Universidad
Rey Juan Carlos

Escuela Técnica Superior de Ingeniería de Telecomunicación

Ingeniería de Telecomunicación - Ingeniería Informática
Especialidad en Informática de Sistemas

PROYECTO FIN DE CARRERA

Structured Prediction of Network Data

Autor: Fernando José Iglesias García

Tutor: Antonio García Marqués

Curso académico 2012-2013

PROYECTO FIN DE CARRERA

Structured Prediction of Network Data

Autor: Fernando José Iglesias García

Tutor: Antonio García Marqués

La defensa del presente Proyecto Fin de Carrera se realizó el día de de 2013, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

Habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 2013.

"There are two problems in modern science:
— too many people use different terminology to solve the same problems;
— even more people use the same terminology to address completely different issues."

Anonymous

RESUMEN

El aprendizaje estructurado es una parte del aprendizaje máquina encargada de la *estimación de parámetros* y la *predicción* de modelos de múltiples variables aleatorias, en donde el estado de dichas variables se trata de forma conjunta. Durante la última década estos modelos han adquirido mucha importancia debido a su aplicación en distintos campos como la visión artificial, el procesamiento de lenguajes naturales y la bioinformática.

El interés principal de este Proyecto de Fin de Carrera es el estudio de *modelos estructurados* generales. Las interrelaciones de los datos asociados con estos modelos se representan mediante los arcos y los nodos de un grafo. El aprendizaje y la predicción estructurada en este tipo de modelos es *difícil* dado que no hay algoritmos eficientes que puedan resolver el problema de inferencia de forma general. Hay que, o bien conformarse con la utilización de algoritmos aproximados, o bien restringir la complejidad de las dependencias cuando hay más de unas pocas variables involucradas.

Este Proyecto de Fin de Carrera presenta en primer lugar una infraestructura conceptual para el aprendizaje estructurado basado en la máquina de vectores soporte. Posteriormente, se presentan distintas alternativas para la estimación de parámetros de la máquina de vectores soporte para el aprendizaje de modelos estructurados. Así mismo, se hacen aplicaciones particulares de esta infraestructura de aprendizaje estructurado a problemas donde la inferencia se puede resolver eficientemente. Por último, se presentan experimentos numéricos cuyos resultados sirven de validación inicial de la metodología analizada y revelan el potencial de la misma.

ABSTRACT

Structured output learning is a field within machine learning occupied with *parameter estimation* and *prediction* of models with multiple random variables, where the state of these variables is treated jointly. During the last decade these models have become rather popular due to their successful application in different domains including computer vision, natural language processing and bioinformatics.

The main interest in this project is in general structured output models. The dependencies of the data associated with these models is represented by the nodes and edges of a graph. Structured learning and prediction in this type of models is *hard* since there is no general efficient algorithm to make inference. One must either rely on approximate algorithms, or restrict the complexity of the model dependencies when more than only a few variables are involved.

A general structured output framework based on the well-known Support Vector Machine method from supervised learning is presented. Distinct alternatives for parameter estimation in this setting are discussed. In addition, two particularizations of the framework for which efficient inference is possible are described. Finally, proof-of-concept experiments demonstrate the suitability and merits of our approach.

ACKNOWLEDGMENTS

First and foremost I must thank Antonio, my project supervisor, for all the support and freedom he has given me to carry out the work presented in this report. He has not only being there during the course of the project though. Antonio has greatly helped me during my studies since I took his course in linear systems, back then when I was still young in my second year in the university. He encouraged me many times to continue my studies abroad, as I ended up doing eventually, and I will never be able to thank him enough for all the good advices he has given me in this matter. He is a reference, both as a researcher and as a person.

This work would not have been possible either without the help and wise guidance from the rest of my professors. To all of you, thank you so much!

Thanks to all the good moments I have shared with my college friends, both in Fuenlabrada and Stockholm. Without them, it would not have been possible to keep up with the hard work during all these years. I do not want to risk writing their names here, one by one, and accidentally forgetting any of them, so I hope they all know they have been, and still are, an important part of my life.

It is thank to the Shogun crew that I got to know about structured learning. I have been sort of hooked on this field for the last year and I am very grateful for all the help I have received from them during this time. It is simply great to be part of your team and develop kick-ass machine learning software with you!

Finally, I have to thank my family of course. Especially to my mother and my uncle. Incredibly, they have been able to bear me every day during many years, and that must have been bloody tough!

To Lina, the one who is bearing me now, tack så mycket for these wonderful last years!

We also acknowledge the financial support from *Ministerio de Educación, Cultura y Deporte*, Rey Juan Carlos University and the Department of Signal Theory and Communications for the scholarship (ID 2097317) received to develop this project.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

1 INTRODUCTION

This project report is organized as follows. First, chapter 1 motivates the context of the project and identifies its main goals. The theoretical background upon which the method used in this project is based is briefly reviewed in chapter 2. In chapter 3 the method itself is described. The experiments and results are discussed in chapter 4. Also, chapter 4 includes descriptions regarding implementation details in order to make the gap between theory and practice as small as possible. Finally, the conclusions and some extensions are left for chapter 5.

The present chapter starts with a short introduction to machine learning in general, followed by a discussion about structured learning in particular. A few representative tasks where methods from these fields have been applied are presented along with the discussion. Then, the project goals are clearly established.

1.1 MACHINE LEARNING

More than half a century ago the pioneer in artificial intelligence and author of the first self-learning program able to play checkers, Arthur Lee Samuel, already defined machine learning [1] as the "field of study that gives computers the ability to learn without being explicitly programmed" (Simon 2013). During the last few decades, machine learning has become a very active and fruitful area of investigation with many researchers involved throughout the world. At the same time, many fields of application such as computer vision, bioinformatics, robotics, or information retrieval, to name a few, have been pushed forward thanks to the use of machine learning.

Broadly speaking, machine learning deals with the design and implementation of software systems capable of learning from data. In particular, a desired property of the agents studied in machine learning is the ability to *generalize*. In this context, generalization has to

1 An introduction to Machine Learning and Pattern recognition can be found in (Bishop 2006). Another introduction to the topic focused on its statistical nature is discussed in (Hastie, Tibshirani, and J. Friedman 2009).

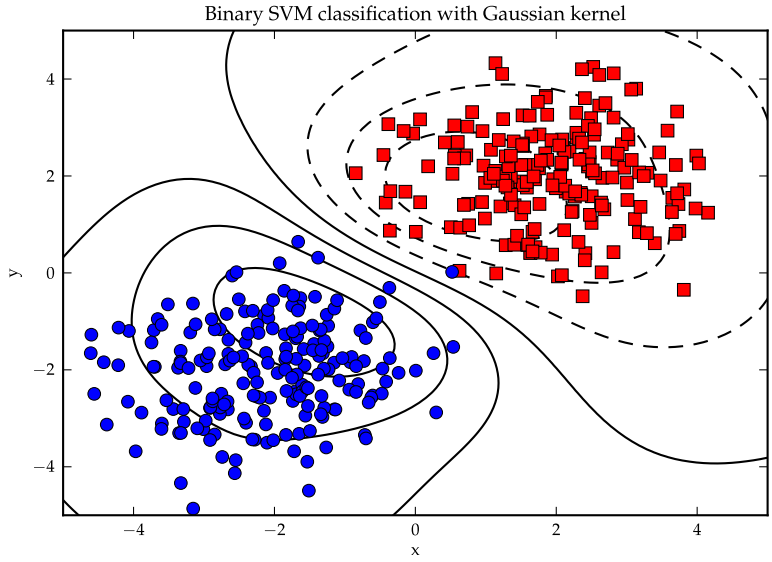


Figure 1.1.: Support Vector Machine with Gaussian kernel applied to a binary classification problem. The use of the kernel function allows the classfier to learn a non-linear boundary. The lines show regions of the space with equal score given by the classifier. Dotted lines and solid lines represent scores with opposite sign.

do with the system being able to extract patterns from data during its learning phase and achieving satisfactory results in new unseen data, using the knowledge acquired. This generalization depends critically on the *representation* of the data and the mathematical models used to assess the performance of a machine learning algorithm.

In the same way machine learning is a branch of artificial intelligence, machine learning algorithms can be divided into different groups according to their taxonomy: supervised learning; unsupervised learning; a combination of these two, known as semi-supervised learning; and reinforcement learning. Let us introduce briefly the first two.

In the supervised learning setting the data samples are composed of two parts: the input, also referred to as *observations* or *features*, and the output or *labels*. The goal is to learn a function using labelled inputs – training data – so that it can later be used in unseen inputs

to predict their labels. One classical example of supervised learning is *classification*, where the output belongs to a finite set of discrete elements. Figure 1.1 shows a simple example of binary classification where one class is represented by circles and a second class is represented by squares. The classifier learns from data a separation of the space into two regions such that points that lie in different regions will be labelled differently. We will talk more about classification later in section 2.1. Another well-known instance of supervised learning is *regression*. In regression, in contrast with classification, the output is not discrete, rather it belongs to a continuous space like the space of real valued vectors. Figure 1.2 illustrates the result of applying regression to a set of data points distributed with the form of a sinc corrupted with Additive White Gaussian Noise (AWGN).

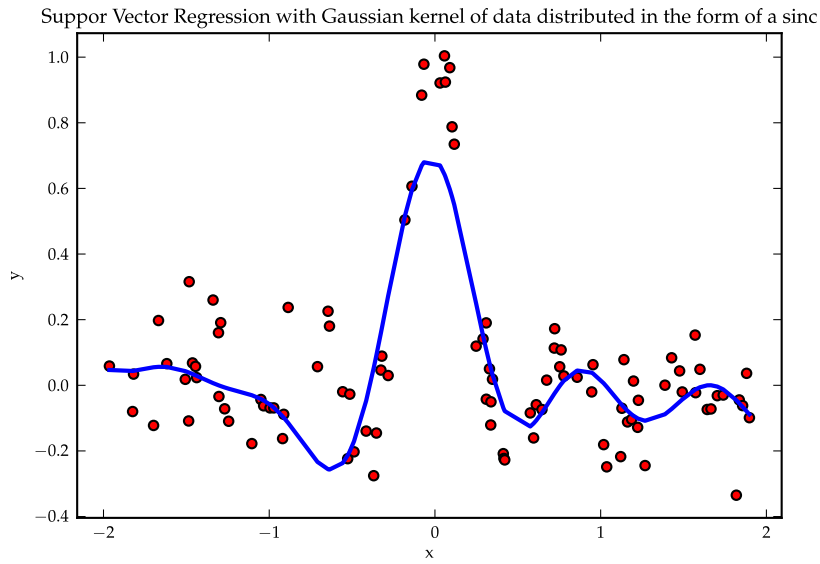


Figure 1.2.: Regression of data with the form of the sinc function corrupted with Additive White Gaussian Noise with standard deviation equal to 0.1. The algorithm used is Support Vector Regression with Gaussian kernel. Note that the estimate (blue solid line) resembles a noiseless sinc even though the data points (red circles) do not seem to follow a sinc.

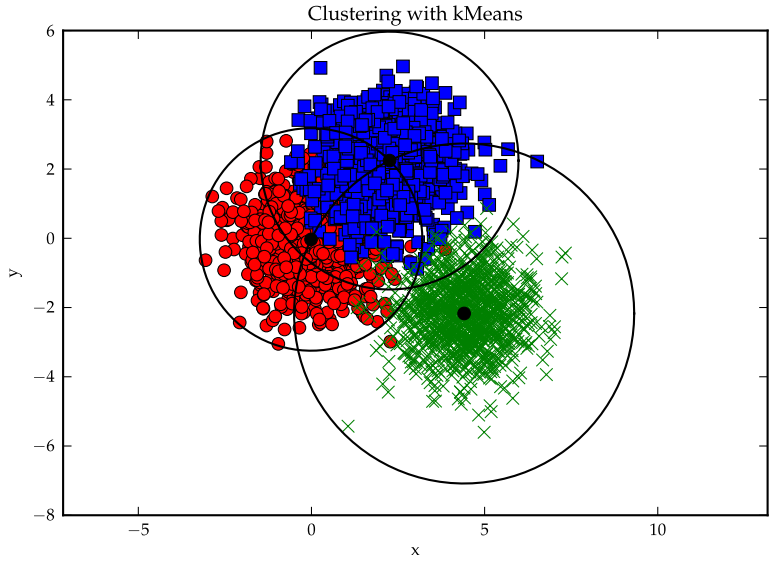
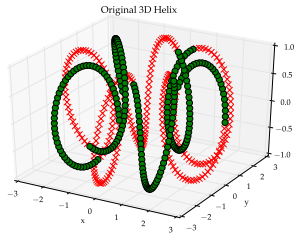


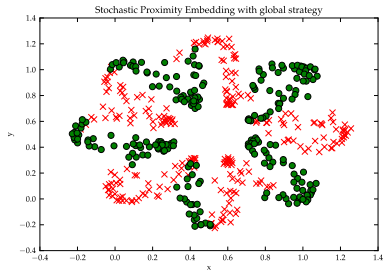
Figure 1.3.: Clustering of two dimensional points into three classes performed with the *kMeans* algorithm. The shape (circle, square or cross) of the points plotted depends on the class assigned by the clustering. The center of the clusters is also shown as well as circunferences representing their spread.

In contrast with supervised learning, the data handled by unsupervised learning algorithms is not labelled or, in other words, only features are available. Not all methods for unsupervised learning share the same goal. There are for instance *clustering* algorithms that aim at grouping the input features into different classes. To do so, the clustering algorithm looks for properties that data samples have in common, in order to associate them with the same class. Figure 1.3 shows an example where input data points were clustered into three classes. Another big family of unsupervised learning algorithms is dimensionality or *dimension reduction*. In dimension reduction the input data lie in a high dimensional space and the aim is to reduce the dimension of the data while preserving its distinguishing properties as much as possible. The purpose of applying dimension reduction may be to make algorithms run faster, as it reduces the size of the input data, or for interpretation purposes since data can be embedded into two or

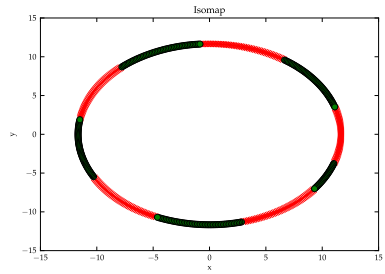
three dimensional spaces. Figure 1.4 depicts the results of applying two different methods for dimension reduction to $3D$ helix-shaped data.



(a) Original helix data.



(b) Stochastic Proximity Embedding with global strategy.



(c) Isomap.

Figure 1.4.: Illustration of dimension reduction. Points in the 3D space following the shape of an helix are embedded into the 2D plane using two algorithms for dimension reduction, stochastic proximity embedding (Agrafiotis 2003) and isomap (Tenenbaum, de Silva, and Langford 2000). Note how the intrinsic properties of the original data are kept in the embedded representations.

1.2 STRUCTURED OUTPUT LEARNING

Structured output learning is, broadly speaking, about using dependencies among variables. In the structured framework neither parameter learning nor inference are done using variables independently. The variables are treated jointly and the information of one variable may as well affect knowledge about the other variables.

One may also think of structured learning as an extension of the classical supervised learning problem of classification. In structured output learning the labels or outputs are objects with their own complex structure, they are multivariate and there exist dependencies among these variables. For instance they can be sequences, trees or even graphs. This is different from standard classification where the labels are simple entities taken from a discrete and finite set.

Structured learning has been successfully applied to several tasks stemming from rather distinct domains. For example, in natural language processing structured prediction is employed to extract the parse tree that represents a given sentence. In Figure 1.5 an example of a parse tree is shown. It is obvious that words in a sentence contain information regarding the location in the tree of other, mostly adjacent, words of the sentence. Say for instance that one word is a verb, then it is likely that the word next to it belongs to the predicate of the sentence.

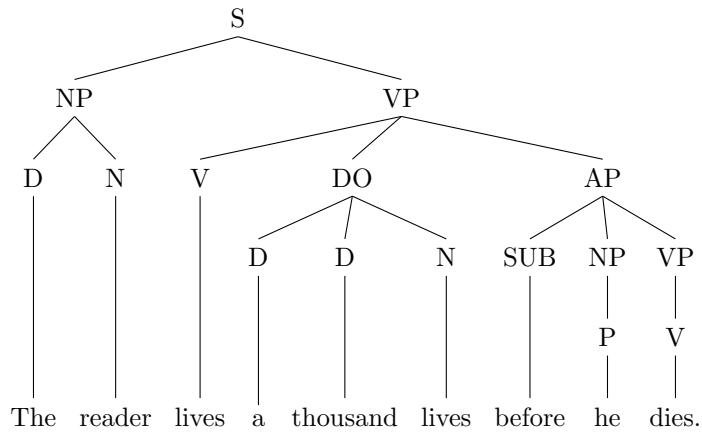


Figure 1.5.: A quote from a novel by George R. R. Martin and its parse tree. In the tree S stands for statement or sentence, NP for noun phrase, VP for verb phrase, D for determinant, N for noun, V for verb, DO for direct object, AP for adverbial phrase, SUB for subordinator and P for pronoun. In natural language processing structured models are trained from pairs (sentence, parse tree) and later used to predict new parse trees for unseen sentences.

Bioinformatics and computational biology are other fields where research in structured learning is fast paced. One task tackled by

structured output prediction in this domain is transcript identification from DNA microarrays (Zeller et al. 2008). This is often used to improve the interpretation of genome data.

Nevertheless, it is very likely that computer vision is the field that has been most fruitful for structured learning applications so far. Some of these applications are Foreground/Background (FG/BG) segmentation, semantic segmentation and face landmark detection. In FG/BG segmentation the aim is to label the pixels of an image as foreground pixels if they belong to some region of interest in the image, or as background pixels otherwise. Semantic segmentation takes one step further grouping the pixels according to some interpretation of the region they belong to. For example, pixels that belong to the sky in an image, to a road or to an animal. Obviously, adjacent classes of adjacent pixels do depend on each other. Imagine for instance eight pixels forming a squared frame with a hole in the middle. If all pixels in the frame belong to the background, then it is very likely that the pixel in the center also belongs to it. This type of dependencies are naturally exploited with structured learning.

(a) Original image (b) FG/BG segmentation

Figure 1.6.: An image of a horse and its binary or FG/BG segmentation. The segmented image is black everywhere but the pixels that belong to the horse in the original image. Data taken from Everingham et al. (*The PASCAL Visual Object Classes Challenge 2012 (VOC2012)*).

Deformable Parts Models (DPM) are also well suited to structured learning and prediction. In human pose detection it is common to use landmarks like the position of the arms and legs, whereas in

(a) Original image (b) Semantic segmentation

Figure 1.7.: An image with some people in a restaurant and its semantic segmentation. The segmented image contains different classes depending on whether the pixels in the original image belong to a table, a chair, a bottle or a person. Data taken from Everingham et al. (*The PASCAL Visual Object Classes Challenge 2012 (VOC2012)*).

facial landmarks detection the eyes and the nose are used, among others (Uricár, Franc, and Hlavác 2012). A straightforward approach is to train and use detectors for each landmark independently. However, using some prior information on the geometrical configuration of the landmark, mostly the typical relative position between landmarks, allows for join detection of the landmarks. A natural way to describe these dependencies and prior information is using an undirected graph, which is what indeed DPM do.

The origin of structured models roots in the Ising model (Ising 1925), proposed almost a century ago, that was later generalized and related to the popular Markov Random Field (MRF). An important difference between the method central to this project, the so-called Structured Output Support Vector Machine (SO-SVM), and these traditional models is that the SO-SVM is a *discriminative model*, while the latter have traditionally been *generative models*. The important distinction is that discriminative models condition on the observed data or, in other words, discriminative models attempt to capture the distribution of the labels conditioned on the observations, meanwhile generative models attempt to represent the joint distribution over labels and observations.

