} (d(x, w_n))
10             e_{n_min} ← e_{n_min} + d_min
11             foreach node ∈ neighbourhood(n_min, ns · (1 − (t−1)/t_max)) do
12                 w_node ← w_node + lr · (1 − (t−1)/t_max) · (x − w_node)
13             end
14             if g and e_{n_min} > −D^{1/2} · log(sf) then
15                 neighbours ← neighbourhood(n_min, 1)
16                 neighbours ← neighbours \ n_min
17                 if |neighbours| < 6 then Grow new nodes. (figure 3.2b)
18                 e_{n_min} ← 1/2 e_{n_min}
19                 foreach n ∈ neighbours do e_n ← 1/6 e_{n_min}
20             end
21         end
22     end
23 end
24 return map
```

Figure 3.1: Pseudo code for the GSOM algorithm

maps growth while the weights constitute the cluster centers of the clustering which is the algorithm's output. Thus the algorithm's output is a planar map of cluster centers quantizing the input dataset.

In order to maintain the map it has to be initialized first, which is done on the first line of the algorithm. Since the GSOM implementation used in this work only supports a hexagonal map topology the map is initialized with seven nodes arranged as a hexagon as in figure 3.2a. Each node is assigned an initial error value of zero and the initial weights may either be random or fixed, based on the values of the input points.



(a) Initially.

(b) After growth has been initiated for the rightmost node.

Figure 3.2: The GSOM map.

After the map is initialized the algorithm commences to run the specified phases sequentially in the order in which they were given. The phase is divided into steps and each step handles one input point. How many times the input dataset is iterated over is specified in the phase parameter p which is an integer and has to be bigger than zero for the phase to do anything at all. The phase then iterates over the input dataset p-times so the number of steps a phase has is p times the number of points in the input dataset as seen in line 4. The algorithm also keeps track of the number of the current step because it is needed for some calculations, so the first thing which is done during each step is to increment the step counter by one.

The remaining actions of a step are devoted to handling one input point x. The first thing which has to be done is finding the so called best matching unit n_{min} which is defined as the node in the map whose weight has minimum distance to the current input point x. After the best matching unit is found its weight's distance to x is saved in d_{min} which is then used to update n_{min}'s accumulated error value $e_{n_{min}}$. Lines 8 to 10 are devoted to the above.

After n_{min} and d_{min} have been calculated the next actions are finding the nodes whose weights are to be altered and carry out the weight adjustment. Finding out which nodes are the ones whose weights have to be altered is done using the `neighbourhood` function

for which the pseudo code is given in figure 3.3.

Input:

- *node* which is a node belonging to a map.
- *size* which is a non negative real number.

Output: A set of nodes.

```
1  size ← round (size)
2  switch size do
3      case 0
4          return {node}
5      end
6      case 1
7          return {node} ∪ (Set of the immediate neighbours of node)
8      end
9      otherwise
10         return ⋃_{n ∈ neighbourhood(node,1)} neighbourhood (n, size − 1)
11     end
12 end
```

Figure 3.3: The `neighbourhood` $(node, size)$ function.

The function returns a set N of nodes where each element of N has a graph distance to *node* which is smaller than or equal to *size* rounded to the nearest integer.

With this knowledge established it can easily be seen that the neighbourhood size which is used in line 11 of figure 3.1 is decreased linearly during the phase. It starts out as the full size specified with ns and linearly decreases until it reaches zero at the end of the phase. The same happens with the learning rate lr which is used in the weight adjustment formula on line 12 of figure 3.1. All the nodes in the neighbourhood of n_{min} which is in effect for the current step of the phase are adjusted by this formula. From the formula it can be seen that if the factor $lr \cdot (1 - fract - 1t_{max})$ evaluates to 1, the weight which is subject to adjustment is made equal to the input point which is handled by the current step. On the other hand it is obvious that a learning rate of zero has no effect. For this reason the condition that $lr \in (0, 1)$ should be apparent. If this is the case, then the weight subject to adjustment will be shifted in the direction of the current input point by a fraction of depending on lr. The effect of lr decreases linearly with each step of the phase.

The next actions of the phase are only carried out if the grow flag g of the current phase is set to $True$. New nodes are only grown in the map if this flag is set. To decide whether new nodes should be grown or not the accumulated error value $e_{n_{min}}$ is checked and if it is greater than $-\sqrt{D} \cdot \log(sf)$ it is considered too large. Here D

is the dimension of the space from which the input dataset is taken, while sf is the spread factor parameter of the current phase. As said before one should have $sf \in (0, 1]$ because then $-\log(sf) \in [0, \infty)$ and a lower spread factor will result in higher error values being allowed before new nodes are grown while a high spread factor will make node growth be initiated earlier. Thus the higher the spread factor is, the more fine grained the clustering will be. If the accumulated error value of n_{min} is deemed too high the next actions depend on whether the n_{min} is an internal or a boundary node.

If a node for which the accumulated error value is deemed too high has less than six immediate neighbours the missing neighbours are created. Their error values are initialized to zero and the weight vectors are initialized as follows. For every created neighbour n_{new} the parent node already has a neighbour n_{old} in the opposite direction. This is a result of the hexagonal map topology and the fact that whenever node growth is initiated, the full set of neighbours for the node in question is grown. The weight of said opposite neighbour is used in weight initialization, which is done through the formula: $w_{nmin} \leftarrow 2 \cdot w_{n_{min}} - w_{n_{old}}$. Since a node can have a maximum of six immediate neighbours as shown in figure 3.2 node new nodes can be from a node which already has six neighbours. In this case the error value still has to be prevented from just growing further without having any effect. This is the reason that n_{min}'s error value is halved and half of the this value is distributed evenly among the nodes adjacent to n_{min}. This is done every time a node's error value is deemed too high, regardless of whether new nodes have been grown or not and makes high error values spread toward the map's boundary, in order to make node growth possible in later steps.

This is all that is needed to specify the GSOM algorithm and the next section will outline the specifics of the GSOM implementation which was used during the course of this work.

