technische universität
dortmund

Master Thesis

# Explicit and Implicit Feature Maps for Structured Output Prediction

Lukas Pfahler

December 2015

Gutachter:
Prof. Dr. Katharina Morik
Dipl.-Inform. Christian Pölitz

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Künstliche Intelligenz (LS-8)
http://www-ai.cs.uni-dortmund.de/

# Contents


**1 Introduction** **1**

1.1 Contribution of this Thesis . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.2 Structure of this Thesis . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

**2 Support Vector Machines for Structured Output Prediction** **5**

2.1 Learning Task . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.1.1 Discrete Structures . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.1.2 Discrete Output Structures . . . . . . . . . . . . . . . . . . . . . 6

2.1.3 Empirical Risk Minimization . . . . . . . . . . . . . . . . . . . . 8

2.2 Structural Support Vector Machines and the Inference Problem . . . 9

2.2.1 Primal $n$-Slack Formulation . . . . . . . . . . . . . . . . . . . . 10

2.2.2 Cutting Plane Optimization . . . . . . . . . . . . . . . . . . . . . 13

2.3 Two Example Instances of Structural Support Vector Machines . . . 17

2.3.1 Sequence Tagging . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

2.3.2 Constituency Parsing . . . . . . . . . . . . . . . . . . . . . . . . . 18

**3 Kernels for Implicit Feature Maps** **21**

3.1 Kernels and Implicit Feature Maps . . . . . . . . . . . . . . . . . . . . 21

3.1.1 Mercer's Theorem . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

3.1.2 Reproducing Kernel Hilbert Spaces . . . . . . . . . . . . . . . . . 23

3.2 Some Important Kernels . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

3.2.1 Gaussian Kernel . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

3.2.2 String Kernel . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

3.2.3 Tree Kernel . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

**4 Towards Hassle-Free Structured Prediction** **29**

4.1 Kernelized Structural Support Vector Machines . . . . . . . . . . . . 29

4.1.1 Dual Program . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

4.1.2 Inference: A Pre-Image Problem . . . . . 33
4.2 Discrete Output Structures and Simple Kernels . . . . . 34
4.2.1 Joint Kernels . . . . . 34
4.2.2 A Class of Kernels for Discrete Output Structures . . . . . 35
4.2.3 Solving the Pre–Image Problem . . . . . 37
4.3 Discrete Output Structures and Loss Functions . . . . . 38
4.4 The Algorithm . . . . . 41
4.4.1 Putting All Together . . . . . 41
4.4.2 Runtime Analysis and Efficiency Improvements . . . . . 42

**5 Related Methods for Structured Prediction 47**
5.1 More on Structural Support Vector Machines . . . . . 47
5.2 Kernel-Based Methods . . . . . 48
5.2.1 Kernel Dependency Estimation . . . . . 48
5.2.2 Joint Kernel Support Estimation . . . . . 50
5.2.3 Multiple Kernel Learning . . . . . 51
5.3 Structured Prediction – a Search Problem . . . . . 52
5.4 Other Non-Linear Models . . . . . 53

**6 A Study on Dependency Parsing 55**
6.1 Dependency Parsing and Related Work . . . . . 55
6.1.1 The Task . . . . . 55
6.1.2 Existing Approaches and Related Work . . . . . 57
6.1.3 The Experiments . . . . . 59
6.2 Dependency Parsing with Explicit Features . . . . . 60
6.2.1 Explicit Feature Map . . . . . 60
6.2.2 Inference . . . . . 61
6.2.3 Model Training using MIRA . . . . . 62
6.2.4 Model Training using Structural SVMs . . . . . 62
6.3 Dependency Parsing with Implicit Features . . . . . 63
6.3.1 Picking a Hash Function . . . . . 64
6.3.2 Picking an Input Kernel . . . . . 66
6.3.3 Testing the Model on the Full Treebank . . . . . 70
6.3.4 Approximate Reconstruction Algorithm . . . . . 71
6.3.5 Analyzing Efficiency Gains . . . . . 73
6.3.6 Presenting my Best Kernel Parser . . . . . 73

**7 Discussion and Outlook** **75**

**Appendices** **85**

**A Glossary** **87**

**B Implementation Details** **89**

B.1 Java Library . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89

B.1.1 Explicit Features . . . . . . . . . . . . . . . . . . . . . . . . 90

B.1.2 Implicit Features . . . . . . . . . . . . . . . . . . . . . . . . 90

B.2 RapidMiner Integration for Dependency Parsing . . . . . . . . . . . 91

# 1 Introduction

Recent advances in Machine Learning have made it possible that more and more people have access to data mining tools and less and less expert knowledge is required to solve Machine Learning tasks. For instance, the most-recent version of the popular data-mining platform RapidMiner[1] can automatically design data-mining processes given a spreadsheet of data. For classification tasks, many solutions including Google Prediction API[2] and Amazon ML[3] provide web-based services where a user can train prediction models by upload a spreadsheet of labeled training data. This model is then accessible via a simple web API.

However, spreadsheets are not the most convenient form of representation for all types of data. There are tasks where we want to classify structured objects like texts, time series, graphs or images. If we want to use the same prediction models as above, we have to find a set of features that describes the interesting properties of particular structures. We speak of a *feature map*, a function that returns for a given structure a fixed-size, real-valued vector of features. It should hold that similar structures are mapped to similar vectors. Finding a suitable feature map is not a trivial task: How can we describe structures that can vary in size in a fixed-size feature vector?

Let us consider one example, where the structures are texts. We can define features based on global statistics like the number of words or the number of characters. Obviously, those features do not carry a lot of semantic information about the text. A popular feature map is the bag-of-words that has a feature for every possible word that indicates the number of occurrences of a particular word in the given text. A bag-of-words feature map can express similarities of two texts via the dot-product of two bag-of-words vectors; the larger the dot product is the more words occur in both texts. But this feature map does not capture the order of words, an important

[1] `https://rapidminer.com/news-posts/rapidminer-v6-0/`
[2] `https://cloud.google.com/prediction/`
[3] `http://aws.amazon.com/machine-learning/`

aspect of language. We could construct a feature map that indicates for each possible position in the text and each possible word in the dictionary whether the word appears at the position in the given text. While this feature map unambiguously describes a text, it cannot capture similarities of two texts using the dot product of two feature vectors. If we take a sentence and compute its feature map and then take the same sentence but insert a single word at the beginning, we get two feature maps that differ in all components. It seems we need something in between, a feature map that captures more information about the structure than just its atomic components but still captures similarities in the vector space.

Kernel Methods fill this void. We can define a function that, given two structures, computes some similarity, e.g., by matching all substructures that appear in both given structures. In many cases, for instance for strings or trees, these matches can be computed efficiently. If the function is positive-definite, it is a member of the class of kernel-functions. Thus it implicitly defines a feature map and computes the dot product. Thus for algorithms that compute a model based on the dot product of feature maps, we can apply the so-called *Kernel Trick* and replace the dot-product by our kernel function. Kernels for structured objects have been successfully applied to classification tasks with structured inputs. They circumvent the need for expert-designed feature maps.

The research in machine learning has progressed from studying isolated classification tasks to the study of tasks where both inputs and outputs are structured objects [14]. Examples for these *Structured Prediction* tasks include Sequence Tagging, Natural Language Parsing or Object Localization. In sequence tagging, we are given a sequence of tokens and predict the label for each of those tokens. We know that there are dependencies between labels, thus we do not want to predict each label of the sequence independently. In Natural Language Parsing, we predict the syntactical structure in form of a parse tree of a given sentence. Object Localization involves finding the bounding boxes and types of objects in a given image.

The state-of-the art methods, including the *Structural Support Vector Machine*, require the definition of a *joint feature map* for input and output pairs. These feature maps need to be designed by experts and usually incorporate domain-specific knowledge. For instance, if we want to predict the part-of-speech tags for English sentences, domain experts could suggest that prefixes and suffixes are a good indicator for word classes. For instance, adverbs usually end on *-ly*, prefixes like *re-*, *ex-* or *in-* are good indicators for verbs, or capitalized words are most-likely named entities. However, this strong dependence on expert-designed features makes methods

for Structured Output prediction hard to use. I hypothesize that we can also use feature maps implicitly defined by kernels for Structured Output learning.

## 1.1 Contribution of this Thesis

In this thesis I use joint kernels instead of explicitly defined joint feature maps for Structured Output prediction. More precisely, I will use the Tensor Joint kernel that combines a kernel for the input structures with a kernel for the output structures. This allows me to reuse kernels defined for classification tasks to solve Structured Output learning tasks. The joint kernel implicitly defines a joint feature map and thereby reduces the amount of feature-engineering required in previous approaches.

I will demonstrate that, when using kernels instead of explicit feature maps, the inference problem of predicting a structured output given an input becomes a *pre-image* problem. I propose to use a simple output kernel, such that solving this pre-image problem is tractable. As this work focuses on discrete structures, this pre-image problem will be a combinatorial optimization problem. The exact structure of this problem will depend on the output domain.

The algorithm I present is based on the Cutting Plane optimization for Structural Support Vector Machines [60, 26]. Model learning requires a lot of computationally-expensive kernel computations. To make training more efficient, I propose an index structure that exploits sparsity in the outputs. This approach decreases learning time substantially.

I will demonstrate the usefulness of this approach on a problem from Natural Language Processing, *Dependency Parsing*. The machine-learning task is to predict the inner syntactic structure of a given sentence. These syntactic structures are modeled by directed edges between the words of the given sentence. I will present a data-driven approach that uses only one handcrafted feature, whereas standard methods apply 60 or more templates to generate features. Experiments based on annotated German data will show that the performance of my kernelized prediction model is comparable to the performance of a model based on expert features.

## 1.2 Structure of this Thesis

I begin by introducing the task of Structured Output Prediction in Chapter 2, focusing on discrete and structured output domains, which I will define precisely. I present the most-frequently used approach for learning prediction models for struc-

tured outputs, the Structural Support Vector Machine [60, 26]. To illustrate the work that has to be done to apply this method, I present two example tasks with structured output from the field of Natural Language Processing.

In Chapter 3, I present the class of kernel functions, which compute the dot product of two implicitly defined feature maps. I introduce two different mathematical justifications for kernels, Mercer's theorem and Reproducing Kernel Hilbert spaces. These results provide characterizations of the class of kernel functions, without the need to specify the feature map used by the kernel. I show three popular kernels and characterize the feature maps they use.

The main contribution of this thesis is presented in Chapter 4. First, I demonstrate how the Dual Program of the Structural SVM optimization problem allows us to use kernels instead of explicitly defined feature maps and show that the inference problem becomes a pre-image problem. Second, I introduce the Tensor kernel that combines two kernels into a joint kernel. I propose a class of kernels for output structures and show the corresponding structure of the pre-image problem. Third, I propose a general loss function that can be used to train a prediction model. Finally, I present an algorithm that efficiently trains a kernelized Structured Prediction model using a Tensor kernel.

In Chapter 5, I present additional research on Structural SVMs as well as other approaches to Structured Output Prediction and relate them to the results in Chapter 4.

I test the algorithm proposed on the Dependency Parsing task in Chapter 6. I begin by introducing the task and presenting different approaches to solve it. One approach by McDonald et al. [38, 42] that won the CoNLL shared task on multilingual dependency parsing [2] will be presented greater detail. This approach involves defining a joint feature map, thus in this I apply the Structural SVM to the learning problem utilizing the feature map proposed by the authors. I compare it to my approach that uses a Tensor kernel with the output kernel proposed in Chapter 4.

Finally, Chapter 7 concludes this thesis and gives an outlook to future research. I want to also mention the glossary in Appendix A which summarizes commonly used definitions, terms and equations.

# 2 Support Vector Machines for Structured Output Prediction

This chapter will first introduce the Machine Learning task of discrete Structured Output Prediction. Then, in Section 2.2, I will introduce the Structural Support Vector Machine (SVM), a method for tackling that learning task and present the algorithm for learning the prediction model. Furthermore I will show two examples of how the Structural Support Vector Machine has been used successfully to solve prediction problems.

## 2.1 Learning Task

In this section I present the class of learning tasks with structured outputs. In these tasks we are interested in finding a function $\hat{y}(x)$ that maps inputs from a set of structured objects $\mathcal{X}$ to elements of another set of structured objects $\mathcal{Y}$. We call $\hat{y}$ a prediction model. The Machine Learning task of Structured Output Learning is learning a model that best fits given training data $\mathcal{D} = [(x_1, y_1), ..., (x_n, y_n)]$ where $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$. In contrast to Structured Output Regression, where the outputs are structured and continuous, this work will focus on prediction tasks with discrete-structured output. Thus I will first define those sets of discrete structures precisely. Looking at known prediction problems with discrete structured output we will discover a similarity that motivates a modified definition for discrete output structures. Then, we introduce a common framework for solving the learning task of structured output prediction – Empirical Risk Minimization.

### 2.1.1 Discrete Structures

Let us first define, what a set of discrete structures is. The following definition builds on an idea by Gärtner and Vembu[19] who describe a set of discrete structures using a finite alphabet.

**Definition 2.1.1** We call $\mathcal{X}$ a set of discrete structures if and only if $\mathcal{X} \subseteq \mathcal{P}(\Sigma)$ where $\Sigma$ is a finite or enumerable infinite alphabet.

Consequently, any discrete structure $x \in \mathcal{X}$ can be constructed by selecting a number of atomic units from an alphabet $\Sigma$ in an arbitrary order. Consider the following examples of discrete structures:

**Example 2.1.1** The set of all token sequences can be expressed using $\Sigma = (\mathbb{N} \times \mathcal{V})$ where $\mathcal{V}$ is the dictionary of possible tokens. For instance, if

$$\mathcal{V} = \{atom, ..., serious, so, why, ..., zombie\}$$

the sequence 'why so serious' can be unambiguously represented by the set $\{(2, so), (1, why), (3, serious)\} \in \mathcal{P}(\Sigma)$.

Note that it is not sufficient to set $\Sigma = \mathcal{V}$ as that would result in a bag-of-words representation that does not describe the order of tokens. While theoretically the cardinality of the alphabet is infinite, in practice there will be a longest possible sequence.

The simplest example of discrete structures is probably the set of atomic labels used in multiclass classification tasks. Thus traditional prediction tasks fit in the framework of discrete structured prediction.

**Example 2.1.2** The set of labels $\mathcal{X}$ can be expressed using $\Sigma = \mathcal{L}$ where $\mathcal{L}$ is the finite set of all possible labels. Now, $\mathcal{X} = \{\{l\} \mid l \in \mathcal{L}\} \subseteq \mathcal{P}(\mathcal{L})$.

The extension to multilabel classification is trivial.

### 2.1.2 Discrete Output Structures

Many prediction tasks with discrete structured outputs show strong dependencies between input and output. Often, the input already constrains the set of possible outputs to only a subset of all possible structures. For instance many prediction tasks in Natural Language Processing are merely annotation tasks, i.e., the output contains the input plus a new layer of annotation like part-of-speech tags, chunks or syntactic structures.

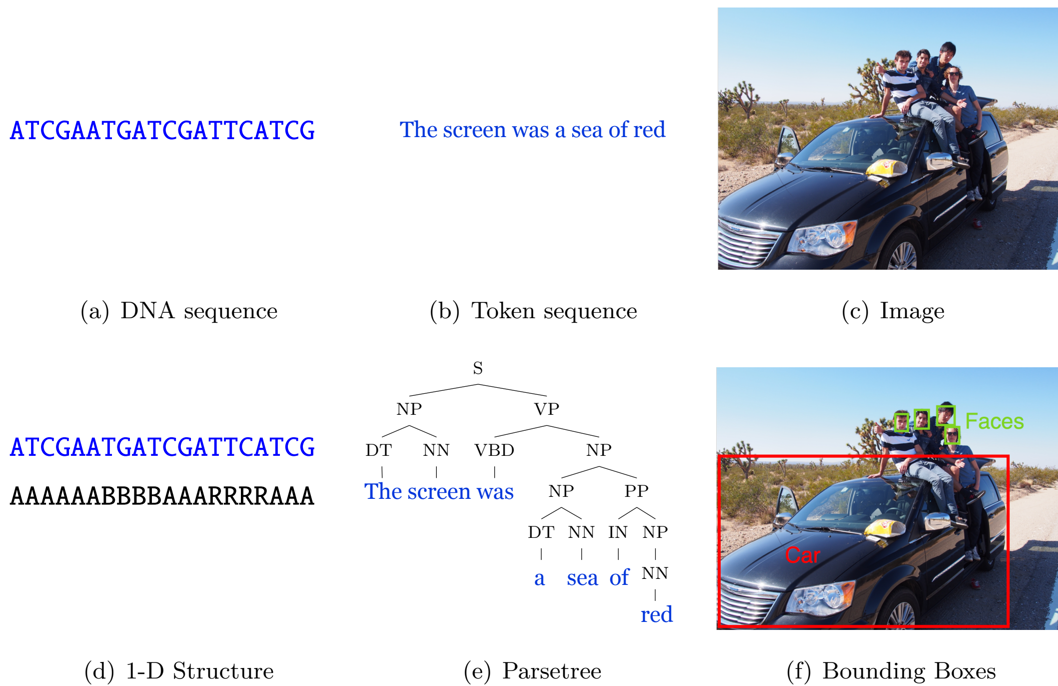



**Figure 2.1:** Inputs and Output Structures: The top row shows the inputs, the bottom row the annotated outputs. We see that the outputs add a layer of annotation to the inputs.

Similar tasks arise in bioinformatics: e.g. for the protein structure prediction task we are given a sequence of amino acids and want to predict its three-dimensional structure. Often this is done by solving a intermediate prediction steps first: 1-D Structure Prediction, which predicts the secondary structure of the molecule by assigning one of three possible labels – $\alpha$-helix, $\beta$-field or random coil – to each amino acid [5]. As implied above, this intermediate problem can be viewed as an annotation tasks. While the previous examples all have discrete structured inputs, the same annotation-idea also applies to problems with non-discrete inputs. E.g., in computer vision tasks like face detection or more general object detection, the input is a continuous matrix of image data. We can, however, characterize the output as an image with an additional layer of discrete annotation like bounding boxes or position-label pairs (See Figure 2.1).

To account for this property, we also define the set of discrete output sets that adds a layer of discrete structured annotation to an input:

**Definition 2.1.2** We call $\mathcal{Y} : \mathcal{X} \rightarrow \mathcal{P}(\mathcal{P}(\Sigma) \times X)$ a set of *discrete output structures* for a set of inputs $\mathcal{X}$ if the alphabet $\Sigma$ is a finite or countably infinite set, $\forall x \in \mathcal{X}$ $|\mathcal{Y}(x)| < \infty$ and $\forall x\ \forall(\sigma, x') \in \mathcal{Y}(x)$ it holds that $x = x'$ and $\sigma \in \mathcal{P}(\Sigma)$.

Note that $\mathcal{Y}(x)$ is the set of all possible outputs for an input $x$, including wrong outputs. It should hold that the correct, hand-labeled output $y$ is a member of $\mathcal{Y}(x)$.

This definition specifies that the input $x$ is a part of the output $(\sigma, x)$. However, an opposing view is also possible; we could define that only the discrete annotation $\sigma$ is output. But a meaningful comparison of two outputs often requires to not only look at the annotation, but also at the input. For now, this work will use the above definition as it also offers the possibility to just ignore the input $x$.

Now we consider an example of a set of discrete output structures using that definition from natural language processing: Part-of-Speech Tagging.

**Example 2.1.3** The set of tagged sequences can be expressed using an alphabet $\Sigma = (\mathbb{N} \times \mathcal{T})$ where $\mathcal{T}$ is the dictionary of possible tags.

For instance if $\mathcal{T} = \{N, V, P, DET, ADJ\}$ the output for the input sequence $x =$'Computers are the future' might be

$$\text{Computers:}N \text{ are:}V \text{ the:}DET \text{ future:}N$$

This can be unambiguously represented by the pair

$$(\{(1, N), (3, DET), (2, V), (4, N)\}, x) \in \mathcal{Y}(x) \subseteq (\mathcal{P}(\Sigma) \times X).$$

The input $x$ poses strong constraints toward the possible outputs $\mathcal{Y}(x)$, e.g. tag sequence and original sequence must have the same length.

**Definition 2.1.3** If $\mathcal{X}$ is a set of inputs and $\mathcal{Y}(\cdot)$ is a set of discrete output structures we define $\mathcal{Y} := \bigcup_{x \in X} \mathcal{Y}(x)$.

### 2.1.3 Empirical Risk Minimization

We have seen that algorithms for Structured Output Learning try to find a prediction model that best fits the data. How do we specify what model fits ‘best’? The most common approach is Empirical Risk Minimization. We associate with every prediction $\hat{y}(x)$ a loss $L(x, y; \hat{y}(\cdot)) \in \mathbb{R}^+$, that quantifies how much error we make predicting the example $(x, y)$ using the model $\hat{y}(\cdot)$. The simplest choice of a loss function $L$ is the 0/1 loss, that is 1 if prediction and true output $y$ differ and 0 if the correct output was predicted.

We call the loss we expect over all possible pairs of inputs and outputs for our predictor $\hat{y}$ *Risk*

$$R(\hat{y}) := \int_{\mathcal{X}\times\mathcal{Y}} L(x, y; \hat{y}(\cdot))dp(x, y)$$

where $p(x, y)$ is the fixed distribution that generates the input/output pairs. We want to choose the prediction model that minimizes the risk. However normally the distribution $p$ is unknown, thus instead we minimize the *Empirical Risk* for the data $\mathcal{D}$

$$R_{emp}(\hat{y}) := \frac{1}{n}\sum_{i=1}^{n} L(x_i, y_i; \hat{y}(\cdot))$$

under the assumption known from other supervised learning tasks that $\mathcal{D}$ is a sufficiently large sample generated i.i.d. from the distribution $p$.