Further discussion about discriminative and generative models is left for section 1.3.

1.3 PROJECT GOALS

The goals of the project presented in this document are twofold. On the one side, we review the current state-of-the-art in structured learning of general output models (i.e. network-graphs). A proof of concept implementation of a simple graph prediction task is developed as well. This is explained in section 4.3.

On the other side, we aim at performing a experimental comparison between generative and discriminative models. Generative models work with the joint distribution of the input and the output spaces, whereas discriminative models condition the outputs $y \in \mathcal{Y}$ on the observations $x \in \mathcal{X}$. The former is actually a density estimation problem, which is provably harder than the latter, see Cherkassky and Mulier (2007); Lafferty, McCallum, and F. Pereira (2001). Moreover, discriminative models work directly with the distribution of interest under a predictive approach or, in other words, with situations where the aim is to map inputs to outputs. For this comparison we use the Hidden Markov Model (HMM) on the generative side and structured models based on the Support Vector Machine (SVM) on the discriminative side. Label sequence learning tasks presented in section 4.1 and section 4.2 are chosen to compare these approaches. Similar work has already been presented in the literature, see for instance Altun, Tsochantaridis, Hofmann, et al. (2003); Collins (2002); Zeller et al. (2008).

2

BACKGROUND

In this chapter, the basic foundations of classification and the Hidden Markov Model (HMM) are introduced. These are very important to fully understand the methodology described in chapter 3. Nonetheless, the aim is not to be as exhaustive as possible and instead provide with basic knowledge; enough to understand the tools used by methods which will be described later on. The reader is referred to (Cherkassky and Mulier 2007) for a deeper analysis of classification and to (Leijon and Eje Henter 2012) for the HMM. The end of the chapter also contains a short discussion about cross-validation, which has been thoroughly used in the experiments described in chapter 4.

Throughout this and the next chapter, optimization problems and other concepts taken from mathematical optimization appear repeatedly. Even so, we consider that it is not within the scope of this document to introduce this branch of mathematics and thus the reader is referred to the good presentations of the topic in the literature, for instance Boyd and Lieven Vandenberghe (2004), and Antoniou and Lu (2007) for an introductory though exhaustive discussion of the topic and Sra, Nowozin, and Wright (2012) for a more advanced treatment focused on machine learning tasks.

2.1 CLASSIFICATION

Given an input sample $x \in \mathcal{X}$, classification deals with the problem of assigning x to a class or category C_k within a set of K classes denoted by $\mathcal{Y}$, i.e. $\mathcal{Y} = \{C_1, C_2, \ldots, C_K\}$. In classification, the number of classes is known a priori and every element $x \in \mathcal{X}$ belongs to one, *and only one*, of the classes C_k. Let us define the categorical variable y as the identifier of the class x belongs to; i.e., $y = k$ means that x is an instance of class C_k. Notice that the output space in classification, denoted here as $\mathcal{Y}$, is restricted to be discrete and finite.

Under a *predictive* approach, the goal of classification is to obtain a model for the mapping $\mathcal{X} \to \mathcal{Y}$ by means of labelled training data $\{(x_i, y_i)\}_{i=1}^{n}$, so that this model can be later utilized to predict the

class y of new instances x not previously seen by the classifier. Both training and prediction (or test) data are Independent and Identically Distributed (iid) according to an underlying joint probability density $p_{\mathcal{X},\mathcal{Y}}(x,y)$ which is unknown. In order to select a suitable model $f(x;\mathbf{w}_0)$ among all the possible ones $f(x;\mathbf{w})$, it is interesting to have a measure of the model quality. Here $\mathbf{w}$ denotes the vector of variables that parametrize the model f, commonly known as the *weight vector*. Let us first define as an intermediate step to achieve this model assessment the *loss function* $L(y,f(x;\mathbf{w}))$ as a measure of the discrepancy between the output of the classifier $f(x;\mathbf{w})$ and the true class y that x belongs to. A very common loss function used in classification is the *zero-one* loss,

$$L(y,f(x;\mathbf{w})) = \begin{cases} 0, & \text{if} \quad y = f(x;\mathbf{w}), \\ 1, & \text{if} \quad y \neq f(x;\mathbf{w}). \end{cases} \tag{2.1}$$

The *indicator function*, which is equal to one if its argument is true and zero otherwise, can be used to rewrite the zero-one loss in a more compact form

$$L(y,f(x;\mathbf{w}) = [[y \neq f(x;\mathbf{w})]]. \tag{2.2}$$

The expected value of the loss function is called *risk functional* (see Equation 2.3) and gives a measure of how well the model performs. However, as it has just been mentioned, the joint distribution is normally not known, thus the risk functional shall not be directly evaluated using Equation 2.3. Alternatively, the training data can be used in a similar way, producing the so-called *empirical risk* as in Equation 2.4. The selection of the most appropriate model or, as it is known widely, learning amounts to finding the $f(x;\mathbf{w}^*)$ that minimizes the empirical risk in Equation 2.4.

$$R(\mathbf{w}) = \int_{\mathcal{X}\times\mathcal{Y}} L(y,f(x;\mathbf{w}))\ p_{\mathcal{X},\mathcal{Y}}(x,y)\ dx\ dy \tag{2.3}$$

$$R_{emp}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} L(y_i,f(x_i;\mathbf{w})) \tag{2.4}$$

Depending on different criteria such as the classifier parametrization or how the risk is attempted to be minimized, there exist a wide range of methods to fit a classifier to data. In the following section the focus is on a type of classifiers called *separating hyperplanes*. These classifiers are relevant for this project because it is rather intuitive to build upon

them the foundation for the Support Vector Machine (SVM), which is the main learning method used through the coming sections.

2.1.1 *Separating Hyperplanes*

In many learning tasks the objects in $\mathcal{X}$ are vectors in a d-dimensional space. Moreover, even for the applications where they are not vectors but rather other type of structured objects (e.g. images, sequences, graphs), it is typical to use a feature function $\phi : \mathcal{X} \to \mathbb{R}^d$ or, as in the structured output learning setting (that will be formally introduced in chapter 3) a joint feature function $\Psi : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^d$ that maps these structured objects to vectors. In other words, it makes sense to assume that the classifier will be dealing with vectors. Let us for ease of notation refer to the objects in the input space $\mathcal{X}$ as vectors $\mathbf{x}$, without loss of generality, regardless whether they were initially vectors, or any of the mappings ϕ or Ψ were applied. Taking into account this observation, classification can be formulated in terms of geometry as the problem of dividing the space $\mathbb{R}^d$ into K different regions, each of them representing a class C_k, such that if the data point $\mathbf{x}$ lies within the region k then its label is $y = k$.

For the sake of clarity, the discussion in this section starts by proposing a classifier that is able to perfectly discriminate two-class linearly separable data. Later on, these two assumptions (linear separability and binary classification) will be dropped, giving rise to a more general setting.

When the data is binary and linearly separable, it is possible to find a hyperplane able to perfectly discriminate all the data points; that is, there exists a hyperplane that partitions the space into two regions such that all the points that belong to one class lie in one region whereas all the points of the other class lie in the other region of the space. An hyperplane is a d-dimensional generalization of a straight line in $\mathbb{R}^2$, Figure 2.1, and it can be represented by the equation

$$\mathbf{w}^T\mathbf{x} + b = 0, \tag{2.5}$$

where $\mathbf{w}$ is the vector normal to the surface of the hyperplane, b is the bias and $\mathbf{x}$ represents any point in the hyperplane. The following geometrical properties of the hyperplane are relevant in order to understand classification with separating hyperplanes:

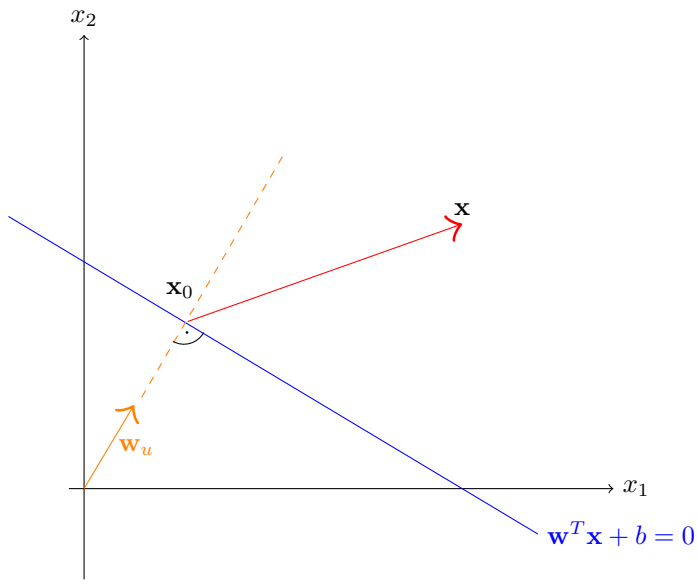


Figure 2.1.: Hyperplane representation in $\mathbb{R}^2$ (i.e. straight line). The figure depicts an hyperplane in the $2D$ plane together with its equation, its unit normal vector $\mathbf{w}_u$ and the vector formed by a point in the hyperplane $\mathbf{x}_0$ and another point $\mathbf{x}$.

- For any point $\mathbf{x}_0$ in the hyperplane $\mathbf{w}^T\mathbf{x}_0 = -b$.
- As a consequence of the first property, for any two points $\mathbf{x}_1$ and $\mathbf{x}_2$ in the hyperplane, $\mathbf{w}^T(\mathbf{x}_1 - \mathbf{x}_2) = 0$, thus $\mathbf{w}_u = \mathbf{w}/\|\mathbf{w}\|$ is the unit vector normal to the surface of the hyperplane.
- More importantly, the *signed* distance between any point $\mathbf{x}$ and the hyperplane is given by

$$\mathbf{w}_u^T(\mathbf{x} - \mathbf{x}_0) = \frac{1}{\|\mathbf{w}\|}\left(\mathbf{w}^T\mathbf{x} - \mathbf{w}^T\mathbf{x}_0\right) = \frac{1}{\|\mathbf{w}\|}\left(\mathbf{w}^T\mathbf{x} + b\right). \tag{2.6}$$

The signed distance measure includes the notion of the *side of the hyperplane*. In other words, if there are two points equally far from the hyperplane, where each of them lies in a different region of the space; then their distances to the hyperplane would have the same absolute

value, although their sign would be different. Therefore, classification using a hyperplane is typically done using the *sign* function

$$f(\mathbf{x}) = \text{sign}(\mathbf{w}^T\mathbf{x} + b), \tag{2.7}$$

for which $\|\mathbf{w}\|$ does not need to be taken into account because it is always positive. Let us continue the discussion introducing techniques whose objective is to find separating hyperplanes.

2.1.1.1 *Perceptron algorithm*

Following the principle of minimizing the empirical risk in Equation 2.4, the perceptron algorithm seeks a *decision boundary* by means of minimizing the distance of misclassified points to a region where they are correctly classified.

Provided that the training data is composed of two different classes labelled as $+1$ and -1, a data point with true label $y_i = +1$ is misclassified if $\mathbf{w}^T\mathbf{x}_i + b < 0$ and, in a similar way, a data point whose ground truth is $y_i = -1$ is misclassified if $\mathbf{w}^T\mathbf{x}_i + b > 0$. Formally, the aim is to minimize the cost

$$C(\mathbf{w}, b) = -\sum_{i \in \mathcal{M}} y_i \, (\mathbf{w}^T\mathbf{x}_i + b), \tag{2.8}$$

where the set $\mathcal{M}$ includes the misclassified points. This function can be easily minimized using stochastic gradient descent (see, e.g., (Antoniou and Lu 2007) and the discussion on stochastic subgradient descent in subsection 3.2.2). The gradients of Equation 2.8 with respect to the normal vector and the bias are, respectively,

$$\frac{\partial C(\mathbf{w}, b)}{\partial \mathbf{w}} = -\sum_{i \in \mathcal{M}} y_i \, \mathbf{x}_i, \tag{2.9}$$

$$\frac{\partial C(\mathbf{w}, b)}{b} = -\sum_{i \in \mathcal{M}} y_i. \tag{2.10}$$

The perceptron algorithm operates iteratively performing the update shown in Equation 2.11 for every point not correctly discriminated. In the equation, t is an identifier for the iteration and η is the so-called *learning rate*, which is useful to tune the influence of the steps taken. The algorithm converges when all the data points are on the appropriate side of the decision boundary.

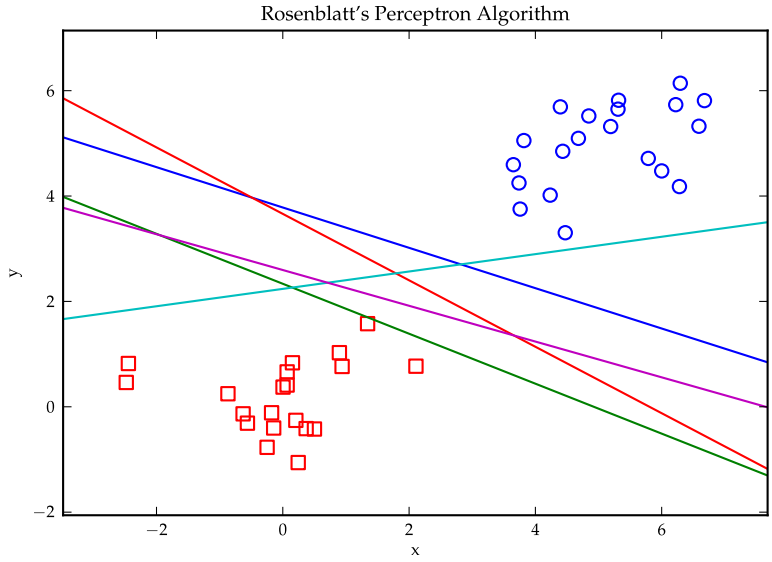


Figure 2.2.: Two-class linearly separable data and several separating hyperplanes found using the perceptron algorithm. One of the classes is represented by circles and the other by squares. For each class, data points are drawn randomly from a Gaussian distribution. Note how different the separating hyperplanes are even if they are found by the same algorithm. This is due to distinct initial weight vector and bias values.

$$\begin{pmatrix} \mathbf{w} \\ b \end{pmatrix}^{(t+1)} \leftarrow \begin{pmatrix} \mathbf{w} \\ b \end{pmatrix}^{(t)} + \eta \begin{pmatrix} y_i \, \mathbf{x}_i \\ y_i \end{pmatrix}. \tag{2.11}$$

The perceptron algorithm can be shown to converge to a separating hyperplane in case of linearly separable data. There are however some issues with the perceptron algorithm, these are:

1. For linearly separable data it is not possible to know to which solution, among the infinitely many feasible ones, the algorithm will converge. This depends strongly on the values used to initialize the normal vector and the bias. This is demonstrated in Figure 2.2.

2. Although the convergence of the algorithm is guaranteed for linearly separable data, the number of steps the algorithm shall take may be rather large. A large value of the learning rate tends to reduce the amount of iterations, albeit it can provoke undesirable oscillatory behaviour.
3. Finally, for non-linearly separable data, the algorithm will not converge.

A neat solution to the first problem is detailed in the next section using the concept of *margin*. In a later section it will be shown how the assumption of linearly separable data can be dropped, arriving finally at the traditional formulation of the SVM for classification.

2.1.2 *Large Margin Separating Hyperplanes*

The margin concept in separating hyperplanes arises from the following intuitive observation: among all the possible hyperplanes that discriminate linearly separable data, those that are far from their nearest data point are more desirable than those that are closer, since the latter will be more sensitive to noise and are likely not to generalize as well as the former for data not used in training. Let us describe in this section how this statement can be formalized so that a method can be built upon it.

Let us define the *margin* as the minimum distance between the hyperplane and the closest point in the dataset (independently of its labelling) and denote it M. Since the same hyperplane can be represented in an infinite number of ways by scaling the normal vector and the bias[1]; as a matter of convention, the chosen one is required to have a unit normal vector, i.e.

$$\|\mathbf{w}\| = 1.$$

1 Recall from Equation 2.5 that a hyperplane is represented by $\mathbf{w}^T\mathbf{x} + b = 0$. In order to see the invariance to normal vector and bias re-scaling, just multiply both sides of the expression by the same scalar value.

As explained in the previous paragraph, the sought hyperplane is the one which gives the maximum margin. These conditions altogether establish the following optimization problem,

$$\begin{aligned} &\max_{M,\mathbf{w},b} \quad M \\ \text{subject to } &y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq M, \quad \forall i = 1,\dots,n, \\ &\|\mathbf{w}\| = 1. \end{aligned} \tag{2.12}$$

If the constraint $\|\mathbf{w}\| = 1$ is dropped, then Equation 2.12 is a Linear Program (LP). This constraint can be dropped by replacing the constraints associated with every point by

$$\frac{1}{\|\mathbf{w}\|}\, y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq M, \tag{2.13}$$

which implicitly forces $\mathbf{w}$ to be unitary and scales the bias with respect to the one used in Equation 2.12. This is equivalent to multiplying both sides of Equation 2.13 by the norm of the normal vector (as the norm is a positive number, the direction of the inequality shall not be modified),

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq \|\mathbf{w}\|\, M. \tag{2.14}$$

Finally, the problem can be now expressed in the following way:

$$\begin{aligned} &\min_{\mathbf{w},b} \quad \frac{1}{2}\mathbf{w}^T\mathbf{w} \\ \text{subject to } &y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \quad \forall i = 1,\dots,n, \end{aligned} \tag{2.15}$$

where the following changes have been applied:

- The normal vector has been fixed equal to the inverse of the margin, $\|\mathbf{w}\| = 1/M$. This can be done without loss of generality since it just involves re-scaling the norm and the bias, in the same way to the step performed in Equation 2.13.
- Thus, providing that $\|\mathbf{w}\| = 1/M$, one shall instead of maximizing the margin M, minimize its inverse, the norm of the normal vector $\mathbf{w}$.
- Further, since the square function is monotonically increasing for positive arguments, maximizing the objective is equivalent to maximize the objective transformed by this function. This

transformation is applied in order to get rid of the square root introduced by the norm. The result is equivalent to the dot product of the vector by itself, i.e. $\|\mathbf{w}\|^2 = \mathbf{w}^T\mathbf{w}$.

- The $1/2$ factor is introduced for convenience as it proves the case to be useful when applying gradient based methods due to the simplification of constants.

The problem in Equation 2.15 is a well-known convex optimization problem, a Quadratic Program (QP) with linear inequality constraints, that can be solved directly in the primal or, if convenient, in the dual by introducing Lagrange multipliers. Although somehow obvious, it is important to note that even though, according to Equation 2.15, there will be no point x_i in the training data lying in the wrong side of the margin, this does not need to be the case for points in the test data. It is merely the *thesis* that achieving a large margin on the training data shall lead to good discrimination of the test data.

In this section, introducing large margin separating hyperplanes, the problem seen in the perceptron algorithm where the solution depends on the initial bias and weight vector has been addressed. Nevertheless, Equation 2.15 leads to no solution if the training data is not linearly separable – the set of feasible solutions given by the constraints is the empty space in this case, no matter the norm of $\mathbf{w}$. In the next section, an approach capable of dealing with non-linearly separable data known as Support Vector Machine (SVM) is introduced. This approach is still based on the margin maximization principle described in this section.

2.1.3 *Support Vector Machine for Classification*

In this section the assumption of linearly separable data maintained so far is dropped. A way of dealing with overlapping data of this kind is to modify the constraints in Equation 2.15 so that points in the training data are allowed to lie in the wrong side of the margin, while still maximizing the margin. In order to allow for misclassifications within the training data the so-called *slack variables*, denoted by ξ_i, are introduced.