3.1.3 Implementation

Previously there have been two implementations of the GSOM algorithm. The first is a Java based one due to Hsu during his work on [12] and was used in [13] while the second one is *C#* based and was written by Chan for [4]. The former is a little dated and has not seen much development since its inception while the latter is tied to its graphical user interface and not portable due to the language it is implemented in. For those reasons the author has decided to re-implement the GSOM algorithm as a Haskell library. Haskell is an actively developed, lazy, functional language which has compilers available for a diversity of platforms. More info on the language including its specification an compilers for it can be found on the Haskell website[1].

One important fact about the implementation in hslibsvm is, that it currently only handles real valued input vectors and is restricted to the euclidean distance measure. The reason for this is connected to the reason for raising the dimension to the power of $\frac{1}{2}$ in the test on line 14 of figure 3.1. In the original proposition this power was omitted.

[1] `http://www.haskell.org`

It has been later shown that map growth should be made dependent on the distance metric in use. This dependence is controlled by raising D to the appropriate power and the square root is the one which is compatible with the Euclidean distance metric. Until now there have been found only ways of adjusting map growth according to *Minkowski metrics* of which the Euclidean one is a special case. As different metrics have not shown to be of significant impact in the algorithm's performance, the author has decided to restrict the implementation using the Euclidean distance measure until ways to adjust map growth to arbitrary metrics have been developed.

There were several reasons for choosing Haskell as an implementation language. The first was its portability. As Haskell compilers are available for various platforms, implementing the GSOM algorithm in Haskell makes it available on all these platforms. The second reason is the fact that Haskell provides a higher level of abstraction than *C* or *C++* while still being compiled to machine code. Thirdly, although even compiled Haskell code is usually not on par in terms of speed with equivalent *C* code, one is able to interface to *C* from Haskell. Thus performance critical bits of the GSOM algorithm can be rewritten in *C* if the Haskell implementation is deemed too slow. Finally, Haskell is one of the languages which pioneered software transactional memory and it is one of the few languages which has a working implementation of it.

Software transactional memory is a concurrency abstraction which was proposed in [21] and the implementation used within the Haskell language is described in [11]. It has a few advantages over a lock based handling of concurrency, most importantly composability, and the absence of the possibility for deadlocks. Using Haskell to implement the GSOM algorithm thus makes it easier to research the possibility of implementing the GSOM algorithm by using multiple threads thus taking advantage of modern multicore processors.

The Haskell library which was written in the course of this work is called *hsgsom* and is available from the Haskell package database[2]. The source code is freely available as a darcs[3] repository and may be browsed under `http://patch-tag.com/r/hsgsom`. It adheres to the specification given in 3.1.2.

3.2 Support Vector Machines

Support vector machines are a supervised learning technique invented by Boser et al. in [2] and a good overview on them can be found in Burges [3]. In their most basic form SVMs are a binary, linear classifier which means that the set of labels has cardinality two and that the classification function which is computed is characterized by an n-dimensional hyperplane, where n is the dimension of the input space $\mathbb{X}$. Classification is carried out by calculating on which side of the hyperplane a given input vector lies. The

[2] `hackage.haskell.org`

[3] A distributed version control system written in Haskell. More information under `http://darcs.net`.

first restriction of having only two labels has been addressed with different approaches and an extensive comparison of some of them can be found in Hsu and Lin [14]. The second restriction of only being able to calculate a classification function for linearly separable data can be addressed by using *kernel functions*.

Kernel functions are used to transform the elements of the input space into a higher dimensional space. In order to understand how kernel functions are used with SVMs, they will be defined in the next section.

3.2.1 Inner Product Spaces and Kernel Methods

A definition of kernel functions can not be done without mentioning another concept of mathematics, namely the notion of inner products, also known as dot products. Inner products are a generalization of the scalar products in Euclidean vector spaces.

3.3 Definition (Inner Product)**.** Let $\mathbb{V}$ be a vector space over the field $\mathbb{R}$ of real numbers. Let furthermore $\alpha \in \mathbb{R}$. Then an **Inner Product** is defined as a function $f\colon \mathbb{V} \times \mathbb{V} \to \mathbb{R}$ which satisfies the following conditions for any $v_1, v_2, v_3 \in \mathbb{V}$:

(i) $f(v_1, v_1) \geq 0 \land f(v_1, v_1) = 0 \iff v_1 = 0$

(ii) $f(v_1, v_2) = f(v_2, v_1)$

(iii) $f(\alpha \cdot v_1, v_2 + v_3) = \alpha \cdot (f(v_1, v_2) + f(v_1, v_3))$

A vector space which is endowed with an inner product is called an **inner product space**.

It should also be noted that any inner product space $(\mathbb{V}, f)$ can be made a metric space by defining the metric $d(v_1, v_2) = \sqrt{f(v_1 - v_2, v_1 - v_2)}$ on it. If no other metric is defined and the notion of a metric or distance is used in the context of an inner product space, the above definition is implicitly assumed for the remainder of this work. This definition is the generalization of the euclidean distance to arbitrary inner product spaces. Directly connected to the implicit metric on an inner product space is the notion of a vector's norm, which is a generalization of the concept the length of a vector. Given that a metric already provides a way of measuring distances in inner product spaces the length of a vector v can easily be defined as the distance to the origin $d(v, 0)$ and this is the definition which will be used throughout this work.

3.4 Definition (Vector Norm)**.** Let $(\mathbb{V}, f)$ be an inner product space and let $d\colon \mathbb{V} \times \mathbb{V} \to \mathbb{R}$ be a metric on it. The **norm** of a vector $v \in \mathbb{V}$ is defined as the distance to the origin $d(v, 0)$ and is written as $\|v\|_d$. If no metric is given the implicit metric on $\mathbb{V}$ is assumed.

Now that the definition of scalar products has been revisited, the notion of a hyperplane can be defined. Hyperplanes are important in the context of SVMs because the classification function a SVM computes is based on a hyperplane.