Directly minimizing the empirical risk over a family of possible models

$$\min_{\hat{y}} R_{emp}(\hat{y})$$

promotes overfitting to the data; imagine a model that remembers all input-output pair of the data which consequently has an empirical risk of zero. Thus we commonly add a regularizing term that penalizes model complexity to prevent overfitting [54, 61].

$$\min_{\hat{y}} R_{emp}(\hat{y}) + \lambda\Omega(\hat{y})$$

where $\lambda > 0$ controls the influence of the regularizer $\Omega(\cdot)$.

## 2.2 Structural Support Vector Machines and the Inference Problem

In the previous section, I introduced the learning task of structured output prediction as well as a technique for training predictors – the regularized empirical risk minimization. In this section I introduce the Structural Support Vector Machine, a machine learning method for solving that task. I start by introducing the so-called $n$-Slack formulation of Structural SVMs [60] and show that it fits the framework of empirical risk minimization. Then I present the cutting-plane optimization algorithm for training Structural SVMS [26]. Finally, I show two examples of how Structural SVMs can be used to solve Structured Output Prediction tasks from Natural Language Processing.

Throughout this chapter, readers familiar with binary SVMs will recognize similarities with SVMs, e.g. in the motivation of the optimization problem via margin

maximization. However, I do not introduce the Structural SVM as an extension of the binary SVM, but try to motivate it from scratch. Thus no prior knowledge about SVMs is required.

### 2.2.1 Primal $n$-Slack Formulation

We are given a multiset of training data $\{(x_i, y_i)\}$ such that for each training example $(x_i, y_i), i = 1, ..., n$ it holds that $y_i \in \mathcal{Y}(x_i)$ is the annotated, true output for the input $x_i \in \mathcal{X}$. For prediction, we want to learn a discriminant function $f : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ that measures compatibility between an input and an output and assigns a real-valued score. To compute a prediction for an input we find the structure that maximizes the discriminant function:

$$\widehat{y}(x) = \arg \max_{y \in \mathcal{Y}(x)} f(x, y).$$

The key idea to Structural Support Vector Machines is to rely on a linear discriminant function that uses a joint feature map $\phi : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^d$ and a weight vector $w \in \mathbb{R}^d$:

$$f(x, y) = \langle w, \phi(x, y) \rangle.$$

Under these assumptions, a first learning task is to estimate $w$ such that the correct label has the highest score for each example $i = 1, \ldots, n$:

$$\forall i = 1, \ldots, n \ \forall y_i' \in \mathcal{Y}(x_i), y_i' \neq y_i : \ \langle w, \phi(x_i, y_i) \rangle > \langle w, \phi(x_i, y_i') \rangle. \tag{2.1}$$

If there is a feasible solution, there are infinitely many possible solutions $w$. Now we apply an idea usually referred to as *Margin Maximization* to further specify which solution is optimal. As depicted in Figure 2.2, we call

$$\delta_i(y') := \frac{\langle w, \phi(x_i, y_i) - \phi(x_i, y') \rangle}{||w||}$$

the margin between the right output $y_i$ and a wrong outputs $y'$ for input $x_i$. The margin induced by $w$ is defined as the smallest margin in the training data

$$\delta := \min_{\substack{i=1,\ldots,n \\ y_i' \in \mathcal{Y}(x_i) \\ y_i' \neq y_i}} \delta_i(y_i') = \frac{1}{||w||} \left[ \min_{\substack{i=1,\ldots,n \\ y_i' \in \mathcal{Y}(x_i) \\ y_i' \neq y_i}} \langle w, \phi(x_i, y_i) - \phi(x_i, y_i') \rangle \right].$$

Thus $\delta$ describes the smallest buffer zone between right output and wrong outputs. The key idea in margin maximization is to maximize this buffer zone to have a more robust separation between the true output and a wrong output.

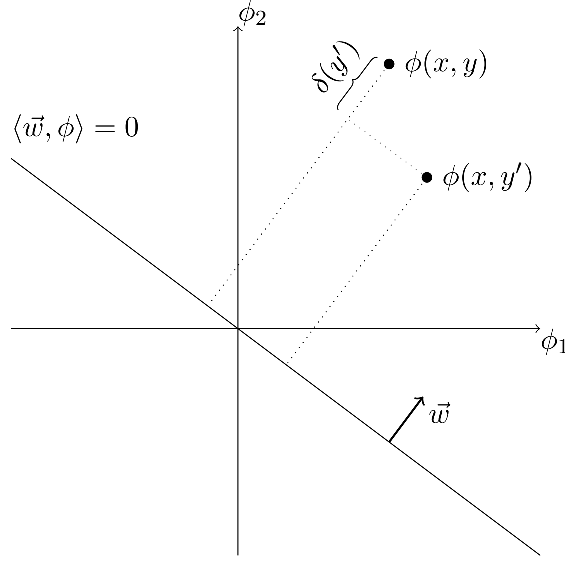



**Figure 2.2:** Illustration of max-margin learning for structured outputs: The idea is to find $\vec{w}$ that maximizes the margins $\delta$ between the correct output $y$ and the wrong outputs $y'$.

Combining these ideas we are going to derive the so-called *n-Slack formulation* of Structural Support Vector Machines. We start by maximizing the margin such that all training data is classified correctly:

$$\max_w \frac{1}{||w||} \min_{\substack{i=1,\dots,n \\ y'_i \in \mathcal{Y}(x_i) \\ y'_i \neq y_i}} \langle w, \phi(x_i, y_i) - \phi(x_i, y'_i) \rangle \tag{2.2}$$

$$\text{s.t. } \forall i = 1, \dots, n \ \forall y'_i \in \mathcal{Y}(x_i), y'_i \neq y_i : \ \langle w, \phi(x_i, y_i) \rangle > \langle w, \phi(x_i, y'_i) \rangle \tag{2.3}$$

Without loss of generality, we set $\min_{i,y'} \langle w, \phi(x_i, y_i) - \phi(x_i, y'_i) \rangle = 1$. This is possible, because for every solution $w^*$ we can construct, by rescaling $w^*$, an equally good solution that fulfills that assumption. Note that this new constraint covers all previous constraints.

$$\max_w \frac{1}{||w||} \tag{2.4}$$

$$\text{s.t. } \min_{\substack{i=1,\dots,n \\ y'_i \in \mathcal{Y}(x_i) \\ y'_i \neq y_i}} \langle w, \phi(x_i, y_i) - \phi(x_i, y'_i) \rangle = 1 \tag{2.5}$$

$$= \max_w \frac{1}{||w||} \tag{2.6}$$

$$\text{s.t. } \forall i = 1, \dots, n \ \forall y'_i \in \mathcal{Y}(x_i) \setminus y_i : \ \langle w, \phi(x_i, y_i) - \phi(x_i, y'_i) \rangle \geq 1 \tag{2.7}$$

The nonlinear constraint (2.5) can be replaced by a set of linear constraints (2.7). We can show, by contradiction, that the minimum over all the left-hand sides of

(2.7) will be 1 at the maximum, thus the equality holds: Let $w^*$ be the optimal solution of problem (2.7). Assume $\exists b > 1$ s.t. $\forall i, y_i' \langle w, \phi(x_i, y_i) - \phi(x_i, y_i') \rangle \geq b$. Then, $\frac{w^*}{b}$ is a feasible solution, too. However its objective value is $\frac{b}{||w^*||} > \frac{1}{||w^*||}$, thus $w^*$ is not optimal. This contradicts the assumption.

Tsochantaridis et al.[60] propose three refinements to this optimization problem:

- Instead of maximizing $\frac{1}{||w||}$ we can minimize $\frac{1}{2}||w||^2$, which is simpler as its partial derivative is simply $\nabla_w \frac{1}{2}||w||^2 = w$.

- Instead of forcing the margin between the correct and all incorrect outputs to be larger than 1, they introduce a general loss function $L : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}^+$ and only require the margin to be larger than the loss:

$$\forall y_i' \in \mathcal{Y}(x_i) : \langle w, \phi(x_i, y_i) - \phi(x_i, y_i') \rangle \geq L(y_i, y_i').$$

  Thus the loss induces an ordering from best outputs to worst outputs in the feature space, along the direction of the normal vector $w$.

- To cope with infeasible training data, we introduce slack variables $\xi_i$ for every example $i = 1, \ldots, n$. This allows the constraints of an example $i$ to be violated by a positive amount $\xi_i$:

$$\forall y_i' \in \mathcal{Y}(x_i) : \langle w, \phi(x_i, y_i) - \phi(x_i, y_i') \rangle \geq L(y_i, y_i') - \xi_i.$$

  However, we want to minimize the amount by which the constraints are violated, thus the objective becomes $\min_{w,\xi} \frac{1}{2}||w||^2 + C \sum_{i=1}^{n} \xi_i$ where $C > 0$ controls the trade-of between model-complexity and exact separation.

Putting it all together we get the $n$-Slack formulation of the Structural Support Vector Machine optimization problem:

$$\begin{aligned}
&\min_{w,\xi} \frac{1}{2}||w||^2 + C \sum_{i=1}^{n} \xi_i \\
&\text{s.t. } \forall y_1' \in \mathcal{Y}(x_1) : \langle w, \phi(x_1, y_1) - \phi(x_1, y_1') \rangle \geq L(y_1, y_1') - \xi_1 \\
&\qquad \vdots \\
&\text{s.t. } \forall y_n' \in \mathcal{Y}(x_n) : \langle w, \phi(x_n, y_n) - \phi(x_n, y_n') \rangle \geq L(y_n, y_n') - \xi_n
\end{aligned} \tag{2.8}$$

We can view this in the framework of regularized empirical risk minimization [60]. We use a regularizer $\Omega(w) = \frac{1}{2}||w||_2^2$ and summing the slack variables is equivalent with summing up the losses $\sum_i l(x_i, y_i; w)$ with the so-called *soft-margin loss* for Structured Prediction [57, 59, 60]

$$l(x, y; w) = \max_{y' \in \mathcal{Y}(x)} \left[ L(y, y') + \langle w, \phi(x, y') - \phi(x, y) \rangle \right]. \tag{2.9}$$

Thus we can rewrite the n-Slack formulation as

$$\min_{w} \frac{1}{n} \sum_{i=1}^{n} l(x_i, y_i; w) + \lambda ||w||_2^2, \tag{2.10}$$

with $\lambda > 0$. Note that for each $C > 0$ in the n-Slack formulation we can choose $\lambda$ so that the problems have the same minimizer.

We can check that the soft-margin loss $l(x, y; w)$ is an upper bound for the loss $L(y, \hat{y}(x))$ that we are really interested in: If $\hat{y}(x) = y$, then

$$0 = L(y, \hat{y}(x)) \leq \underbrace{l(x, y; w)}_{\geq 0}.$$

If $\hat{y} \neq y$, then $\langle w, \phi(x, \hat{y}(x)) - \phi(x, y) \rangle \geq 0$. Thus

$$L(y, \hat{y}(x)) + \langle w, \phi(x, \hat{y}(x)) - \phi(x, y) \rangle \geq L(y, \hat{y}(x))$$

and thus $L(y, \hat{y}(x)) \leq l(x, y; w)$.

You can interpret $l(x, y; w)$ as a stricter version of $L(y, \hat{y}(x))$, that not only penalizes wrong outputs, but also violations of separation margins.

### 2.2.2 Cutting Plane Optimization

We have seen two equivalent formulations of Structural Support Vector Machine training – the n-Slack formulation (2.8) and the regularized empirical risk minimization (2.10). Unfortunately, it is not feasible to directly optimize these problems because of the vast size of $|\mathcal{Y}|$. The n-Slack formulation is a quadratic program that we could solve using standard solvers. However, we have $\mathcal{O}(n|\mathcal{Y}|)$ linear constraints which makes that approach infeasible. The regularized empirical risk is a convex function over an unconstrained domain. But computing the loss function as well as the subgradient involves maximizing over all members of $\mathcal{Y}(x_i)$ for $i = 1, ..., n$, which is infeasible if done naively. Still, if we have an efficient subroutine for computing the soft margin loss for Structured Prediction, we can apply subgradient-descent methods to compute the optimal model [55]. Joachims et al. [26] present a different approach, using so-called *Cutting Plane Optimization.*

Following the cutting plane optimization framework of Teo et al. [59], we can understand the algorithm as a method that incrementally builds an approximation of the empirical risk and minimizes only this approximation.

First we note that the soft-margin loss is a convex function of $w$ and thus the empirical risk is also a convex function. A convex function $f$ is always bounded below by its first order Taylor approximation

$$f(w) \geq f(w_0) + \langle (w - w_0), \nabla_w f(w_0) \rangle.$$

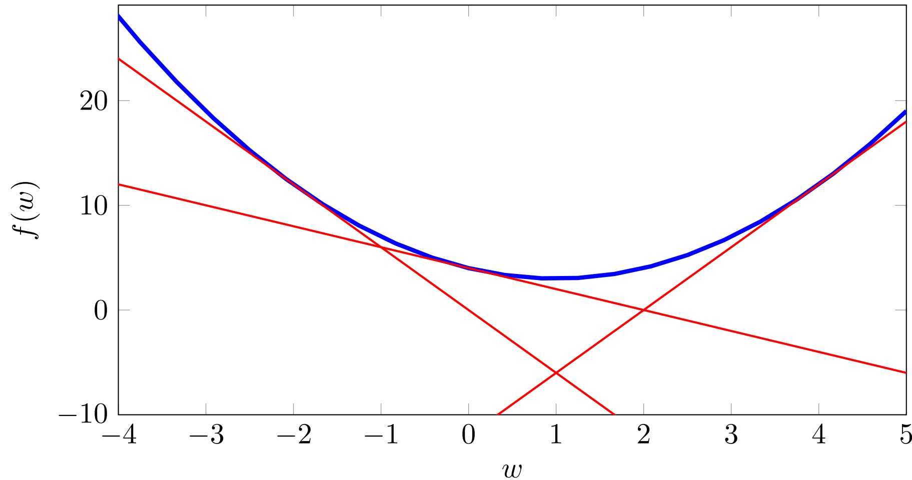



**Figure 2.3:** A convex function lower bounded by three first order approximations.

The same holds if $f$ is not differentiable; we can use any subgradient of $f(w_0)$.

We can combine a number of Taylor approximations at locations $w_0, ..., w_k$ to get a tighter approximation

$$f(w) \geq \max_{i=0,\ldots,k} \left[ f(w_i) + \langle (w - w_i), \nabla_w f(w_i) \rangle \right]$$

as depicted in Figure 2.3.

This implies Algorithm 1 which iterates two steps: Find the optimal, regularized model

$$w_i := \arg\min_w \widetilde{R}_{emp}(w) + \lambda ||w||_2^2 = \arg\min_w \max_{j<i} \left[ \langle a_i, w \rangle + b_j \right] + \lambda ||w||_2^2$$

using the current approximation of the empirical risk $\widetilde{R}_{emp}$ and then update the approximation. We refine the approximation using the first order Taylor approximation $g_i(w) = \langle a_i, w \rangle + b_i$ with $a_i = \nabla_w R_{emp}(w_i)$ and $b_i := R_{emp}(w - i) - \langle a, w_i \rangle$ developed at $w_i$. Intuitively, $w_i$ will converge when $\widetilde{R}_{emp}$ converges to $R_{emp}$ in a neighborhood of the true minimizer $w^*$.

**Algorithm 1** Cuttingplane Optimization

Choose initial $w \in \mathbb{R}^d$, initialize approximation $\widetilde{R}_{emp}(w) := 0$

**for** $i = 1, ..., \infty$ **do**

$w_i := \arg\min_w \widetilde{R}_{emp}(w) + \lambda ||w||_2^2 = \arg\min_w \max_{j<i} \left[ \langle a_i, w \rangle + b_j \right] + \lambda ||w||_2^2$

$a_i := \nabla_w R_{emp}(w_i)$, $b_i := R_{emp}(w - i) - \langle a, w_i \rangle$

Add Taylor approximation $g_i(w) = \langle a_i, w \rangle + b_i$ to $\widetilde{R}_{emp}$

**if** converged **then return** $w$

Now, let us investigate the convergence of this schema in more detail. Let $J(w) := R_{emp}(w) + \lambda||w||^2$ denote the regularized empirical risk for a model $w$. We denote by $w^*$ and $J^*$ the true minimizer and the corresponding objective value of $J(w^*)$. Let

$$J_i(w) := \widetilde{R}^{(i)}_{emp}(w) + \lambda||w||_2^2 = \max_{j \le i} [\langle a_i, w \rangle + b_j] + \lambda||w||_2^2$$

denote the approximation of the regularized empirical risk that uses all $a_j, b_j, j \le i$.

In the $i$th iteration, we first compute $w_i = \arg\min_w J_{i-1}(w)$ and then update the new approximation $J_i(w)$ by computing $a_i, b_i$.

The Taylor expansion is exact for all $w_i$, thus $J(w_j) = J_j(w_j)\ \forall j \le i$. Consequently, we obtain an upper bound on the true minimizer

$$J^* \le \min_{j \le i} J_j(w_j) =: J_i^+.$$

The series $J_i^+$ is monotonically decreasing, as we minimizer over a larger set with each iteration.

Furthermore $J_i^- := J_{i-1}(w_i) \le J^*$ lower bounds the optimal solution, as $J_{i-1}(w) \le J(w)$ for all $w$ and thus the inequality particularly holds for the respective minimums. Additionally, $J_i^-$ is a monotonically increasing series, as $J_i(w) \ge J_{i-1}(w)$ for all $w$.

Hence, the expression $J^+ - J^-$ quantifies the largest approximation error we could produce. We can upper bound this error by $J^+ - J^- \le J_i(w_i) - J_{i-1}(w_i)$ and stop, once this upper bound is below a user-specified tolerance $\varepsilon > 0$.

Now we can write down the algorithm presented by Joachims et al. [26]. Instead of weighting the regularizer $||w||^2$ with $\lambda$ as in [59, 55], we stay in the traditional notation for Support Vector Machines as in [26] and weight the empirical risk with $C$ and the regularizer with $\frac{1}{2}$. We will begin by investigating the specific Taylor approximations that arise in Structural SVM training: for the first order approximation we have to compute the subgradient of the empirical risk $R_{emp} = \frac{1}{n}\sum_{i=1}^{n} l(x_i, y_i; w)$ at $w_i$ with $l(x, y; w)$ defined according to (2.9). The soft-margin loss is a piecewise linear function of $w$, the subgradient depends only on the largest element of the maximization, thus it holds that $a_i = \frac{1}{n}\sum_{i=1}^{n} \phi(x_i, \hat{y}_i) - \phi(x_i, y_i)$ with

$$\hat{y}_i := \arg \max_{y' \in \mathcal{Y}(x_i)} \left[ L(y_i, y') + \langle w_i, \phi(x_i, y') - \phi(x_i, y_i) \rangle \right]. \tag{2.11}$$

Consequently we see that $b_i = \frac{1}{n}\sum_{i=1}^{n} L(y_i, \hat{y}_i)$. In order to store a new first order approximation we just have to remember $\hat{y} = (\hat{y}_1, ..., \hat{y}_n)$. Thus we can store $\widetilde{R}_{emp}$ by storing in a set $\mathcal{W} \subseteq \mathcal{Y}^n$ all $\hat{y}$ that we have computed so far. The size of $\mathcal{W}$ will be incremented by 1 each iteration.

**Algorithm 2** Cuttingplane Optimization For Structural SVMs

**Require:** Data Set $\mathcal{D}$, $C > 0$, $\varepsilon > 0$,

$\mathcal{W} := \emptyset$

**repeat**

$w, \xi := \arg\min_{w,\xi\geq 0} \ \frac{1}{2}||w||_2^2 + C\xi$

s.t. $\frac{1}{n}\sum_{i=1}^{n}\langle w, \phi(x_i, y_i) - \phi(x_i, \bar{y}_i)\rangle \geq \frac{1}{n}\sum_{i=1}^{n} L(y_i, \bar{y}_i) - \xi \ \forall \bar{y} \in \mathcal{W}$

**for** $i = 1, ..., n$ **do**

$\hat{y}_i := \arg\max_{y' \in \mathcal{Y}(x_i)} L(y_i, y') + \langle w_i, \phi(x_i, y')\rangle$ // Oracle Call

$\mathcal{W} := \mathcal{W} \cup \{\hat{y}\}$

**until** $\frac{1}{n}\sum_{i=1}^{n}\langle w, \phi(x_i, y_i) - \phi(x_i, \hat{y}_i)\rangle \geq \frac{1}{n}\sum_{i=1}^{n} L(y_i, \hat{y}_i) - (\xi + \varepsilon)$

As hinted above we need to have an efficient subroutine to compute the $\hat{y}_i$, which we will refer to as the *most violated constraints*. We call the subroutine computing the most-violated constraints *Separation Oracle*.

Instead of solving the problem $\min_w \frac{1}{2}||w||_2^2 + C \cdot \widetilde{R}_{emp}(w)$ as an unconstrained optimization problem, we can solve a quadratic program [59], given the current working set $\mathcal{W}$:

$$\begin{aligned} \min_{w,\xi\geq 0} \ & \frac{1}{2}||w||_2^2 + C\xi \\ \text{s.t.} \ & \frac{1}{n}\sum_{i=1}^{n}\langle w, \phi(x_i, y_i) - \phi(x_i, \bar{y}_i)\rangle \geq \frac{1}{n}\sum_{i=1}^{n} L(y_i, \bar{y}_i) - \xi \ \forall \bar{y} \in \mathcal{W}. \end{aligned} \tag{2.12}$$

Notice the similarities with the n-Slack formulation (2.8). In contrast there is only one slack variable, which is why Joachims et al. call this formulation the 1-Slack formulation [26]. We call each $\bar{y} \in \mathcal{W}$ a cutting plane, as in Equation (2.12), the $\bar{y}$ is responsible for one linear inequality constraint, that cuts the space into a feasible and an infeasible halfspace. The complete algorithm can be found in Algorithm 2.