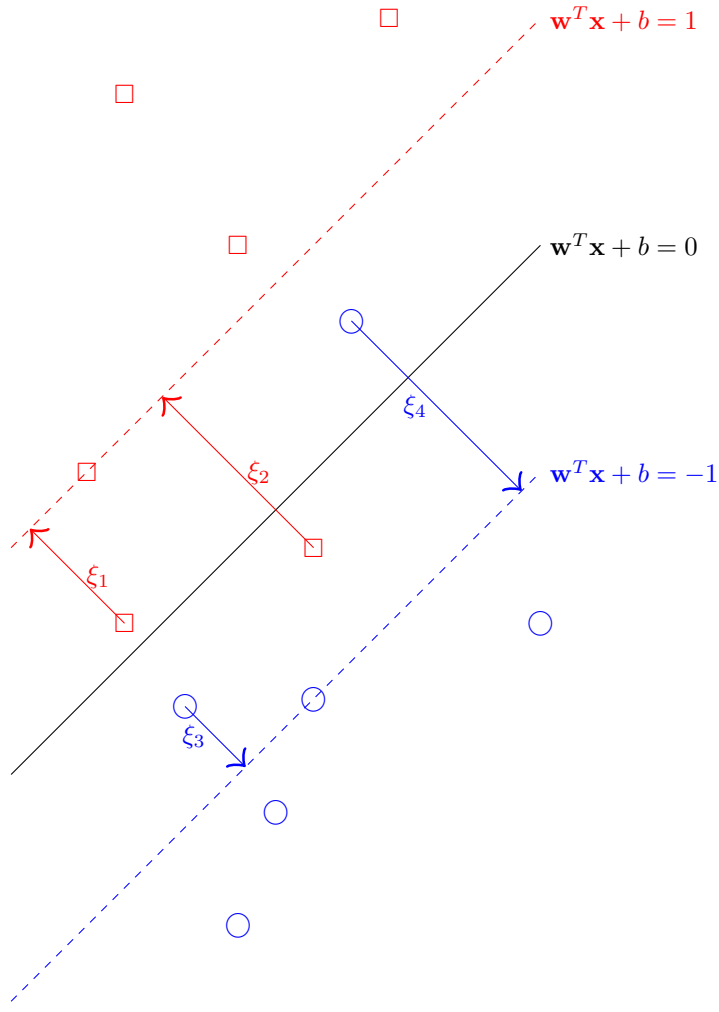


Figure 2.3.: Hyperplane learnt by the Support Vector Machine algorithm for binary non-linearly separable data. The data points with labels $y = +1$ are represented with squares, and those labelled with $y = -1$ with circles. For the examples not correctly classified their respective slack variables are shown. Note that for each class there is a data point lying on the line of the margin. These points are the so-called *support vectors*.

Recall that the constraints in the optimization problem shown in Equation 2.15 are of the form

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \quad \forall\, i = 1, \ldots, n.$$

Figure 2.3 depicts how the slack variables must be introduced in the constraints. It results easier to understand this by studying the cases $y_i = +1$ and $y_i = -1$ independently; afterwards it is possible to put together both cases in a succinct manner. In particular,

$$\left.\begin{array}{l} \text{if } y_i = +1 \rightarrow (\mathbf{w}^T\mathbf{x}_i + b) \geq \quad 1 - \xi_i \\ \text{if } y_i = -1 \rightarrow (\mathbf{w}^T\mathbf{x}_i + b) \leq -1 + \xi_i \end{array}\right\} \quad y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i. \tag{2.16}$$

In addition to the constraints, the objective in Equation 2.15 must be also transformed since it is desirable to maintain most of the points in the right side of the margin or, equivalently, the distances to their correct side of the margin as small as possible. The geometrical interpretation of the slack variables is precisely the distance to the adequate region of the margin. Thus, it is reasonable to introduce the slack variables in the objective to be minimized. The new constraints introduced in the paragraph above together with the objective including the slack variables establish the SVM formulation for classification:

$$\begin{aligned} \min_{\mathbf{w},b,\xi} \quad & \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{n}\xi_i \\ \text{subject to } & y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i, \quad \forall\, i = 1, \ldots, n, \\ & \xi_i \geq 0, \qquad\qquad \forall\, i = 1, \ldots, n. \end{aligned} \tag{2.17}$$

The last group of constraints on the slack variables in Equation 2.17 are necessary for consistency. It does not make sense to regard negative values for ξ_i since that would only be possible if the training example i is already on the correct side of the margin, providing the other set of constraints. Note that non-zero values of the slack only occur for points incorrectly discriminated. Apart from this, the *cost* parameter or *regularization* C in Equation 2.17 appears because the objective now is composed of two different contributions. This parameter allows a trade-off between how large the margin will be – a large margin is achieved with small values of C – and the fit to the training data – favoured with large values of C. The latter case would produce in an extreme case a high variance model *overfitted* to the data, whereas the

model of the former would be *biased* with little or even no consideration of the training data in an extreme case. An appropriate way of choosing the value of the cost parameter is to divide the data into different sets (training, test and validation) and perform any form of cross-validation, see section 2.3. As a final comment, note that other penalizations or costs different than $\sum \xi_i$ could have been used. For instance, it would have been possible to use a quadratic penalization $\sum \xi_i^2$. The penalization $\sum \xi_i$ is known as the l_1-norm (note that the slack variables are positive, which is why the absolute value is not included) and it is used because it reinforces *sparsity*. In other words, a large number of zero slack variables.

It is worth mentioning before the end of this section that the discussion of the SVM presented here is based on its geometrical interpretation. Nonetheless, this interpretation is not unique. A deeper interpretation based on Vapnik's Statistical Learning Theory (STL) is well explained in (Cherkassky and Mulier 2007).

2.1.4 *Multiclass Support Vector Machine*

This section introduces a generalization of the binary SVM described in subsection 2.1.3. This generalization is based on the work by Crammer and Singer (2002) in multiclass kernel machines. Nevertheless, this approach is not the only possible one to make the SVM amenable to multiclass classification. Let us start this section with a brief review of other approaches to multiclass classification and later focus on the multiclass SVM by Crammer and Singer (2002), which is more relevant taking into account the scope of this project.

There is a framework for multiclass problems called Error-Correcting Output Codes (ECOC) whose main idea is to create a multiclass classifier built upon a set of binary classifiers. The simplest ECOC are *one-vs-rest* and *one-vs-one*. The one-vs-rest [2] strategy uses K – recall that K denotes the number of classes – binary classifiers $f_k(x; \mathbf{w}_k)$ with $k = 1, \dots, K$. The data used to train each classifier belongs to one of two classes: the data labelled with $y = k$ to one class, and all the other data labelled with $y \in \{1, \dots, K\} \setminus \{k\}$ to the other class. The overall classifier assigns

2 One-vs-rest is also known as one-vs-all in part of the Machine Learning community. However, here the name one-vs-rest is used since one-vs-all might be misleading as it literally means to discriminate examples that belong to a class k from examples that belong to one of the classes $1, \dots, K$, including k.

a class to the example x by maximizing some confidence value or score given by each $f_k(x; \mathbf{w}_k)$. In contrast with one-vs-rest, one-vs-one requires $K(K-1)/2$ classifiers; that is, it simply trains a classifier between every two pair of classes. At prediction time, all the classifiers are applied to the test example x, and the most voted class (i.e. the one that has been output more times among the $K(K-1)/2$ classifications) is chosen. These two, and other output codes for multiclass learning problems, are well discussed in (Dietterich and Bakiri 1995). However, breaking the multiclass classification problem into several *independent* binary problems does not account for the possible correlations between the classes. In structured learning these correlations are particularly important since, as it will be described in chapter 3, it is crucial to capture the interdependencies among classes. The approach described next based on Crammer and Singer (2002) takes into consideration these correlations.

Let us introduce a parametrization of the multiclass classifier using a matrix W of size $K \times d$, where K denotes again the number of classes and d the dimension of the input space, either because $\mathcal{X}$ is actually $\mathbb{R}^d$ or because any of the mappings discussed in subsection 2.1.1 has been applied. The classification rule is formally defined as

$$f(\mathbf{x}; W) = \underset{k=1,\ldots,K}{\operatorname{argmax}} \mathbf{W_k}^T \mathbf{x}, \tag{2.18}$$

where $\mathbf{W_k}$ denotes the kth row of the matrix W, which can be of course regarded as a vector. The inner-product $\mathbf{W_k}^T \mathbf{x}$ is called the *confidence* or *similarity score* of $\mathbf{x}$ for class k. During training the goal is to find a matrix W that minimizes the empirical risk, which was described in section 2.1, using a set of labelled data $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$. For convenience, the empirical risk particularized for this scenario is written in Equation 2.19. Unfortunately, direct minimization of the empirical risk for multiclass classification entails a significant computational burden (Crammer and Singer 2002). For this reason, an approach which takes measure of model quality using a modified empirical risk is sought instead.

$$R_{emp}(W) = \frac{1}{n} \sum_{i=1}^{n} [[f(\mathbf{x}; W) \neq y_i]] \tag{2.19}$$

One upper bound on the empirical risk for one example is shown in Equation 2.20, where δ_{ij} is the Kronecker delta. This expression

can be proven by studying the two possible cases $k = y_i$ (correct classification) and $k \neq y_i$ (misclassification) separately. The upper bound in Equation 2.20 (also known as loss) renders two properties:

- It is equal to zero if the similarity score for the true class $\mathbf{W_{y_i}}^T \mathbf{x}_i$ is larger than, or equal to one plus the maximum score for the rest of the classes, i.e. $1 + \max_{k=1,\dots,K} \mathbf{W_k}^T \mathbf{x}_i$ with $k \neq y_i$. The fact of taking into account this offset equal to one should remind the reader of the margin concept from the binary SVM.

- It is non-zero in case of misclassification or correct classification with short margin. On the one hand, for misclassified examples the loss is positive and linearly proportional to the difference between the score of the correct label and the score of the incorrect label given by the classifier. On the other hand, for correctly classified examples with a short margin it is proportional to the difference between the score and the margin.

$$[[\underset{k=1,\dots,K}{\operatorname{argmax}} \mathbf{W_k}^T \mathbf{x}_i \neq y_i]] \leq \max_{k=1}^{K} \left\{ \mathbf{W_k}^T \mathbf{x}_i - \delta_{y_i k} + 1 \right\} - \mathbf{W_{y_i}}^T \mathbf{x} \qquad (2.20)$$

These properties can be visualized in Figure 2.4. In the left-most figure the test sample denoted by a circle is correctly classified with a large-enough margin and there is no loss. In the center figure the sample is correctly classified but with a small margin, thus there is already a loss different from zero (denoted by the double arrow). In the right-most figure the sample is incorrectly classified and the loss is considerably larger.

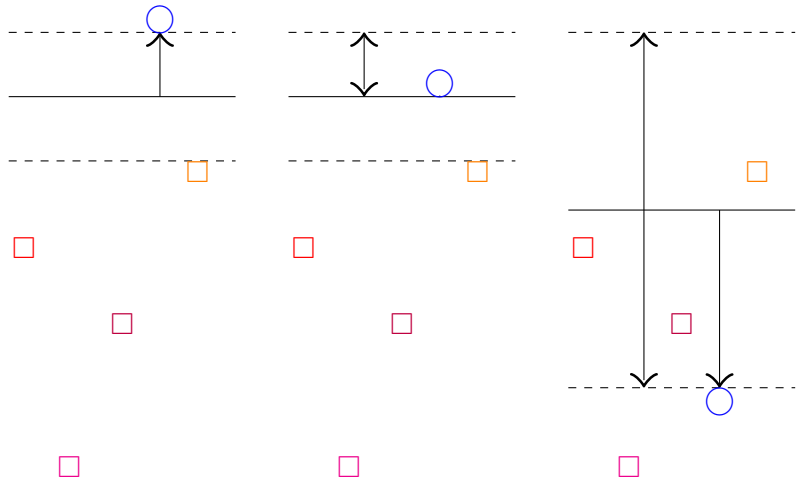
In a similar fashion to subsection 2.1.2, let us first assume that the training data is linearly separable by a multiclass classifier of the form defined in Equation 2.18; that is, there exists a matrix W such that the loss for all the examples is equal to zero,

$$\max_{k=1,\dots,K} \left\{ \mathbf{W_k}^T \mathbf{x}_i - \delta_{y_i k} + 1 \right\} - \mathbf{W_{y_i}}^T \mathbf{x} = 0 \quad \forall\, i = 1 \dots, n,$$

which can be rewritten as

$$\mathbf{W_{y_i}}^T \mathbf{x} - \mathbf{W_k}^T \mathbf{x} + \delta_{y_i k} \geq 1 \quad \forall\, i = 1, \dots, n,\ k = 1, \dots, K.$$

Once these constraints are established, a new optimization problem can be formulated introducing a regularization term for W due to the same



(a) Correct classification with large margin.

(b) Correct classification with small margin.

(c) Misclassification.

Figure 2.4.: Illustration of the margin concept in multiclass classification with the Support Vector Machine by Crammer and Singer (2002). In the figures the simple arrow denotes the normal from the hyperplane to the test example, denoted by a circle. The double arrow symbolizes the loss incurred. The squares denote examples from different classes.

margin maximization and good generalization desirable properties that were introduced in subsection 2.1.2:

$$\begin{aligned} \min_{W} \quad & \frac{1}{2}\|W\|_2^2 \\ \text{subject to } & \mathbf{W_{y_i}}^T\mathbf{x} - \mathbf{W_k}^T\mathbf{x} + \delta_{y_i k} \geq 1 \quad \forall\, i = 1,\ldots,n,\ k = 1,\ldots,K, \end{aligned} \tag{2.21}$$

where $\|A\|_2^2$ denotes the *Frobenius* norm of the real valued matrix A which is simply

$$\|A\|_2^2 = \sum_{i,j} A_{ij}^2 = \sum_i \mathbf{A_i}^T\mathbf{A_i}, \tag{2.22}$$

being i and j indices for the row and columns of A, respectively. One more time, as in subsection 2.1.3, slack variables are needed in case the training data is non-linearly separable. Very similar to Equation 2.17

the primal formulation of the multiclass classification problem for data not necessary linearly separable becomes finally:

$$\begin{aligned} \min_{W,\xi} \quad & \frac{1}{2}\|W\|_2^2 + C\sum_{i=1}^{n}\xi_i \\ \text{subject to } & \mathbf{W_{y_i}}^T\mathbf{x} - \mathbf{W_k}^T\mathbf{x} + \delta_{y_i k} \geq 1 - \xi_i \quad \forall\, i = 1,\ldots,n,\ k = 1,\ldots,K, \\ & \xi_i \geq 0, \qquad \forall\, i = 1,\ldots,n. \end{aligned} \tag{2.23}$$

It is worth mentioning that even though Equation 2.17 and Equation 2.23 may seem different at first sight, they represent in fact the same principle. Equation 2.23 is just a natural generalization of Equation 2.17 for problems with more than two classes. For instance, the parametrization used for the multiclass classifier is a matrix W while a vector $\mathbf{w}$ is used in the binary case. This is nonetheless only a matter of representation and the same development could have been done in the multiclass case using a vector, let us call it $\mathbf{w_m}$ for multiclass, instead of a matrix. In particular, by means of: a) setting $\mathbf{w_m}$ equal to a row-wise vectorized version of W; and b) using a mapping $\psi(\mathbf{x}_i, y_i)$ instead of $\mathbf{x}$ directly in Equation 2.23, the very same problem is still being represented; using a vector instead of a matrix in a more similar fashion to Equation 2.17 though. This mapping ψ would only set the vector $\mathbf{x}$ unmodified with an appropriate shift depending on the value of y_i, leaving the rest of the elements equal to zero. This insight will become more relevant in the structured output learning framework described in chapter 3 where mappings such as this one are continuously used.

2.2 HIDDEN MARKOV MODELS

In this section a brief introduction to Hidden Markov Models following the presentation by Leijon and Eje Henter (2012) is given. The aim of the discussion is to refresh concepts like conditional independence, latent states only perceived through observations, and the Markov chain structure. These concepts will become relevant later on in subsection 3.3.1 to better understand the problem of label sequence learning.

A Hidden Markov Model (HMM) is a tuple with two objects: a *Markov chain* and an array of *output probability distributions*, that here is denoted with B. The scope of this report is restricted to Markov chains whose

state variable at a given point or instant of the chain is represented by a value from a finite discrete set $\mathcal{S}$ of $|\mathcal{S}|$ possible states. However, this is not a limitation of the Markov chain structure. Continuous state Markov chains are commonly associated with a Linear Dynamical System (LDS). The Markov chain of a HMM is in turn composed of two objects, a *initial state probability distribution* q and a *transition probability matrix* A. For the sake of simplicity a HMM is often written as (q, A, B), without making explicit the Markov chain object.

HMMs are used to describe sequence data, i.e. data where each element is not independent of the other elements, but rather provides information about both past and future elements of the sequence. The word Markov is used to say that the state sequence generated by a HMM is a – usually *first-order* – Markov chain. A first-order Markov chain models a sequence of random variables such that each element in the sequence is *conditionally independent* of all the other elements given its immediate predecessor. The order of the Markov chain increases accordingly with the number of predecessors each element is allowed to statistically depend on. For instance, in a second-order Markov chain it is necessary to account for two consecutive values of the sequence in order to keep all the available knowledge regarding the value of the random variable that follows.

It is important to realize that conditional independence does not mean that an element of the chain is independent of future and past elements. Intuitively, the main idea is that in a Markov chain the information flows all along the elements of the sequence. In the moment one of the random variables is known, then the chain is divided into two independent parts and the information cannot go through the fixed element. This idea is represented in Figure 2.5. At first, S_0 (state at time 0) has an influence on $S_1, S_2, S_3, \ldots, S_T = S_{1:T}$. On the contrary, from the moment the value of S_t is known (denoted by a darker colour in the figure) S_0 cannot provide knowledge about $S_{t+1}, S_{t+2}, S_{t+3}, \ldots, S_T = S_{t+1:T}$ other than the one already contained in S_t. The information flow is broken at S_t, with all previous information relevant for S_{t+1} onwards summarized in S_t.

Once the concept of conditional independence between states or beliefs of the Markov chain is clear, let us continue with the formal definition of a HMM. As mentioned earlier, a first-order Markov chain is characterized by an initial state probability distribution q and a transition probability matrix A. The distribution q is a probability mass

that can be represented by a vector of $|\mathcal{S}|$ elements that gives, for every state, the probability of the state taking place at the beginning of the sequence,

$$q_{s_0} = P\left[S_0 = s_0\right] \quad \forall\, s_0 \in \mathcal{S}. \tag{2.24}$$

The transition matrix A is composed of $|\mathcal{S}|$ rows and $|\mathcal{S}|$ columns and its element a_{ij} corresponds to the probability of the Markov chain moving to state j from state i, Equation 2.25. This matrix expresses the first-order Markov property of the chain, and needs to be left stochastic (i.e. each of its columns sums to one). Although more general and richer types of HMM exist, the models used in this report are assumed to be represented with transition matrices that do not vary over time, if not explicitly written otherwise.

$$A_{s_t,s_{t+1}} = a_{s_t s_{t+1}} = P\left[S_{t+1} = s_{t+1} | S_t = s_t\right] \tag{2.25}$$

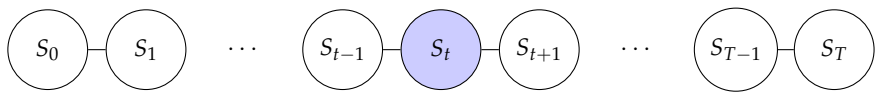


Figure 2.5.: Graphical model for a Markov chain. The lines between states denote conditional dependencies. The state S_t appears shaded meaning that this state is known and all the information previous to S_t is summarized at this node, making the states S_{t+1}, S_{t+2}, and so forth until S_T independent of everything previous to S_t.