3.5 Definition (Hyperplane)**.** Let $\mathbb{V}$ be an inner product space of dimension d with the inner product f. A **hyperplane** $h(n,c) \subseteq \mathbb{V}$ is defined as:

$$h(n,c) = \{v \in \mathbb{V} \mid f(n,v) = c\}.$$

If the context allows it the hyperplane $h(n,c)$ may also be written in the shorter form h. The vector $n \in \mathbb{V}$ is called the **normal vector** of h while the parameter $c \in \mathbb{R}$ is called the **constant component** of h. Every hyperplane $h(n,c)$ partitions the space $\mathbb{V}$ into three disjoint subsets:

- The set $h(n,c)$ whose elements are called the points on or of h.
- The **positive side** of h which is also called the **positive half space** of $\mathbb{V}$ with respect to h:
$$h_+(n,c) = \{v \in \mathbb{V} \mid f(n,v) > c\}.$$
- The **negative side** of h which is also called the **positive half space** of $\mathbb{V}$ with respect to h:
$$h_-(n,c) = \{v \in \mathbb{V} \mid f(n,v) < c\}.$$

Every hyperplane defines a mapping $R_{h(n,c)} \colon \mathbb{V} \to \mathbb{R}, v \mapsto f(n,v) - c$. Once again $R_{h(n,c)}$ may be written in the shorter form R_h if the values of n and c can be derived from the context.

This leaves us with the only thing left to define being kernel functions. Informally, kernel functions are those functions which implicitly transform their input values from their domain into a vector space and calculate the result of an inner product between the transformed input values. The precise definition is the following:

3.6 Definition (Kernel Function)**.** Let $\mathbb{X}$ be a set and let $\mathbb{R}$ be the field of real numbers. A function $K \colon \mathbb{X} \times \mathbb{X} \to \mathbb{R}$ is a **Kernel Function** if and only if there exists a vector space $\mathbb{V}$ over $\mathbb{R}$, a function $\phi \colon \mathbb{X} \to \mathbb{V}$ and an inner product $f \colon \mathbb{V} \times \mathbb{V} \to \mathbb{R}$ such that:

$$\forall x_1, x_2 \in \mathbb{X} : K(x_1, x_2) = f(\phi(x_1), \phi(x_2))$$

Unless the target vector space and the corresponding inner product are known, the above definition is neither helpful for the construction of kernel functions nor does it give a useful criterion for testing whether a given function K actually is a kernel function. To solve those kinds problems the following theorem, taken from Cristianini and Shawe-Taylor [9], can be used:

3.7 Theorem. *Let $\mathbb{X}$ be a finite input space with $K(x,z)$ a symmetric function on $\mathbb{X}$. Then $K(x,z)$ is a kernel function if and only if the matrix*

$$\mathbf{K} = (K(x_i, x_j))_{i,j=1}^{n},$$

is positive semi-definite (has non-negative eigenvalues).

The above theorem is a special case of *Mercer's Theorem*, which is a far more general result. In [9] Cristianini and Shawe-Taylor give a good explanation of Mercer's Theorem and how the above result can be derived from it. With the necessary mathematical basics established the inner workings of SVMs can now be explained.

3.2.2 The concept behind SVMs

As has been pointed out earlier, SVMs in their original form have been designed to be a binary, linear classifier. The reason for this is that a SVM actually tries to find a hyperplane separating the input points according to their labels. That is, it tries to find a hyperplane h such that all the points labeled with one label are on one side of h while the input points labeled with the other label are on the other side of h. This formulation hints at the fact that the input dataset X should be taken from an inner product space, because hyperplanes are defined using inner products. But even if a hyperplane exists which separates the input data set in such a way it is usually not unique, a point which is illustrated in figure 3.4. This raises the question of how the separating hyperplane is chosen among the available possibilities.

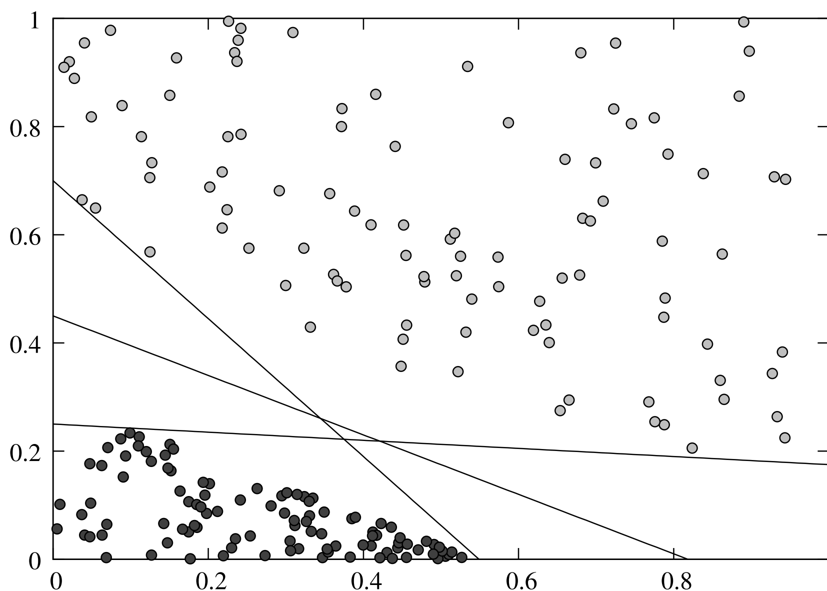


Figure 3.4: Multiple hyperplanes separating point sets.

The answer is that SVMs try to maximize the *geometric margin* between the points which have different labels. The hyperplane which achieves this is called the maximum margin hyperplane h_{max}. What is meant with the term geometric margin is illustrated in figure 3.5 where the same set of points as in figure 3.4 is used but only the maximum margin hyperplane separating the two classes is shown. The dashed lines indicate parallels of h_{max} which have been pushed towards each of the two classes until they touch the

first labeled point. The distance between those parallels is called the geometric margin or, for the remainder of this document, just the margin.