In order to apply the Structural SVM to a specific learning problem, one has to

- define a joint feature mapping $\phi(x, y)$ suitable for the given problem.
- define a suitable loss function $L(y, y') : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}_0^+$.
- find a method to solve the inference problem

$$\widehat{y}(x) = \arg\max_{y \in \mathcal{Y}(x)} \langle w, \phi(x, y)\rangle.$$

This is generally not trivial when we cannot just iterate all members of $\mathcal{Y}$ for the given problem of interest. However, for many formulations of $\phi$ one can find polynomial time algorithms that solve the problem by incorporating knowledge on the structure of $\mathcal{Y}(x)$ and $\phi$ [60, 26, 57].

- find a separation oracle, i.e. a method to find the most violated constraint for an example $(x, y)$:

$$\hat{y} = \arg \max_{y' \in \mathcal{Y}(x)} \langle w, \phi(x, y') \rangle + L(y, y').$$

Similar to finding a prediction, this problem is generally not trivial for large $|\mathcal{Y}|$. To find an efficient algorithm for a given problem, one has to incorporate knowledge on the structure of $\mathcal{Y}$, the structure of $\phi$ and the structure of $L$. Ideally, one can use the same optimization routine used for the inference problem. To do so, the loss function needs to fulfill certain requirements.

In the next section we will see examples of how to specify instances of Structural Support Vector Machines for different problems.

## 2.3 Two Example Instances of Structural Support Vector Machines

In this section we will see two examples of structured outputs tasks that can be solved using the Structural Support Vector Machine. Both tasks arise in Natural Language Processing, have token sequences as inputs (c.f. Example 2.1.1) and have outputs that can be represented by discrete output structures.

### 2.3.1 Sequence Tagging

The first task is sequence tagging which was already briefly introduced in Example 2.1.3. Given an input sequence $x$ we want to assign each token a label or tag from the set of tags $\mathcal{T}$. Thus $y$ is a sequence of tags of same length as $x$.

A naive approach is to assign the tags for each token individually. But Joachims et al. [26] propose to not only model the dependencies between each token $x_i$ and tag $y_i$, but also the dependencies between each $y_i$ and $y_{i-1}$. They assign a feature vector $\phi(x_i) \in \mathbb{R}^d$ to each token that has binary features that describe the whole token, its prefixes and suffixes etc. Then the features that model the dependencies

between input and output

$$\phi_1(x, y) = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \phi(x) \\ 0 \\ \vdots \\ 0 \end{pmatrix} \in \mathbb{R}^{d \times |\mathcal{T}|}$$

are created by stacking the features of $x$ into a position according to $y$.

The features that model the dependencies between two consecutive tags $y_i$ and $y_{i-1}$, $\phi_2(y_i, y_{i-1}) \in \mathbb{R}^{|\mathcal{T}|^2}$, are defined according to

$$[\phi_2(y_i, y_{i-1})]_{s,t} = 1_{(y_i = s)} 1_{(y_{i-1} = t)} \; \forall s, t \in \mathcal{T}.$$

Thus each possible transition from one tag to the other has a binary feature. The $\phi$ used for the Structural SVM is

$$\phi([x_1, ..., x_l], [y_1, ..., y_l]) = \begin{pmatrix} \phi_1(x_1, y_1) \\ 0 \end{pmatrix} + \sum_{i=2}^{l} \begin{pmatrix} \phi_1(x_i, y_i) \\ \phi_2(y_i, y_{i-1}) \end{pmatrix}.$$

Thus the score $\langle w, \phi(x, y) \rangle$ gives weight to both the compatibility of a token and a tag and the compatibility of the transitions between tags.

To compute the highest scored sequence $\hat{y}(x) = \arg\max_{y \in \mathcal{Y}(x)} \langle w, \phi(x, y) \rangle$ out of the $|\mathcal{T}|^l$ possible sequences, we rely on Viterbi's algorithm, which is a dynamic programming algorithm that needs $\mathcal{O}(|\mathcal{T}|^2 * l)$ runtime.

Joachims et al. propose to use the number of misclassified tokens $L(y, y') = \sum_{i=1}^{l} 1_{(y_i \neq y'_i)}$ for the loss function. Since the loss function is also decomposable over the transitions in $y$, we can also use the Viterbi algorithm in the separation oracle.

Experiments show, that the Structural SVM outperforms a generative Hidden Markov Model and has a performance insignificantly better than discriminative Conditional Random Fields at Named Entity Recognition tasks [60]. Tsochantaridis et al. claim that this demonstrates the benefit of a large margin approach, particularly regarding the ability to generalize from training to test data.

### 2.3.2 Constituency Parsing

The second task is constituency parsing, where given a token sequence $x = [x_1, ..., x_n]$ we want to predict its inner syntactical structure using a context free grammar and return the corresponding parse tree.

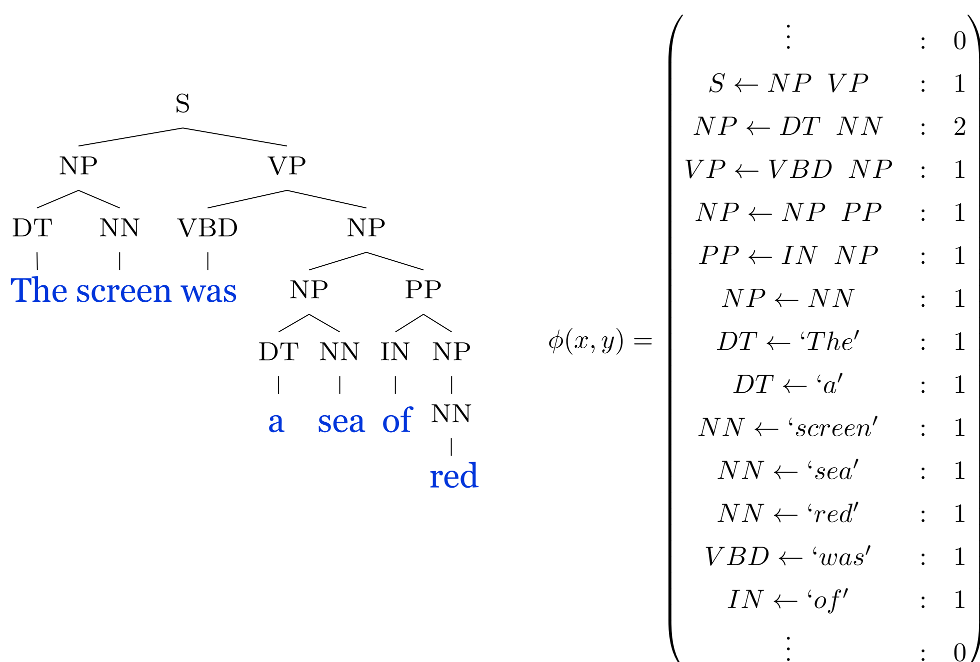



**Figure 2.4:** An example constituency parse tree with corresponding feature vector

In Figure 2.4 we see an example on a constituency parse tree. Notice how the input is part of the output, since the leafs of the parse tree are the tokens of the input. I will not give a formal definition of the corresponding output space. It suffices to note that the discrete symbols that form the parse tree describe which rules of the context-free grammar were used to generate which spans of the token sequence. E.g. in Figure 2.4 the rule $S \leftarrow NP\ \ VP$ was used to generate the whole sequence and the rule $NP \leftarrow DT\ \ NN$ was used to generated the subsequence "*a sea*".

In Natural Language Parsing, there are two kinds of rules: On the one hand, there are rules that generate a non-terminal symbol from one or many non-terminal symbols. On the other hand, there are rules that generate a non-terminal symbol from one terminal symbol, i.e., token from the input. This separates NL parsing from general context-free parsing, where other kinds of rules would be possible. The hard separation of rules into two categories is due to the conventions used for annotating the corpus, i.e. creating the treebank: Particularly, each token is generated from a part-of-speech non-terminal symbol.

Joachims et al. [26, 60] as well as Taskar et al. [57] propose to solve the problem using Structural Support Vector Machines. To obtain a context-free grammar they rely on annotated data, in this case a treebank, and simply use all rules observed

in the data [26]. They propose a $\phi(x, y)$ that has one feature for every rule in the context free grammar. For a rule $r$ the value of $[\phi(x, y)]_r$ is the number of times the rule was used in the tree $y$ to generate $x$.

Thus, the prediction problem $\hat{y}(x) = \arg\max_{y \in \mathcal{Y}(X)} \langle w, \phi(x, y) \rangle$ has to construct, using rules from the grammar, the parse tree that has the highest score. Each rule $r$ that is used adds an amount according to the corresponding weight $w_r$ to the total score. The arg-max can be computed in polynomial time using the CYK algorithm for weighted context-free grammars. The CYK algorithm is a dynamic programming approach that computes the best parse tree in $\mathcal{O}(n^3)$ where $n$ is the length of the input sequence.

The simplest choice of a loss function is $L(y, y') = 1_{y \neq y'}$. In order to solve the separation oracle problem efficiently, they rely on a modified CYK parser that returns the output with the second largest score value. This can still be done in polynomial time [24]. Other loss functions like the F1-measure [60] or F1-related measures [57] are possible, but I will not discuss them here.

Experiments show that the max-margin approach outperforms a probabilistic context free grammar, a generative approach [60]. While the train error is similar, the test error for Structural SVM training is significantly better. We should note that the Structural SVM particularly outperforms the generative model with respect to the loss function used for training: If we choose the 0/1 loss, the accuracy is superior, if we choose the F1-loss, the F1-score is superior. This speaks for the ability to adopt to a particular use case or application domain.

# 3 Kernels for Implicit Feature Maps

In the previous chapter we have seen that Structural Support Vector Machines require you to explicitly define a feature map $\phi(x, y)$ for an input-output pair. In this section we discover a class of functions, the so-called kernel functions, that allows us to use feature maps that are only implicitly defined. We will look into the mathematical foundation of kernels and lay the theoretical justification for the next chapter of this work which will use kernel-induced feature maps for structural prediction with the Structural Support Vector Machine.

## 3.1 Kernels and Implicit Feature Maps

Let us first define what a kernel function is.

**Definition 3.1.1** We call a function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ a *kernel*, if $k$ can be expressed as $k(x, x') = \langle \phi(x), \phi(x') \rangle_{\mathcal{H}}$ for a feature map $\phi : \mathcal{X} \to \mathcal{H}$ where $\mathcal{H}$ is a Hilbert space.

**Example 3.1.1** Let $\mathcal{X} = \mathbb{R}^n$. Then $k(x, x') = \langle x, x' \rangle^p$ is the so-called *polynomial kernel*. We set $\phi : \mathbb{R}^n \to \mathbb{R}^{(n^p)}$ where $[\phi(x)]_s = \prod_{i=1}^{p} x_{s_i}$ with $s \in \{1, ..., n\}^p$ has one feature for every possible product of $p$ features of $x$.

Using kernels is a popular technique in Machine Learning: Many ML algorithms rely on linear dot products of feature vectors of the data. Whenever the raw features do not yield good results, one way to make the algorithm work is to apply a transformation of features. E.g. the Support Vector Machine tries to find a linear separation of two classes in the feature space. When the problem is not linearly separable, a transformation into a different feature space might make the problem linearly separable. The reason that kernels are popular in Machine Learning is that

a kernel implicitly performs this feature transformation and applies the dot product in the transformed feature space, all in one step. Considering Example 3.1.1, it is obvious that computing the kernel is less computationally expensive than computing the transformation $\phi(x)$ and then the dot product of the transformed vectors. Particularly with respect to memory usage, which grows exponentially with the degree $p$ when we apply the transformation. This also demonstrates the strength of kernel methods: We can implicitly transform into higher dimensional or even infinitely dimensional Hilbert spaces, while the runtime of the learner does not depend on the target dimensionality, but only on the complexity of the kernel.

On first sight, the class of functions that are kernels seem restricted. However there are results that show that a large group of functions are kernels.

### 3.1.1 Mercer's Theorem

Let us first consider the class of Mercer kernels, following the definitions and theorems given in [53, 66]. Mercer's Theorem states:

**Theorem:** Let $\mathcal{X} \subseteq \mathbb{R}^n$ be a compact set and $(\mathcal{X}, \mu)$ be a finite measure space with a measure[1] $\mu$. Let $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a symmetrical function, i.e. $k(x, x') = k(x', x)$. If the integral operator $T_k : L_2(\mathcal{X}) \to L_2(\mathcal{X})$

$$T_k f(x) = \int_{\mathcal{X}} k(x, y) f(y) d\mu(y) \tag{3.1}$$

is positive for all $f \in L_2$, i.e.

$$\int_{\mathcal{X}} \int_{\mathcal{X}} k(x, y) f(y) f(x) d\mu(y) d\mu(x) \geq 0,$$

then for the normalized eigenfunctions[2] of $T_k$, $\psi_j \in L_2(\mathcal{X})$ with eigenvalues $\lambda_j > 0$, sorted in non-increasing order it holds that

$$k(x, y) = \sum_{j=1}^{N} \lambda_j \psi_j(x) \psi_j(y)$$

with either $N \in \mathbb{N}$ or $N = \infty$. In the latter case, the series converges absolutely and uniformly for all $x, y$ with $\mu(x) > 0$ and $\mu(y) > 0$.

The consequence of this theorem is that every function that fulfills Mercer's condition, i.e. the condition of the theorem, is a kernel function with an implicit mapping

$$[\phi(x)]_j = \sqrt{\lambda_j} \psi_j(x), \ j = 1, ..., N$$

[1] We can think of measures as unnormalized probability density functions $\mu : \mathcal{X} \to \mathbb{R}^+$

[2] An eigenfunction is a function $\psi_j(x)$ such that $T_k \psi_j(x) = \lambda_j \psi_j(x)$ for the corresponding eigenvalue $\lambda_j$.

This holds also if $N = \infty$. But uniform convergence also implies that given $\varepsilon > 0$ there exists $\tilde{N} \in \mathbb{N}$ such that $k$ can be approximated within $\varepsilon$ accuracy as a dot product using only the first $\tilde{N}$ eigenfunctions [53].

### 3.1.2 Reproducing Kernel Hilbert Spaces

Mercer Kernels, i.e. kernel functions that obey Mercer's condition, are a large subgroup of kernels. However, in the context of this work we want to consider discrete structured input spaces $\mathcal{X} \not\subseteq \mathbb{R}^n$. Thus we have to consider another class of kernels: The class of positive definite kernels [23].

**Theorem:** A symmetric function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a kernel, if for each $n \in \mathbb{N}$, for all $x_i \in \mathcal{X}$ for $i = 1, ..., n$ the Gram matrix $K_{ij} = k(x_i, x_j)$ is positive definite.

To see that this theorem holds, first, we define a feature mapping $\phi : \mathcal{X} \to \mathbb{R}^{\mathcal{X}}$ that maps each $x \in \mathcal{X}$ to a function from $\mathcal{X} \to \mathbb{R}$ such that $\phi(x) = k(x, \cdot)$ or $[\phi(x)](x') = k(x, x')$ respectively.

For the vector space $\mathcal{H} = \overline{span}\{\phi(x) \mid x \in \mathcal{X}\}$, the completed linear span, we define a dot product for two $f, g \in \mathcal{H}$, $f := \sum_{i=1}^{m} \alpha_i k(x_i, \cdot)$ and $g := \sum_{i=1}^{m'} \beta_i k(x'_i, \cdot)$ with $\alpha_i, \beta_i \in \mathbb{R}$ and $x_i \in \mathcal{X}$, $i = 1, ..., m$, as well as $x'_i \in \mathcal{X}$ for $i = 1, ..., m'$ as

$$\langle f, g \rangle_{\mathcal{H}} = \sum_{i=1}^{m} \sum_{j=1}^{m'} \alpha_i \beta_j k(x_i, x'_j).$$

Let us check that $\langle \cdot, \cdot \rangle_{\mathcal{H}}$ is a dot product: Since $k$ is a symmetric function, $\langle \cdot, \cdot \rangle_{\mathcal{H}}$ is symmetric. It is obviously linear. It is also positive definite, since the positive definiteness of $k$ implies that $\langle f, f \rangle_{\mathcal{H}} = \sum_{i=1}^{m} \sum_{h=1}^{m} \alpha_i \alpha_j k(x_i, x_j) \geq 0$.

Thus we have shown that it fulfills all the requirements for a dot product. One can show, that $\mathcal{H}$ is indeed a Hilbert space, i.e. a real vector space with an inner product, for details see [18]. We call the space Reproducing Hilbert Space, as for each $x \in X$, $f \in \mathcal{H}$ the reproducing property $\langle k(x, \cdot), f \rangle_{\mathcal{H}} = f(x)$ holds.

## 3.2 Some Important Kernels

In this section I want to present some important kernels that are widely used for Machine Learning tasks.

### 3.2.1 Gaussian Kernel

First I want to discuss the Gaussian kernel, or RBF kernel, since it is very widely used and demonstrates how powerful implicit feature maps induced by kernels really are.

For a compact subset $\mathcal{X} \subseteq \mathbb{R}^d$ the kernel is defined as $k(x, x') = e^{-\gamma||x-x'||_2^2}$. First we have to verify that the RBF kernel is indeed a kernel. We rewrite

$$e^{-\gamma||x-x'||_2^2} = e^{-\gamma||x||^2} e^{-2\gamma\langle x,x'\rangle} e^{-\gamma||x'||_2^2}.$$

We use, without proof, the property that for arbitrary real-valued functions $f$ a function $f(x)k(x, x')f(x')$ is a kernel if $k$ is a kernel. Thus we have to show that $e^{-\gamma\langle x,x'\rangle}$ is a kernel. We can apply the power series and see that

$$e^{-\gamma\langle x,x'\rangle} = \sum_{i=0}^{\infty} \frac{(-\gamma)^i \langle x, x'\rangle^i}{i!} \tag{3.2}$$

We use, without proof, the property that (infinite) linear combinations of kernels are kernels. We have seen that the polynomial dot product is a kernel, thus the Gaussian kernel is indeed a kernel. We even see one possible, infinite dimensional feature mapping. For each $i$ in (3.2) we can define

$$\phi_i(x) = \sqrt{\frac{2(-\gamma)^i}{i!}} \phi_{poly}^{(i)}(x)$$

where $\phi_{poly}^{(i)}(x)$ is the implicit feature map of the polynomial kernel of degree $i$ discussed in Example 3.1.1. Now we can write down the infinite feature map

$$\phi(x) = e^{-\gamma||x||_2^2} \cdot [\phi_0(x)^t, \phi_1(x)^t, \phi_2(x)^t, ...]^t$$

Now let us consider the dimensionality of the implicit feature map $\phi$ in the framework of Reproducing Kernel Hilbert spaces: We see that the completed linear span $\mathcal{H} = \overline{span}\left\{\left(x' \mapsto e^{-\gamma||x-x'||_2^2}\right) \mid x \in \mathcal{X}\right\}$ is infinitely dimensional since $\mathcal{X}$ is a compact subspace of $\mathbb{R}^d$ and we cannot express a member of $\phi(x) \in \mathcal{H}$ as a linear combination of members $\phi(x_i) \in \mathcal{H}$ with $x_i \neq x$.

If we apply Mercer's theorem we gain more insight into the feature space $\mathcal{H}$. Rahimi and Recht [50, 48, 49] show that the kernel $k$ can be written as

$$k(x, x') = \int_{\mathbb{R}^d} \int_{-\pi}^{\pi} \phi_{z,b}(x)\phi_{z,b}(x')\lambda_{z,b} \mathrm{d}b \, \mathrm{d}z$$

with $\phi_{z,b}(x) = \cos(\langle z, x\rangle + b)$ and $\lambda_{b,z} = \frac{1}{2\pi} \frac{1}{\sqrt{(2\pi\gamma)^d}} e^{-\gamma||z||}$. Thus we see that we again have a infinite dimensional feature map defined indexed by all elements of

$\mathbb{R}^d \times [-\pi, \pi]$. While this does not seem useful at first sight, it lends itself to sampling. Rahimi and Recht demonstrate that sampling these features according to the probability density $p(z, b) \mapsto \lambda_{z,b}$ approximates the RBF-feature map well using less than 1,000 of the infinitely many features [48].

### 3.2.2 String Kernel

Now I want to introduce a kernel for discrete structures, token sequences with a dictionary of possible tokens $\Sigma$ to be more precise. Lohdi et al. propose the use of a string subsequence kernel [34], that compares all substrings up to a specified length $n$ in two sequences. That is why I will call this kernel *n-Gram kernel* from now on.

For a token sequence $x$ let $x_i$ denote the $i$th token for $i = 1, ..., |x|$ where $|x|$ is the number of tokens in the token sequence. Let $x[i : j]$ denote the substring $[x_i \; x_{i+1} \; ... \; x_j]$ of size $j - i + 1$. A first idea is to use a kernel with features for each possible substring $s \in \Sigma^n$ such that $[\phi(x)]_s$ specifies the number of occurrences of $s$ in $x$. The authors go one step further and allow matching even when gaps are present in the subsequence. For $I = (i_1, ..., i_l)$ with $i_k < i_{k+1}$ we denote by $x[I] := x_{i_1} \, x_{i_2} \ldots x_{i_l}$ a subsequence of length $|x[I]| = i_l - i_1 + 1$. For a specified decay factor $\lambda \in [0, 1]$ they define

$$[\phi(x)]_s = \sum_{I \text{ s.t. } x[I]=s} \lambda^{|x[I]|}.$$

Thus a exact matching of a subsequence contributes more than an inexact matching where gaps are present in $x$.