The other component of the HMM, the array B of output probability distributions, has $|\mathcal{S}|$ elements defining the probability density of the observation $X_t = x_t$ for each state,

$$b_{s_t}(x_t) = f_{X_t|S_t}\left(x|s_t\right). \tag{2.26}$$

Again, similar to the transition matrix A, these probabilities densities are fixed in time. Note also that the observations are in general vector valued and continuous. They are neither restricted to be scalars nor discrete. This definition for the output distributions makes explicit the independence and conditional independence relationships in the HMM. Figure 2.6 represents these relations using tools from probabilistic graphical models (Koller and N. Friedman 2009). The observation at time $t = t'$ is conditionally independent of all the other observations and states at time $t \neq t'$ given the state of the Markov chain at time t'.

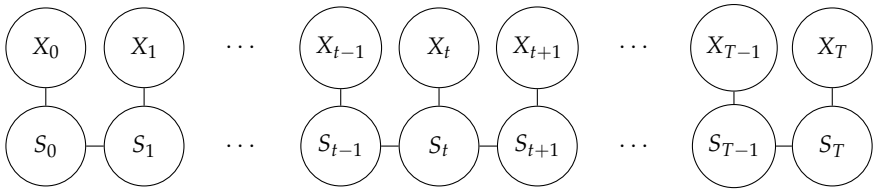


Figure 2.6.: Graphical model for a Hidden Markov Model (HMM). The lines between states, and between states and observations denote conditional dependencies.

In addition to the compact notation and easy interpretation, the HMM structure allows for efficient computation of marginal probabilities through two well-known algorithms: the *forward* and *backward* procedures (Rabiner 1989). These algorithms are instances of dynamic programming. One more notable and efficient procedure applicable to the HMM is the *Viterbi* algorithm. The Viterbi algorithm is used for inference in HMMs and it will be described in detail in subsection 3.3.1.

The HMM discussion in this section has introduced so far a few new concepts, such as conditional independence, and actors, states and observations for instance. At first, it might lead to confusion trying to understand a connection between them and other ideas related to learning from data previously introduced. However, the feature vectors and labels used in classification are no different from the observations and states, respectively, from the HMM nomenclature. Needless to say, it is also possible to estimate the parameters of a HMM (A, B and q) using data. Moreover, relationships like conditional independence are exploited in the algorithms used to estimate these model parameters.

Sequences of states, in contrast with the labels used in classification in section 2.1, are complex in the sense that they are composed of multiple variables and not of a single element taken from a finite discrete set. In chapter chapter 3 the concepts seen in classification shall be generalized to deal with structured or multivariate labels.

2.3 CROSS-VALIDATION

It is rather common within machine learning to deal with problems that depend on one or several parameters that are not in the search space of the learning algorithm. In other words, there are values that may modify – often critically – the goodness of a model for

which no straightforward way of choosing them exist. An example of such parameter is the C that appears in the SVM training algorithm in Equation 2.17. As a matter of convention we will refer to these variables as free parameters in this section. In this cases, a common approach is to use *model selection* and *cross-validation* (Bishop 2006).

The basic idea behind *model selection* is to train a model several times using the same data with different combinations of the free parameters. The combination that gives the best result in a test data set different from the training data is then chosen. Let us describe more in detail how model selection with cross-validation is performed in a simple scenario.

Firstly, the training data is separated into non-overlapping folds. Then, the same model, this is the exactly same choice for all the free parameters, is trained using all the folds except from one that is used for validation. The error in the test fold corresponds to the *cross-validation error for that fold*. This model is trained as many times as folds are, leaving out one different fold for validation each time. The average of the errors obtained during this process is the *cross-validation error* for the model. This process is repeated for all the possible models to evaluate, choosing at the end the model that has performed with minimum cross-validation error. The cross-validation error is an estimate of the performance of the model in unseen data, providing that the data represents accurately the real problem domain.

At this moment one may ask what behaviour is to be expected from the model, taking into account the cross-validation error obtained. On the one hand, the estimate could be too optimistic if the performance of the model in the real world is worse than the estimate given by the cross-validation error. On the other hand, it could be that the real world performance is better. With the model selection process explained so far the estimate is almost certainly optimistic. This stems from the fact that the model selection process has biased the error since the model with minimum error is chosen out of many possible models. Consider for example that a thousand different models are trained. Just owing to randomness, at least a few models are expected to have low cross-validation error.

In order to avoid this bias introduced by the model selection process, cross-validation shall be applied twice. First, the training data is separated into two parts. One used for model selection as described above, and the other part for evaluation. The key idea is that the

evaluation fold is never seen during training. Once the model with minimum cross-evaluation error has been found, its performance is measured in the evaluation fold. The performance obtained this time (measured with the cross-validation error) is likely to be worse than the real world performance, since not all the available data have been used for training.

3

STRUCTURED OUTPUT SUPPORT VECTOR MACHINE

This chapter describes the Structured Output Support Vector Machine (SO-SVM) learning framework from (Tsochantaridis et al. 2005). As it shall be shown, it generalises the binary and multiclass SVM which were explained in chapter 2.

3.1 INTRODUCTION

In chapter 2 the problem of classification was presented as learning a mapping $\mathcal{X} \to \mathcal{Y}$ using labelled training data $\{(x_i, y_i)\}_{i=1}^n$ where the elements of $\mathcal{Y}$ consist of single numbered labels. From now on the focus is placed on output spaces $\mathcal{Y}$ whose objects are *structured*, which means that they are not constrained to be represented by discrete entities within a finite set but are rather allowed to take the form of multiple dependent output variables or complex structures such as sequences, trees, or graphs.

The naive approach one may come up with first would be to regard each of the objects $y_i \in \mathcal{Y}$ as a different class C_i and perform multiclass classification accordingly. However, this approach is doomed to be intractable due to the large number of classes – actually, *exponentially* many – which arises making a one-to-one correspondence between structured objects and classes. At the same time, the amount of training examples such an approach requires would be prohibitive, providing that each class should appear at least a few times in the training data.

An alternative approach consists in mapping the structured output objects to an space of *fixed* dimension d. The mapping shall be done jointly (i.e. using inputs $x \in \mathcal{X}$ and outputs $y \in \mathcal{Y}$ at the same time), so that both the correlations among the output variables each $y \in \mathcal{Y}$ is composed of, and the input-output correlations between pair of examples (x, y) are accounted for. Thus, it can be represented without loss of generality as a function $\Psi : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^d$. Intuitively, the essence of this joint feature mapping is to put together in each of its dimensions information that comes from more than one dimension in the original

data representation. The word fixed at the beginning of this paragraph is intendedly used to emphasize that even in case the original elements in $\mathcal{Y}$ are free to have different sizes, the mapping Ψ shall transform them into vectors in $\mathbb{R}^d$, no matter their original size. In other words, consider a problem where the elements $y \in \mathcal{Y}$ are trees, being y_a and y_b two instances of them. Let us define the application $f_n : \mathcal{Y} \to \mathbb{N}$ that given a tree computes its number of nodes. The fact that the joint space is fixed implies that both $\Psi(x, y_1)$ and $\Psi(x, y_2)$ are in $\mathbb{R}^d$, $\forall\, x \in \mathcal{X}$ even if $f_n(y_1) \neq f_n(y_2)$.

Once it has been explained how the structured objects are transformed into a vector space representation typical of the SVM approach, let us continue the explanation describing the form of the predictor. Using a predictive strategy, the aim of learning is to find a function $f : \mathcal{X} \to \mathcal{Y}$ using labelled training data. This is done by means of a *discriminant function* F which assigns an score to every input-output pair, i.e. $F : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$. A prediction for a given $x \in \mathcal{X}$ is performed searching the $y \in \mathcal{Y}$ which maximizes the $\mathbf{w}$-parametrized F. Formally,

$$f(x; \mathbf{w}) = \underset{y \in \mathcal{Y}}{\operatorname{argmax}}\, F(x, y; \mathbf{w}). \tag{3.1}$$

In the SO-SVM setting, the function F is assumed to be linear in $\mathbf{w}$ and a joint feature space where input-output combinations are mapped with $\Psi(x, y)$,

$$F(x, y; \mathbf{w}) = \mathbf{w}^T \Psi(x, y). \tag{3.2}$$

Therefore, it is useful to think of F as a family of functions parametrized by $\mathbf{w}$ and the goal is to choose one of these functions so that the score of $F(x_i, y^*; \mathbf{w})$ is maximum for y^* very close to the true y_i, being (x_i, y_i) training data samples.

When making predictions over structured output spaces $\mathcal{Y}$, it is often a good idea to use a loss function different from the typical zero-one loss used in classification. The zero-one loss would assign a value of one to every object different from the object it is compared with, no matter how big or small this difference is, and this is probably not the most desired behaviour in structured learning. Let us make this point clear using a particular scenario. Consider the learning phase of a SO-SVM where, as it shall be explained more in depth later in section 3.2, the intermediate prediction results generated by the model are compared to the ground truth. Imagine a sequence of length T is outputted by the predictor. Now, a prediction that only differs from the ground truth

in one element is intuitively much better than one whose T elements are all wrong. Using the zero-one loss these two predictions cannot be ranked differently, both of them would get a loss equal to one as they are not equal to the ground truth. Thus, it is obvious that in structured learning loss functions more involved than the zero-one loss need to be used.

The loss function depends heavily on the application of the structured output framework – recall that we are working in a general setting where $\mathcal{Y}$ can represent a space composed of trees, graphs, graph matchings, or images, among other types of objects. This is, different instances of the loss function can be used depending on the structure of the labels. In order to use the same notation regardless of the application, they are in general denoted by $\Delta : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$. Every possible structured loss must satisfy

$$\Delta(y, y') > 0 \; \forall \; y \neq y', \tag{3.3}$$

$$\Delta(y, y) = 0. \tag{3.4}$$

A typical loss function used for the SO-SVM is the *Hamming* loss (also known as the Hamming distance). In this context, the Hamming loss takes two objects which share the same structure and basically counts the number of variables that have not got the same value in both objects. This loss is widely used due to its broad applicability as it can be computed for different structures, also because it is *decomposable* (later on, the meaning of this property and its advantages will be more apparent) and because it can be implemented easily.

Once the loss function has been introduced, the risk functional – recall from section 2.1 that the risk is defined as the expected value of the loss function – for the structured learning scenario can be specified, see Equation 3.5. Once again, the joint distribution $p_{\mathcal{X},\mathcal{Y}}$ is generally not known so, instead of dealing with Equation 3.5 directly, the empirical risk in Equation 3.6 is minimized.

$$R^{\Delta}(\mathbf{w}) = \int_{\mathcal{X}\times\mathcal{Y}} \Delta(y, f(x; \mathbf{w})) \; p_{\mathcal{X},\mathcal{Y}}(x, y) \; dx \; dy \tag{3.5}$$

$$R^{\Delta}_{emp}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} \Delta(y_i, f(x_i; \mathbf{w})) \tag{3.6}$$

In the next section, the margin concept from the SVM theory is introduced in the structured output learning setting.

3.1.1 *Margin Maximization*

First of all, following the organization in subsection 2.1.1, the linearly separable case where there exists a weight vector $\mathbf{w}$ such that the empirical risk in Equation 3.6 is equal to zero is presented. Based on it, the soft-margin (or non-linearly separable) case is derived.

Assuming that there exists a weight vector $\mathbf{w} \in \mathbb{R}^d$ such that $f(x_i; \mathbf{w}) = y_i$ for all the training data pairs $\{(x_i, y_i)\}_{i=1}^n$, using Equation 3.1 and Equation 3.2, it must be true

$$\max_{y \in \mathcal{Y} \setminus y_i} \mathbf{w}^T \Psi(x_i, y) < \mathbf{w}^T \Psi(x_i, y_i), \quad \forall i = 1, \ldots, n. \tag{3.7}$$

These are in total n – the size of the training data – *non-linear* inequalities. They are non-linear because of the presence of the max operator. However, these non-linear inequalities can be transformed into $n(|\mathcal{Y}| - 1)$ equivalent *linear* inequalities ($|\mathcal{Y}| - 1$ linear for each non-linear inequality),

$$\mathbf{w}^T (\Psi(x_i, y_i) - \Psi(x_i, y)) > 0, \quad \forall\, i = 1, \ldots, n,\ \forall\, y \in \mathcal{Y} \setminus y_i \tag{3.8}$$

As long as the inequalities in Equation 3.8 are all feasible at once, there are several vectors $\mathbf{w}$ solving Equation 3.8 (remember Figure 2.2). Under a margin maximization approach, among all the feasible solutions, the chosen one corresponds to the weight vector which maximizes the so-called margin. In the SO-SVM, the margin can be related to the difference between the score given to y_i and the second highest score given to $\widehat{y}_i = \operatorname{argmax}_{y \in \mathcal{Y} \setminus y_i} \mathbf{w}^T \Psi(x_i, y)$. The aim of margin maximization is to find the $\mathbf{w}^*$ that maximizes this difference for all the examples at the same time. Let us formalize these ideas next.

First, writing the maximum margin objective along with the constraints in Equation 3.8, we have

$$\begin{aligned} &\max_{M, \mathbf{w}} \quad M \\ &\text{subject to } \mathbf{w}^T (\Psi(x_i, y_i) - \Psi(x_i, y)) \geq M, \quad \forall\, i = 1, \ldots, n,\ \forall\, y \in \mathcal{Y} \setminus y_i \\ &\qquad\qquad \|\mathbf{w}\| = 1, \end{aligned} \tag{3.9}$$

where $\mathbf{w}$ is compelled to be a unitary vector as a matter of convention. Secondly, by means of applying equivalences identical to the ones intro-

duced in subsection 2.1.1 to obtain Equation 2.15 from Equation 2.12, the following QP is obtained:

$$\begin{aligned} \min_{\mathbf{w}} \quad & \frac{1}{2}\mathbf{w}^T\mathbf{w} \\ \text{subject to } & \mathbf{w}^T(\Psi(x_i, y_i) - \Psi(x_i, y)) \geq 1, \quad \forall\, i = 1, \ldots, n,\ \forall\, y \in \mathcal{Y} \setminus y_i, \end{aligned} \tag{3.10}$$

which is a hard-margin optimization because it does not allow for errors in the training set. In order to allow for errors in the training set and be able to learn a model from non-linearly separable data, slack variables like the ones used in subsection 2.1.3 have to be introduced,

$$\begin{aligned} \min_{\mathbf{w},\xi} \quad & \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{n}\xi_i \\ \text{s.t. } & \mathbf{w}^T(\Psi(x_i, y_i) - \Psi(x_i, y)) \geq \ \Delta(y_i, y) - \xi_i,\ \forall\, i = 1, \ldots, n,\ \forall\, y \in \mathcal{Y} \setminus y_i, \\ & \xi_i \geq 0, \quad \forall i = 1, \ldots, n. \end{aligned} \tag{3.11}$$

In the constraints above, the margin has been scaled by $\Delta(y_i, y)$ in order to account for different predictions in a more complex fashion than utilizing the zero-one loss.

Note that the standard multiclass SVM by Crammer and Singer (2002) in Equation 2.23 and the SO-SVM in Equation 3.11 are much alike. In principle, one could use a standard SVM or QP solver to address Equation 3.11; if there were not that many constraints, in the order of $\mathcal{O}(n\,|\mathcal{Y}|)$ actually. In section 3.2 an approach, not the only one though, of dealing with these large number of constraints is described.

The problem in Equation 3.11 is written as a constrained QP. It can be transformed into an unconstrained problem substituting in the objective the expression of the slack variables. This is obtained from the first group of constraints in Equation 3.11, writing the slack variables on one side of the inequality and the rest of the terms on the other side. This is,

$$\xi_i \geq \Delta(y_i, y) - \mathbf{w}^T\left(\Psi(x_i, y_i) - \Psi(x_i, y)\right),\ \forall\, i = 1, \ldots, n,\ \forall\, y \in \mathcal{Y} \setminus y_i.$$

The substitution is done in Equation 3.12. This other point of view elucidates the importance of the max or argmax for the SO-SVM, which could seem latent from Equation 3.11 as it is implicit in the constraints. In reality, it is precisely in the computation of this operation where the

major computational burden of the SO-SVM is. Note that Equation 3.12 would be a convex problem if it were not because of the $\max_{y \in \mathcal{Y}}$ part. However, as this space is not convex (it consists of *discrete* objects), the overall problem results non-convex. Notwithstanding this complexity, it is common to regard Equation 3.12 convex as long as there is a "black box" able to compute the max efficiently. The fact that the max can be computed efficiently depends heavily on the structure of $\mathcal{Y}$. In section 3.3 some examples where fast computation is feasible will be presented.

$$\min_{\mathbf{w}} \quad \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{n}\left[\max_{y \in \mathcal{Y}}\Big(\Delta(y_i, y) + \mathbf{w}^T\Psi(x_i, y)\Big) - \mathbf{w}^T\Psi(x_i, y_i)\right] \tag{3.12}$$

At this point, it is worth mentioning the relation between the SO-SVM and another method for structured learning widely used, the Conditional Random Field (CRF) (Lafferty, McCallum, and F. C. Pereira 2001). CRFs have been successfully used to label sequence data, and more recently for image segmentation (Szummer, Kohli, and Hoiem 2008). Parameter estimation in a CRF is done via the following problem (Nowozin and Christoph H Lampert 2011),

$$\min_{\mathbf{w}} \quad \frac{1}{2\sigma^2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{n}\left[\log\sum_{y\mathcal{Y}}\exp\Big(\mathbf{w}^T\Psi(x_i, y) - \mathbf{w}^T\Psi(x_i, y_i)\Big)\right]. \tag{3.13}$$

This resembles Equation 3.12, and it does so even more taking into account that the $\log\sum_y \exp$ combination may be interpreted as sotfmax, i.e. a smooth approximation of the max in Equation 3.12. This connection between the CRF and the SO-SVM suggests that both methods are somewhat similar, even though they are derived from very different strategies; these are maximum likelihood in the CRF case, and margin maximization in the SO-SVM. It also puts forward into consideration the interpretation of the SO-SVM as a method for parameter learning in probabilistic models.

In this section the SO-SVM has been defined, generalizing the binary and multiclass SVM used in classification that were described chapter 2. Three distinct components of the SO-SVM have arisen within its definition: the structured or Δ loss, the joint feature map Ψ, and the argmax, also known as inference solver or separation oracle. Nevertheless, it has not been specified yet how these components

can be computed. This stems from the fact that they depend on structure of $\mathcal{Y}$. A few instances shall be described in section 3.3. Before that, the next section is occupied with the SO-SVM training, solving either Equation 3.11 or Equation 3.12.

3.2 TRAINING

In this section a few strategies to train a SO-SVM solving either Equation 3.11 or Equation 3.12 are described. First, a method that solves the constrained version of the problem using cutting planes is presented. Second, a couple of methods based on subgradient updates that work with the unconstrained formulation are detailed. All these methods use the primal formulation of the problem, but methods leveraging its dual counterpart are also available (Teo et al. 2010).