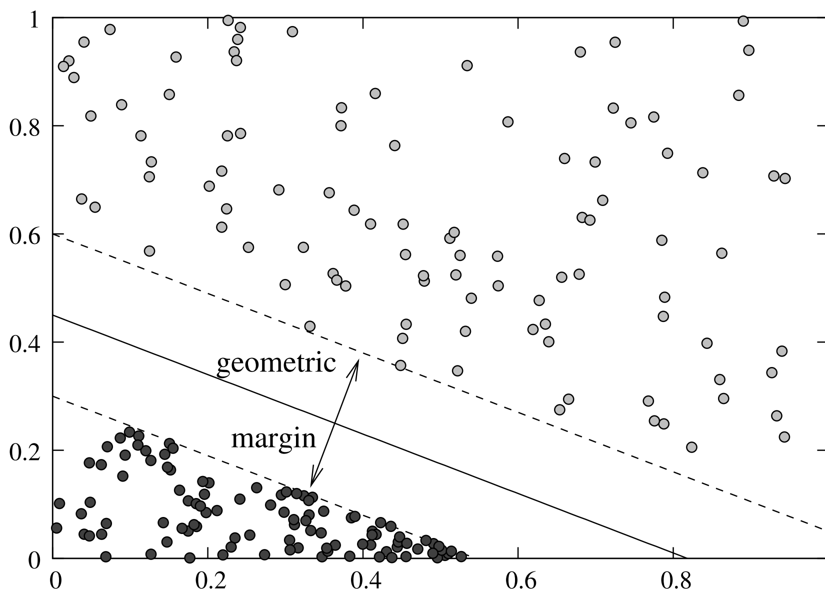


Figure 3.5: The maximum margin hyperplane separating the point sets.

Figure 3.5 also is a good way to explain how SVMs got their name. The hyperplanes which define the boundary of the margin are only dependent on the points they are touching. One can remove all the other input points and the margin would still stay the same. As these vectors thus *support* the hyperplanes on the margin's boundary, they are called the *support vectors*. The notion of support vectors will be formally defined later but the explanation above provides a good intuition.

Generally, for a given classification problem $(X, \{y_1, y_{-1}\}, l)$ and a given separating hyperplane $h(n, c)$, the margin is defined as the distance between the two hyperplanes $h_1(n, c_1)$ and $h_{-1}(n, c_{-1})$ where h_i is the hyperplane having minimum distance to the subset of X which is labeled y_i while still maintaining that:

$$\forall x \in X\colon \operatorname{sign}(R_h(v)) = \operatorname{sign}(R_{h_1}(v)) = \operatorname{sign}\left(R_{h_{-1}}(v)\right).$$

Here $\operatorname{sign}\colon \mathbb{R} \to \mathbb{R}$ is the usual sign function given by:

$$\operatorname{sign}(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}$$

and the distance between two sets is defined using the implicit metric d on $\mathbb{V}$:

$$\bar{d}\colon 2^{\mathbb{V}} \times 2^{\mathbb{V}} \to \mathbb{R}$$
$$\bar{d}(A, B) = \begin{cases} 0 & A \cap B \neq \emptyset \\ \min\limits_{a \in A, b \in B} (d(a, b)) & A \cap B = \emptyset \end{cases}$$

Though this definition narrows the choice of separating hyperplanes down it still does not make it unique because given the above definition parallel hyperplanes still have identical margins. This can be overcome by specifying that among the hyperplanes having the same margin, the one which has equal distance to the differently labeled point sets should be chosen. In other words, the separating hyperplane which is computed by a SVM is the one which maximizes the geometric margin and lies in the middle of said margin.

The next thing which should be explained is how SVMs actually find the maximum margin hyperplane. This is accomplished by formulating an optimization problem, where the separating hyperplane is the main parameter of the objective function and the value to be optimized is the geometric margin. As it turns out posing the search for the maximum margin separating hyperplane as an optimization task yields a quadratic program which can be solved using known quadratic programming algorithms.

The quadratic program is constructed as follows. The goal is to find a hyperplane $h(n, c)$ separating two points sets in such a way that the margin between the two sets is maximized. As one can see this amounts to finding the right parameters n and c because they define the hyperplane. Since these parameters then also define the hyperplanes on the boundary of the margin we can define these hyperplanes dependent on them as $h_{-1}(n, c-1)$ and $h_{+1}(n, c+1)$. If constructed in this way the two boundaries of the margin are guaranteed to be on different sides of h as they should be. The construction is also well defined because as one can deduce from definition 3.5 that hyperplanes are invariant under rescaling, that is for a given hyperplane $h(n, c)$ one has that:

$$\forall \lambda \in \mathbb{R}\colon h(\lambda \cdot n, \lambda \cdot c) = h(n, c)$$

This means that the definitions of h_{-1} and h_1 only amount to a normalization and do not change the optimization result. On the other hand this construction is highly convenient because it opens the possibility of finding a formula for the margin width w which is only dependent on the normal vector of the separating hyperplane as the w is just the distance between h_{-1} and h_1. This formula can be found in the following way: let $v_{-1} \in h_{-1}$ and let $v_1 \in h_1$ and let both v_i have minimal distance to each other. Geometry says that $v_1 - v_{-1}$ has to be perpendicular to both h_{-1} and h_1. But as both hyperplanes have the normal vector n it follows that $v_1 - v_{-1} = \lambda n$ for a $\lambda \in \mathbb{R}$. Two things can be deduced from this:

$$w = \|\lambda n\| = \lambda \|n\| \tag{3.1}$$

$$v_1 = v_{-1} + \lambda n \tag{3.2}$$

Now all which is left is to find λ. Definition 3.5 shows that $f(n, v_1) - c = 1$ which

gives:

$$\begin{array}{rc} & f(n, v_1) - c = 1 \\ \overset{\text{equn. 3.2}}{\Rightarrow} & f(n, v_{-1} + \lambda n) - c = 1 \\ \overset{\text{defn. 3.3}(iii)}{\Rightarrow} & \lambda f(n, n) + \underbrace{f(n, v_{-1}) - c}_{=-1} = 1 \\ \Rightarrow & \lambda f(n, n) - 1 = 1 \\ \Rightarrow & \lambda = \frac{2}{f(n,n)} \end{array}$$

Given the implicit metric on the inner product space it follows from equation 3.1 that

$$w = \lambda \sqrt{f(n, n)} = 2 \frac{\sqrt{f(n, n)}}{f(n, n)} = \frac{2}{\sqrt{f(n, n)}}.$$

Unfortunately, this is not very suitable as an objective function for an optimization problem since it contains a square root. But since all values in the function are positive neither squaring nor multiplication with positive factors alters the parameters at which the optimum is obtained which is why in practice instead of maximizing $\frac{2}{\sqrt{f(n,n)}}$ one rather chooses to minimize $\frac{1}{2} f(n, n)$.