Lohdi et al. present a Dynamic Programming algorithm that computes the value of the kernel $k(x, x') = \langle \phi(x), \phi(x') \rangle$ for two sequences $x$,$x'$ in $\mathcal{O}(n \cdot |x| \cdot |x'|)$. Also they propose to normalize the kernel $k'(x, x') = \frac{k(x,x')}{\sqrt{k(x,x)k(x',x')}}$.

The authors demonstrate the usefulness of their approach for a text classification task where classification using their kernel outperformed standard linear models with n-gram features for the right choice of $n$ and $\lambda$ [34].

### 3.2.3 Tree Kernel

Duffy and Collins [7] propose to use a kernel for natural language parse trees like the ones discussed in Section 2.3.2. They propose to compare two parse trees $x$ and $x'$ by comparing all possible tree fragments or subtrees. A subtree is a connected

subgraph with more than one node, such that for each selected node, either all productions[3] or none of the productions are included.

Let $N(x)$ denote the set of nodes in $x$ respectively. For a subtree $i$ we define the indicator function $I_i(x,n)$ that indicates whether the tree $x$ has the subtree $i$ rooted in node $n \in N$. Then we define a kernel by its feature map

$$[\phi(x)]_i = \sum_{n \in N(x)} I_i(x,n)$$

Duffy and Collins show, that the kernel $k(x,x') = \langle \phi(x), \phi(x') \rangle$ can be computed in $\mathcal{O}(|x|\,|x'|)$ via the following equation

$$k(x,x') = \sum_{n \in N(x)} \sum_{n' \in N(x')} \sum_i I(x,n) \cdot I(x',n') = \sum_{n \in N(x)} \sum_{n' \in N(x')} C(n,n')$$

where $C(n,n') = \sum_i I(x,n) \cdot I(x',n')$. $C(n,n')$ can be computed recursively:

- $C(n,n') = 0$ if the productions at $n$ and $n'$ differ
- $C(n,n') = 1$ if the non-terminal symbols at $n$ and $n'$ are the same and both $n$ and $n'$ are pre-terminal symbols, i.e. non-terminal symbols that produce exactly one terminal symbol.
- otherwise the non-terminal symbols at $n$ and $n'$ are the same and $n$ and $n'$ are not both pre-terminal symbols. We denote by $nc(n)$ the number of children of node $n$ and by $ch(n,j)$ the $j$th child of node $n$. Then

$$C(n,n') = \prod_{j=1}^{nc(n)} [1 + C(ch(n,j), ch(n',j))]$$

Duffy and Collins also propose to normalize the kernel such that larger kernels do not produce larger kernel values. Also, they observe that the kernel value between two copies of the same tree is extremely large while, in comparison, the kernel value between two different trees is much smaller. This is due to the exponential number of subtrees in a tree. They propose to down-weight larger subtree matches for the kernel computation using a parameter $\lambda \in [0,1]$, such that

$$[\phi(x)]_i = \sum_{n \in N(x)} \sqrt{\lambda^{|i|}} I_i(x,n)$$

where $|i|$ is the size of a subtree $i$. We can compute this kernel using a modified definition of $C(n,n')$:

[3] A production is a set of non-terminal or terminal symbols produced by a rule of the context free grammar. Thus the production of a node correspond to the respective children in the tree.

- $C(n, n') = 0$ if the productions at $n$ and $n'$ differ
- $C(n, n') = \lambda$ if the non-terminal symbols at $n$ and $n'$ are the same and both $n$ and $n'$ are pre-terminal symbols, i.e. non-terminal symbols that produce exactly one terminal symbol.
- $C(n, n') = \lambda \prod_{j=1}^{nc(n)} [1 + C(ch(n, j), ch(n', j))]$ otherwise

The authors demonstrate the use of this kernel for both Natural Language Parsing and Sequence Tagging. In their experiments, they use a Structured Perceptron, a learning algorithm related to the Structural SVM, to identify the best possible output structure from a set of candidate structures generated by a generative parse model or tagging model [7, 9]. Using this kernel-based reranking approach they were able to increase the performance of the generative models substantially.

# 4 Towards Hassle-Free Structured Prediction

Structural Prediction as we have seen it in Chapter 2 requires a lot of algorithmic work. One has to design feature maps, find compatible inference algorithms and efficient separation oracle algorithms.

In this chapter I present a method that eliminates some of these challenges, while – to be fair – also creating some new challenges. To do so, I will combine the things discussed in the previous chapters: We are going to build on Structural Support Vector Machines, but instead of using explicit, hand-designed feature maps we will use implicit feature maps induced by kernel functions. Using kernels changes the inference problem and the separation oracle problem: these problems will become so-called Pre–Image problems, which are generally tricky to solve.

I tackle the pre–image problem by defining a class of kernels for discrete output structures for which the pre-image problem can be solved. Thus I obtain an algorithm for discrete structured output prediction that requires less engineering than the Structural SVM.

## 4.1 Kernelized Structural Support Vector Machines

In this section I will introduce the Kernelized Structural Support Vector Machine that uses implicit features. In order to see how kernels can be applied we have to derive the Lagrangian dual program of the Structural SVM optimization problem. Solving the dual instead of the primal changes the way we store learned model, i.e. the optimal solution of the dual problem. Instead of one weight vector we will store a linear combination of constraints. This affects the inference problem and the

separation oracle computations: In order to obtain a prediction or a most-violated constraint, one has to solve a so-called pre-image problem for the kernel used.

### 4.1.1 Dual Program

Like with Support Vector Machines, we can also derive and solve the dual problem. First let us consider the properties of the primal optimization problem (2.12). The objective is a quadratic function that is convex. The inequalitiy constraints are affine functions. Thus the primal problem is a convex optimization problem. We can construct a feasible solution $w = \vec{0}$ and $\xi = \Delta$ for $\Delta = \max_{i,\bar{y}} L(y_i, \bar{y}_i)$. Thus we can apply Slater's theorem that guarantees that a) the Karush-Kuhn-Tucker (KKT) conditions provide a necessary and sufficient optimality criterion and b) strong duality holds for the Lagrangian dual program.

First, let us write out the KKT conditions for the primal problem. The Lagrangian

$$\begin{aligned}\mathcal{L}(w, \xi, \alpha, \beta) = &\frac{1}{2}||w||_2^2 + C\xi \\ &+ \sum_{\bar{y} \in \mathcal{W}} \alpha_{\bar{y}} \left[ \frac{1}{n} \sum_{i=1}^{n} L(y_i, \bar{y}_i) - \xi - \frac{1}{n} \sum_{i=1}^{n} \langle w, \phi(x_i, y_i) - \phi(x_i, \bar{y}_i) \rangle \right] - \beta\xi \quad (4.1)\end{aligned}$$

has Lagrangian multipliers $\alpha_{\bar{y}}$ for each constraint and one multiplier[1] $\beta$ for the constraint $\xi \geq 0$. As noted above, the KKT conditions provide a sufficient and necessary condition for optimality: A solution $(w^*, \xi^*)$ is optimal if and only if there exist Lagrangian multipliers $\alpha_{\bar{y}}$ and $\beta$ such that the following Karush-Kuhn-Tucker conditions hold:

- **Stationarity:** $\nabla_w \mathcal{L}(w^*, \xi*, \alpha, \beta) = 0$. Consequently we have

$$w^* = \sum_{\bar{y} \in \mathcal{W}} \alpha_{\bar{y}} \frac{1}{n} \sum_{i=1}^{n} \phi(x_i, y_i) - \phi(x_i, \bar{y}_i). \quad (4.2)$$

  Following the terminology familiar from Support Vector Machines, we call every $\bar{y} \in \mathcal{W}$ with $\alpha_{\bar{y}} \neq 0$ *Support Vector*. In contrast to SVMs, a support vector is not a single structure, but a linear combination of multiple structures.

- **Stationarity II:** $\nabla_\xi \mathcal{L}(w^*, \xi^*, \alpha, \beta) = 0$. Thus

$$C - \sum_{\bar{y} \in \mathcal{W}} \alpha_{\bar{y}} - \beta = 0. \quad (4.3)$$

[1] This multiplier is missing in [26]

- **Primal Feasibility:** $\forall \bar{y} \in \mathcal{W}: \; g_{\bar{y}}(w^*, \xi^*) \leq 0$ with

$$g_{\bar{y}}(w^*, \xi^*) := \frac{1}{n}\sum_{i=1}^{n} L(y_i, \bar{y}_i) - \xi^* - \frac{1}{n}\sum_{i=1}^{n}\langle w^*, \phi(x_i, y_i) - \phi(x_i, \bar{y}_i)\rangle$$

- **Primal Feasibility II:** $\xi^* \geq 0$

- **Dual Feasibility:** $\alpha_{\bar{y}} \geq 0 \; \forall \bar{y} \in \mathcal{W}$ and $\beta \geq 0$.

- **Complementary Slackness:** $\alpha_{\bar{y}} \cdot g_{\bar{y}}(w^*, \xi^*) = 0 \; \forall \bar{y} \in \mathcal{W}$. Thus if $\alpha_{\bar{y}} > 0$ we know that $g_{\bar{y}}(w^*, \xi^*) = 0$. Consequently we have that

$$\alpha_{\bar{y}} > 0 \Rightarrow \xi^* = \frac{1}{n}\sum_{i=1}^{n} L(y_i, \bar{y}_i) - \frac{1}{n}\sum_{i=1}^{n}\langle w^*, \phi(x_i, y_i) - \phi(x_i, \bar{y}_i)\rangle. \tag{4.4}$$

- **Complementary Slackness II:** $\beta \cdot \xi^* = 0$. Thus if $\beta > 0$ we have zero slack $\xi^* = 0$ and if we have positive slack $\xi^* > 0$ we have $\beta = 0$.

Now we can begin obtaining the dual problem by formulating the Lagrangian dual problem:

$$\max_{\alpha \geq 0, \beta \geq 0} \inf_{w, \xi} \mathcal{L}(w, \xi, \alpha, \beta). \tag{4.5}$$

Slater's condition holds, thus strong duality holds i.e., the solutions of problem (2.12) and (4.5) have zero duality gap i.e., are equal. Plugging $w^*$ from Equation (4.2) into the Lagrangian dual and plugging in $\sum_{\bar{y} \in \mathcal{W}} \alpha_{\bar{y}} + \beta - C = 0$ from Equation (4.3) while adding constraints on $\alpha$ yields

$$\begin{aligned} \max_{\alpha, \beta} \; & -\frac{1}{2}\sum_{\bar{y} \in \mathcal{W}}\sum_{\bar{y}' \in \mathcal{W}} \frac{\alpha_{\bar{y}}\alpha_{\bar{y}'}}{n^2}\left\langle \sum_{i=1}^{n}\left(\phi(x_i, y_i) - \phi(x_i, \bar{y}_i)\right), \sum_{i=1}^{n}\left(\phi(x_i, y_i) - \phi(x_i, \bar{y}'_i)\right)\right\rangle \\ & + \sum_{\bar{y} \in \mathcal{W}} \alpha_{\bar{y}} \frac{1}{n}\sum_{i=1}^{n} L(y_i, \bar{y}_i) \\ \text{s.t.} \; & \sum_{\bar{y} \in \mathcal{W}} \alpha_{\bar{y}} + \beta - C = 0 \text{ and } \alpha_{\bar{y}} \geq 0 \; \forall \bar{y} \in \mathcal{W}, \beta \geq 0. \end{aligned} \tag{4.6}$$

For notational brevity we define

$$\begin{aligned} H(\bar{y}, \bar{y}') &= \frac{1}{n^2}\left\langle \sum_{i=1}^{n}\left(\phi(x_i, y_i) - \phi(x_i, \bar{y}_i)\right), \sum_{i=1}^{n}\left(\phi(x_i, y_i) - \phi(x_i, \bar{y}'_i)\right)\right\rangle \\ &= \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\left[\langle \phi(x_i, y_i), \phi(x_j, y_j)\rangle - \langle \phi(x_i, y_i), \phi(x_j, \bar{y}'_j)\rangle\right. \\ &\qquad \left. - \langle \phi(x_i, \bar{y}_i), \phi(x_j, y_j)\rangle + \langle \phi(x_i, \bar{y}_i), \phi(x_j, \bar{y}'_j)\rangle\right] \end{aligned}$$

for the inner product of two constraints $\bar{y}$ and $\bar{y}'$, and

$$L(\bar{y}) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, \bar{y}_i)$$

for the averaged loss of a constraint. Since $\beta > 0$ and $\beta = C - \sum_{\bar{y}\in\mathcal{W}} \alpha_{\bar{y}}$ we can omit $\beta$ from the optimization and write the final dual program

$$\begin{aligned} &\max_{\alpha} -\frac{1}{2}\sum_{\bar{y}\in\mathcal{W}}\sum_{\bar{y}'\in\mathcal{W}} \alpha_{\bar{y}}\alpha_{\bar{y}'}H(\bar{y},\bar{y}') + \sum_{\bar{y}\in\mathcal{W}} \alpha_{\bar{y}}L(\bar{y}). \\ &\text{s.t.} \sum_{\bar{y}\in\mathcal{W}} \alpha_{\bar{y}} \leq C \text{ and } \alpha_{\bar{y}} \geq 0 \ \forall \bar{y} \in \mathcal{W}. \end{aligned} \tag{4.7}$$

Because strong duality holds, we can obtain an optimal primal solution $(w^*, \xi^*)$ from an optimal dual solution $\alpha^*$ using the stationary point condition in Equation (4.2)

$$w^* = \sum_{\bar{y}\in\mathcal{W}} \alpha^*_{\bar{y}} \frac{1}{n}\sum_{i=1}^{n} \left[\phi(x_i, y_i) - \phi(x_i, \bar{y}_i)\right].$$

Thus the product $\langle w^*, \phi(x, y)\rangle$ can be solved by computing

$$\langle w^*, \phi(x,y)\rangle = \sum_{\bar{y}\in\mathcal{W}} \alpha^*_{\bar{y}} \frac{1}{n}\sum_{i=1}^{n} \left[\langle \phi(x_i, y_i), \phi(x,y)\rangle - \langle \phi(x_i, \bar{y}_i), \phi(x,y)\rangle\right].$$

The complementary slackness conditions yields

$$\xi^* = \begin{cases} 0 & \text{if } \sum_{\bar{y}\in\mathcal{W}} \alpha^*_{\bar{y}} < C \\ L(\bar{y}) - \sum_{\bar{y}'} \alpha^* \bar{y}' H(\bar{y}, \bar{y}') & \text{otherwise with one } \alpha^*_{\bar{y}} > 0. \end{cases}$$

The important property of the dual reformulation is that optimizing the Dual problem and computing $\langle w^*, \phi(x, y)\rangle$ only requires us to compute dot products of feature vectors. Since we have seen that a kernel function $k : (\mathcal{X} \times \mathcal{Y}) \times (\mathcal{X} \times \mathcal{Y}) \to \mathbb{R}$ computes the dot product of two feature vectors, we can apply what is commonly referred to as the *Kernel Trick*. We can replace all the dot products of feature vectors with kernel applications and rewrite

$$\begin{aligned} H(\bar{y}, \bar{y}') = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} &\left[k((x_i, y_i), (x_j, y_j)) - k((x_i, y_i), (x_j, \bar{y}'_j))\right. \\ &\left. - k((x_i, \bar{y}_i), (x_j, y_j)) + k((x_i, \bar{y}_i), (x_j, \bar{y}'_j))\right] \end{aligned} \tag{4.8}$$

as well as

$$\langle w^*, \cdot\rangle = \sum_{\bar{y}\in\mathcal{W}} \alpha^*_{\bar{y}} \frac{1}{n}\sum_{i=1}^{n} \left[k((x_i, y_i), \cdot) - k((x_i, \bar{y}_i), \cdot)\right].$$

### 4.1.2 Inference: A Pre-Image Problem

Now let us consider the inference problem

$$\widehat{y}(x) = \arg\max_{y \in \mathcal{Y}(x)} \sum_{\bar{y} \in \mathcal{W}} \alpha^*_{\bar{y}} \frac{1}{n} \sum_{i=1}^{n} \left[ k((x_i, y_i), (x, y)) - k((x_i, \bar{y}_i), (x, y)) \right]. \tag{4.9}$$

This problem is closely related to a problem well-known from kernel methods: the *Pre-Image problem*[31].

Say we have a kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, which implicitly maps to a Hilbert Space $\mathcal{H}_k$. Assume we are given an element $g \in \mathcal{H}_k$, not in its explicit feature structure, but as a linear combination $g : \sum_i \alpha_i k(x_i, \cdot)$. Now the challenge is to find an element $x \in \mathcal{X}$, such that $\phi_k^{-1}(g) = x$ or $\phi_k(x) = g$ respectively. There are some problems: $\phi$ is only implicitly given, $\phi_k^{-1}$ is unknown and might not be unique, and when $g$ is the output of a numerical prediction algorithm, there is likely no exact match $x \in \mathcal{X}$. Thus we solve an approximation instead:

$$\tilde{x} = \arg\min_{x \in \mathcal{X}} ||\phi(x) - g||^2 \tag{4.10}$$

$$= \arg\min_{x \in \mathcal{X}} k(x, x) - 2 \sum_i \alpha_i k(x_i, x). \tag{4.11}$$

Notice that Equation(4.9) does not require knowledge about $\phi$, but relies solely on the kernel $k$. However, depending on the structure of $\mathcal{X}$ and $k$, the problem is generally hard to solve.

For continuous spaces $\mathcal{X} \subseteq \mathbb{R}^n$ several approaches to solve the problem are known: For RBF-kernels a fixed-point iteration scheme can be applied to solve the non-convex optimization problem and calculate an approximate pre-image[43]. Kwok and Tsang propose a method that exploits the relationship between the distances in feature space $\mathcal{H}_k$ and output space $\mathcal{X}$ for isotropic and dot-product kernels to directly compute an approximate solution for the pre-image problem[31].

For discrete structured spaces the pre-image problem becomes a combinatorial optimization problem. For string outputs and the special case of n-gram kernels, Cortes et al.[11] proposed a method to solve the pre-image problem by transforming it into a graph-problem that involves finding Euler circuits. While the pre-image problem can be expressed using only implicit feature mappings via kernel computations, their method requires to explicitly compute the feature map $\phi_l$. Recently Giguère et al.[22, 20] investigated the pre-image problem for the Generic String Kernel[21], a kernel that generalizes five string kernels commonly used in bioinformatics. They show that for some parametrization of the General String Kernel and

when the kernel is not normalized, pre-image computation is tractable. When normalization is applied, the combinatorial optimization is intractable, however a upper bound can be computed efficiently.

Looking back to the inference problem (4.9) we see that it has essentially the same structure as problem (4.11): We maximize over the argument of a linear combination of kernel functions. The main difference is that the kernel functions in (4.9) have a $x$ and a $y$ part and we only maximize over $y$ while $x$ is fixed. When $\mathcal{Y}(\mathcal{X})$ is a discrete output structure, the pre-image problem (4.9) is a combinatorial optimization problem where the solver has to select the elements of the alphabet that maximize the kernel expressions while taking into account the constraints posed by $\mathcal{Y}(x)$.

The same applies to finding the maximally-violated constraints in the separation oracle calls

$$\arg\max_{y' \in \mathcal{Y}(x)} \langle w, \phi(x, y') \rangle + L(y, y') \tag{4.12}$$

$$= \arg\max_{y \in \mathcal{Y}(x)} \sum_{\bar{y} \in \mathcal{W}} \alpha^*_{\bar{y}} \frac{1}{n} \sum_{i=1}^{n} [k((x_i, y_i), (x, y)) - k((x_i, \bar{y}_i), (x, y))] + L(y, y'). \tag{4.13}$$

We maximize a sum of kernel expressions, however there is an additional term for maximizing the loss $L(y, y')$. The loss term might change the structure of the pre-image problem. Thus it is important to choose loss functions that still allow us to solve the pre-image problem.

## 4.2 Discrete Output Structures and Simple Kernels

We have seen, that a kernel function that is defined for pairs of input and output structures $k : (\mathcal{X} \times \mathcal{Y}) \times (\mathcal{X} \times \mathcal{Y}) \to \mathbb{R}$ can be used instead of defining explicit feature mappings. In this section we will investigate how we can define such kernels.

### 4.2.1 Joint Kernels

It is not immediately clear how we can define a kernel over the product of two spaces $\mathcal{X}$ and $\mathcal{Y}$. However people have already proposed kernels for discrete spaces like sequences or trees. Thus a straight forward way to define kernels for joint product spaces is to combine known kernels for the individual spaces $\mathcal{X}$ and $\mathcal{Y}$.