3.2.1 *Cutting Plane Algorithm*

The main difficulty to solve the SO-SVM optimization in Equation 3.11 is the high number of constraints, which is in the order of $\mathcal{O}(n\,|\mathcal{Y}|)$. In a label sequence learning application for instance, suppose the length of a sequence is $T = 100$ and the labelling is simply binary. The total number of sequences in $\mathcal{Y}$ is equal to 2^{100}. Obviously, it is not possible to solve a QP with that many constraints, actually it is not even possible to hold them in any computer's memory. However, dealing with binary sequences of 100 elements is actually a small task. The structure of $\mathcal{Y}$ has to be exploited so that problems in this setting can be tackled. In particular, note that for each training example i in Equation 3.11 the weight vector sought must satisfy

$$\mathbf{w}^T(\Psi(x_i, y_i) - \Psi(x_i, y)) \geq \Delta(y_i, y) - \xi_i, \quad \forall\, y \in \mathcal{Y} \setminus y_i,$$

thereby if there was a way to find the y^* which maximizes

$$\mathbf{w}^T\Psi(x_i, y) + \Delta(y_i, y) \tag{3.14}$$

and ensure that this expression is less than

$$\mathbf{w}^T\Psi(x_i, y_i) + \xi_i, \tag{3.15}$$

then all the other constraints would be automatically fulfilled, for a fixed vector $\mathbf{w}$. The optimization oracle (introduced in the previous

section) will be instrumental to maximize Equation 3.14, known as the *loss-augmented* inference , prediction or maximization oracle.

The pseudocode in Algorithm 3.1 leverages this. It starts off with an empty set of constraints Γ. On every iteration, it loops through the training examples looking for the *most violated* constraint and adds it to the active set of constraints of that training example Γ_i. Once all the training examples have been studied, the QP is solved including all the constraints in the working sets of active constraints of each example. Thus, this algorithm is doing a *batch update*, going through all the data before the weight vector is modified. This whole process is repeated until there is no variation in the set of active constraints, or a particular number of iterations have been reached, in case it has been specified.

Algorithm 3.1 Cutting plane n-slack SO-SVM training with margin scaling

Require: Regularization $C > 0$, training data $\{(x_i, y_i)\}_{i=1}^{n}$ *iid*

$\Gamma_i \leftarrow \emptyset \quad \forall\, i = 1, \ldots, n$
repeat
  for $i = 1 \rightarrow n$ **do**
    $y_i^* \leftarrow \operatorname{argmax}_{y \in \mathcal{Y}} \mathbf{w}^T \Psi(x_i, y) + \Delta(y_i, y)$
    if $\Gamma_i = \emptyset$ **then**
      $\Gamma_i \leftarrow \Gamma_i \cup y_i^*$
    else
      if $\mathbf{w}^T \Psi(x_i, y_i^*) + \Delta(y_i, y_i^*) \; > \; \max_{y \in \Gamma_i} \mathbf{w}^T \Psi(x_i, y) + \Delta(y_i, y)$ **then**
        $\Gamma_i \leftarrow \Gamma_i \cup y_i^*$
      end if
    end if
  end for
  $\mathbf{w} \leftarrow \operatorname{argmin}_{\mathbf{w}} \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{n} \xi_i$
    s.t. $\mathbf{w}^T(\Psi(x_i, y_i) - \Psi(x_i, y_j)) \geq \; \Delta(y_i, y_j) - \xi_i,$
    $\forall\, i = 1, \ldots, n, \forall\, y_j \in \Gamma_i$
until no Γ_i has been updated during the last iteration

One of the interesting properties of Algorithm 3.1 is that it is proven to converge to the global optimum (provided that the optimization oracle is solved exactly) in a polynomial number of iterations, even if $\mathcal{Y}$ has an exponential number of elements or it is unbounded. Such proof is out of the scope of this report, a thorough analysis can be found

in (Tsochantaridis et al. 2005). It is necessary that the maximization performed at line 4 in Algorithm 3.1 is solved exactly so that the method finds the optimal hyperplane. However, it is still possible to get good results using strategies for approximate inference. The impact of using approximate methods has been studied in the work by Finley and Joachims (2008); Domke (2013). It is nevertheless more important that the inference is solved efficiently. Obviously, if the argmax is not solved efficiently it does not really make a difference that the algorithm converges in a polynomial number of iterations.

In this aspect, the choice of the loss function plays an important role. Once the optimization oracle $\operatorname{argmax}_{y \in \mathcal{Y}} \mathbf{w}^T \Psi(x_i, y)$ can be solved efficiently, the Δ loss must be decomposable so that the loss-augmented oracle $\operatorname{argmax}_{y \in \mathcal{Y}} \mathbf{w}^T \Psi(x_i, y) + \Delta(y_i, y)$ can be solved efficiently as well. A loss that is decomposable can be divided into individual contributions, which can be integrated into the joint feature map $\Psi(x_i, y)$ so that the loss-augmented argmax does not become any harder (Pletscher and Kohli 2012).

3.2.2 *Subgradient Descent*

Apart from the cutting plane optimization algorithm described in the previous section and other methods based on cutting planes, there are other approaches to train the SO-SVM. Note that all the terms in the summation in Equation 3.12 are convex functions in $\mathbf{w}$ [1]. Therefore, standard convex optimization methods such as those based on gradient descent updates can be applied. Nonetheless, Equation 3.12 is not differentiable because of the presence of the max function. The non-differentiability of the SO-SVM objective is illustrated in Figure 3.1. To bypass this, gradients shall be replaced by *subgradients* (Antoniou and Lu 2007). Formally, $v \in \mathbb{R}^D$ is a subgradient of the function $f : \mathbb{R}^{\mathbb{D}} \rightarrow \mathbb{R}$ at $\mathbf{w_0}$ if

$$f(\mathbf{w}) \geq f(\mathbf{w_0}) + \mathbf{v}^T (\mathbf{w} - \mathbf{w_0}) \quad \forall\, \mathbf{w} \in \mathbb{R}^D. \tag{3.16}$$

This definition is a bit more general than the gradient and, when f is actually differentiable, the gradient of f is actually its only subgradient.

1 Recall that it is the space $\mathcal{Y}$ what makes the whole SO-SVM training problem hard since it is not a convex space. It is not convex because it is formed by non-continuous elements (e.g. graphs whose nodes may take one out of two values) and combinatorial spaces of this type are not convex.

The standard subgradient descent minimization for SO-SVM training is shown in Algorithm 3.2. In this algorithm, it has been used that for a function

$$f(\mathbf{x}) = \max_{\mathbf{x}} \left[f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_r(\mathbf{x}) \right],$$

where the functions $f_i(\mathbf{x}) \; \forall \; i \in [1, r]$ are convex; at any point $\mathbf{x}$ there is (at least) one i such that $f(\mathbf{x}) = f_i(\mathbf{x})$, and the subgradient of f_i is also a subgradient of f at $\mathbf{x}$ (Antoniou and Lu 2007).

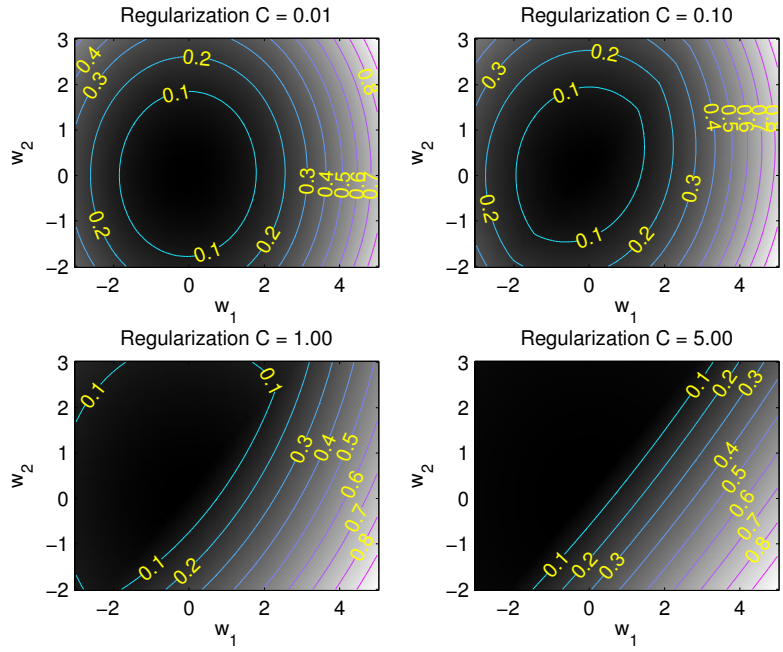


Figure 3.1.: Structured Output Support Vector Machine objective function with input space $\mathcal{X} = \mathbb{R}$ and binary labels, i.e. $\mathcal{Y} = \{-1, +1\}$. The training set is composed of the following (x_i, y_i) pairs: $\{(-10, +1), (-4, +1), (6, -1), (5, -1)\}$. The joint feature map is $\Psi(x, -1) = (x, 0)$ and $\Psi(x, +1) = (0, x)$. The objective is shown for four different values of the regularization strength C. With larger regularization values, the empirical loss in the training set becomes more important and its non-differentiability appears more visible.

One important drawback of the subgradient minimization is that it requires n, the number of training examples, argmax computations

Algorithm 3.2 Subgradient descent SO-SVM training

Require: Regularization C, maximum number of iterations T, learning rate η

$\mathbf{w}_{cur} \leftarrow \mathbf{0}$
for $t = 1 \rightarrow T$ **do**
  for $i = 1 \rightarrow n$ **do**
    $\hat{y}_i \leftarrow \text{argmax}_{y \in \mathcal{Y}} \Delta(y_i, y) + \mathbf{w}^T \Psi(x_i, y)$
  end for
  $\mathbf{v} \leftarrow \mathbf{w}_{cur} + C \sum_{i=1}^{n} [\Psi(x_i, y_i) - \Psi(x_i, \hat{y}_i)]$
  $\mathbf{w}_{cur} \leftarrow \mathbf{w}_{cur} - \eta \mathbf{v}$
end for
$\mathbf{w}^* \leftarrow \mathbf{w}_{cur}$

(loss-augmented prediction actually) in order to update the weight vector once. A naive idea to improve this facet consists of subsampling the data in the step of computing the gradient, and only compute descent directions with sampled batches. When only one randomly drawn example is used for every update, the method receives the name of *stochastic subgradient descent*. Pseudocode for this algorithm is shown in Algorithm 3.3. With the stochastic updates the total number of argmax computations is considerably reduced.

Algorithm 3.3 Stochastic subgradient descent SO-SVM training

Require: Regularization C, maximum number of iterations T, learning rate η

$\mathbf{w}_{cur} \leftarrow \mathbf{0}$
for $t = 1 \rightarrow T$ **do**
  $(x_i, y_i) \leftarrow$ randomly chosen training pair
  $\hat{y} \leftarrow \text{argmax}_{y \in \mathcal{Y}} \Delta(y_i, y) + \mathbf{w}^T \Psi(x_i, y)$
  $\mathbf{v} \leftarrow \mathbf{w}_{cur} + C(\Psi(x_i, y_i) - \Psi(x_i, \hat{y}_i))$
  $\mathbf{w}_{cur} \leftarrow \mathbf{w}_{cur} - \eta \mathbf{v}$
end for
$\mathbf{w}^* \leftarrow \mathbf{w}_{cur}$

All in all, on the one side, subgradient methods are attractive because they are easy to implement and understand. On the other side, it may be difficult to tune the learning rate. A value too small would involve a very large number of iterations until convergence is reached,

while a too large value may lead to oscillatory behaviour and non-convergence. There are other online methods that automatically search for the learning rate (Lacoste-Julien et al. 2013). Moreover, subgradient based methods have low convergence rate, it is equal to $\mathcal{O}(\sqrt{\epsilon})$. In other words, it takes $\mathcal{O}\left(1/\epsilon^2\right)$ iterations to get $\mathbf{w}_{cur}$ closer to $\mathbf{w}^*$ by ϵ. Bundle methods (Teo et al. 2010) improve this rate to $\mathcal{O}\left(1/\epsilon\right)$.

3.3 STRUCTURED MODELS

In this section a few models or applications of the SO-SVM are described. This involves specifying how the joint feature mapping $\Psi : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^d$ and the optimization oracle argmax : $\mathcal{X} \times \mathbb{R}^d \to \mathcal{Y}$ are computed in these applications.

3.3.1 *Label Sequence Learning*

The goal in label sequence learning is to learn a function capable of predicting a sequence of labels from a sequence of observations. This is different from making predictions for individual class labels, as in multiclass classification, because interactions among the variables that form the sequence are taken into account.

Formally let y be a sequence of T labels $l_1, l_2, \ldots, l_T = l_{1:T}$ and x be the sequence of observations associated with y. Each of the labels may take a value in $\mathcal{L}$ and the total number of possible label values is denoted by $|\mathcal{L}|$. Following the HMM structure, the basic model consists of unary and pairwise terms. The unary terms model the dependencies among observations and states in a HMM, whereas the pairwise terms specify the relationships between neighbour states. The joint feature mapping assembles the sufficient statistics and it is

$$\Psi(x, y) = \sum_{i=1}^{T} \Psi_u(x_i, y_i) + \sum_{i=1}^{T-1} \Psi_p(y_i, y_{i+1}). \tag{3.17}$$

The operator $+$ between unary and pairwise contributions in Equation 3.17 is not the usual vector addition operator. Note that Ψ_u and Ψ_p need not have the same dimension, thus vector addition between them does not have any sense. The $+$ here denotes concatenation – it stacks one vector on top of the other – so that the unary and pairwise contributions are not combined together. This could also be done by

extending Ψ_u and Ψ_p with elements equal to zero. In that case, $+$ would actually denote vector addition.

The unary sufficient statistics Ψ_u are

$$\Psi_u(x_i, y_i) = \begin{bmatrix} \mathbb{I}_1(y_i)\ \phi(x_i) \\ \mathbb{I}_2(y_i)\ \phi(x_i) \\ \vdots \\ \mathbb{I}_{|\mathcal{L}|}(y_i)\ \phi(x_i) \end{bmatrix}, \tag{3.18}$$

where the function ϕ maps an observation to a vector and it appears to make explicit the case where observations are objects other than vectors or, even if they are vectors, a feature extractor is used. $\mathbb{I}_{\texttt{expr}}(x)$ denotes the indicator function introduced in chapter 2. On the other side, the pairwise sufficient statistics have the form

$$\Psi_p(y_i, y_j) = \begin{bmatrix} \mathbb{I}_1(y_i)\ \mathbb{I}_1(y_j) \\ \mathbb{I}_1(y_i)\ \mathbb{I}_2(y_j) \\ \vdots \\ \mathbb{I}_1(y_i)\ \mathbb{I}_{|\mathcal{L}|}(y_j) \\ \mathbb{I}_2(y_i)\ \mathbb{I}_1(y_j) \\ \vdots \\ \mathbb{I}_2(y_i)\ \mathbb{I}_{|\mathcal{L}|}(y_j) \\ \vdots \\ \mathbb{I}_{|\mathcal{L}|}(y_i)\ \mathbb{I}_1(y_j) \\ \vdots \\ \mathbb{I}_{|\mathcal{L}|}(y_i)\ \mathbb{I}_{|\mathcal{L}|}(y_j) \end{bmatrix}. \tag{3.19}$$

In the SO-SVM training algorithm the computation of the maximizer $y^* \in \mathcal{Y}$,

$$y^* = \operatorname*{argmax}_{y \in \mathcal{Y}} \mathbf{w}^T \Psi(x_i, y) + \Delta(y_i, y),$$

is an important step of the training process. This same operation without the loss term is also performed at prediction time. In label sequence learning, y^* corresponds to the most probable sequence of states in a HMM. If the feature vector x_i has duration T and the state model has $|\mathcal{S}|$ states, then there are $|\mathcal{S}|^T$ possible state sequences in total that could have produced the observations. Obviously, searching among these $|\mathcal{S}|^T$ is prohibitive even for not too large values of T. Fortunately,

this problem is essentially the same as the one of finding the most probable or Maximum-A-Posteriori (MAP) hidden state sequence given an observation sequence and a first-order HMM.

The algorithm that solves this task in polynomial time (in the length T of the sequence) is the Viterbi algorithm and it is an instance of dynamic programming. The key idea behind the algorithm consists in relying on recursion and organizing the computations for $t \in [0, T]$ so that they can be built upon previous computations. Let us explain further how this is achieved in the Viterbi algorithm. First of all, note that at any time t there are exactly $|\mathcal{S}|$ values that the random variable S_t may take. Moreover, the optimal value state sequence *must* pass through one, and only one, of these possible states. In other words, at each time step $t \in [0, T]$ there are actually only $|\mathcal{S}|$ candidates to be part of the solution state sequence. This fact can be exploited to maximize separately partial sequences ending at time $t \leq T$.

Intuitively, keeping in mind the Markov property for both states and observations of the HMM, it seems reasonable that in order to find the optimal state sequence up to time step t only two aspects need be accounted for:

1. The partial most probable sequences that end in every possible state at time $t - 1$.
2. Make the state transition that leads to the maximum increment in the objective function.

Note that the first point is actually achieved solving the original problem, but only taken into account the first $t - 1$ time steps. This is a common feature of dynamic programming; a problem is tackled using a bottom-up approach, solving equivalent subproblems incrementally. This reasoning, which is at the core of the Viterbi algorithm, is formalized next.

Formally, finding the MAP hidden state sequence is

$$\hat{s}_{0:T} = \underset{s_{0:T}}{\operatorname{argmax}} P\left[s_{0:T} | x_{0:T}\right]. \tag{3.20}$$

Using Bayes rule this can be rewritten as

$$\hat{s}_{0:T} = \underset{s_{0:T}}{\operatorname{argmax}} \frac{P\left[s_{0:T}, x_{0:T}\right]}{P\left[x_{0:T}\right]} = \underset{s_{0:T}}{\operatorname{argmax}} P\left[s_{0:T}, x_{0:T}\right].$$

The objective can be written recursively as follows,

$$\begin{aligned} P\left[s_{0:T}, x_{0:T}\right] &= P\left[s_{t+1:T}, x_{t+1:T} | s_{0:t}, x_{0:t}\right] P\left[s_{0:t}, x_{0:t}\right] = \\ &= P\left[s_{t+1:T}, x_{t+1:T} | s_t\right] P\left[s_{0:t}, x_{0:t}\right]. \end{aligned}$$

Maximizing over all the possible hidden state sequences:

$$\max_{s_{0:T}} P\left[s_{0:T}, x_{0:T}\right] = \max_{s_t} \left(\max_{s_{0:t-1}} \left(P\left[s_{0:t}, x_{0:t}\right]\right) \max_{s_{t+1:T}} \left(P\left[s_{t+1:T}, x_{t+1:T} | s_t\right]\right) \right). \tag{3.21}$$

Both inner maximizations in Equation 3.21 are functions only of s_t, once the maximizations over all possible partial state sequences $s_{0:t-1}$ and $s_{t+1:T}$ are performed in the first and second inner max, respectively. Therefore, only one candidate for each $s_t \in \mathcal{S}$ needs be stored to compute the MAP hidden state sequence. This concept is illustrated in Figure 3.2.

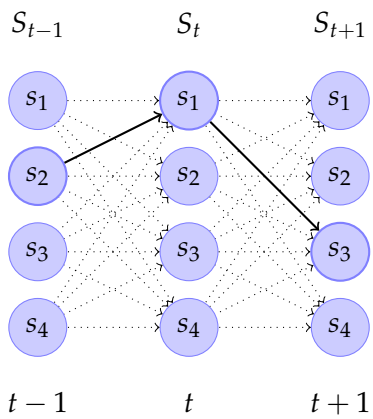


Figure 3.2.: Trellis diagram showing the central idea of the Viterbi algorithm. The horizontal axis is for the time and the vertical for the different values that a state random variable can take in every time step. For simplicity, the Markov chain has only four states. There is an exponential number of partial state sequences $s_{0:t}$, however it is enough to only take into account the best one to obtain the complete Maximum-A-Posteriori sequence of hidden states. The candidates that need to be remembered are indicated by solid lines, while all the possible transitions by dotted lines.