With this in place the optimization problem which is solved by a SVM can be stated. Given a classification problem $(X \subseteq \mathbb{V}, \{-1, 1\}, l)$ with $(\mathbb{V}, f)$ being an inner product space, then a separating hyperplane with maximum geometric margin is found by solving the quadratic optimization problem:

$$\begin{array}{rl} \text{minimize:} & \frac{1}{2} f(n, n) \\ \text{s.t. } \forall x \in X\text{:} & l(x)(f(n, x) - c) \geq 1 \end{array}$$

The constraints ensure that each point is outside and on the correct side of the margin while the returned decision function is given by:

$$l'(v) = \text{sign}\left(R_{h(n,c)}(v)\right).$$

Now the notion of support vectors may be formally defined. After obtaining the results of solving the optimization problem above, the set of support vectors SV is defined as:

$$SV = \{x \in X \mid (f(n, x) - c) = 1\}$$

That is, it the set of vectors with active constraints. From optimization theory it is known that these vectors are sufficient to compute the optimization result. All other input points may be removed from the classification problem without affecting the resulting classification function.

This explains how the SVMs compute a decision function for a classification problem where the different classes can be separated by a hyperplane in the given space but what

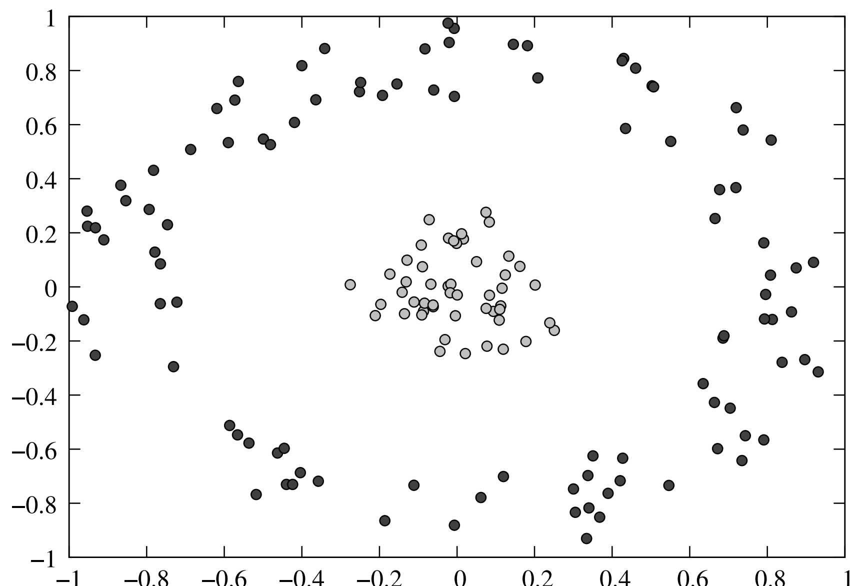


Figure 3.6: Two point sets which are not linearly separable.

happens if this is not the case as for example in figure 3.6? This is where the kernel functions are useful.

As by their definition (3.6), kernel functions allow us to compute the value of an inner product in a different space than the one from which the input points are taken. It can also clearly be seen that in the optimization problem which is solved by a SVM the only operations which are not among real numbers are inner products. Therefore, one may replace the application of inner products by applications of kernel functions. Therefore, for a given classification problem $(X \subseteq \mathbb{X}, \{-1, 1\}, l)$ there is no longer the need for an inner product on $\mathbb{X}$ but only a kernel function $K \colon \mathbb{X} \times \mathbb{X} \to \mathbb{R}$. The optimization problem to solve then simply becomes:

$$\begin{aligned} \text{minimize:} \quad & \tfrac{1}{2} K(n, n) \\ \text{s.t. } \forall x \in X: \quad & l(x)\,(K(n, x) - c) \geq 1 \end{aligned}$$

This simple substitution provides two improvements. Firstly, the only restriction on the input space is that there should exist a suitable kernel function for it. Secondly, the space in which the inner product corresponding to the kernel function is computed may actually be of higher dimension than the input space. This may allow data which is not linearly separable in the input space to be so in the kernel space.

Consider for example the function:

$$\begin{aligned} & K \colon \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R} \\ & K\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}\right) = x_1 y_1 + x_2 y_2 + \left(x_1^2 + x_2^2\right)\left(y_1^2 + y_2^2\right). \end{aligned}$$

This function is the result of mapping the euclidean plane $\mathbb{R}^2$ to the 3-dimensional euclidean space $\mathbb{R}^3$ via the function

$$\begin{aligned} &\phi\colon \mathbb{R}^2 \to \mathbb{R}^3 \\ &\phi\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ x_1^2 + x_2^2 \end{pmatrix} \end{aligned}$$

and then applying the canonical inner product

$$\begin{aligned} &f\colon \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R} \\ &f\left(\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}\right) = x_1 y_1 + x_2 y_2 + x_3 y_3. \end{aligned}$$

This makes K a kernel function by definition 3.6 and figure 3.7 shows the result of applying ϕ to the points from figure 3.6 and a hyperplane separating the two point sets.

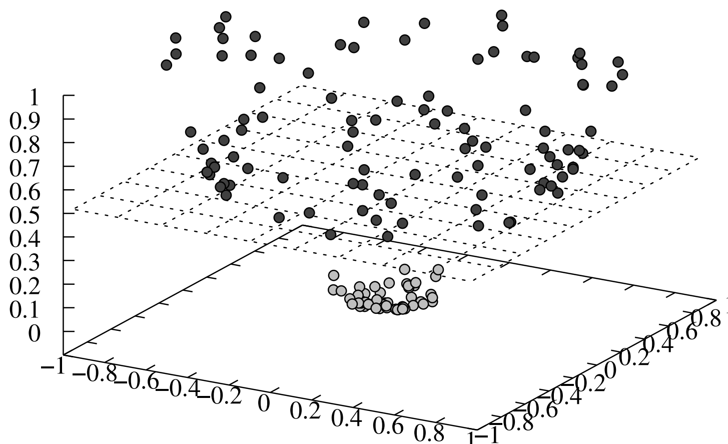


Figure 3.7: Points from figure 3.6 made linear separable through the transformation ϕ.