We will use the most popular joint kernel used[26, 64, 21]: the *tensor product kernel*

$$k((x, y), (x', y')) = k_{\mathcal{X}}(x, x') \cdot k_{\mathcal{Y}}(y, y'),$$

where $k_{\mathcal{X}} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is the input kernel and $k_{\mathcal{Y}} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ is the output kernel. The reason this kernel is called tensor product kernel is that its implicit feature map is the tensor product of the implicit feature maps corresponding to $k_{\mathcal{X}}$ and $k_{\mathcal{Y}}$. Say that $k_{\mathcal{X}}(x, x') = \langle \phi_{\mathcal{X}}(x), \phi_{\mathcal{X}}(x') \rangle$ and $k_{\mathcal{Y}}(y, y') = \langle \phi_{\mathcal{Y}}(y), \phi_{\mathcal{Y}}(y') \rangle$ with mappings $\phi_{\mathcal{X}} : \mathcal{X} \to \mathbb{R}^c$ and $\phi_{\mathcal{Y}} : \mathcal{Y} \to \mathbb{R}^d$. Then $k((x, y), (x', y')) = \langle \phi(x, y), \phi(x', y')$ with

$$\phi(x, y) = \phi_{\mathcal{X}}(x) \odot \phi_{\mathcal{Y}}(y)$$

where the tensor product is defined as

$$[\phi_{\mathcal{X}}(x) \odot \phi_{\mathcal{Y}}(y)]_{j \cdot c + i} := [\phi_{\mathcal{X}}(x)]_i \cdot [\phi_{\mathcal{Y}}(y)]_j \text{ for } i = 1, ...c, j = 1, ...d.$$

We can easily verify the correctness of the feature map:

$$\begin{aligned}
\langle \phi(x, y), \phi(x', y') \rangle &= \sum_{i=1}^{c} \sum_{j=1}^{d} ([\phi_{\mathcal{X}}(x)]_i \cdot [\phi_{\mathcal{Y}}(y)]_j) \cdot ([\phi_{\mathcal{X}}(x)]_i \cdot [\phi_{\mathcal{Y}}(y)]_j) \\
&= \sum_{i=1}^{c} ([\phi_{\mathcal{X}}(x)]_i \cdot [\phi_{\mathcal{X}}(x)]_i) \sum_{j=1}^{d} ([\phi_{\mathcal{Y}}(y)]_j) \cdot [\phi_{\mathcal{Y}}(y)]_j) \\
&= k_{\mathcal{Y}}(y, y') \sum_{i=1}^{c} ([\phi_{\mathcal{X}}(x)]_i \cdot [\phi_{\mathcal{X}}(x)]_i) \\
&= k_{\mathcal{Y}}(y, y') k_{\mathcal{X}}(x, x') = k_{\mathcal{X}}(x, x') k_{\mathcal{Y}}(y, y')
\end{aligned}$$

When we use the tensor product kernel in pre-image problem (4.9), the solution depends only on $k_{\mathcal{Y}}$:

$$\widehat{y}(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{\bar{y} \in \mathcal{W}} \alpha^*_{\bar{y}} \frac{1}{n} \sum_{i=1}^{n} k_{\mathcal{X}}(x_i, x) \left[ k_{\mathcal{Y}}(y_i, y) - k_{\mathcal{Y}}(\bar{y}_i, y) \right]. \tag{4.14}$$

Thus $k_{\mathcal{X}}$ can have arbitrary complexity that induces intractable pre-image problems. As long as $k_{\mathcal{Y}}$ has a pre-image problem that can be solved efficiently, we can efficiently perform inference.

### 4.2.2 A Class of Kernels for Discrete Output Structures

In this section I will design a class of kernels specifically for discrete output structures. Remember that every discrete output structure $y \in \mathcal{Y}$ is a pair $(S, x)$. For notation brevity I will denote by $\sigma \in y$ the access to the atomic symbols from $S \subseteq \Sigma$ that describe $y$. Also, whenever there is no risk of confusion, $x$ will denote the corresponding input structure. Furthermore we write $|y| := |S|$.

According to Gärtner [18] a common choice for kernels for discrete structures is the intersection kernel

$$k(y, y') = \sum_{\sigma \in y} \sum_{\sigma' \in y'} 1_{\sigma = \sigma'}$$

that counts the number of symbols that two structures have in common. However for many problems this exact matching of symbols is too strict. When you think of $\sigma$ and $\sigma'$ as symbols in a layer of annotation, then whether the symbols express the same fact or not depends on the underlying inputs $x$ and $x'$. For instance in Part-of-Speech tagging, comparing two symbols $(3, N)$ and $(2, N)$ has little value without knowing the tokens of the underlying inputs. Thus I propose a kernel that utilizes a hash function $h : \Sigma \times \mathcal{X} \to \mathbb{N}$ that determines if two symbols are equivalent given the corresponding input structures:

**Definition 4.2.1** A kernel $k : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ is called *atomic*, if $\mathcal{Y}$ is a set of discrete output structures and $k$ is of form

$$k(y, y') = \sum_{\sigma \in y} \sum_{\sigma' \in y'} 1_{h(\sigma,x)=h(\sigma',x')}$$

for a hash function $h : \Sigma \times \mathcal{X} \to \mathbb{N}$.

**Example 4.2.1** Consider the set of token sequences $\mathcal{Y}$ as in Example 2.1.1 with the hash function $h : (\mathbb{N} \times \mathcal{V}) \times \mathcal{X} \to \mathbb{N}$ for which it holds that $h((i, t), x) = h((i', t'), x')$ if and only if $t$ and $t'$ are the same tokens. This results in the bag-of-words kernel that for both sequences counts the number of occurrences of tokens and multiplies the corresponding counts.

**Example 4.2.2** Now let $h' = g \circ h$ for a function $g : \mathbb{N} \mapsto \mathbb{Z}_n$. Applying $h'$ instead of $h$ leads to a hash kernel that maps the tokens to one of $n$ buckets depending on the hash function $h'$. This leads to a compressed bag-of-word representation.

**Lemma 4.2.1** An atomic kernel $k : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ with a hash function $h : \Sigma \times \mathcal{X} \to \mathbb{N}$ can be expressed explicitly as $k(y, y') = \langle \phi(y), \phi(y') \rangle$ with $\phi_i(y) = |\{\sigma \in y \mid h(\sigma, x) = i\}|$.

*Proof.* We can simply verify that

$$\begin{aligned}
\langle \phi(y), \phi(y') \rangle &= \sum_{i=0}^{\infty} |\{\sigma \in y \mid h(\sigma, x) = i\}| \cdot |\{\sigma \in y' \mid h(\sigma, x') = i\}| \\
&= \sum_{i=0}^{\infty} \left[ \sum_{\sigma \in y} 1_{h(\sigma',x)=i} \right] \cdot \left[ \sum_{\sigma' \in y'} 1_{h(\sigma',x)=i} \right] \\
&= \sum_{\sigma \in y} \sum_{\sigma' \in y'} 1_{h(\sigma,x)=h(\sigma',x')} = k(y, y').
\end{aligned}$$

□

### 4.2.3 Solving the Pre–Image Problem

Now we can examine the pre-image problem for atomic kernels. When the image is given by a linear combination $\sum_i \alpha_i k(y_i, \cdot)$ the following optimization problem arises:

$$\min_{y \in \mathcal{Y}(x)} \sum_i \alpha_i k(y_i, y) = \min_{y \in \mathcal{Y}(x)} \sum_i \alpha_i \sum_{\sigma' \in y_i} \sum_{\sigma \in y} 1_{h(\sigma', x_i) = h(\sigma, x)} \tag{4.15}$$

$$= \min_{y \in \mathcal{Y}(x)} \sum_{\sigma \in y} \sum_i \alpha_i \sum_{\sigma' \in y_i} 1_{h(\sigma', x_i) = h(\sigma, x)} \tag{4.16}$$

$$= \min_{\lambda} \sum_{\sigma \in \Sigma} \lambda_\sigma \sum_i \alpha_i \sum_{\sigma' \in y_i} 1_{h(\sigma', x_i) = h(\sigma, x)} \tag{4.17}$$

$$\text{s.t. } \lambda_\sigma \in \{0, 1\} \; \forall \sigma \in \Sigma \tag{4.18}$$

$$\text{and } (\{\sigma \in \Sigma \mid \lambda_\sigma = 1\}, x) \in \mathcal{Y}(x) \tag{4.19}$$

We call $w_\sigma := \sum_i \alpha_i \sum_{\sigma' \in y_i} 1_{h(\sigma', x_i) = h(\sigma, x)}$ *sufficient weights* for $\sigma$, as it is sufficient to know all these weights to construct the optimal output. This construction problem obviously is a combinatorial optimization problem, which is in general hard to solve. However, depending the constraint $(\{\sigma \in \Sigma \mid \lambda_\sigma = 1\}, x) \in \mathcal{Y}(x)$ efficient solving is possible:

For some instances of $\mathcal{Y}(\cdot)$, efficient, i.e. polynomial time, algorithms that compute exact solutions are known. Particularly when $\sigma \in \Sigma$ describe edges in a graph and $\mathcal{Y}(\cdot)$ describes a subset of graphs like paths, trees or matchings we know efficient algortihms to compute the optimal structure given the sufficient weights [10, 15, 56].

For other instances of $\mathcal{Y}(\cdot)$ a set of linear constraints that characterizes the constraint exists [37, 52]. Then, the pre-image problem becomes an Integer Linear Program (ILP). Although ILPs are known to be NP-complete, efficient optimization routines exist [27].

In cases where we neither know an polynomial algorithm to construct the structure from sufficient weights nor know a linear set of constraints, it is possible to use heuristic approximations. For instance, we might specify the number of symbols we want to select and always select the symbols with the highest weights. Another possibility seems to be selecting all the symbols with positive weight. In cases where we have an oracle function that decides, whether a $\lambda$ describes a valid output structure, we could apply a black-box optimization routine or evolutionary algorithm that tries to solve the pre-image problem.

When we have the pre-image problem that arises in the Structural Support Vector Machine inference problem (4.14), the image is given in a slightly different

form. We can easily verify that the sufficient weights are now defined as

$$w_\sigma := \sum_{\bar{y} \in \mathcal{W}} \frac{\alpha^*_{\bar{y}}}{n} \sum_{i=1}^{n} k_\mathcal{X}(x_i, x) \left[ \sum_{\sigma' \in y_i} 1_{h(\sigma', x_i) = h(\sigma, x)} - \sum_{\sigma' \in \bar{y}_i} 1_{h(\sigma', x_i) = h(\sigma, x)} \right]. \quad (4.20)$$

## 4.3 Discrete Output Structures and Loss Functions

Now let us think about a general loss function for discrete output structures $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$. Ideally, a loss function should be normalized, i.e. it should map to $[0, 1]$ such that a loss of 1 is the biggest possible loss. Furthermore, it should be symmetric. And most importantly, a loss function should be compatible with the inference pre-image problem

$$\arg \min_{y' \in \mathcal{Y}(x)} \sum_i \alpha_i k(y', y_i)$$

in a sense that we can apply the same optimization routine to solve the separation oracle pre image problem

$$\arg \min_{y' \in \mathcal{Y}(x)} \sum_i \alpha_i k(y', y_i) + L(y', y).$$

Trivially, we could use the 1/0 loss $L(y, y') = 1_{y \neq y'}$. However, it cannot express similarities sufficiently. Furthermore, solving the separation oracle pre-image problem now is finding the structure with the second-largest discriminant score. There is no universally valid way to generalize an combinatorial optimization problem solver to not compute the largest value, but the second largest. Thus the 1/0 loss does not fulfill our requirement of compatibility with the inference problem.

When we have defined an output kernel $k : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$, one possible loss function is the squared Euclidean distance of the implicit feature maps

$$L_\mathcal{H}(y, y') = ||\phi_k(y) - \phi_k(y')||^2 = k(y, y) - 2k(y, y') + k(y', y').$$

When we consider the separation oracle pre-image problem, it is obvious that the term $-2k(y, y')$ is just another term in the the linear combination that is given. The term $k(y, y)$ is constant, however the term $k(y', y')$ is different. Let us investigate

the effect of optimizing over a kernel computation $k(y', y')$ for atomic kernels $k$:

$$\begin{aligned}
&\arg\min_{y' \in \mathcal{Y}(x)} L_{\mathcal{H}}(y', y) \\
=&\arg\min_{y' \in \mathcal{Y}(x)} k(y', y') - 2k(y, y') \\
=&\arg\min_{y' \in \mathcal{Y}(x)} \sum_{\sigma \in y'} \left[ \sum_{\sigma' \in y'} 1_{h(\sigma,x)=h(\sigma',x)} - 2 \sum_{\sigma' \in y} 1_{h(\sigma,x)=h(\sigma',x)} \right] \\
=&\arg\min_{\lambda} \sum_{\sigma \in \Sigma} \lambda_\sigma \left[ \sum_{\sigma' \in \Sigma} (\lambda_{\sigma'} \cdot 1_{h(\sigma,x)=h(\sigma',x)}) - 2 \sum_{\sigma' \in y} 1_{h(\sigma,x)=h(\sigma',x)} \right] \\
&\text{s.t. } \lambda_\sigma \in \{0,1\} \, \forall \sigma \in \Sigma \text{ and } (\{\sigma \in \Sigma \mid \lambda_\sigma = 1\}, x) \in \mathcal{Y}(x)
\end{aligned}$$

Obviously the objective is no longer linear in $\lambda$, thus we cannot necessarily use the same optimization algorithm for solving the separation-oracle pre-image problem as we can use for the inference problem. Thus, $L_{\mathcal{H}}$ does not fulfill our central requirement for a suitable loss function.

When $k(y', y')$ were constant, i.e. $\forall x \in \mathcal{X} \exists c \forall y \in \mathcal{Y}(x) : k(y, y) = c$, we could also omit that term and directly use the loss function $L_{\mathcal{H}}$. One possibility seems to be normalizing the kernel $k'(y, y') = \frac{k(y,y')}{\sqrt{k(y,y)k(y',y')}}$. However, as also stated by Giguere et al.[22], this will also make the pre-image problem harder, as the term $k(y', y')^{-\frac{1}{2}}$ will now appear in every element of the linear combination

$$\begin{aligned}
&\arg\min_{y' \in \mathcal{Y}(x)} \sum_i \alpha_i k'(y', y_i) + L_{\mathcal{H}}(y', y) \\
=&\arg\min_{y' \in \mathcal{Y}(x)} \sum_i \alpha_i k'(y', y_i) - 2k'(y', y) + 2 \\
=&\arg\min_{y' \in \mathcal{Y}(x)} \sum_i \frac{\alpha_i k(y', y_i)}{\sqrt{k(y', y')k(y_i, y_i)}} - \frac{2k(y', y)}{\sqrt{k(y', y')k(y, y)}}.
\end{aligned}$$

Thus normalizing is not an option for a loss function and $L_{\mathcal{H}}$ does not seem an appropriate choice.

A straightforward alternative to $L_{\mathcal{H}}$ is a generalization to the hamming loss, that basically measures how many symbols of the alphabet of $y'$ were selected correctly with respect to the true $y$:

$$L_h(y, y') = \frac{|y| + |y'| - 2|y \cap y'|}{|\Sigma|} = \frac{|y| + |y'| - 2 \sum_{\sigma \in y} \sum_{\sigma' \in y'} 1_{\sigma=\sigma'}}{|\Sigma|}$$

While optimizing $\min_{y' \in \mathcal{Y}} L_h(y, y')$ is compatible with the separation-oracle pre-image problem for atomic kernels, because as above we also minimize an linear objective of binary variables $\lambda$ over a set of constraints

$$(\{\sigma \in \Sigma \mid \lambda_\sigma = 1\}, x) \in \mathcal{Y}(x),$$

there are two major problems: First $|\Sigma|$ might be infinite. Second, even for finite $\Sigma$, $L_h$ puts lots of weight on the symbols $\sigma$ that have not not been selected in either $y$ or $y'$. Thus the resulting loss values are hard to interpret, particularly for large $|\Sigma|$. A quick fix is to just eliminate the denominator, thus the loss counts how many symbols have to be swapped in order to get the correct output:

$$L_h(y, y') = |y| + |y'| - 2 \sum_{\sigma \in y} \sum_{\sigma' \in y'} 1_{\sigma=\sigma'}.$$

One drawback of this is that larger structures will always produce larger losses than smaller structures. Thus the loss should be normalized with respect to the size of the structure. One nice normalization would be scaling the loss with $(|y| + |y'|)^{-1}$ or $(2\sqrt{|y| \cdot |y'|})^{-1}$. However, both these normalization term yield to a loss that is no longer linear in $\lambda$ whenever $|y'|$ is not known prior to the optimization. One quick approximation is just to assume $|y'| = |y|$ and neglecting symmetry. This yields a loss that is 0 when $y = y'$. If $|y'| \leq |y|$ the loss maps to the interval $[0, 1]$, if $|y'| > |y|$ the loss can also be greater 1.

In conclusion I propose the use of the approximately normalized, generalized hamming loss

$$L(y', y) = \frac{|y| + |y'| - 2 \sum_{\sigma \in y} \sum_{\sigma' \in y'} 1_{\sigma=\sigma'}}{2 \cdot |y|}.$$

It is apparent that the loss function is compatible with the inference pre-image problem: If $w_\sigma$ are the sufficient weights for the inference problem, the sufficient weights $\widetilde{w}_\sigma$ for the corresponding seperation oracle problem are

$$\widetilde{w}_\sigma := w_\sigma + \frac{1}{|y|} \left( \frac{1}{2} - \sum_{\sigma' \in y} 1_{\sigma=\sigma'} \right). \tag{4.21}$$

Finally we should note that, depending on the application, the exact matching of alphabet symbols may be to strict. For instance, when the alphabet symbols describe bounding boxes in object recognition tasks, exact comparison is ill-suited. Instead, one should apply another loss function that measures the similarity of two alphabet symbols that is more appropriate than the proposed 0/1 loss function $1_{\sigma=\sigma'}$. On the other hand, many NLP tasks like tagging, chunking or parsing usually rely on the proposed 0/1 loss, which justifies the use in this setting.

## 4.4 The Algorithm

Now we have seen all the elements required to formulate our algorithm for discrete structured output prediction using implicit feature maps based on the Structural Support Vector Machine and Kernels.

### 4.4.1 Putting All Together

To train a prediction model that uses joint tensor kernels, we use the Cutting Plane optimization algorithm presented in Section 2.2.2. Instead of using explicit feature maps $\phi(x, y)$ and computing a weight vector $w$, we use a joint kernel $k((x, y), (x', y')) = k_{\mathcal{X}}(x, x') \cdot k_{\mathcal{Y}}(y, y')$ and compute a model represented by a linear combination kernel computations. To compute the model using kernels, we solve the Dual program seen in Section 4.1.1.

While the input kernel $k_{\mathcal{X}}$ can be chosen arbitrarily, the output kernel $k_{\mathcal{Y}}$ will be an atomic kernel specified by an arbitrary hash function $h : \Sigma \times \mathcal{X} \to \mathbb{N}$. We use a generic loss function, the approximately normalized, generalized Hamming loss, that is valid for all kinds of discrete output structures.

As seen above, the inference or prediction problem now becomes a pre-image problem:

$$\begin{aligned}
&\max_{y' \in \mathcal{Y}(x)} \sum_{\bar{y} \in \mathcal{W}} \frac{\alpha_{\bar{y}}}{n} \sum_{j=1}^{n} k_{\mathcal{X}}(x_j, x) \left[k(y_j, y') - k(\bar{y}_j, y')\right] \\
= &\max_{\lambda} \sum_{\sigma \in \Sigma} \lambda_\sigma w_\sigma \\
&\text{s.t. } \lambda_\sigma \in \{0, 1\} \; \forall \sigma \in \Sigma \text{ and } (\{\sigma \in \Sigma \mid \lambda_\sigma = 1\}, x_i) \in \mathcal{Y}(x)
\end{aligned}$$

with sufficient weights $w$ defined according to (4.20).

In order to solve this, we have to either (a) know an algorithm that constructs the optimal structure in a set of constraints $y \in \mathcal{Y}(x)$ from the sufficient weights or (b) represent the constraints that describe the set of valid discrete output structures as linear inequalities and apply a generic solver for Integer programs. From the optimal $\lambda^*$ we can then construct the predicted output as

$$\hat{y}(x) = (\{\sigma \in \Sigma \mid \lambda^*_\sigma = 1\}, x) .$$

The separation oracle problems are pre-image problems, too. Since we chose our loss function to be compatible with the inference problem, we can use the same optimization routine with modified sufficient weights, according to (4.20) and (4.21),

for the example $(x_i, y_i)$:

$$\widetilde{w}_\sigma := \sum_{\bar{y} \in \mathcal{W}} \frac{\alpha^*_{\bar{y}}}{n} \sum_{j=1}^{n} k_{\mathcal{X}}(x_j, x_i) \left[ \sum_{\sigma' \in y_j} 1_{h(\sigma', x_j) = h(\sigma, x_i)} - \sum_{\sigma' \in \bar{y}_j} 1_{h(\sigma', x_j) = h(\sigma, x_i)} \right] + \frac{1}{|y_i|} \left( \frac{1}{2} - \sum_{\sigma' \in y_i} 1_{\sigma = \sigma'} \right).$$

To train the model, the algorithm repeats the following steps until convergence, as depicted in Algorithm 3: Solve the Dual program, compute a new cutting plane by solving the separation oracle pre-image problems and compute the inner products of all cutting planes according to (4.8) to formulate the new Dual program that includes the new cutting plane. As proposed by Joachims et al. [26] we prune cutting planes $\bar{y}$ that have not been used in 50 iterations. A cutting plane it not used, when the corresponding $\alpha_{\bar{y}} = 0$.

The algorithm terminates, when the most-violated constraint returned by the separation oracle is violated by less than $\xi + \varepsilon$. This guarantees, that our solution has a duality gap upper bounded by $\varepsilon$, as seen in Section 2.2.2.