The algorithm computes the part

$$\max_{s_{0:t-1}} P\left[s_{0:t}, x_{0:t}\right]$$

recursively, for incrementing values of t. At $t = T$ the second inner maximization in Equation 3.21 is empty and

$$\max_{s_{0:T}} P\left[s_{0:T}, x_{0:T}\right]$$

is solved. Note that it is the max and not the argmax, which is our goal, what is obtained so far. In addition, a traceback matrix with pointers to previous states in the sequence must be saved so that

$$\operatorname*{argmax}_{s_{0:T}} P\left[s_{0:T}, x_{0:T}\right]$$

can be recovered from

$$\max_{s_{0:T}} P\left[s_{0:T}, x_{0:T}\right].$$

To summarize, the Viterbi algorithm computes these structures:

1. The probability vector χ, which has $|\mathcal{S}|$ elements. There is one element in χ for every possible value each state S_t may take. We denote the element in χ that corresponds to the value s_t taken by S_t as χ_{s_t}. This is,

$$\chi_{s_t} = \max_{s_{0:t-1}} P\left[s_{0:t-1}, s_t, x_{0:t}\right]. \tag{3.22}$$

2. The backpointer matrix ζ, which has dimensions equal to $|\mathcal{S}| \times T$. The element $\zeta_{s_t,t}$ of this matrix denotes the value s_{t-1} of S_{t-1} such that the partial MAP sequence $s_{0:t}$ ends in $S_t = s_t$. Formally,

$$\zeta_{s_t,t} = \operatorname*{argmax}_{s_{t-1}\in\mathcal{S}} a_{s_{t-1}s_t}\, \chi_{s_{t-1}}. \tag{3.23}$$

The computation of the vector χ in Equation 3.22 at time t can be expressed in function of the same vector χ' computed one time step earlier,

$$\begin{aligned}
\chi_{s_t} = \max_{s_{0:t-1}\in\mathcal{S}^t} P\left[s_{0:t}, x_{0:t}\right] = \max_{s_{0:t-1}\in\mathcal{S}^t} \left(P\left[s_{0:t-1,x_{0:t-1}}\right] P\left[s_t, x_t|s_{t-1}\right]\right) \quad = \\
= P\left[x_t|s_t\right] \max_{s_{t-1}\in\mathcal{S}} \left(P\left[s_t|s_{t-1}\right] \max_{s_{0:t-2}\in\mathcal{S}^{t-1}} P\left[s_{0:t-1,x_{0:t-1}}\right]\right) \quad = \\
= P\left[x_t|s_t\right] \max_{s_{t-1}\in\mathcal{S}} \left(P\left[s_t|s_{t-1}\right] \chi'_{s_{t-1}}\right),
\end{aligned}$$

which is precisely,

$$\chi_{s_t} = b_{s_t}(x_t) \max_{s_{t-1} \in \mathcal{S}} \left(a_{s_{t-1} s_t} \chi'_{s_{t-1}} \right). \tag{3.24}$$

Finally, pseudocode for the Viterbi algorithm is presented in Algorithm 3.4. The time complexity of the algorithm is $\mathcal{O}\left(T |\mathcal{S}|^2\right)$, this is linear in the length of the sequence although quadratic with the number of states.

Algorithm 3.4 Viterbi algorithm

Require: A HMM (q, A, b), a sequence of observations $x_{0:T}$

for $s_0 \in \mathcal{S}$ **do**
  $\chi_{s_0} \leftarrow q_{s_0} b_{s_0}(x_0)$
end for
for $t = 1 \to T$ **do**
  $\chi' \leftarrow \chi$
  for $s_t \in \mathcal{S}$ **do**
    $\zeta_{s_t, t} \leftarrow \operatorname{argmax}_{s_{t-1} \in \mathcal{S}} \left(a_{s_{t-1} s_t} \chi'_{s_{t-1}} \right)$
    $\chi_{s_t} \leftarrow b_{s_t}(x_t) \max_{s_{t-1} \in \mathcal{S}} \left(a_{s_{t-1} s_t} \chi'_{s_{t-1}} \right)$
  end for
end for
$\hat{s}_T \leftarrow \operatorname{argmax}_{s_T \in \mathcal{S}} \chi_{s_T}$
for $t = T - 1 \to 0$ **do**
  $\hat{s}_t \leftarrow \zeta_{\hat{s}_{t+1}, t+1}$
end for

3.3.2 *Graph Labelling*

In graph labelling the objects in $\mathcal{Y}$ are formed by several variables that have dependency relations among each other. Recall that this is also the case of label sequence learning. However, in label sequence learning the dependencies are expressed in the form of a *chain* graph. Graph labelling generalizes this aspect and the graph that represents the variables with its dependencies may have an arbitrary shape. Thus, similar to label sequence learning, the variables of the output $y \in \mathcal{Y}$ correspond to the nodes in a graph, while the edges of the graph represent their dependencies. Formally, one variable is conditionally independent of the rest of the variables of the graph given

its neighbours; two nodes are neighbours if there is an edge in the graph connecting them (Koller and N. Friedman 2009).

The goal of the predictor is to assign a class variable to each of the nodes of the graph. Of course, the assignments are decided jointly, in contrast with deciding each vertex variable independently. As usual, the predictor is trained using labelled data $\{(x_i, y_i)\}_{i=1}^{n}$ and it is later used to map unseen inputs $x \in \mathcal{X}$ to outputs $y \in \mathcal{Y}$. Both input and output objects are, in general, graphs.

The joint feature mapping in this model is very similar to the mapping used in label sequence learning in Equation 3.17, in particular it is

$$\Psi(x, y) = \sum_{i \in \mathcal{V}} \Psi_u(x_i, y_i) + \sum_{(i,j) \in \mathcal{E}} \Psi_p(y_i, y_j), \tag{3.25}$$

where $\mathcal{V}$ and $\mathcal{E}$ denote the set of nodes and the set of edges of the graph y, respectively. Ψ_u and Ψ_p are one more time the unary and pairwise terms and they are equivalent to the ones used in label sequence learning in Equation 3.18 and Equation 3.19. They account for properties of the vertex variables and interactions between one-hop neighbours, respectively. The most straightforward structured loss function commonly used in graph labelling is the Hamming loss.

Using general structured output models (i.e. general network-graphs) leads to an argmax function which cannot be solved efficiently. This problem is in fact equivalent to inference on general probabilistic graphical models, which is a well-known NP-hard problem (Koller and N. Friedman 2009). A workaround is either to identify a possible subset of problems for which efficient inference algorithms do exist, or to use approximate inference algorithms. Label sequence learning may indeed be seen as an instance of a subset of problems for which efficient inference is possible. In this case via Viterbi decoding as previously seen. The expense comes at the cost of not being able to tackle general structured output models, but only chain graphs. Another example where a restriction to simpler models leads to efficient inference is the case of tree structures. Inference is solved in this case via the Cocke-Kasami-Younger (CKY) parser. These are all instances of a more general algorithm called (Loopy) Belief Propagation (BP) to solve inference on graphical models (Klein, Brefeld, and Scheffer 2008).

The literature about algorithms for approximate inference and simplifications leading to tractable inference is immense. The work by Boykov (2001); Finley and Joachims (2007); Finley and Joachims

(2008); Szummer, Kohli, and Hoiem (2008) are only a few examples of these algorithms used together with the SO-SVM. In this report we will present a method to solve the argmax for general structured output models based on a Linear Program (LP) relaxation. Implementation of alternative methods, as well as design of our own approximations are left as future work (see chapter 5 for additional details).

The crux of the LP relaxation is easy: use binary encoding of the label values of each node and minimize the configuration of label values for all the nodes jointly. With binary encoding it is meant that for each node a vector with as many elements as possible values the node may take is used. Only one of the elements in the vector is allowed to be non-zero. This element is equal to one and denotes the label value taken by the selected node. The Integer Linear Program (ILP) that achieves this goal is

$$\min_{\mu} \sum_{i \in \mathcal{V}} \sum_{y_i \in \mathcal{Y}_i} \theta_i(y_i)\mu_i(y_i) + \sum_{\substack{\{i,j\} \\ \in E}} \sum_{\substack{(y_i,y_j) \in \\ \mathcal{Y}_i \times \mathcal{Y}_j}} \theta_{i,j}(y_i,y_j)\mu_{i,j}(y_i,y_j) \tag{3.26}$$

$$\text{subject to} \sum_{y_i \in \mathcal{Y}_i} \mu_i(y_i) = 1, \quad \forall\, i \in V \tag{3.27}$$

$$\sum_{y_j \in \mathcal{Y}_j} \mu_{i,j}(y_i,y_j) = \mu_i(y_i), \quad \forall\, \{i,j\} \in \mathcal{E}, \forall\, y_i \in \mathcal{Y}_i \tag{3.28}$$

$$\mu_i(y_i) \in \{0,1\}, \quad \forall\, i \in \mathcal{V}, \forall\, y_i \in \mathcal{Y}_i \tag{3.29}$$

$$\mu_{i,j}(y_i,y_j) \in \{0,1\}, \quad \forall \{i,j\} \in \mathcal{E}, \forall\, (y_i,y_j) \in \mathcal{Y}_i \times \mathcal{Y}_j, \tag{3.30}$$

where $\mu_i(y_i)$ denotes the binary encoding of node i taking value y_i. The pairwise terms $\mu_{i,j}(y_i,y_j)$ are introduced to account for the variable dependencies in the argmax computations although their values are fixed by the values of their corresponding unaries $\mu_i(y_i)$ and $\mu_j(y_j)$, as it can be observed from the second restriction above. The terms $\theta_i(y_i)$ are computed using the observations and the model parameters corresponding to the unaries in the weight vector. The terms $\theta_{i_j}(y_i,y_j)$ are computed using only the model parameters associated with pairwise interactions in the weight vector. In the computer vision literature θ_i and $\theta_{i,j}$ are typically referred to as *unary and pairwise potentials*, respectively. The objective is often called *energy*, which is why in computer vision the inference problem in graphical models is generally known as energy minimization. The energy is equal to

$-\mathbf{w}^T\Psi(x_i, y)$ for the loss-augmented prediction used in training), thus the minimization above is indeed the argmax.

By dropping the constraints in Equation 3.29 and Equation 3.30, the ILP becomes a LP, which can be solved efficiently by general purpose optimization solvers. More important, in case the graph has got no loops (i.e., it is a tree) or the pairwise interactions are all submodular, the relaxation is tight and the solution will be integral (Nowozin and Christoph H. Lampert 2011). See Pletscher and Kohli (2012) for a discussion about submodularity. In other cases where the relaxation is not tight, a final threshold can be applied to obtain discrete label values.

4

EXPERIMENTS AND RESULTS

This chapter describes the process followed to evaluate the SO-SVM performance and further study its characteristics both qualitatively and quantitatively. The experiments are instances of label sequence learning and graph labelling tasks. The SO-SVM models used are the ones described in subsection 3.3.1 and in subsection 3.3.2. Throughout this chapter we will refer to the SO-SVM used in label sequence learning as Hidden Markov Support Vector Machine (HM-SVM). The HMM introduced in section 2.2 is used in the experiments for comparison.

The present chapter is divided as follows. In section 4.1 and section 4.2 label sequence learning experiments with synthetic data are presented. In section 4.1 a HMM is explicitly specified to generate data from its model parameters, while in section 4.2 data are generated without obeying a HMM structure. Finally, graph labelling experiments based on a simplification of the segmentation problem in computer vision are analysed in section 4.3.

The software for the experiments is implemented in C++, Python and Matlab. In addition, we have used the Structured Output learning software framework of the Shogun Machine Learning Toolbox (Sonnenburg et al. 2010). This framework was implemented by me during Google Summer of Code 2012 [1].

As an important side note, the regularization in the SO-SVM learning stage in Equation 3.11 is not the same as the one used in the implementation of the cutting plane algorithm used in the label sequence learning tasks. In the implementation in Shogun the regularization is multiplying the term with the norm of the weight vector rather than the term with the sum of the slacks. Fortunately, this does not entail any big difference and it just have to be kept in mind that the regularization used throughout the two first sections of this chapter would actually be $1/C$ in Equation 3.11.

1 Link to project abstract.

4.1 SIMULATED HMM DATA

First of all, a Markov chain with two states is specified. The fixed, or time-invariant, transition probabilities are randomly generated so that the probability of remaining in the same state is considerably larger than the probability of moving to the other state. This is usually the case in real-world applications where HMMs are used, such as Automatic Speech Recognition (ASR) (Rabiner 1989). The state model given by this Markov chain is represented in Figure 4.1. The start and stop states are included to model differently the beginning and end from the remaining part of the sequences. The distribution of the observations is Gaussian, centered on a value given by the state and with constant variances randomly drawn from a specified interval. Sample data generated by this type of HMM is shown in Figure 4.2.

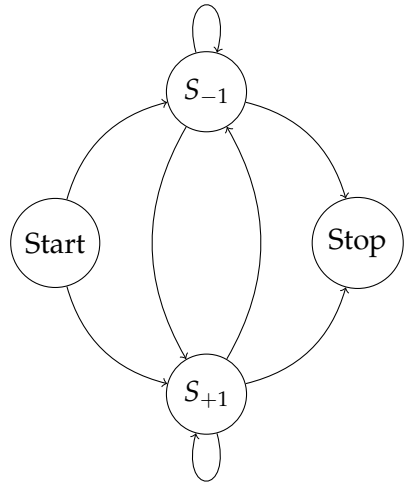


Figure 4.1.: Binary state model used in experiments.

Three predictors are used in this experiment. The first one is simply the ground truth model that will be referred to as GT-HMM from now on. Obviously, this model does not require any training phase and prediction consists basically of the Viterbi algorithm.

The second predictor is a HMM whose parameters are estimated from training data in a maximum likelihood fashion. This will be referred to as ML-HMM. The parameters of this model are estimated as follows:

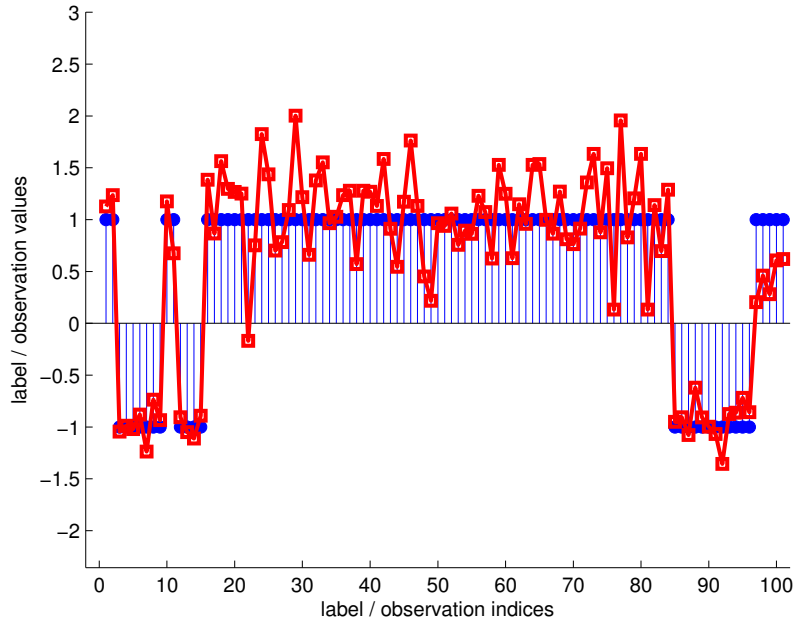


Figure 4.2.: Segment of a sample sequence from dataset #1. The observations are normally distributed, centered on the state value ($+1$ for the positive state and -1 for the negative) and with standard deviation between 0 and 1.

- The transition probabilities are obtained using frequency counts. For example, in order to estimate A_{ij} the number of transitions from s_i to s_j are divided by the total number of transitions from s_i. This is in fact the interpretation of the transition probabilities.

- The observation scores are calculated differently though [2]. A non-parametric model based on Piece-wise Linear Functions (PLiFs) is used to model the observations (Zeller 2010). PLiFs are used to deal with continuous data without forcing a fixed model (e.g. a Gaussian distribution, or a mixture of Gaussians), using a small number of parameters. We say non-parametric because no parameters from a distribution such as mean or variance are

2 Note how the term scores starts to be used here instead of probability or distribution when talking about the observations. Due to the model used, saying probabilities might lead to confussion since they cannot be interpreted as such any more.

directly estimated from data. A PLiF is composed of two parts: the supporting nodes or values in the abscissa, and the scores associated with each supporting node.

- The supporting nodes are chosen using only the observations in the training set. Every pair of adjacent supporting nodes delimit an interval between them. The supporting nodes are chosen so that roughly the same number of observations in the training set fall into each of these intervals.

- The scores are estimated using an approach similar to how the transition probabilities are estimated using frequency terms during training. Let us first explain how the PLiFs are used. Assume a PLiF with its supporting nodes and scores is specified. The score given to an observation that is exactly equal to one of the supporting nodes is exactly the score associated with that node. When an observation falls between two nodes, its score is computed by linear interpolation of the closest nodes' scores. Mathematically, let s denote the supporting nodes, v scores and x an observation. Let also P be the number of supporting points. The supporting nodes are sorted so that $s_p < s_{p+1} \; \forall \; p \in [1, P]$. Then, the PLiF is defined as shown in Equation 4.1.

$$f(x) = \begin{cases} v_1 & \text{if} \quad x < s_1 \\ \frac{v_p\left(s_{p+1}-x\right) \; + \; v_{p+1}\left(x-s_p\right)}{s_{p+1}-s_p} & \text{if} \quad s_p \leq x < s_{p+1} \\ v_P & \text{if} \quad x \geq s_P \end{cases} \tag{4.1}$$

  Let us now explain how PLiF scores are learned from data. The crux is to count how often each parameter has been used for decoding, in a similar way to how it is done with the transition probabilities as indicated above. Since the PLiF nodes are discrete and the observations are continuous, the counts are split between the nodes adjacent to the observation, proportionally to how close the observed value is to the mentioned supporting nodes. Let $c_p(x)$ denote the contribution given by observation x to the total count

of supporting node s_p. Then, $c_p(x)$ is computed as shown in Equation 4.2.

$$c_p(x) = \begin{cases} 0 & \text{if} \quad x \leq s_{p-1} \quad \vee \quad x \geq s_{p+1} \\ 1 & \text{if} \quad p = 1 \quad \wedge \quad x \leq s_1 \\ \frac{x - s_{p-1}}{s_p - s_{p-1}} & \text{if} \quad 1 < p < P \quad \wedge \quad s_{p-1} < x \leq s_p \\ \frac{s_{p+1} - x}{s_{p+1} - s_p} & \text{if} \quad 1 < p < P \quad \wedge \quad s_p < x < s_{p+1} \\ 1 & \text{if} \quad p = P \quad \wedge \quad x \geq s_P \end{cases} \tag{4.2}$$

The third and last predictor is a HM-SVM. Training is performed solving the problem in Equation 3.11 with the cutting plane algorithm described in subsection 3.2.1, where the joint feature mapping Ψ is the one defined in subsection 3.3.1 and the argmax required by the training algorithm is solved using Viterbi. The regularization value C is selected via cross-validation. Once the best regularization value is obtained as described in section 2.3, the model is trained using the whole training set and evaluated in the test data. Likewise the ML-HMM, the observations in the HM-SVM are modelled using PLiFs.

Table 4.1.: HM-SVM cross-validation accuracy in the Gaussian HMM dataset #1.