With this explanation some last notes about kernel functions are in order. It should be mentioned that not only the optimization problem can be altered by substituting a kernel function for every occurrence of an inner product but so can the computed classification function. As this function only consists of a hyperplane, which is essentially defined by the inner product of its normal vector with the test vector, this inner product may also be substituted with a kernel function. The next thing to point out is that kernel functions also have a drawback in that they obscure the metric that is actually used to find the maximum margin. Consequently, it can be hard to understand the results a SVM gives because one might not really know in which space the input dataset is actually separated. Nonetheless, SVMs have worked so well in practice that this concern has not had to much of an impact. The final note about kernel methods is that the

above mathematical explanation does not even convey their full usefulness. Since a lot of work has been done on quadratic programming and optimization theory in general the concept of duality may be applied to the optimization problem described above. Using duality it is possible to formulate an optimization problem which is equivalent to the one above but has a different structure. This is called the dual form. The dual form of the SVM optimization problem is special in that it only contains inner products of the input vectors and even the classification function is solely characterized by these. Therefore, the dual form lifts the necessity of fully specifying the kernel function and instead makes it sufficient to know the values of the kernel function for every pair of input points.

This finishes the outline kernel methods and leaves an explanation of how SVMs are generalized to handle classification problems with more than two labels. There are two main approaches, both consisting of multiple SVMs and choosing among them. The first approach trains a SVM for every label, where one label is taken as is and the remaining labels constitute the opposite class for this particular SVM. The second approach trains one SVM for every pair of labels and the classification function is a combination of the result. These approaches have been compared in [14].

The variant of SVMs which is explained here does not allow for training errors to occur, which may cause over fitting. This situation was remedied in [8] where Soft-Margin-SVMs where proposed which allow for training errors but allow the user to specify a parameter which defines how much a training error is penalized.

The interested reader is referred to [23], [3] and [9] for a thorough treatment of SVMs and kernel methods.

3.2.3 LibSVM: Some remarks about the implementation

For the purpose of this work the author has not written his own SVM implementation but chose to use an existing library. This library is called *LibSVM* and is freely available from its website[4]. It is distributed under a 3 clause BSD-Style license and its implementation details are available in [6]. The library supports a relatively large number of features of which only a subset has been used in this work. The ones which where used are explained in the following chapter along with the explanation of how they are used in the new algorithm.

As LibSVM is written in *C++* with a *C* interface and the software for this work is written in Haskell, a Haskell binding has also been written for the purpose of this work. The Haskell binding is released as *hslibsvm* and is available from the Haskell package database[5]. Its source code is available as a darcs[6] repository and can be browsed under `http://patch-tag.com/r/hslibsvm`.

[4] `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`

[5] `hackage.haskell.org`

[6] A distributed version control system written in Haskell. More information under `http://darcs.net`.

Chapter 4

Combining the GSOM and SVM Algorithms

This chapter provides more details of the approach taken to combine the GSOM algorithm with SVMs. The motivation will be re-established and the specifics of the algorithm combination implementation will be provided. This will be followed by a discussion of possibilities that arise by combining the two algorithms as proposed. The problems with the presented approach will also be considered. The actual implementation will then be presented and the chapter will close with the results of evaluating the algorithm on the benchmark datasets used in [7].

Since both the GSOM and the SVM algorithms have been explained, a more thorough explanation can be given of why a combination of both algorithms may be beneficial. In semi-supervised learning there is always the question of what the ratio of labeled to unlabeled data is. In [7] the primary consideration is the case where the number of unlabeled points is not much greater than the number of labeled points. It is even suggested that a setting in which there are vastly more unlabeled points—when compared to labeled ones—be called 'semi-UNsupervised' learning. The problem is that it is relatively easy to generate large input datasets, while acquiring labels for the points means somehow interpreting them. This is usually done by humans and it is a slow and sometimes expensive process. To alleviate this databases where labeled points are stored have been created so that researchers may have a pool of labeled data accessible with which they can work. But new research areas where old labeled data may not be applied are are constantly explored and data generating technologies are becoming faster so that traditional label generation strategies has a hard time keeping up. For this reason new ways to make use of sparse labeled datasets are essential to harnessing the vast amounts of data being produced.

SVMs alone are poorly suited to cases in which there are only few labeled points. Although the classification function found by a SVM only depends on the support vectors and not many of them are needed, if there are not enough labeled points the margin may actually be chosen too wide which results in *over fitting* and thus the accuracy with which

new points are classified drops. This is where combining GSOM and SVMs might be beneficial. The clustering computed by a GSOM helps in identifying a set S of unlabeled points which can be deemed sufficiently similar to a known labeled one in order to assign the same label to the points in S. With this new set of labeled points a SVM may be trained which may result in a better classification function.

4.1 A detailed explanation of the combination approach

The idea for combining GSOM and SVM into the new algorithm *GSOMSVM* is straightforward. Given a semi-supervised learning problem $(X_l \cup X_u, Y, l)$ the GSOMSVM consists of the following sequence of steps:

1. Run GSOM with $(X_l \cup X_u)$ as input storing the resulting clustering X_c.
2. Detect clusters x_c with $x_c \cap X_l \neq \emptyset$.
3. Assign the labels of the labeled points in each x_c to all the points in x_c thus creating a new decomposition $X_l' \cup X_u'$.
4. Use X_l' to train a SVM resulting in l'.
5. Use l' to label the points in X_u'.
6. Output labeled input dataset and/or l'.

As one might notice there is a problem in step 3. It stems from the fact that there may be multiple, differently labeled points in the cluster x_c. This case is called a *conflict* because the differing labels can be viewed as giving conflicting suggestions on how to label the points in the cluster.

As this case indicates, when the clustering is not of sufficiently small granularity, conflicts are handled by splitting the clusters in which the conflicts occur into multiple ones. This is done by treating the labeled points in x_c as new cluster centers and assigning each remaining point of x_c to the cluster center to which it has minimum distance. If there are still points left which are tied between two or more cluster centers they are discarded in order to not infer the wrong labels.