**Algorithm 3** Atomic Kernel Structured SVM

**Require:** Data Set $\mathcal{D}$, $C > 0$, $\varepsilon > 0$, Input Kernel $k_{\mathcal{X}}$, Hash Function $h$

$\mathcal{W} := \emptyset, H := [\,]$

$k :=$AtomicKernel$(h)$

**repeat**

$\alpha, \xi = \arg \max_{\substack{\alpha \geq 0, \beta \geq 0 \\ \sum \alpha_{\bar{y}} = C}} -\frac{1}{2} \sum_{\bar{y} \in \mathcal{W}} \sum_{\bar{y}' \in \mathcal{W}} \alpha_{\bar{y}} \alpha_{\bar{y}'} H(\bar{y}, \bar{y}') + \sum_{\bar{y}' \in \mathcal{W}} \alpha_{\bar{y}} L(\bar{y})$

**for** $i = 1, ..., n$ **do**

$\hat{y}_i := \arg \max_{y' \in \mathcal{Y}(x_i)} \sum_{\bar{y} \in \mathcal{W}} \frac{\alpha_{\bar{y}}}{n} \sum_{j=1}^{n} k_{\mathcal{X}}(x_j, x_i) \left[ k(y_j, y') - k(\bar{y}_j, y') \right] + L(y_i, y')$

$\mathcal{W} := \mathcal{W} \cup \{\hat{y}\}$

Prune $\mathcal{W}$

Update $H$ matrix

**until** $L(\hat{y}) - \sum_{\bar{y} \in \mathcal{W}, \bar{y} \neq \hat{y}} \alpha_{\bar{y}} H(\bar{y}, \hat{y}) < \xi + \varepsilon$

### 4.4.2 Runtime Analysis and Efficiency Improvements

While in the case of explicit feature maps, a cutting plane can be represented as just one high dimensional vector, in the case of implicit feature maps, a cutting plane is

a sum of kernel computations of size $n$ where $n$ is the number of training examples. This means that computing the dot product of an example with a cutting plane has $\mathcal{O}(n)$ kernel computations. Recall that the model is a linear combination of cutting planes, which means that calculating the discriminant function for an input output pair $(x, y)$ takes $\mathcal{O}(n \cdot s)$ kernel computations, where $s$ is the number of support vectors, i.e. cuttingplanes with $\alpha_i > 0$. The same applies for the separation oracle pre-image problems, where the pre-image is given as a linear combination of size $\mathcal{O}(n \cdot s)$. Each iteration, we have to solve $n$ of these pre-image problems, adding up to a total size of $\mathcal{O}(n^2 \cdot s)$ summands. If the algorithm finishes in $I$ iterations, this adds up to pre-image computations with a total of $\mathcal{O}(n^2 \cdot sI) \subseteq \mathcal{O}(n^2 \cdot I^2)$ summands.

In addition, computing the inner product of two cutting planes – as required in formulating the Dual program – involves $\mathcal{O}(n^2)$ kernel computations. In the $j$th iteration, we have to compute $j$ of these inner products. Thus, if the algorithm finishes in $I$ iterations, the number of kernel computations required to formulate the Dual programs is $\mathcal{O}(n^2 \cdot I^2)$.

For datasets with larger[2] $n$, we should spend some thoughts on an efficient implementation. Obviously we should cache kernel computations as much as possible. Particularly the input kernel lends itself to caching; whenever enough memory is available we should precompute the full $n \times n$ matrix of input kernel values. The output kernel is harder to precompute, as each iteration another $n \times n$ values need to be stored.

We choose the output kernel to be an atomic kernel. This lends itself to optimizations specifically for atomic output kernels. Recall that the biggest speed advantage of explicit features is the fact that cutting planes can be compressed into a single vector. While we cannot make the feature map explicit for the joint kernel, we can make the feature map for atomic output kernels explicit.

Naively implemented, the product of a cutting plane $\bar{y}$ and an input output pair $x, y$

$$\frac{1}{n}\sum_{i=1}^{n} k_{\mathcal{X}}(x_i, x)\left[k_{\mathcal{Y}}(y_i, y) - k_{\mathcal{Y}}(\bar{y}_i, y)\right]$$

takes $O(n)$ input kernel computations as well as $O(n)$ output kernel computations. A naive implementation with atomic kernels is depicted in Algorithm 4.

Obviously computing the atomic kernels using two loops results in quadratic runtime. Inspired by Hash Joins [45] known from the database community we can

[2] on personal computers built in 2015, $n > 1000$ can be large, on multi-core machines for high performance computing, $n > 10000$ can be large.

**Algorithm 4** Naive Product

**Require:** Cutting Plane $\bar{y}$, Input/Output pair $(x, y)$

$s := 0$
**for** $i = 1, ..., n$ **do**
  $\kappa := k_{\mathcal{X}}(x_i, x)$
  **for** $\sigma \in y$ **do**
    **for** $\sigma' \in y_i$ **do**
      **if** $h(\sigma, x) = h(\sigma', x_i)$ **then** $s := s + \kappa$
    **for** $\sigma' \in \bar{y}_i$ **do**
      **if** $h(\sigma, x) = h(\sigma', x_i)$ **then** $s := s - \kappa$
**return** $s$

implement this nearly linear using a hash table as depicted in Algorithm 5. First, we compute the counts of all the hashed symbols $h(\sigma, x)$ for $\sigma \in y$ in a hash table. Note that we only need to do this once. Then for each output $y_i$ and $\bar{y}_i$ from data and $\bar{y}$ respectively, we can compute the number of matches in linear time by looking it up in the hash table.

**Algorithm 5** Hash Table Product

**Require:** Cutting Plane $\bar{y}$, Input/Output pair $(x, y)$

Initialize empty hash table $C : \mathbb{N} \mapsto \mathbb{N}$, $s := 0$
**for** $\sigma \in y$ **do**
  $C[h(\sigma, x)] = C[h(\sigma, x)] + 1$
**for** $i = 1, ..., n$ **do**
  $\kappa := k_{\mathcal{X}}(x_i, x)$
  **for** $\sigma' \in y_i$ **do**
    $s := s + \kappa \cdot C[h(\sigma', x_i)]$
  **for** $\sigma' \in \bar{y}_i$ **do**
    $s := s - \kappa \cdot C[h(\sigma', x_i)]$
**return** $s$

This is a good way to compute the product, whenever the cost of computing the input kernel dominates the cost of computing the product, because we have to make sure that we compute the input kernel at most once for each $x_i$. This is why, in Algorithms 4 and 5, we decided to sum over $i$ in the outer loop.

However, when we have precomputed the input kernel or can cache at least $n$ values, we can rearrange the computations in a way that might need more input kernel computations without running into speed issues. Note that we iterate over

all symbols of all the $2n$ structures of in the data and $\bar{y}$. Large portions of these symbols will be irrelevant to $y$. Thus I propose an algorithm that changes the order of the summations and loops first over the symbols of $y$ and second over $i = 1, ..., n$. To make this more efficient, I propose to use an indexing scheme that stores for each hashed symbol all $i$ where $y_i$ or $\bar{y}_i$ contains the symbol as well as the factor of the corresponding summand. This allows us to not sum over all data items, but only the relevant ones.

**Algorithm 6** Index Cuttingplane

**Require:** Cutting Plane $\bar{y}$

Initialize empty hash table $\bar{C} : \mathbb{N} \mapsto (\mathbb{N} \mapsto \mathbb{R})$
**for** $i = 1, ..., n$ **do**
  **for** $\sigma \in y_i$ **do**
    $\bar{C}[h(\sigma, x_i)][i] := \bar{C}[h(\sigma, x_i)][i] + 1$
  **for** $\sigma \in \bar{y}_i$ **do**
    $\bar{C}[h(\sigma, x_i)][i] := \bar{C}[h(\sigma, x_i)][i] - 1$
**return** $\bar{C}$

We have to compute this index for each cutting plane; this is done by Algorithm 6. We see that for each $i = 1, ..., n$ we aggregate the input kernel computations by hashvalues of symbols in a two-dimensional hash table. If the input kernel is a linear dot product, we can also aggregate the inputs $x$ for each hashed symbol into a single vector and our index reduces to the explicit representation of the feature vector of our joint tensor kernel.

**Algorithm 7** Indexed Cuttingplane Product

**Require:** Index $\bar{C}$ of Cutting Plane $\bar{y}$, Input/Output pair $(x, y)$

Initialize empty hash table $C : \mathbb{N} \mapsto \mathbb{N}$, $s := 0$
**for** $\sigma \in y$ **do**
  $C[h(\sigma, x)] = C[h(\sigma, x)] + 1$
**for** $(h, n) \in C$ **do** //loop all hashed symbols in $y$
  **for** $(i, \alpha) \in \bar{C}[h]$ **do** //all entries in $\bar{C}[h]$
    $\kappa := k_{\mathcal{X}}(x_i, x)$ //input kernel computation or cache lookup.
    $s := s + n \cdot \alpha \cdot \kappa$
**return** $s$

Now using these aggregated index $\bar{C}$ of $\bar{y}$ we can compute the product more efficiently as depicted in Algorithm 7. Running this algorithm results in at most

$|y| \cdot n$ lookups of input kernel values from cache, which, in turn, will result in at most $n$ input kernel computations for cache sizes larger than $n$. Thus the worst-case running time of Algorithm 5 and Algorithm 7 are equal. However for a good choice of $h$, the feature maps of the atomic kernel will be extremely sparse. Thus we never have to loop over all $n$ examples but only a small fraction. This results in a better running time.

Once we have indexed all constraints, we can go one step further and apply the same indexing to the complete model

$$\sum_{\bar{y} \in \mathcal{W}} \alpha_{\bar{y}} \frac{1}{n} \sum_{i=1}^{n} k_{\mathcal{X}}(x_i, x) \left[ k_{\mathcal{Y}}(y_i, y) - k_{\mathcal{Y}}(\bar{y}_i, y) \right].$$

All we have to do is merge all indexes with $\alpha_{\bar{y}} > 0$ into a single index structure by aggregating all entries with the same $\sigma$ and $x$. This results in a more compact form of the model, where we have to loop over less entries in order to compute the score function for an input/output pair.

Moreover, we can directly get the sufficient weights for the pre-image problems (4.20) from the model index structure: For each possible $\sigma$ in the output structure, we can query the examples $i$ that contribute to the corresponding sufficient weights and perform only the necessary kernel computations.

Representing constraints as aggregated indexes has more benefits: When the input kernel $k_{\mathcal{X}}$ is cached we can apply the algorithm to compute the inner product $H(\bar{y}, \bar{y}')$ of two cutting planes. We gain speed improvements because we only need to sum up input kernel values for hashed symbols that appear in both cutting planes. Thus we can prune large parts of the computation.

In conclusion, we see that the title of this thesis is actually very appropriate, as we use both implicit feature maps in the input kernel *and* an output kernel that has been made explicit by the indexing scheme proposed in this section. It is possible to apply the same ideas to arbitrary output kernel that have a sparse, explicit feature map.

# 5 Related Methods for Structured Prediction

In this chapter, I want to give an overview over other research on Structured Output Prediction and draw connections to the approach proposed in Chapter 4. In the first section I present work that focuses on the linear models of the Structural SVM. In the second section I present kernel methods for structured outputs. In the third section I present a different approach to Structured Output Prediction that casts the inference problem as a search problem in the space of all possible outputs. Lastly, I want to present other models that are neither linear nor apply kernels.

## 5.1 More on Structural Support Vector Machines

In Chapter 2, I presented the Structural Support Vector Machine developed by Tsochantaridis et al. [60]. However, the authors presented two different variants, Margin-Rescaling and Slack-Rescaling, of which I only presented Margin-Rescaling. The Slack-Rescaling variant has a slightly different optimization problem where we minimize $\frac{1}{2}||w||_2^2 + C\sum_i \xi_i$ such that $\langle w, \phi(x_i, y')\rangle \geq 1 - \frac{\xi_i}{L(y', y_i)}$ for all $y' \in \mathcal{Y}(x_i), y' \neq y_i$, $i = 1, ..., n$. This changes the separation oracle which becomes

$$\arg\max_{y' \in \mathcal{Y}(x)} \left[L(y', y)(1 - \langle w, \phi(x, y')\rangle)\right].$$

As we can see the loss is no longer added to the linear score function, but multiplied. If both loss and score function decompose over the discrete symbols of the alphabet $\Sigma$, as proposed in Chapter 4, this multiplication makes the separation oracle non-linear. This is why I chose to present the Margin-Rescaling variant and use it for Algorithm 3. The authors demonstrate in their experiments, that the prediction performance of both variants is essentially equally good.

When applying the cutting plane optimization presented in Section 2.2.2, a substantial amount of time is spent on computing the next cutting plane. Yu and

Joachims [68] show that it is possible to compute a new cutting plane by only solving the separation oracle problem for a sample of the training data. This is particularly helpful in settings where the runtime complexity of solving the problem is very high.

Since the Cutting Plane algorithm was proposed in 2009, different optimization routines for solving the Structural SVM training objective have been proposed. Most notably is an approach presented by Lacoste-Julien et al. [32] which applies a randomized block-coordinate Frank-Wolfe algorithm. It only needs a number of separation oracle calls that is independent of $n$ – in case of the cutting plane algorithm this number is linear in $n$ – however it does not guarantee a solution that has a duality gap smaller than $\varepsilon$, but only a guarantee that in expectation the duality gap is smaller $\varepsilon$.

We should also mention the Pegasos algorithm [55] which directly optimizes the primal objective in form of the regularized empirical risk functional in an online fashion. It computes a solution whose objective value is close to the optimal objective value with high probability, while only needing a number of oracle calls independent of $n$ [32]. Pegasos can also be used with kernels by expressing the model $w$ as a linear combination of kernel computations, however the model grows by one kernel computation and structure $y_t$ every iteration.

Choosing a good joint feature map is essential to linear models. Often, we factorize $\phi(x, y) = \sum_{\sigma \in y} \phi(x, \sigma)$. However, it can be useful to define mappings for larger subsets or cliques of symbols, which, unfortunately, makes the inference problem harder. Weiss and Taskar [63] propose an algorithm that iteratively solves the inference problem for clique-factorized feature maps by using increasingly complex factorizations. We could think about output kernels that also compare larger subsets of symbols and use a similar scheme to solve the resulting pre-image problem.

## 5.2 Kernel-Based Methods

In this section we will see three different methods for Structured Output Prediction that involve kernels.

### 5.2.1 Kernel Dependency Estimation

Weston et al. [65] propose kernel dependency estimation (KDE) for structured output learning with kernels. Given a set of input output pairs $(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}$, they choose an input kernel $k_{\mathcal{X}}(\cdot, \cdot) : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ and an output kernel $k_{\mathcal{Y}}(\cdot, \cdot) : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$.

As shown in Chapter 3 the kernels $k_\mathcal{X}, k_\mathcal{Y}$ implicitly map into Hilbert spaces $\mathcal{H}_\mathcal{X}$ and $\mathcal{H}_\mathcal{Y}$ respectively. They propose to learn a function $f : \mathcal{X} \to \mathcal{Y}$ that best fits the given training data by decomposing it into functions $g : \mathcal{X} \to \mathcal{H}_\mathcal{Y}$ and $\phi_\mathcal{Y}^{-1} : \mathcal{H}_\mathcal{Y} \to \mathcal{Y}$ as illustrated in Figure 5.1.

To learn a prediction function $g : \mathcal{X} \to \mathcal{H}_\mathcal{Y}$, they first map the given output structures $y$ into the Hilbert space $\mathcal{H}_l$. Weston et al. propose to apply kernel-PCA for this problem [65], which yields real-valued coefficients for all principal components of only implicitly known space $\mathcal{H}_\mathcal{Y}$. Then they can learn a mapping from $\mathcal{X}$ to the principal components of $\mathcal{H}_\mathcal{Y}$ that best fits the given training data using the kernel $k_\mathcal{X}$ and techniques for real-valued regression like kernel-ridge-regression or Support Vector Regression. Cortes et al. [11] solve both steps in one by directly learning a linear mapping $g(x) = \mathcal{W}\phi_\mathcal{X}(x)$ from $\mathcal{X}$ to $\mathcal{H}_\mathcal{Y}$ using the kernels $k_\mathcal{X}$ and $k_\mathcal{Y}$. They propose to learn $\mathcal{W}$ using kernel ridge regression, i.e. solving

$$\arg\min_{\mathcal{W}} \sum_i ||\mathcal{W}\phi_\mathcal{X}(x_i) - \phi_\mathcal{Y}(y_i)||_2^2 + \gamma||\mathcal{W}||_F^2$$

where $\gamma > 0$ is a regularization coefficient. The optimization problem has a closed form solution

$$\mathcal{W}^* = M_\mathcal{Y}(K + \gamma I)^- 1 M_\mathcal{X}^T$$

where $K_{ij} = k_\mathcal{X}(x_i, x_j)$, $M_\mathcal{X} = [\phi_\mathcal{X}(x_1), ..., \phi_\mathcal{X}(x_n)]$ and $M_\mathcal{Y} = [\phi_\mathcal{Y}(y_1), ..., \phi_\mathcal{Y}(y_n)]$.

When we have learned a function $g : \mathcal{X} \to \mathcal{H}_l$ and want to perform a prediction for a given $x \in \mathcal{X}$ we have to find an $y \in \mathcal{Y}$ whose feature map $\phi_l(y)$ best fits the computed feature map $g(x)$. This problem is another instance of the Pre-Image problem introduced in Section 4.2.3:

$$\arg\min_{y \in \mathcal{Y}} ||\phi_\mathcal{Y}(y) - g(x)||_2^2.$$

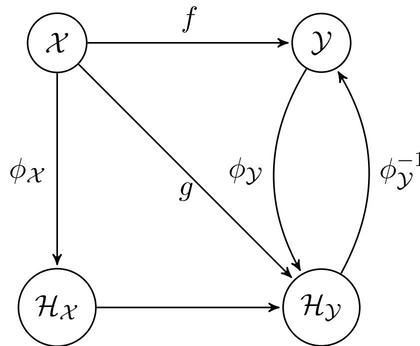



**Figure 5.1:** The kernel dependency estimation framework

When $g(x) = \mathcal{W}\phi_{\mathcal{X}}(x)$ and $\mathcal{W}$ is learned using kernel ridge regression, the problem can be rewritten using only kernel computations

$$\arg\min_{y\in\mathcal{Y}} = k_{\mathcal{Y}}(y,y) - 2L_y^T(K+\gamma I)^{-1}K_x$$

where $[L_y]_i = k_{\mathcal{Y}}(y, y_i)$ and $[K_x]_i = k_{\mathcal{X}}(x, x_i)$.

Similar to my proposed approach in Chapter 4 we can choose $k_{\mathcal{X}}$ to be an arbitrary kernel, but $k_{\mathcal{Y}}$ has to be a kernel for which the pre-image problem can be solved. The pre-image problem differs from the problem in Section 4.2.3 because of the term $k_{\mathcal{Y}}(y, y)$ which is not present in the Structural SVM. As discussed in Section 4.3 we thus cannot directly use Atomic output kernels. There are two possible solutions: Either we assume $k(y, y)$ to be constant and omit the term or we adapt the pre-image problem solvers to the modified problem.

### 5.2.2 Joint Kernel Support Estimation

Lampert and Blaschko [33] propose a generative model for Structured Output Prediction based on Joint kernels $k : (\mathcal{X}\times\mathcal{Y})\times(\mathcal{X}\times\mathcal{Y}) \to \mathbb{R}$. They model probabilities of an output $y$ appearing with an input $x$ as

$$p(x,y) = \frac{1}{Z}e^{\langle w,\phi(x,y)\rangle}$$

where $Z$ is a normalizing constant. We predict by selecting

$$\hat{y}(x) = \arg\max_{y\in\mathcal{Y}(x)} p(x,y).$$

Instead of using the explicit feature map $\phi(x, y)$ they propose to use a joint kernel as presented in Section 4.2.1. The authors demonstrate that the model becomes

$$\hat{y}(x) = \arg\max_{y\in\mathcal{Y}(x)} \sum_{i=1}^{n} \alpha_i k\left((x,y),(x',y')\right)$$

thus when we use a tensor kernel with Atomic Output kernel as proposed in Sections 4.2.1 and 4.2.2 the inference problem is a pre-image as discussed in Section 4.2.3.

The authors show that $\alpha_i \in \mathbb{R}$ can be trained using the one-class Support Vector Machine. This implies that we do not have to solve inference problems $\hat{y}(x_i)$ during training, but work solely on the data in the training set, which is a huge benefit in terms of runtime. In comparison, approaches based on Structural SVMs have to solve $n$ separation oracle problems $\max_y\langle w, \phi(x, y)\rangle + L(y, y')$ in every iteration, which is computationally expensive. However, experiments by Lampert and Blaschko demonstrate that for tasks other than object localization using RBF-kernels the model cannot compete with Structured SVMs [33].

### 5.2.3 Multiple Kernel Learning

Martins et al. [36, 35] presented an approach for Structured Prediction based on Multiple Kernel Learning (MKL). For a number $M$ of different feature maps $\phi_i(x, y), i = 1, ..., M$, we perform inference by minimizing

$$\hat{y}(x) = \arg \min_{y \in \mathcal{Y}(x)} \sum_{i=1}^{M} \langle w_i, \phi_i(x, y) \rangle$$

for the model $w = [w_1^T, ..., w_1^M]^T$. We want to learn the model $w$ by minimizing the regularized empirical risk functional

$$\min_{w} \lambda \Omega(w) + \frac{1}{n} L(x_i, y_i; w)$$

where the loss function is the structured soft margin introduced in Section 2.2.1 and

$$\Omega(w) = \sum_{i=1}^{M} ||w_i||_2^2$$

is a block-structured regularizer, which promotes sparsity in the number of feature maps $\phi_i$ used in the final model. If we choose $M = 1$, the optimization problem is equivalent to the problem presented in Chapter 2. Thus we can view the MKL-approach as another way of solving the training optimization problem.

The authors give an online proximal algorithm that optimizes the regularized empirical risk function. Online proximal algorithms optimize the objective functional by repeating two steps for a specified number of epochs for each example of the training data: First, they take a stochastic subgradient descent step according to the subgradient $\partial L(x_i, y_i; w)$ of the loss function for the example $(x_i, y_i)$ considered. As discussed in Section 2.2.2 this requires a computation of the separation oracle. Then they take a *proximinal* step $prox_{\Omega}(w) = \arg\min_{w'} \frac{1}{2}||w - w'||_2^2 + \Omega(w')$ that applies regularization.

Martins et al. present variants that work for explicit feature maps and for implicit feature maps based on $M$ different joint kernels. The kernelized variant has to remember all results to the separation oracle performed during execution of the algorithm as the model $w$ becomes a linear combination of kernel computations similar to the case of kernelized Structural SVMs. Thus each iteration, one has to store one additional structure.