Regularization strength	500	50	5	0.5	0.05
Accuracy [%]	99.648	99.648	99.648	99.648	99.648

The first dataset (dataset #1) is formed by 200 examples of length 250 each. The observations are 10-dimensional, Gaussian distributed with mean centered on $+1$ for observations generated from positive states and on -1 for negative ones; the variances are uniformly generated from the interval $[0, 1]$ for every feature-state combination (so they are different both among features and states). The dataset is divided into training and test sets, each with with 100 examples. For the HM-SVM, the best value of the regularization parameter is chosen performing 5-fold cross-validation with 20 examples in each fold. The results of cross-validation appear in Table 4.1. The cross-validation accuracy is the same for all the choices of regularization, which means the pattern is rather easy to learn from the data used. Test accuracy and error for the three predictors described above appear in Table 4.2. All of them share roughly the same performance, which serves to validate the SO-SVM approach.

Table 4.2.: Test accuracy and error in the Gaussian HMM dataset #1.

	Test Accuracy [%]	Test Error [%]
GT-HMM	99.632	0.368
ML-HMM	99.628	0.372
HM-SVM	99.632	0.368

4.2 ARTIFICIAL DATA

The two models used in this section are the ML-HMM and the HM-SVM. The GT-HMM shall not be used since no HMM is specified for the data generated in this section. Both the ML-HMM and HM-SVM used for the experiments here are identical to those described in the previous section. Therefore, we will first describe how the data generation is performed and later pass directly to the analysis of the results.

Similar to section 4.1, the data in this experiment belongs to one out of two states, so that the same state model in Figure 4.1 still holds. To generate the sequences of labels, there are two magnitudes of interest: the minimum (and maximum) *number of blocks* and the minimum (and maximum) *block length*. For each label sequence to generate, all the elements are initialized to -1. Then, a random number of blocks between the minimum and maximum number of blocks is chosen. For each of these blocks, a random length between the minimum and maximum block length is drawn as well as a random start position for the block within the sequence. All the elements within a block are set to $+1$. Once the procedure to generate the labels has been described, we move to explain how the corresponding observations are generated. For each feature, a *distortion proportion* of all the labels are flipped to the other state value, i.e. negative labels are substituted with positive labels and vice versa. Lastly, AWGN centered on the distorted labels is also introduced. The proportion of distorted labels can be modified to study the robustness against noise of the approaches. The distortion in the labels has been introduced to simulate flawed ground truth labelling. This is a typical problem in real-world supervised learning tasks. In many situations the data is labelled manually by human beings and it is often hard to avoid a few errors in the labelling process. In addition, to make the prediction task harder, a few features are completely replaced with white noise. This is used to represent data with features that do

not give any information about the state. The Matlab code used for data generation is given in section B.2.

As in section 4.1, the complete data set is formed by 200 sequences of length 250 and with features in $\mathbb{R}^{10}$. There are two possible states, identified with the values $+1$ and -1, and the observations are Gaussian distributed with standard deviation equal to 4 for every state-feature combination. These observations are also affected by the flipping of the label values as described in the paragraph above.

For the HM-SVM, the best regularization strength is found again with 5-fold cross-validation. This model selection process is repeated for every value of the proportion of label distortion introduced. The cross-validation accuracy results as a function of the regularization and the proportion of distortion appear in Table 4.3 and in Figure 4.3. The overall conclusion is that a higher regularization value performs worse because it limits the weight vector considerably by punishing high values of its elements with high cost. Lower regularization values perform generally better, at the cost of a notably time consuming training process as it can be verified in Table 4.4. The results shown in the tables are obtained by taking the average of the results in each fold. To get an idea of performance variation within the same fold for different models, see Figure 4.4.

Table 4.3.: Cross-validation accuracy of the HM-SVM as a function of the proportion of label distortion and regularization strength. The results are also graphically represented and described in Figure 4.3. For each level of distortion, the best accuracy value is in bold font.

	Proportion of label distortion				
Regularization strength	0%	10%	20%	30%	40%
500	87.624	80.14	62.764	60.78	58.912
50	95.856	93.612	87.38	78.224	58.684
5	95.82	94.288	89.88	84.088	66.62
0.5	**96.188**	94.756	90.408	85.208	**69.932**
0.05	96.156	94.768	**90.548**	85.256	69.364
0.005	96.08	**94.976**	90.216	**85.856**	69.652

The results in the test set for the HM-SVM and ML-HMM predictors are listed in Table 4.5. For the HM-SVM, the best regularization is chosen for each proportion of distortion value using the cross-validation

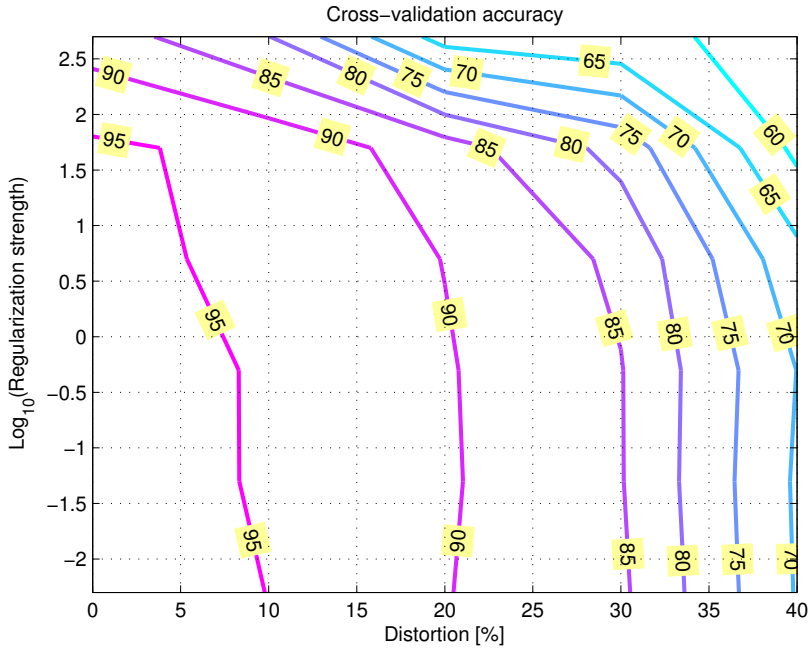


Figure 4.3.: Cross-validation accuracy of the HM-SVM with the synthetic data described in section 4.2. The level curves correspond to different values of the cross-validation accuracy as a function of the regularization strength (y axis) and the distortion level (x axis). More specifically, 5-fold cross-validation is performed for the SVM regularization parameter. The values of the regularization parameter used are 500, 50, 5, 0.5, 0.05 and 0.005. Their base 10 logarithm are along the y axis. Different levels of noise are introduced flipping the label values used to produce the observations. The proportion of label distortion introduced is 0, 10, 20, 30 and 40 percent of the total amount of labels. The results indicate that smaller values of regularization generalize better. The exact accuracy values for each pair regularization-proportion of noise appear in Table 4.3.

Table 4.4.: Average training time for the HM-SVM (measured in seconds) for different values of the regularization parameter. 5-fold cross-validation is performed, thus the results are the average of each training run, using 4-folds with 20 sequences of length 250 each and 10-dimensional features. The proportion of noise used is equal to 0%. A large regularization value restrains considerably the weight vector so that the search finishes earlier. A small regularization value leaves more freedom to choose the values of the weight, vector and the training time increases accordingly.

Regularization	500	50	5	0.5	0.05	0.005
Time [s]	106.50	186.466	325.17	486.42	648.83	674.23

performance shown in Table 4.3. As it can be seen, the HM-SVM outperforms the ML-HMM in this task, maintaining the prediction accuracy over 70 [%] even for the data sets with high level of noise.

Table 4.5.: Test accuracy of the HM-SVM and the HMM as a function of the proportion of label distortion. The results indicate that the HM-SVM is more robust against noise since its performance does not decay as much as the HMM's performance does.

	Proportion of label distortion				
	0%	10%	20%	30%	40%
	Test accuracy [%]				
HM-SVM	97.232	95.220	92.396	85.132	70.888
ML-HMM	93.376	94.816	90.232	80.740	60.960

4.3 GRAPH LABELLING

For the experiments in this section the graphs used are square images. The nodes of a graph correspond to the pixels in an image and the edges of each node are the connections given by the 4-neighbourhood of each pixel (i.e. the set formed by the pixels above, below, left and right). The outputs or labels are straight patterns and their corresponding features are generated from the labels by adding Gaussian noise. A sample label

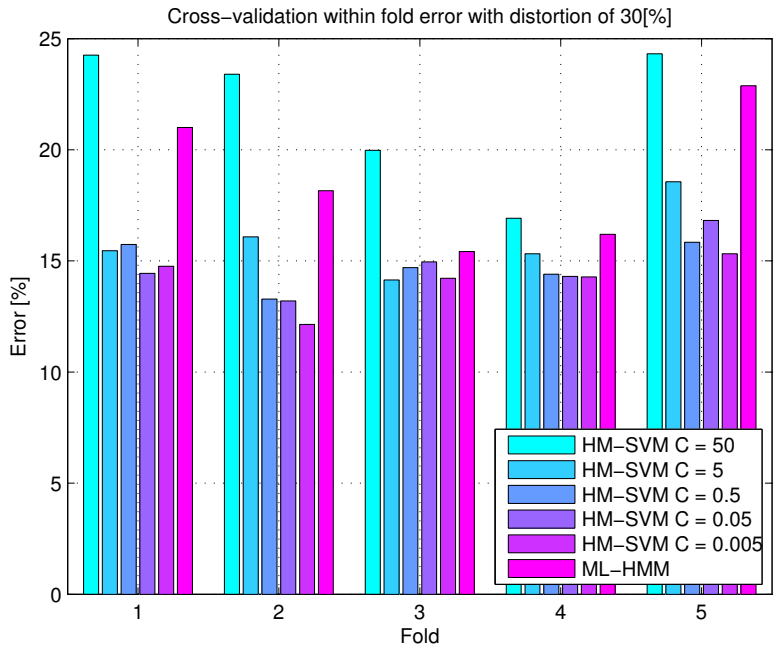
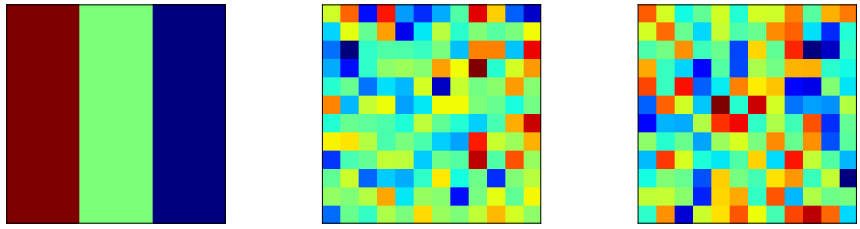


Figure 4.4.: Cross-validation within fold error with distortion of 30% in dataset #2 for the ML-HMM and the HM-SVM with several choices of the regularization. On the x axis the bars are grouped according to the training fold. The y axis represents the error in the validation fold, measured in %. Although it does not occur in general, it can be seen that some folds are harder to learn than others for *all* the models. This is, the ML-HMM and most of the HM-SVM achieve their worst performance in the same fold, the fifth in particular.

and part of their corresponding features are shown in Figure 4.5. Note that the features are generated from the labels similarly to previous experiments. Also, the labels used in this section may take more than two values, as it can be seen from Figure 4.5.



(a) Label (b) Features (c) More features

Figure 4.5.: Sample data used in the graph labelling experiment. Each label has three features although only two of them are shown in the figure, the third feature has a similar pattern. At first, each label is of dimensions $3 \times 12 \times 12$ with all its elements equal to zero. Then, a column of ones is set in the elements along the first dimension. These columns are to the left, centered and to the right of each 12×12 matrix. The features are created by means of adding white noise to these matrices. Finally, the label shown above is obtained by taking the indices that give the maximum value of the label acros the first dimension. The Python source used for the data generation just explained is given in section B.1.

The training phase of the SO-SVM is done via the subgradient descent algorithm explained in subsection 3.2.2. This is different from the previous experiments where the training method used was the cutting plane algorithm from subsection 3.2.1. This modification in the training algorithm has been made merely to verify that another method can produce good results.

The subgradient descent solvers as well as the structured model (i.e. the structured loss function, the joint feature mapping, and the argmax) are implemented in Python. In order to make this structured model for graph labelling compatible with other solvers implemented in C++, it uses the Simplified Wrapper and Interface Generator (SWIG) director

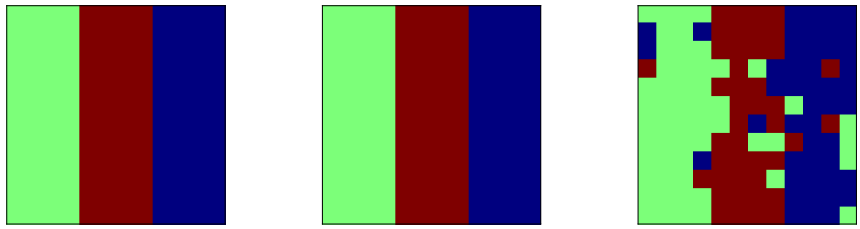
classes. It is necessary to use a director class because the C++ code for the SO-SVM solver needs to use components from the structured model. Informally, one can think of it as if the C++ code calls Python code.

In addition to examine the Linear Program (LP) relaxation for general graph inference explained in subsection 3.3.2 and the SO-SVM training based on subgradients, the aim of the experiments discussed in this section is to demonstrate the superior modelling ability of structured output learning against a model that ignores dependencies among variables. The latter type of model has been trained in the same manner as the former (subgradient descent, argmax with LP relaxation), getting rid of the parameters of the weight vector that correspond to pairwise interactions and setting to zero the pairwise potentials used in the LP relaxation inference in Equation 3.26.

Both with subgradient descent and with stochastic subgradient descent, the learning rate used is 0.01 and it is not modified during the course of training [3]. The number of iterations until convergence is reached are set manually, studying the value of the primal objective in Equation 3.12. To solve the LP in the argmax the Python package for convex optimization CVXOPT is used (Dahl and L Vandenberghe 2006).

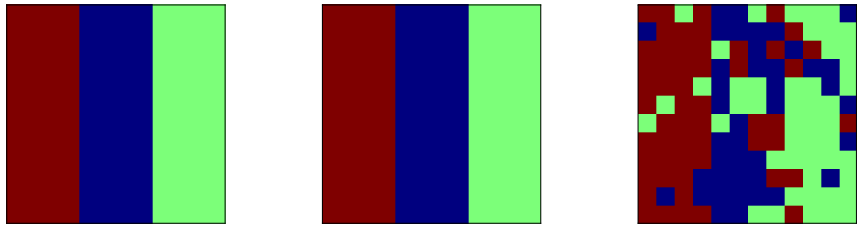
The training set is formed by twenty 12×12 images. Since the goal of the experiments is to compare how using dependencies among the output variables (i.e. structured output modelling) achieves better performance than ignoring them, for simplicity the same training data is used for validation. Figure 4.6 shows a ground truth label, the prediction made by the structured model (called unaries and pairwise in the figure) and the prediction made by the model that treats variables independently (unaries only in the figure). From the figure it is clear how accounting for dependencies in the model outperforms the other case. Figure 4.7 shows results from a similar experiment, where the noise used to create the features has been increased. As it can be seen, the structured approach is also more robust against noise.

3 In (sub-)gradient descent based algorithms it is common to decrease the learning rate or equivalent parameter so the (sub-)gradient updates in advanced steps of training are smaller than the updates produced at the beginning. However, we choose to use a small learning rate and keep it constant during the whole training process. This choice makes easier the election of the learning rate at the expense of requiring more iterations until convergence.



(a) Ground Truth (b) Unaries and pairwise (c) Unaries only

Figure 4.6.: Prediction results for a sample instance in the graph labelling task with features affected by *low* level noise. From left to right the pictures show the ground truth label, the prediction made with the model with unaries and pairwise interactions, and the prediction with the model that only learns unary interactions. The model with both type of interactions outperforms the latter.



(a) Ground Truth (b) Unaries and pairwise (c) Unaries only

Figure 4.7.: Prediction results for a sample instance in the graph labelling task with features affected by *high* level noise. From left to right the pictures show the ground truth label, the prediction made with the model with unaries and pairwise interations, and the prediction with the model that only learns unary interactions. In this case, the results for the model with only unaries are considerably worse than those for the case with lower level of noise, see Figure 4.6. The model with pairwise and unary interactions can still make a perfect prediction though. Thus, the complete model is more robust against noise.

5

CONCLUSIONS AND FUTURE WORK

This project analyzed how SO-SVMs can be used as a discriminative tool to predict structured (graph-network) outputs by leveraging the inherent structure of the data. Two main messages that the project tried to convey were the following. First, in many tasks, accounting for the complex (networked) structure of the data by treating the variables jointly, rather than making decisions about them independently, is critical. Second, discriminative modelling is more suitable for prediction tasks than generative modelling. In the next pages, we summarize the main findings of this project, analyze those findings within the context of discriminative structured prediction, and suggest future lines of work.

The project began by reviewing general machine learning concepts that were relevant for our problem. We discussed that prediction, also called inference, in the structured output setting is in general a hard problem owing to the combinatorial nature of the output space $\mathcal{Y}$. Since training (or parameter estimation) of the SO-SVM requires a slightly modified version of the prediction oracle as well, learning the SO-SVM from training data pairs is hard too.

Then, parameter estimation of the SO-SVM was explained in detail. In order to tackle the training phase of the SO-SVM two approaches were discussed. One of them worked with the constrained formulation of the problem using cutting planes. These cutting planes represent the most violated constraint for each training instance and they are found by the so-called loss-augmented inference. The second approach relied on subgradient algorithms (similar to standard gradient algorithms for non-differentiable problems). After experimenting with both types of algorithms, we concluded that the subgradient methods were preferred because they are greatly easier to implement and do not rely on any external solver. However, these methods also have problems of their own: the choice of the learning rate may not be easy and their convergence properties are not the best. Pointers to alternative algorithms dealing with these drawbacks were given in the discussion.

In addition, the generic structured output framework was particularized using two applications of the framework for which efficient inference is feasible. In the first particularization, the structured objects were restricted to be chain graphs or, as they are also commonly called, sequences. In this case, inference was solved exactly via the Viterbi algorithm. The algorithm was described in depth, including a brief discussion on its computational complexity. More importantly, this model could be easily reinforced introducing higher order dependencies into the variables of the chain graph. In other words, making the variables along the sequence dependent on two or more previous variables, instead of only using the last one. This would entail straightforward modifications in the joint feature mapping and the argmax solver presented in this report and would require increasing the computational complexity of the algorithm. The second particularization of the framework did not consider any constraint on the structure of the graphs, so that it was possible to deal with general structured output models. A method for inference in this scenario based on a linear programming relaxation was discussed. Both in the first and in the second particularization, experiments were performed to validate the approach. Moreover, in the latter case we compared models that do and models that do not account for structure. For this task, the experiment chosen was a simplification of the segmentation problem widely known in computer vision. The conclusion was that structured models outperform models that make decisions for all the variables independently. The fact that the results obtained by means of structured modelling achieved a better performance stems from the ability of the SO-SVM to account for the dependencies of the variables of the output space $\mathcal{Y}$, *as long as* it improves training performance. This can be seen in either Equation 3.11 or Equation 3.12. The first term in the objective of these equations is minimized by setting all the elements of the weight vector equal to zero. They become non-zero in the moment the empirical risk in the training set, measured by the second term in the equations, improves, paying off the cost incurred in the first term.