One can see that the heavy lifting really is done in the GSOM and SVM algorithms and understanding them is crucial in order to understand how they can work together.

4.2 Potential Benefits and Issues

The GSOMSVM algorithm depends heavily on one assumption, namely that the metric used conveys information about the distribution of labels. The algorithm can only

operate correctly if points which are near each other are also more likely to be labeled equally. More precisely, the assumption is that a high point density signifies a high probability for the occurrence of a single label. If that assumption is violated the trained SVM will very likely give wrong predictions. Therefore it is important to be sure that these assumptions are correct for a given model.

Fortunately, SVMs are in most cases very compatible with the Euclidean distance measure used in hsgsom. This stems from the fact that the canonical inner product used in an $\mathbb{R}$ valued vector space—which is the sum of the product of the components—defines the Euclidean distance measure as the implicit metric of the space. This can easily be seen from the remarks after definition 3.3. Even if one uses kernel functions this compatibility is not necessarily lost. As shown in [15], the most successfully applied family of kernel functions, which are radial basis functions, are compatible with linear SVMs.

4.3 About the Implementation

The GSOMSVM algorithm has been implemented as a command line application in Haskell. It is not yet part of the Haskell package database but its source code is available as a darcs repository and may be freely browsed under `http://patch-tag.com/r/gsomsvm`. The gsomsvm package depends on both, hsgsom and hslibsvm.

Apart from the input data and the GSOM phase specifications, the application may receive three additional parameters. One is the hsgsom soft-margin SVM. The parameter in question is called nu and it should be in the interval $(0, 1]$. As stated in [6], it is a lower bound on the fraction of support vectors and an upper bound on the fraction of training errors.

The next parameter is due to the fact that gsomsvm is for now restricted to radial basis functions as kernels. It is not difficult to add support for other kernels in the future but the decision to only support this family of kernel functions has been made because with correct parameter selection they can make standard inner products obsolete according to [15]. The same publication also proposes a successful heuristic for parameter selection for radial basis functions. The main benefit of using radial basis functions is that they compute an inner product in an infinite dimensional Hilbert space, which practically alleviates most restrictions on separability. The only drawback is that data which is hard to separate may only be classified by an over fitting classifier. The formula for radial basis functions is

$$K(x_1, x_2) = e^{-\gamma \|x_1 - x_2\|^2}$$

from which it follows that the parameter is γ. The parameter controls the generalization properties of the kernel. Higher values usually result in better separation but can lead to over fitting.

The last parameter is the result of a special feature of LibSVM. The library not only allows for a simple classification to be done but also can train a classifier having

probability information attached to it. That means the trained classifier is able to estimate the probability with which a point will be classified as having a certain label. This feature is used by gsomsvm to enable a threshold parameter t which is chosen from the interval $[0, 1]$. If it is supplied only points which are classified with a probability of at least t are included in the output.

4.4 Evaluating the algorithm

To get an understanding of how the proposed algorithm works in practice it has been evaluated on a set of benchmark data which is also used in [7]. The datasets are available from the book's website[1].

The datasets which where used for evaluation here are sets 1 to 7. The remaining sets 8 and 9 where of a format unsuitable for the evaluation of GSOMSVM. The format of the rest of the datasets is as follows. Except for dataset 4 every set consists of 1500 input points, with 214 dimensions. The 4th dataset instead consists of 400 input points 117 dimensions. Every dataset has two labels except dataset 6 which has 5 labels. For benchmarking purposes all the labels for the input points are provided but additionally so called *splits* are given.

These splits give the decomposition of the input dataset into labeled and unlabeled subsets in order for a semi-supervised learning problem to be correctly specified. The splits are given as the lists of indices which denote which points of the input dataset are to be treated as the labeled points. For every dataset 12 splits each containing ten and 12 splits each containing 100 labeled points are provided. According to Chapelle et al. there is no bias in the choice of labeled points except for the fact that each split contains every label at least once.

The evaluation has been carried out by first running the GSOMSVM algorithm on each semi-supervised learning problem defined by each given split. Afterwards the accuracy of GSOMSVM has been determined by using the classifier to classify the points of the respective input dataset and using the given labels to test whether the classification was correct. The accuracy is then calculated as the percentage of input points which have been correctly classified. The results of computing the average accuracy for all splits of one size in one dataset are given in table 4.1.

	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6	Set 7
small	78.44%	77.01%	59.87%	51.81%	62.18%	34.26%	68.14%
large	94.68%	90.59%	89.88%	66.63%	71.28%	77.33%	84.26%

Table 4.1: The average accuracy results for each dataset and split size.

It should be explicitly noted that for the calculation of accuracy the points which

[1] `http://www.kyb.tuebingen.mpg.de/ssl-book/`

where already given as labeled points have been included. There are two reasons for this choice. The reason behind this is that a soft margin SVM has been used and that the clustering and label inference step compute a set of labels which differs from the one originally given. As it is possible for a soft margin SVM to classify training data with a different label than what is originally given in the labeling and as the fact that additional labels are computed during by GSOMSVM this chance is only increased. This opens the possibility to actually lower the accuracy by including the original training data in the accuracy calculation. Therefore the original training data given through the splits is used included in the accuracy calculation.

The parameters which where used for the evaluation are given as follows. For the GSOM three phases have been specified. One growing and two smoothing phases, in that order. The two smoothing phases had no growing flag set and as such the spread factor was irrelevant for them. For the growing phase the spread factor was set to 0.1, which is a relatively conservative choice leading to nearly the coarsest possible clustering. This has been done in order to see whether this leads to labels being wrongly inferred, which would have a negative effect on performance. The remaining parameters for the three phases where $(5, 0.1, 3)$, $(50, 0.05, 2)$ and $(50, 0.01)$ in that order. Each tuple should be read as (*passes*, *learning rate*, *neighbourhood size*). The low number of passes, big neighbourhood and high learning rate for the first pass where chosen so that the map does not grow to strong but already establishes a good mapping over the input dataset. The high number of passes for the next two phases open the possibility for lowering the other to parameters while still ensuring a good distribution of clusters over the input data set.