We can directly apply the joint tensor kernel introduced in Sections 4.2.1 and 4.2.2. If we have identified a set of suitable input kernels and a set of suitable Atomic output kernels via a set of different hash functions, we can directly optimize over $M$

different joint kernels. The regularization penalizes the selection of unnecessary kernel functions which yields more compact and efficient models than a model based on all $M$ kernels.

## 5.3 Structured Prediction – a Search Problem

Daumé et al. [13] propose a totally different approach of Structured Output Learning where we model the inference problem as a search problem in the space of (partial) output structures and guide this search using a machine-learned model.

First one has to define a search space $\mathcal{S}$ where each state $s \in \mathcal{S}$ is either a terminal state or it has a number of *successor* states $succ : \mathcal{S} \to \mathcal{P}(\mathcal{S})$. Each successor is associated with an *action* $a$ that transitions a state to the respective successor. Every search space has an initial state denoted by $\emptyset$. The authors distinguish between two kinds of search spaces: concrete search spaces are defined directly over the alphabet of the discrete output space $\Sigma$ such that $\mathcal{S} = \mathcal{P}(\Sigma)$, while abstract search spaces are defined independently of $\Sigma$ and apply a function $f : \mathcal{S} \to \mathcal{Y}$ to obtain the prediction. In case of concrete search spaces we just define $f(y) = y$ and use the same notation for both cases. Obviously for concrete spaces, the actions correspond to selections of atomic symbols $\sigma \in \Sigma$ and all states $s \in \mathcal{Y} \subseteq \mathcal{P}(\Sigma)$ are terminal.

We assume that for each $y \in \mathcal{Y}(x)$ there is a way to obtain a sequence of actions $a_1, ..., a_T$ or equivalently a sequence of states $s_0, ..., s_T$ that, starting from the initial state $s_0 = \emptyset$ describe a transition into the state $f^{-1}(y)$.

For prediction we assume that we have a model, that given the input $x$ and a non-terminal state $s \in \mathcal{S}$ picks an action or successor state. Then we perform inference by performing a greedy search that, starting at $\emptyset$, queries the model for the next action and takes that action until we end up in a terminal state. We denote the result of this greedy search by $g(x) \in \mathcal{S}$ and predict by returning $\hat{y}(x) = f(g(x))$.

Now I will briefly present a simple algorithm for learning the model that guides the greedy search.

The exact *Imitation Learning* approach learns a model that tries to imitate the actions needed to produce the training data $(x_1, y_1), ..., (x_n, y_n)$ exactly [14]. For each example $(x_i, y_i)$ we take the sequence of actions $a_1, ..., a_{T_i}$ that constructs the output $y_i$. For each $a_j$ of this sequence, we create classification examples based on a feature vector $\phi(x_i, a_1, ..., a_{j-1})$ and the action $a_j$ which takes the role of the label. We collect these examples $(\phi(x_i, a_1, ..., a_{j-1}), a_j)$ for each example $i$ and each action

$j$ and train a classifier based on this labeled data. Then we can use this classifier to guide the search $g(x)$ and hence to predict $\hat{y}(x) = f(g(x))$.

Models trained in this fashion have one shortcoming: As they are only trained on optimal outputs, they cannot necessarily recover from wrong decision in early states of the prediction, since there is no training data for states that contain errors. Thus more advances approaches interleave policy execution and learning [51, 62, 13]. One example for an algorithm is SEARN proposed by Daumé et al. [13]; I will describe only the essential idea. For each example $x_i, y_i$ we generate a prediction $\hat{y}(x)$ that is based on a sequence of states $s_0, ..., s_T$. For each state $s_j$ we can compute the loss that results in taking an action $a$ and then continuing the prediction according to the current prediction model; we denote this loss by $l_a$. We generate a training example

$$(\phi(x_i, s_0, ..., s_j), \arg\min_a l_a)$$

and use these training examples to train a classifier that guides the search. This process is repeated for a specified number of iterations. This way we learn a model that is not just based on the transitions observed in the data, but also on wrong transitions that are evaluated by a general loss function.

More advanced methods apply more complex search strategies than greedy search and apply ranking models to choose between a number of terminal states, similar to the idea described by Collins and Duffy [8]. One idea by Doppa et al. [14] is to create another search space of complete outputs. The initial state is the output by a search-based structured prediction algorithm. The succeeding states of a state $s$ result from changing one decision in the search process that lead to the output $f(s)$. We can guide the search in this second search space of complete outputs with another machine-learned model presented by Doppa et al. [14].

## 5.4 Other Non-Linear Models

Currently non-linear models like Neural Networks or *Deep Neural Networks* are very popular. Their design allows them to combine features in a non-linear matter and thus generate new features in a data-driven approach. One can also apply these networks to structured prediction. Neural Networks are mostly engineered to solve specific structured learning tasks like object detection or sequence tagging. Pei et al. [47] present an approach that solves the same large-margin empirical loss minimization problem as introduced in Section 2.2.1, however instead of using a linear score $\langle w, \phi(x, y)\rangle$ they propose to use a score function computed by an

Neural Network. They assume that the score is factorized according to the atomic components

$$score(x, y) = \sum_{\sigma \in y} NN(x, \sigma)$$

where $NN(x, \sigma)$ is a value computed by the Neural Network. Thus inference

$$\hat{y}(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{\sigma \in y} NN(x, \sigma)$$

can be done using the same construction algorithms as in the linear or kernelized models, the only difference being that the sufficient weights are not computed by a Neural Network. The network can be trained by stochastic gradient descent using Back Propagation.

We see that methods for Structured Predictions vary mostly on two fronts: How inference is performed – exact or by greedy search – and what function determines the compatibility of an input and an output – a linear, kernelized or neural network function.

# 6 A Study on Dependency Parsing

In this chapter, I will investigate the use of explicit and implicit features for Structured Output Prediction for a particular prediction problem that arises in Natural Language Processing: Dependency Parsing. First, I will introduce the task as a Structured Prediction problem using the definitions from Chapter 2. Then I will briefly present related work on Dependency Parsing. I will learn a prediction model based on explicit features designed by domain experts McDonald et al. [39] and the Structural SVM presented in Chapter 2. Then, I will learn another prediction model using implicit features utilizing the Kernelized Structural SVM and an atomic output kernel as presented in Chapter 4. I will present experiments on parsing German newspaper text using both models.

## 6.1 Dependency Parsing and Related Work

In this section the Structured Output learning task of Dependency Parser Training will be introduces. I will show different methods to solve the tasks and relate them to the methods presented in Chapters 2 and 5.

### 6.1.1 The Task

Dependency parsing is a method of automatic syntactic analysis of natural language based on ideas from traditional dependency grammars that have a long history in theoretical linguistics. Dependency Parsing is popular in the natural language community, e.g., for information extraction or machine translation, because it helps uncovering predicate-argument structure within sentences [30].

In dependency parsing, we model the syntactic structure of a sentence by binary relations between the words of a sentence. Dependency relations hold between the

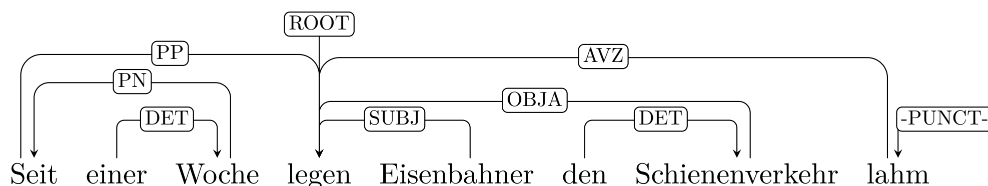



**Figure 6.1:** Example dependency tree from the TüBa-D/Z tree bank.

*dependent* and the word which it depends on, the *head*. Each relation has a *dependency type*. We model head-dependent dependencies as labeled, directed edges in a tree, as you can see in Figure 6.1 [30]. Take for instance the German language: Foth proposed a dependency grammar that relies on 34 different syntactical dependency types [17]. He groups the dependency types into two categories: complementary and modifying: Complementary dependents are all words who appear frequently with their respective head. This group includes subjects, objects, determiners, auxiliary verbs, etc. Modifying dependents are all other relations, including adverbial modifiers, attributes, etc. Modifying dependents are never mandatory. Foth also provides annotation guidelines that formally specify how to annotate a sentence with dependency relations. He claims that almost all structures that occur frequently in the German language can be modeled by this dependency grammar.

A common restriction of the set of valid dependency trees is the restriction to the class of projective dependency trees: A dependency tree is projective, if for each edge $(h, d)$ there is a path from $h$ to each token between $h$ and $d$. Intuitively this means that we can draw the edges annotating the sentence without crossings. From a linguistic view point, projectivity is an invalid restriction, as many languages require non-projective dependency trees for accurate syntax analysis. However, for instance for the English, non-projective constructions appear with little frequency [30].

The problem we want to solve in this study and in Dependency Parsing in general is: Given a sentence or sequence of tokens, compute the best fitting dependency parse tree for that sequence. Usually dependency parsing is performed after other layers of annotation have already been added. Particularly, part-of-speech (POS) tags are assumed to be available for each token. Furthermore, we assume that for each token the corresponding lemma, i.e. the corresponding canonical form, is available. Thus we can define the input space of annotated token sequences by a set of discrete structures. The following definition extends Example 2.1.1.

**Definition 6.1.1** The set of all *annotated token sequences* is set of discrete structures $\mathcal{X}_{seq} \subseteq \mathcal{P}(\Sigma)$ with $\Sigma := \mathcal{N} \times \mathcal{T} \times \mathcal{P} \times \mathcal{L}$ where $\mathcal{T}$ is the dictionary of tokens, $\mathcal{P}$ is the set of part-of-speech tags and $\mathcal{L}$ is the dictionary of lemmas if

$$x \in \mathcal{X}_{seq} \Leftrightarrow \; \forall i = 1, ..., |x| \; \exists \, t \in \mathcal{T} \; \exists \, p \in \mathcal{P} \; \exists \, l \in \mathcal{L} \text{ s.t. } (i, t, p, l) \in x.$$

Now we can formally define the set of discrete output structures of (unlabeled) dependency parse trees, that adds another layer of annotation on the syntactical structure of annotated token sequences.

**Definition 6.1.2** Given a set of annotated input sequences $\mathcal{X}_{seq}$ the discrete output structure set of *unlabeled dependency trees* $\mathcal{Y}_{dep}$ is constructed by an alphabet of edges $e \in (\mathbb{N} \times \mathbb{N})$ such that

$$\mathcal{Y}_{dep} \subseteq (\mathcal{P}(\mathbb{N} \times \mathbb{N}) \times \mathcal{X}).$$

A structure $(E, x) \in (\mathcal{P}(\mathbb{N} \times \mathbb{N}) \times \mathcal{X}_{seq})$ is an unlabeled dependency tree for an input sequence $x \in \mathcal{X}_{seq}$ if and only if the set of edges $E$ describes a spanning arborescence with nodes $\{0, ..., |x|\}$ where the node 0 corresponds to the root of the arborescence and the nodes $1, ..., |x|$ correspond to the annotated tokens of the input sequence $x$.

### 6.1.2 Existing Approaches and Related Work

In this section we will see a brief overview of the work on Dependency Parsing.

Traditional methods for building parser $f : \mathcal{X}_{seq} \rightarrow \mathcal{Y}_{dep}$ rely on hand-crafted grammars. That can be either context-free grammars, so that one can parse using the CYK-algorithm briefly mentioned in Section 2.3.2 or constraint grammars that model parsing as a constraint-satisfaction problem [30].

Recent approaches apply Machine Learning to automatically train parsers from annotated training data, so-called dependency-treebanks. Kübler et al. [30] identify two classes of data-driven dependency parser learners, transition- and graph-based.

In transition-based parsing, we incrementally build parse trees and use machine learning to determine the next parser decision, as presented in Section 5.3. The most popular model employs a shift-reduce parser that sequentially constructs a parse tree. Starting with a sequence $n$ partial trees that contain only one token, the parser always considers a window two neighboring partial trees, starting with the left-most pair. It can take one of three possible actions: SHIFT shifts the window one step to the right. LEFT connects the right tree of the window to the root of

the left tree. Analogously, RIGHT connects the left tree to the root of the right tree. After connecting the trees, the window is extended to the left such that the new tree is on the right position. Yamada and Matsumoto [67] propose to use Support Vector Machines to decide which action to take given the current state of the parsing algorithm. Thus the features used for predicting contain information about the neighboring tree fragments, including tokens and POS-tags. Chen and Manning [4] propose to use Neural Networks.

In graph-based parsing, we assign scores to possible parse-trees and parse by selecting the output parse tree with the highest score. We can define the score of a parse-tree $y$ for a sentence $x$ as a linear function of features for the parse-tree. This coincides with the general ideas of Structural SVMs seen in chapter 2. Naturally, the question is how we define the feature map for a parse tree.

McDonald et al. propose to factor the feature map of a full parse tree into a sum of edge features[39]

$$\phi(x, y) = \sum_{\sigma \in y} \phi(x, \sigma)$$

and show that then the problem of predicting the best unlabeled parse tree becomes a Maximum Spanning Tree problem [42] that can be solved in polynomial time[6, 15, 56]. We will look into this idea in greater detail in the following sections.

Later research has focused on more complex ways of factoring the feature map. McDonald and Pereira [41] propose to factor the feature map into a sum of features for pairs or neighboring edges. Two edges $(i, j), (i', j')$ of a parse tree are neighboring if $i = i'$ and $j + 1 = j'$. Thus they decompose

$$\phi(x, y) = \sum_{(i,j) \in y \text{ and } (i,j+1) \in y} \phi(x, (i, j), (i, j + 1)).$$

They call this decomposition a second order factorization and show that inference is feasible for projective parsing using Eisner's algorithm [16]. Also, they show if the feature map is decomposed into sums of features of $m$ neighboring edges, a m-th order factorization, an adapted version of Eisner's algorithm that runs in $\mathcal{O}(n^m)$ can be used. However, for non-projective second-order parsing exact inference is NP-hard. Thus, they propose an approximative algorithm that runs in $\mathcal{O}(n^3)$.

Koo and Collins [29] propose a number of different factorizations that include up to three edges. Possible factorizations can score dependencies $(i, j)$, siblings $(i, j), (i, j + 1)$, as mentioned above, tri-siblings $(i, j), (i, j + 1), (i, j + 2)$, grand-children $(i, j), (j, k)$, grand-siblings $(i, j), (j, k), (j, k + 1)$. They show that inference for factorizations over parts of three edges for projective parsing can be done in $\mathcal{O}(n^4)$ runtime.

Martins et al. [37] demonstrate that the inference problem of dependency parsing can be reformulated as an Integer Linear Program. This allows us to define linear cost functions that can score arbitrary subsets of edges, including but not limited to the factorizations introduced above. This approach is not limited to projective parsing, however solving ILPs is known to be NP-hard. Still solvers are available that are able to find good solutions to the inference problem, as demonstrated in the experiments performed by Martins et al. [37].

Furthermore one should mention the work of Bonet [1], who tries to make parser learning more efficient by reducing the size of the feature maps computed. He proposes to use a hash kernel that hashes the features into a smaller number of buckets. Thus the model becomes smaller, inference is faster and thus training is faster, too.

### 6.1.3 The Experiments

The experiments and comparisons presented in this chapter all rely on training data from the TüBa-D/Z treebank [58]. It contains 65,524 sentences annotated with dependency relations. The sentences are taken from German newspaper articles published in the ‘taz’ newspaper. The average length of sentences in the treebank is 17.8 words and the total number of tokens is to 1,164,766. While it is originally annotated using a constituency grammar, it was converted to dependency grammar annotation using the annotation guidelines by Foth [17].

The loss function used in all experiments and comparisons will be the unlabeled attachment score (UAS), i.e., the fraction of tokens that have been assigned the wrong head, as my study will only be on unlabeled dependency parsing. McDonald et al. have shown that the dependency labels can be added in a second step using an additional classifier [40], thus I will not focus on that.

All experiments rely on my own implementation of Algorithms 2 and 3, for more details see Appendix B. As this implementation was developed during the course of this thesis, not all experiments were performed with the final version of the software, but with earlier, work-in-progress versions. I believe that all results are still valid, meaningful and comparable, as later modifications of the software mostly addressed performance issues.

For the Chu-Liu/Edmonds algorithm, I gratefully use a publicly available Java implementation by Sam Thomson[1].

[1] `https://github.com/sammthomson/ChuLiuEdmonds`

## 6.2 Dependency Parsing with Explicit Features

In this section I present my results for Dependency Parsing with explicit features based on Structural SVMs. The results presented here are based on McDonald's work [42, 40, 38].

### 6.2.1 Explicit Feature Map

McDonald et al. [42, 40, 38] propose to factor the feature map of a pair of token sequence and dependency parse tree into a sum of edge features:

$$\phi(x, y) = \sum_{\sigma \in y} \phi(x, \sigma).$$

They define binary edge features for an edge $(h, d) \in \mathbb{N}^2$ based on a number of different templates:

- **Head-Features:** For each possible token, PoS-tag and lemma, we use one feature that indicates if the head of the edge has that particular token, PoS-tag or lemma. Additionally, we use features for the combinations of token and PosTag, as well as lemma and PoS-tag.
- **Dependent-Features:** Defined analogously for the dependent
- **Head-Dependent-Features:** We have one feature that indicates that head and dependent have a particular pair of tokens. Analogously, features for PoS-tags and lemmas are defined. Additionally, we use features for pairs of pairs of tokens and PoS-tags, and lemmas and PoS-tags.
- **In-Between-Features:** For the PoS-tags of head and dependent, we have one feature for each possible PoS-tag that indicates how often that PoS-tag appears in between the PoS-tags of the head and the dependent.
- **Neighbor-Features:** Similar to the In-Between-Features, we use features that indicate the PoS-tags for the left and right neighbors of head and dependent, respectively.

In Figure 6.2 we see an example of the edge features defined by these templates. Additionally, we extend each of these feature templates with the so-called *attachment distance*, i.e., the distance between head and dependent in the input sequence. For instance, we have a feature that is set to 1 if head and token have a particular pair of tokens and the distance of head and dependent in the sentence is $k$ tokens. Negative

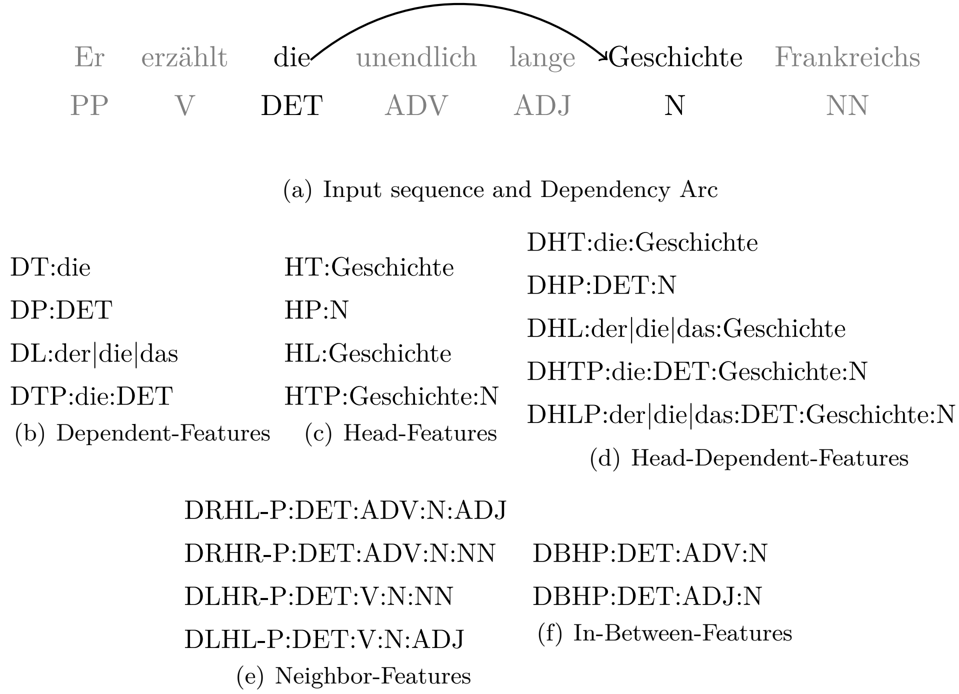



**Figure 6.2:** An example token sequence with dependency edge (a) and the features resulting from the templates presented in this section (b)-(f).

distances indicate that the dependent stands left of the head in the input sequence. We discretize the possible distance into seven categories: $1, 2, 3, 4, 5, > 5, > 10$, as higher distance are less common and thus aggregation for distances greater 5 gives us more reliable estimates for the weights.

### 6.2.2 Inference

Now that we have seen how to define a feature map $\phi(x, y)$, we have to think about an efficient inference algorithm for solving

$$\arg \max_{y \in \mathcal{Y}_{dep}(x)} \langle w, \phi(x, y) \rangle.$$

Plugging in the definition of $\phi$ we see that

$$\begin{aligned}
\arg \max_{y \in \mathcal{Y}_{dep}(x)} \langle w, \phi(x, y) \rangle &= \arg \max_{y \in \mathcal{Y}_{dep}(x)} \sum_{e \in y} \langle w, \phi(x, e) \rangle \\
&= \arg \max_{\lambda} \sum_{e \in \Sigma} \lambda_e \cdot \langle w, \phi(x, e) \rangle \\
&\quad \text{s.t. } \{e \mid \lambda_e = 1\} \in \mathcal{Y}_{dep} \text{ and } \lambda_e \in \{0, 1\}
\end{aligned}$$

where $\lambda_e$ are binary decision variables that indicate, whether an edge in the full graph $e$ is selected in the output or not. We know that a parse tree is a connected, cycle-free graph. We know further, that the tree is rooted in the artificial

*root* node. Thus our algorithm needs to construct a directed tree, that is rooted in the *root*-node and spans all tokens. Consequently, we have identified our inference problem to be a Maximum Spanning Tree (MST) problem in a directed, weighted graph [42]. The weights $w_e$ for an edge $e$ in the complete graphs are defined according to $w_e := \langle w, \phi(x, w) \rangle$. In order to solve the MST problem, we rely on Chu-Liu/Edmonds' algorithm [6, 15]. See [42] for pseudo-code and an example application of the algorithm. Tarjan proposed an efficient implementation that returns the maximum spanning tree in a complete graph in $\mathcal{O}(n^2)$ runtime, where $n$ is the number of nodes [56].