In the second particularization of the structured output framework, the algorithm for inference in general structured output models presented is not the only one that could have been applied. There are actually many alternatives and the literature in this field, inference in graphical models, is vast. In computer vision for instance, it is

very common to use solvers based on graph-cuts that leverage max-flow/min-cut algorithms in segmentation tasks like the ones introduced in chapter 1. Dual decomposition methods leveraging the structure of the graphs and recent advances in distributed (stochastic) optimization are also a worth exploring alternative. A possible branch of future work from this project is to study and compare these methods for inference in different tasks.

Finally, we also compared generative modelling against the discriminative approach in a label sequence learning task with synthetic data. The results obtained agree with the theory in the sense that generative modelling (or density estimation) is a harder task than discriminative modelling. Recall that generative models, as denoted by their name, provide a model that describes how the data is generated. Formally, this is captured in the prior distribution of the output objects $P_{\mathcal{Y}}(y)$ and in the conditional distribution of the observations $p_{\mathcal{X}|\mathcal{Y}}(x|y)$; or equivalently $p_{\mathcal{X},\mathcal{Y}}(x,y)$ using Bayes' rule. On the other hand, discriminative modelling intends to represent the decision boundaries between the input classes. The linear SVM used in this project does this using separating hyperplanes as explained in chapter 2. These boundaries may be interpreted probabilistically [1] as the posterior $P_{\mathcal{Y}|\mathcal{X}}(y|x)$. Note that generative models actually have discriminative properties since it is possible to get $P_{\mathcal{Y}|\mathcal{X}}(y|x)$ from the other two distributions using again Bayes' rule. This is nevertheless not true in the other direction. From this observation, it is clear, at least intuitively, that generative modelling is indeed harder than discriminative modelling because the scope of generative modelling is broader. These and other differences between generative and discriminative methods were studied by Ng and Jordan (2002). Back to the experiments in section 4.2, the results obtained with the SO-SVM are more robust against noise and generalize better. As an extension, it would also be interesting to verify that this is also the case in some task with real data, as it has been shown in the literature for instance in (Altun, Tsochantaridis, Hofmann, et al. 2003).

1 Discriminative modelling is not easily interpreted in a probabilistic framework in every case. There are some methods which actually do attempt to learn the distribution $P_{\mathcal{Y}|\mathcal{X}(y|x)}$ directly such as logistic regression (Bishop 2006), whereas other methods learn mappings from inputs to outputs like the SVM. These methods are normally called probabilistic and non-probabilistic, respectively. Still, they are all discriminative.

All the numerical experiments in this project were basically illustrative and served to confirm the theoretical findings. Clearly, a future line of work is to apply SO-SVMs (together with some of the extensions discussed) to well-known applications dealing with graph-structured data such as social networks, genomics, traffic anomalies, authorship identification, to name a few (Kolaczyk 2009). Successful implementation will require tuning of the corresponding algorithms to the application at hand.

We limited the scope of this report to structured models with discrete variables. The reason for this was that the literature on structured learning with discrete variables is richer than for the continuous case. There is nonetheless work done in structured learning with continuous variables like the Structured Output-Associative Regression or SOAR by Bo and Sminchisescu (2009). Clearly, consideration of Gaussian variables and linear generation and observation models would render the problem more tractable and it is another interesting future line of work.

Also interesting is the consideration of *kernels* that map the structured data into a (high dimensional) Euclidean space. Kernel functions empower considerably the SVM by making it capable of learning non-linear decision boundaries without the need of changing considerably the problem formulation. Of course, the SO-SVM can be complemented with kernels as well, see (Tsochantaridis et al. 2005) for a discussion on this topic. In fact, kernel methods are heavily used when dealing with graph-structured data.

A

TOOLS

All the work related to this project was developed in a laptop computer [1], using different software development tools. The total amount of time devoted to the project is roughly equal to 800 hours; approximately, half of it was devoted to reading, one quarter to the implementation of experiments, and the remaining quarter to writing this report.

Next, the libraries and other software used in this project are listed and briefly discussed in this chapter. The aim of this chapter is to give an idea of how these tools have been used during the course of the project. We do not aim at describing the tools in detail. To that end, pointers to more information are given where it is necessary.

A.1 PROGRAMMING LANGUAGES

The implementation of the methodology and experiments presented in this report is separated into three programming languages: C++, Python and Matlab. Next, we describe why they were chosen and what they have been used for.

- C++ was chosen for its great performance and to produce bindings from the C++ source code so it could be used from other programming languages (e.g. Python). The generic structured output framework class hierarchy, the cutting plane algorithm explained in subsection 3.2.1 and the HM-SVM described in subsection 3.3.1 are implemented in C++.
- Python was used to leverage its features for fast development and prototyping. The subgradient descent algorithm explained in subsection 3.2.2, the graph-network labelling model subsection 3.3.2 and the ML-HMM from section 4.2 are implemented in Python. One part of the experimental figures included in this report were produced using the Matplotlib Python plotting library.

1 Lenovo Thinkpad X1 Carbon with an Intel processor i7@2.00 GHz and 4 GB RAM.

- The HM-SVM implemented in C++ and the ML-HMM in Python are inspired by the Matlab implementation by Gunnar Raetsch and Georg Zeller [2]. Part of this code was modified and used to generate data for the label sequence learning experiments in chapter 4. Moreover, some figures in this report were produced in Matlab.

A.2 SHOGUN

The Shogun Machine Learning toolbox (Sonnenburg et al. 2010) was extensively used in this project. For instance, the methods for classification, regression, clustering and dimension reduction used to produce the figures shown in chapter 1 are in Shogun. More importantly, the SO-SVM central to this project is in Shogun too.

A.3 THE SIMPLIFIED WRAPPER AND INTERFACE GENERATOR

We used SWIG to generate code bindings of the structured output framework so that this code could be used from other programming languages. In particular, we used the bindings for Python. Nonetheless, bindings for other languages like C#, Java, Ruby, Octave and R, to name a few, could be generated as well.

SWIG director classes [3] were used to leverage the bindings for Python. Using the director classes we were able to extend the structured output framework in Shogun using Python. By means of using the director classes, code implemented in C++ can use methods (in our case they were the structured loss function, the joint feature map and the optimization oracle for graph labelling, see subsection 3.3.2) that are implemented in Python.

A.4 GIT

was used for the revision control and source code management involved in this project owing to its flexibility. There is a git repository in github were the source code can be found available at `https://github.com/iglesias/linal`.

2 Link to the Matlab implementation of Hidden Markov Support Machines in mloss.org.
3 Link to director classes in SWIG.

A.5 MOSEK AND CVXOPT

Mosek and CVXOPT (Dahl and L Vandenberghe 2006) are general-purpose optimization libraries. On the one hand, the Mosek C/C++ API was used to solve the QP in the SO-SVM cutting plane algorithm explained in subsection 3.2.1. On the other hand, CVXOPT for Python was used to solve the LP relaxation of the argmax for general structured output (graph-network) models presented in subsection 3.3.2.

B

DATA GENERATION

B.1 GRAPH LABELLING

The function implemented in Python to generate data for the experiments in section 4.3 is shown below.

```
import numpy

def generate_blocks_multinomial(n_samples, noise, n_rows, n_cols):
  '''
  Generates three state graph simulated data, where the features
  are generated from the graph labels by adding Gaussian noise.

  Written by Fernando J. Iglesias Garcia, 2013. Inspired by
  pystruct written by Andreas Mueller.
  '''
  # labels
  Y = numpy.zeros((n_samples, n_rows, n_cols, 3))
  # vertical stripes of ones along the last dimension of Y
  for i in xrange(n_samples):
    ridxs = numpy.random.permutation(range(3))
    # left stripe
    Y[i, :,   :4, ridxs[0]] = 1
    # center stripe
    Y[i, :,  4:8, ridxs[1]] = 1
    # right stripe
    Y[i, :, 8:16, ridxs[2]] = 1

  # observations introducing noise in the labels
  X = Y + noise * numpy.random.normal(size=Y.shape)
  # finish labels format
  Y = numpy.argmax(Y, axis=3).astype(numpy.int32)

  return X, Y
```

B.2 LABEL SEQUENCE LEARNING

Part of the Matlab function used to generate data for the experiments in section 4.2 is listed below. The goal is not to provide a fully operating and executable function but rather to illustrate how the data generation process works.

```
function [label, signal] = simulate_data(num_exm, exm_len, ...
        distort ,prop_distort, num_features, block_len, num_blocks)

% Generates two state simulated data, where features are
% generated from the label sequence with some labels
% swapped and Gaussian noise added.
%
% Written by Fernando J. Iglesias Garcia, 2013. Inspired by the
% HM-SVM toolbox written by Gunnar Raetsch and Georg Zeller.

% generate label sequences containing from num_blocks(1)
% to num_blocks(2) blocks of positive labels, each of length
% between block_len(1) and block_len(2)
for i = 1:num_exm,
  exm_id = [exm_id i*ones(1,exm_len)];
  pos_id = [pos_id 1000*i + (1:exm_len)];
  l = -ones(1,exm_len);
  rnb = num_blocks(1) + ceil((num_blocks(2)- ...
        num_blocks(1)).*rand(1)) - 1;
  for j = 1:rnb,
    rl = block_len(1) + ceil((block_len(2)- ...
        block_len(1)).*rand(1)) - 1;
    rp = ceil((exm_len-rl).*rand(1));
    l(rp:rp+rl) = 1;
  end
  label = [label l];
end

% generate features by: i) introducing label noise flipping a
% proportion of labels and ii) adding gaussian noise
for i = 1:num_features,
  l = label;
  perm_idxs = randperm(length(l));
  distort_idxs = perm_idxs(1:round(length(l)*prop_distort));
  l(distort_idxs) = -l(distort_idxs);
  signal(i,:) = l+noise*randn(size(label));
end
```

BIBLIOGRAPHY

Agrafiotis, Dimitris K. (2003). „Stochastic Proximity Embedding.“ In: *Journal of Computational Chemistry* 24.10, pp. 1215–1221 (cit. on p. 5).

Altun, Yasemin, Ioannis Tsochantaridis, Thomas Hofmann, et al. (2003). „Hidden markov support vector machines.“ In: *Machine Learning-International Workshop*. Vol. 20. 1, p. 3 (cit. on pp. 9, 69).

Antoniou, Andrea and Wu-Sheng Lu (2007). *Practical Optimization: Algorithms and Engineering Applications*. Springer (cit. on pp. 11, 15, 41, 42).

Bishop, Christopher M (2006). *Pattern Recognition and Machine Learning*. Secaucus, NJ, USA: Springer-Verlag (cit. on pp. 1, 30, 69).

Bo, Liefeng and Cristian Sminchisescu (2009). „Structured output-associative regression.“ In: *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, pp. 2403–2410 (cit. on p. 70).

Boyd, Stephen and Lieven Vandenberghe (2004). *Convex Optimization*. Cambridge University Press (cit. on p. 11).

Boykov, Yuri (2001). „Fast approximate energy minimization via graph cuts.“ In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 8.4, pp. 413–1239 (cit. on p. 50).

Cherkassky, Vladimir and Filip Mulier (2007). *Learning from Data*. IEE Press (cit. on pp. 9, 11, 22).

Collins, Michael (2002). „Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms.“ In: *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*. Association for Computational Linguistics, pp. 1–8 (cit. on p. 9).

Crammer, Koby and Yoram Singer (2002). „On the Algorithmic Implementation of Multiclass Kernel-based Vector Machines.“ In: *Journal of Machine Learning Research* 2.2, pp. 265–292 (cit. on pp. 22, 23, 25, 37).

Dahl, Joachin and L Vandenberghe (2006). „CVXOPT: A python package for convex optimization.“ In: *Proc. Eur. Conf. Op. Res* (cit. on pp. 64, 73).

Dietterich, Thomas G. and Ghulum Bakiri (1995). „Solving Multiclass Learning Problems via Error-Correcting Output Codes.“ In: *arXiv preprint cs/9501101* (cit. on p. 23).

Domke, Justin (2013). „Learning Graphical Model Parameters with Approximate Marginal Inference.“ In: (cit. on p. 41).

Everingham, M., L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. *The PASCAL Visual Object Classes Challenge 2012 (VOC2012)* (cit. on pp. 7, 8).

Finley, Thomas and Thorsten Joachims (2007). „Parameter learning for loopy markov random fields with structural support vector machines.“ In: *Proceedings of the ICML Workshop on Constraint Optimization and Structured Output Spaces* (cit. on p. 50).

Finley, Thomas and Thorsten Joachims (2008). „Training structural SVMs when exact inference is intractable.“ In: *Proceedings of the 25th international conference on Machine learning (ICML)*, pp. 304–311 (cit. on pp. 41, 50).

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (Feb. 2009). *The Elements of Statistical Learning*. Springer (cit. on p. 1).

Ising, Ernst (1925). „Beitrag zur Theorie des Ferromagnetismus.“ In: *Zeitschrift für Physik A Hadrons and Nuclei* 31.1, pp. 253–258 (cit. on p. 8).

Klein, Thoralf, Ulf Brefeld, and Tobias Scheffer (2008). „Exact and approximate inference for annotating graphs with structural SVMs.“ In: *Machine Learning and Knowledge Discovery in Databases*. Springer, pp. 611–623 (cit. on p. 50).

Kolaczyk, Eric D (2009). *Statistical analysis of network data*. Springer (cit. on p. 70).

Koller, Daphne and Nir Friedman (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press (cit. on pp. 28, 50).

Lacoste-Julien, Simon, Martin Jaggi, Mark Schmidt, and Patrick Pletscher (2013). „Block-Coordinate Frank-Wolfe Optimization for Structural SVMs.“ In: *Proceedings of the 30th International Conference on Machine Learning (ICML)* (cit. on p. 44).

Lafferty, John, Andrew McCallum, and Fernando Pereira (2001). „Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data.“ In: *Proceedings of the Eighteenth International Conference on Machine Learning (ICML)*, pp. 282–289 (cit. on p. 9).

Lafferty, John, Andrew McCallum, and Fernando CN Pereira (2001). „Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data.“ In: (cit. on p. 38).

Leijon, Arne and Gustav Eje Henter (2012). *Pattern Recognition: Fundamental Theory and Exercise Problems*. KTH Electrical Engineering (cit. on pp. 11, 26).

Ng, Andrew Y and Michael I Jordan (2002). „On Discriminative vs. Generative classifiers: A comparison of logistic regression and naive Bayes.“ In: (cit. on p. 69).

Nowozin, Sebastian and Christoph H. Lampert (2011). „Structured Learning and Prediction in Computer Vision.“ In: *Foundations and Trends in Computer Graphics and Vision* 6.3-4, pp. 185–365 (cit. on p. 52).

Nowozin, Sebastian and Christoph H Lampert (2011). *Structured learning and Prediction in Computer Vision*. Now Publishers (cit. on p. 38).

Pletscher, Patrick and Pushmeet Kohli (2012). „Learning low-order models for enforcing high-order statistics.“ In: *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*. JMLR: W&CP 22, pp. 886–894 (cit. on pp. 41, 52).

Rabiner, Lawrence R. (1989). „A tutorial on hidden Markov models and selected applications in speech recognition.“ In: *Proceedings of IEEE*. Vol. 77, pp. 257–286 (cit. on pp. 29, 54).

Simon, Phil (2013). *Too Big to Ignore: The Business Case for Big Data*. Wiley (cit. on p. 1).

Sonnenburg, Sören, Gunnar Rätsch, Sebastian Henschel, Christian Widmer, Jonas Behr, Alexander Zien, Fabio de Bona, Alexander Binder, Christian Gehl, and Vojtěch Franc (2010). „The SHOGUN machine learning toolbox.“ In: *The Journal of Machine Learning Research* 99, pp. 1799–1802 (cit. on pp. 53, 72).

Sra, Suvrit, Sebastian Nowozin, and Stephen J Wright (2012). *Optimization for Machine Learning*. Mit Pr (cit. on p. 11).

Szummer, Martin, Pushmeet Kohli, and Derek Hoiem (2008). „Learning CRFs Using Graph Cuts.“ In: *10th European Conference on Computer Vision (ECCV)*, pp. 582–595 (cit. on pp. 38, 51).

Tenenbaum, Joshua Brett, V. de Silva, and J. C. Langford (2000). „A global geometric framework for nonlinear dimensionality reduction.“ In: *Science* 290.5500, pp. 2319–23 (cit. on p. 5).

Teo, Choon Hui, SVN Vishwanthan, Alex J Smola, and Quoc V Le (2010). „Bundle methods for regularized risk minimization.“ In: *The*

Journal of Machine Learning Research 11, pp. 311–365 (cit. on pp. 39, 44).

Tsochantaridis, Ioannis, Thorsten Joachims, Thomas Hofmann, and Yasemin Altun (2005). „Large Margin Methods for Structured and Interdependent Output Variables." In: *Journal of Machine Learning Research* 6, pp. 1453–1484 (cit. on pp. 33, 41, 70).

Uricár, Michal, Vojtech Franc, and Václav Hlavác (2012). „Detector of facial landmarks learned by the structured output SVM." In: *VISAPP* 12, pp. 547–556 (cit. on p. 8).

Zeller, Georg (2010). „Machine Learning Algorithms for the Analysis of Data from Whole-Genome Tiling Microarrays." PhD thesis. Universitätsbibliothek Tübingen (cit. on p. 55).

Zeller, Georg, Stefan R Henz, Sascha Laubinger, Detlef Weigel, and Gunnar Rätsch (2008). „Transcript normalization and segmentation of tiling array data." In: *Pac Symp Biocomput*. Vol. 13, pp. 527–538 (cit. on pp. 7, 9).

NOTATION

PROBABILITY

SYMBOL	MEANING
$P[\cdot\,]$	probability mass function
$p(\cdot\,)$	probability density function
$\mathbb{I}_{\texttt{expr}}(x)$	indicator function returning 1 if the Boolean expression `expr` involving variable x is true and 0 otherwise.

STRUCTURED OUTPUT LEARNING AND PREDICTION

SYMBOL	MEANING
$\mathcal{Y}_i$	domain of the i-th output variable
y_i	individual output variable, $y_i \in \mathcal{Y}_i$
$\mathcal{Y}$	product space of all the individual output domains
y	all the output variables of an example
$\mathcal{X}$	domain of the input variables
x	all the input variables of an example
$\Psi(x, y)$	feature map of an example pair (x, y)
$\Delta(y, y^*)$	structured loss incurred when predicting y^* instead of y
$\mathbf{w}$	(linear) parameters of a structured output SVM
η	learning rate in (sub-)gradient descent algorithms
$\text{sign}(\cdot\,)$	sign function that takes of one two values depending on the sign of its argument

ACRONYMS

MAP	Maximum-A-Posteriori
MRF	Markov Random Field
CRF	Conditional Random Field
SVM	Support Vector Machine
LP	Linear Program
ILP	Integer Linear Program
QP	Quadratic Program
BP	Belief Propagation
SO-SVM	Structured Output Support Vector Machine
HMM	Hidden Markov Model
ECOC	Error-Correcting Output Codes
LDS	Linear Dynamical System
iid	Independent and Identically Distributed
HM-SVM	Hidden Markov Support Vector Machine
PLiFs	Piece-wise Linear Functions
ASR	Automatic Speech Recognition
SWIG	Simplified Wrapper and Interface Generator
DPM	Deformable Parts Models
CKY	Cocke-Kasami-Younger
FG/BG	Foreground/Background
STL	Statistical Learning Theory
AWGN	Additive White Gaussian Noise

INDEX

COLOPHON

This document was typeset in LaTeX using the typographical look-and-feel `classicthesis`. Most of the graphics in this report are generated using `pgfplots` and `pgf/tikz`. The bibliography is typeset using `biblatex`.