The svm parameters nu and γ where set to 0.1 and $frac1D$ where D denotes the dimension of the input space respectively. The only exception is dataset 6 for nu was set to 0.001 because every higher value made the optimization problem infeasible. The rationale behind the choice of nu was to give to rigid constraints on the number of support vectors. The parameter γ was chosen this way because making it dependent on the number of dimensions is in accordance with the fact that a high number of dimensions raises the probability of the input data set being linearly separable. This stems from the fact that hyperplanes have better generalization properties the higher their dimension.

Chapter 5

Conclusions and Future Work

In this work it has been shown how to combine the GSOM and SVM algorithms in order to solve semi-supervised learning problems. Both algorithms and their foundations have been explained in order to show which assumptions have to hold and under which circumstances these algorithms can be applied together fruitfully.

The results that where obtained from benchmark datasets show mixed success. On the one hand there where datasets for which the algorithm performed well—yielding up to 94% accuracy with only 6.66% labeled data and up to 78% accuracy with only 0.07% labeled data. But there were also cases where the accuracy dropped to 34% for sparsely labeled data.

A result which is surprisingly bad, considering that the classification problem was a binary one. For comparison, a random classifier that would uniformly guess labels would achieve an expected accuracy of 50%, assuming the labels are uniformly distributed. Interestingly, the dataset on which the algorithm exhibited such bad performance had a structure which severely restricted the parameters available to the SVM. This is a good indicator that further research into how the structure of the input dataset influences the performance of GSOMSVM is necessary.

Generally, the results look promising and show room for improvement. Looking at how the GSOMSVM is influenced by its different parameters—how they work together, how properties of the input dataset can be used to find good parameters and estimate performance qualities—might very well give good insights into how to handle sparsely labeled data in the context of semi-supervised learning.

Bibliography

[1] Damminda Alahakoon, Saman K. Halgamuge, and Bala Srinivasan. Dynamic self-organizing maps with controlled growth for knowledge discovery. *IEEE Transactions on Neural Networks*, 11(3):601–614, May 2000.

[2] B. E. Boser, I. M. Guyon, and Vladimir Naumovich Vapnik. A training algorithm for optimal margin classifiers. *Proceedings of the fifth Annual ACM Workshop on Computational Learning Theory*, pages 144–152, 1992.

[3] Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Min. Knowl. Discov.*, 2(2):121–167, 1998.

[4] Chon-Kit Kenneth Chan. *Development of Clustering Algorithms for Grouping DNA Fragments in Environmental Genomics*. Doctor of philosophy, Department of Mechanical Engineering, The University of Melbourne, Victoria, Australia, Department of Mechanical Engineering, The University of Melbourne, Victoria 3010 Australia, May 2008.

[5] Chon-Kit Kenneth Chan, Arthur L. Hsu, Sen-Lin Tang, and Saman K. Halgamuge. Using growing self-organising maps to improve the binning process in environmental whole-genome shotgun sequencing. *Journal of Biomedicine and Biotechnology*, 2008:10, 2008. URL `http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2235928`.

[6] Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[7] O. Chapelle, B. Schölkopf, and A. Zien. *Semi-Supervised Learning*. MIT Press, Cambridge, MA, 2006. URL `http://www.kyb.tuebingen.mpg.de/ssl-book`.

[8] Corinna Cortes and Vladimir Naumovich Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, September 1995.

[9] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and other kernel-based methods*. Cambridge University Press, 2000.

[10] Mark de Berg, Marc van Krefeld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry - Algorithms and Applications*. Springer, second, revised edition, 2000.

[11] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. *PPoPP*, June 15-17 2005.

[12] Arthur L. Hsu. *Interactive Data Mining with Dynamic Self-Organising Maps*. Doctor of philosophy, Department of Mechanical and Manufacturing Engineering, The University of Melbourne, Victoria, Australia, Department of Mechanical Engineering, The University of Melbourne, Victoria 3010 Australia, December 2005.

[13] Arthur L. Hsu, Sen-Lin Tang, and Saman K. Halgamuge. An unsupervised hierarchical dynamic self-organizing approach to cancer class discovery and marker gene identification in microarray data. *Bioinformatics*, 19(16):2131–2140, 2003. doi: 10.1093/bioinformatics/btg296. URL `http://bioinformatics.oxfordjournals.org/cgi/content/abstract/19/16/2131`.

[14] Chih-Wei Hsu and Chih-Jen Lin. A comparison of methods for multiclass support vector machines. *IEEE Transactions on Neural Networks*, 13(2):415–425, March 2002.

[15] S. Sathiya Keerthi and Chih-Jen Lin. Asymptotic behaviors of support vector machines with gaussian kernel. *Neural Computation*, 15(7):1667–1689, July . doi: 10.1162/089976603321891855. URL `http://www.mitpressjournals.org/doi/abs/10.1162/089976603321891855`.

[16] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78:1464–1480, 1990.

[17] Teuvo Kohonen. *Self-Organizing Maps*, volume 30 of *Springer series in information sciences*. Springer, third edition, 2001.

[18] Tom Michael Mitchell. *Machine Learning*. McGraw-Hill Series in Computer Science. McGraw-Hill, 1st edition, 1997.

[19] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, November 1958.

[20] Toby Segaran. *Programming Collective Intelligence*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, first edition, August 2007.

[21] N Shavit and D. Touitou. Software transactional memory. *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, 1995.

[22] Jude W. Shavlik and Thomas G. Dietterich. *Readings in Machine Learning*. The Morgan Kaufmann Series in Machine Learning. Morgan Kaufmann Publishers, Inc., 2929 Campus Drive, Suite 260, San Mateo, California 94403, 2nd edition, 1991.

[23] Vladimir Naumovich Vapnik. *The Nature of Statistical Learning Theory*. Springer, Springer Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010, USA, 1995.

[24] L.K. Wickramasinghe and L.D. Alahakoon. Dynamic self organizing maps for discovery and sharing of knowledge in multi agent systems. *Web Intelligence and Agent Systems: An international journal*, 3:31–47, 2005.

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 26 August 2009:

Stephan Günther