### 6.2.3 Model Training using MIRA

McDonald et al. propose to learn the model $w$ using a modified Margin Infused Relaxed Algorithm (MIRA) [12] that is closely related to the Structural Perceptron. It estimates $w$ in an online fashion, i.e. it changes $w$ according to one training example $(x, y)$ at a time. Starting with an initial choice of $w_0$, it considers an example $(x_t, y_t)$ and updates $w_{i+1}$, such that the change

$$||w_{i+1} - w_i||_2^2$$

is minimized subject to correctly parsing $x_t$ with a margin at least as great as the loss of the best prediction $\widehat{y}_t = \arg\max_{y' \in \mathcal{Y}_{dep}(x_t)} \langle w_i, \phi(x_t, y_t) \rangle$ performed with the old weights $w_i$. Thus each iteration it has to solve the following Quadratic Program:

$$\begin{aligned} &\min_{w_{i+1}} ||w_{i+1} - w_i||_2^2 \\ &\text{s.t. } \langle w, \phi(x_t, y_t) - \phi(x, \widehat{y}_t) \rangle \geq L(y_t, \widehat{y}_t) \end{aligned} \tag{6.1}$$

Finally, it sets $w$ to be the average of all $w_i$. It has been empirically shown that this averaging reduces overfitting [39].

### 6.2.4 Model Training using Structural SVMs

I propose to use the Structural SVM instead, as it offers more control over the generalization performance via setting $C$. In order to apply Structural Support Vector Machines, we have to also provide an algorithm for solving the separation oracle problem

$$\arg \max_{y' \in \mathcal{Y}_{dep}(x)} \langle w, \phi(x, y') \rangle + L(y, y')$$

for the unlabeled attachment score loss function $L(y, y')$. Luckily, the UAS Loss is equivalent to the loss function proposed in Chapter 4 given two parse trees, thus

| C | Train Error | Test Error | $\Delta$ |
|---|---|---|---|
| 0.1 | 15.67% | 15.92% | 0.25% |
| 1 | 9.44% | 9.73% | 0.29% |
| 10 | 7.43% | 8.01% | 0.48% |
| 100 | 5.27% | 6.32% | 1.05% |
| 200 | 3.48% | 5.04% | 1.56% |

**Table 6.1:** Empirical loss for different values of C. Training and testing were performed using 50% of the data respectively. $\Delta$ is the difference of train and test error.

we know that we can apply the same MST algorithm for solving the separation oracle problem. The only difference is, that now the edge weights $w_e$ are defined, analogously to equation (4.21), as

$$w_e := \langle w, \phi(x, e) \rangle + \frac{1}{|x|} \cdot 1_{\{e \notin y\}}$$

This definition holds, because both $\phi$ and the loss function decompose over the edges $e \in y'$.

I train the model on 50% of the TüBa-D/Z treebank, which amounts to more than 37,000 training instances and test on the remaining 50%. I set $\varepsilon := 0.05$ and analyze the train and test loss for different values of $C$. From the features described above I use those that occur more than 5 times. This amounts to a dimensionality of 541,968 different features. Without this pruning step, it would amount to 7,150,021 different features.

In Table 6.6 we see the parsing results using the Structural SVM. The value reported is the Unlabeled Attachment Score. We see that models trained with higher values of $C$ perform better both in training and testing. When $C$ increases, the gap between train and test loss increases. The best test error achieved is 5.04%.

## 6.3 Dependency Parsing with Implicit Features

In the section before, we have seen that we can train accurate dependency parsers based on explicit features handcrafted by domain experts. In particular, we have used 541,968 features based on 60 different templates. In this section I will investigate the hypothesis that we can train accurate dependency parsers based on implicit features as proposed in Chapter 4. In order to do run Algorithm 3, we have to specify a hash function for the atomic kernel, pick an input kernel and then specify a construction algorithm that builds an output structure based on the sufficient

weights predicted by the model. We can view the first two choices as parameters of the algorithm; I will perform a parameter optimization on a small development set and then validate the parameters on the full data. The hash function will be the only component of the model that is truly handcrafted to the task. But for the input kernel, I will rely on available and well-tried kernel functions for token sequences or strings.

As for the construction algorithm, we can again use the Maximum Spanning Tree algorithm by Chi-Liu/Edmonds: The alphabet models edges in a graph, we are provided with sufficient weights for all the edges of the full graph and we know that the output has to be a directed tree that connects all the tokens. The main difference in using implicit features is that the sufficient weights are the result of the kernel computations according to Equation (4.20) on page 38, not of linear products of a weight vector and a vector of edge features as in section 6.2. Section 6.3.4 also investigates the use of an approximate construction algorithm instead of the Chi-Liu/Edmonds algorithm.

### 6.3.1 Picking a Hash Function

We begin by selecting a hash function $h : \Sigma \times \mathcal{X}_{seq}$. I use a small development set of 10,000 examples, 5,000 for training and 5,000 for testing, to test the usefulness of a set of candidate hash functions. To keep these comparisons as simple as possible, I will use no input kernel, i.e., $k(x, x') = 1 \; \forall x, x'$. Thus we will see how appropriate the pure atomic output kernel is for predicting dependency parse trees.

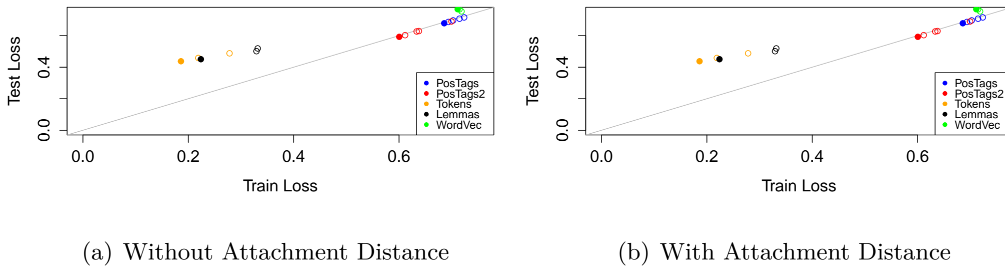
The hash function has to be capable of comparing two edges $e$ and $e'$ in two trees $y$ and $y'$. Different views of similarity are possible here:

- **Tokens:** Two edges are equal if the tokens at their respective head and dependent match.

- **Lemmas:** Two edges are equal if the lemmas at their respective head and dependent match.

- **POS Tags:** Two edges are equal if the POS tags at their respective head and dependent match. Note that there are two sets of POS tags available, a coarse set (PosTag) and a fine-grained set (PosTag2).

- **Neural Word Embeddings:** Mikolov et al. [44] recently presented a Neural Network based approach to project words into Euclidean vector spaces in a way that captures word similarities or analogies [3]. I cluster these so-called




(a) Without Attachment Distance

(b) With Attachment Distance

**Figure 6.3:** A comparison for different hash functions. On the left we see the hash functions not using the attachment distance, on the right we see hash functions using the attachment distance. Points mark the train and test loss of a model for a particular $C$. Filled points indicate the best model I was able to find by optimizing $C$. This Figure is based on Table 6.2 and Table 6.3.

word-embeddings into around 1250 clusters of similar words using k-Means. Now two edges are equal if the tokens of their respective head and dependent fall into the same cluster.

From the experiments with explicit feature we know that the attachment direction and distance between two tokens and the direction of the edges are important features for parsing. Thus for each of the hash functions $h$ presented above I propose another hash function that only matches two edges $e, e'$ for sequences $x, x'$ if the distance of the tokens and the direction of the edge are the same and $h(e, x) = h(e', x')$.

For each candidate hash function, I run a small grid search over increasing values for $C$, until the training loss does not improve any longer. In Figure 6.3 we see, that overall, hash functions that incorporate the distance between head and dependent perform better. Furthermore, the only two hash functions that achieve a decent test error are based on POS tags using the distance of head and dependent. What is particularly notable is that both the hash function using the PosTag and PosTag2 achieve perfect generalization, i.e. the training loss is the same as the test loss. Unfortunately this comes at the cost of not being able to model the data very well; the smallest train error we can achieve with PosTag and PosTag2 is 17.91% and 14.72% respectively. (c.f. Table 6.3). In comparison a hash function based on tokens and the distance of head and dependent can fit the data nearly perfectly with a training loss of 3.40%. Of course such a hash function does not generalize well, the test loss is 34.74%, more than 10 times the training loss.

Using PosTag2 instead of PosTag results not only in less test error, but also in a faster runtime. The reason is that PosTag2 has more different possible tags and thus

| Hash Function $h$ | $C = 0.1$ train | $C = 0.1$ test | $C = 1$ train | $C = 1$ test | $C = 10$ train | $C = 10$ test |
|---|---|---|---|---|---|---|
| tokens | 27.82% | 48.79% | 21.89% | 45.85% | 18.61% | 43.76% |
| tokens::dir | 5.56% | 36.65% | 3.79% | 35.40% | 3.40% | 34.74% |
| lemmas | 32.94% | 50.15% | 33.18% | 51.90% | 22.39% | 45.05% |
| lemmas::dir | 7.99% | 35.84% | 5.23% | 33.84% | 4.33% | 33.10% |
| posTags | 72.37% | 71.57% | 68.58% | 67.87% | 69.36% | 68.70% |
| posTags::dir | 22.87% | 22.58% | 21.65% | 21.22% | 19.07% | 18.80% |
| posTags2 | 69.96% | 69.03% | 60.04% | 59.26% | 61.16% | 60.32% |
| posTags2::dir | 22.08% | 21.71% | 19.01% | 18.84% | 16.44% | **16.39**% |
| wordVecs | 71.86% | 75.41% | 71.47% | 76.71% | 71.14% | 76.77% |
| wordVecs::dir | 45.01% | 53.23% | 36.31% | 47.32% | 36.23% | 48.15% |

**Table 6.2:** A comparison of empirical losses achieved with different hash functions $h$ for different $C$ using no input kernel. Training and testing was performed using 5,000 samples, respectively.

the resulting feature vectors are more sparse. Hence the optimizations presented in section 4.4.2 are more effective.

I conclude that a hash function based on the PosTag2 and the distance between head and dependent is the best choice for dependency parsing. I conjecture that using a joint kernel for a suitable choice of input kernel will overcome the shortcoming that the output kernel cannot adapt to the data more closely.

### 6.3.2 Picking an Input Kernel

After selecting a suitable hash function, we have to select an input kernel for our joint tensor kernel. I want to consider only kernels that are widely used and are not tailored specifically to the task of dependency parsing. I came up with the following selection of kernels for token sequences or strings.

- **Bag-of-Words Kernel:** Joachims has shown that the Bag-of-Word kernel performs great for text classification [25]. It creates a vector with the number of occurrences of each token for both token sequences $x$ and $x'$ respectively. Then it takes the linear dot product of these *bag-of-words* vectors:

$$k_{bow}(x, x') = \langle bow(x), bow(x') \rangle.$$

| Hash Function $h$ | $C = 50$ train | $C = 50$ test | $C = 100$ train | $C = 100$ test | $C = 200$ train | $C = 200$ test |
|---|---|---|---|---|---|---|
| posTags | 71.48% | 70.78% | 70.27% | 69.66% | | |
| posTags::dir | 18.45% | 18.29% | 17.91% | 17.69% | 18.32% | 18.04% |
| posTags2 | 63.34% | 62.68% | 63.77% | 62.92% | | |
| posTags2::dir | 15.32% | 15.46% | 14.72% | **14.79**% | 14.96% | 15.09% |

**Table 6.3:** A comparison of empirical losses achieved with PosTag based hash functions $h$ higher values of $C$ using no input kernel. Training and testing was performed using 5,000 samples, respectively.

Obviously, it does not give us any benefits to use this kernel as a "black box" rather than using the explicit feature vectors $bow(x)$ and a linear kernel. On the contrary we have seen in Chapter 4 that this decreases the efficiency massively since some computations are no longer linear but quadratic in the number of training examples.

A joint kernel using the bag-of-words input-kernel and the Atomic output-kernel discussed in the last section seems appropriate in the context of dependency parsing: Some edges will be more likely when particular tokens occur in the input. We implicitly have features for each combination of occurring token and occurring edge hashed according to the respective PoS-tags.

- **Polynomial Bag-of-Words Kernel:** We have seen in Chapter 3 that a polynomial kernel of degree $d$ has implicit features for all combinations or products of $d$ features. While the implicit feature space grows exponentially with $d$, but the expense of the kernel computation does not grow. Thus a polynomial bag-of-words kernel

$$k_{poly}^{(d)}(x, x') = (k_{bow}(x, x') + 1)^d$$

can match subsets of words that occur in both inputs $x$ and $x'$ and the computation cost is independent of $d$.

In the context of dependency parsing, I suspect that the polynomial kernel of degree $d \geq 2$ is more powerful than $d = 1$. We can now use implicit features that combine subsets of tokens with edge features.

- **N-Gram Kernel:** We can apply the n-Gram kernel proposed by Lodhi et al. [34] presented in Chapter 3. It matches subsequences up to a specified

length $l$ while also allowing inexact matching via a specified discount factor $\lambda \in [0, 1]$. It can be computed in a Dynamic Programming fashion and I rely on the Mallet[2] implementation available online. Using the n-Gram kernel in a joint kernel creates dependencies between substrings of the input and edges in the output. This way particular phrases can be used as evidence for particular edges.

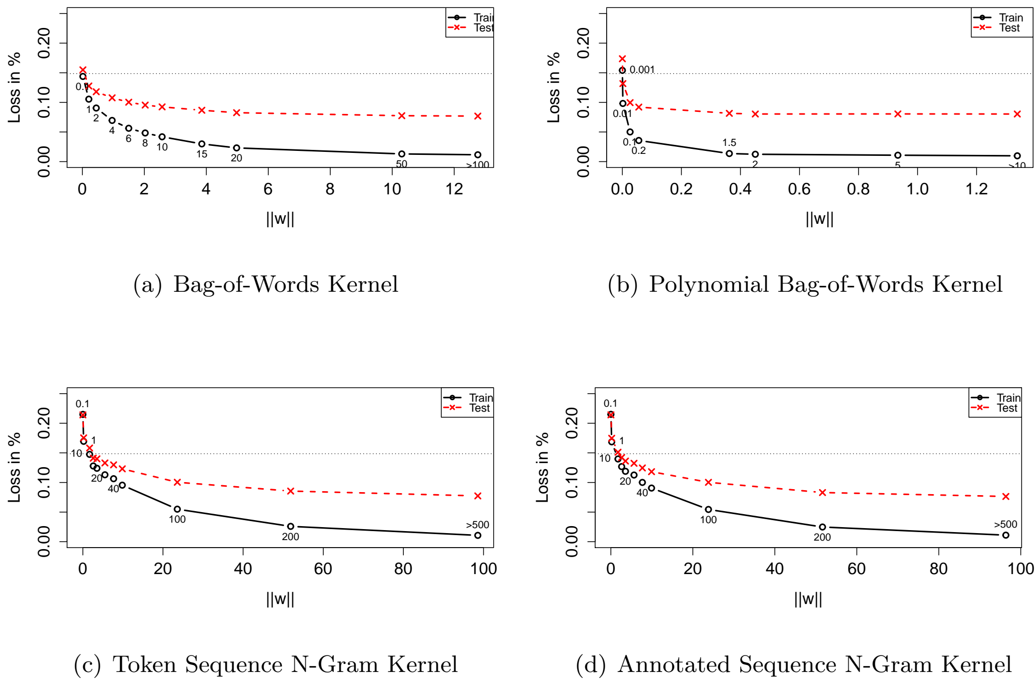
- **N-Gram Kernel for Annotated Sequences:** The n-Gram Kernel matches only the tokens of the input, however we have more layers of annotations available. I propose to modify the n-Gram kernel in a fashion that also matches PoS-tags and lemmas.

Now I compare the prediction performance for four different input kernels: The Bag-of-Words kernel, the polynomial Bag-of-Words kernel of degree $d = 2$, the N-Gram Kernel and the N-Gram Kernel modified for Annotated Sequences. Both N-Gram kernels have the maximum length of matched sequences set to 6 and use the default decay factor $\lambda = 0.5$ provided by the Mallet implementation used for these experiments.

I use the same development set as in the section above. The model is optimized for increasing values of $C$, until the training error does not improve anymore. The cutting plane optimization threshold is set as $\varepsilon = 0.05$. In Figure 6.4 we see the results of this comparison: For different hash functions, the train/test error of different models is plotted against the model complexity, i.e. the regularizing term $||w||_2^2$. The small numbers indicate which value of $C$ was used for a particular model.

First of all, we notice that using an input kernel improves the prediction in comparison to using solely the output kernel. Also, we see that the test loss is always higher than the train loss. I attribute this to overfitting. However, it is not particularly dangerous, as both train and test loss are monotonically decreasing. The test error converges at some point, but does not increase again. Thus in the context of dependency parsing, a high model complexity is not necessarily a bad thing. Apparently, a more complex model memorizes more edges, that appeared in the training data and can benefit from a small fraction of these memorized facts when applied to new, unseen data. Still, for completely unseen inputs, where none of these apply, it can still use more general features. Thus, we should generally use high values for $C$. However, Joachims et al. show that the number of iterations

[2] `http://mallet.cs.umass.edu/`




(a) Bag-of-Words Kernel

(b) Polynomial Bag-of-Words Kernel

(c) Token Sequence N-Gram Kernel

(d) Annotated Sequence N-Gram Kernel

**Figure 6.4:** Test/Train loss are plotted against the model complexity for different input kernels. The horizontal dotted line shows the loss achieved using no input kernel. The numbers next to the points indicate the value of $C$ used for the respective model.

needed to converge depends linearly on $C$, thus models with higher complexity need longer to train [26].

The best model for the Bag-of-Words input kernel achieves a train loss of 1.16% and a test loss of 7.68%. This verifies the hypothesis that the tokens occurring anywhere in the input sequence influence the local likelihood of edges in the output tree.

The polynomial BoW input kernel peaks at a train loss of 0.98% and a test loss of 8.04%. While this is a smaller train loss than the BoW kernel, the test loss is higher and the gap between train and test loss is larger. The kernel uses pairs of tokens as implicit features which allows to fit to the training data more closely. This explains the danger to start overfitting.

Using an n-Gram kernel for token sequences results in a training loss of 1.07% with a test loss of 7.73%. This validates our intuition, that the occurrence of certain sub-sequences somewhere in the input gives evidence to the generation of edges.

When we use a modified n-Gram kernel that also compares the annotation layers of the input, we get a training loss of 1.11% and test loss of 7.64%. We see that

| $C$ | Train Loss | Test Loss | $\Delta$ | $C$ | Train Loss | Test Loss | $\Delta$ |
|---|---|---|---|---|---|---|---|
| 10 | 15.72% | 16.25% | 0.53% | 0.1 | 11.71% | 13.63% | 2.92% |
| 100 | 13.15% | 14.34% | 1.19% | 1 | 6.39% | 10.17% | 3.78% |
| 500 | 8.53% | 11.54% | 3.01% | 10 | 2.04% | 9.52% | 7.50% |
| 1000 | 4.92% | 9.53% | 4.61% | | | | |

**Table 6.4:** Training and Testing the model on the Full Treebank, N-Gram Kernel on the left, Polynomial BoW Kernel on the right

this is insignificantly better than using only the tokens. We should however note that adding the necessary layers of annotations to new, unobserved sequences is computationally expensive. Thus this small advantage comes at high cost.

This experiment suggests that implicit features defined by the input kernel in a joint kernel make better, handcrafted features obsolete. In contrast to the explicit features introduced above, which are defined locally on particular edges, the joint kernel extends very simple local features specified on edges with complex, implicit features defined globally, i.e. on the whole inputs. The input kernel allows us to use implicit feature maps of arbitrary complexity; particularly the n-Gram kernel has a feature space that is very high dimensional. In the next section, I will evaluate its performance on a large treebank. To the best of my knowledge, the approach of using complex, global features has never been used for dependency parsing.

### 6.3.3 Testing the Model on the Full Treebank

In this section I want to evaluate the performance of a parser based on joint tensor kernels on the full TüBa-D/Z treebank. Of the four kernels discussed in the last section, I choose the N-Gram kernel, as it is less expensive to compute than the Annotated n-Gram kernel, and the Polynomial Bag-of-Words kernel, as it takes greater advantage of the implicit feature map than the Bag-of-Words kernel. I use a hash function based on the PoS2-tag and the attachment distance.

I train a parser on 50% of the TüBa-DZ treebank and test on the remaining 50%. As before, I set $\varepsilon = 0.05$ and test the performance for increasing values of $C$. As we see in Table 6.4, both input kernels can achieve a test loss of around 9.5%. The loss decrease with increasing values of $C$, as seen in earlier experiments. Thus the results reported for the small development set of 10,000 examples carries over to the full treebank. The test error is not as low as achieved on the development set,

I believe that it may improve for higher values of $C$. However, training with higher $C$ is too time-demanding.

### 6.3.4 Approximate Reconstruction Algorithm

For inference and for finding the most-violated constraint we rely on the Chu-Liu/Edmonds algorithm that finds the best tree structure that connects all tokens with directed edges from the sufficient weights. In Section 4.2.3 I briefly mentioned that it might be possible to use approximate construction algorithms, whenever no efficient exact algorithm is at hand.

In this section, I want to evaluate the effects of using an approximate construction algorithm. More specifically, I will use an algorithm, that for each token selects the highest scored incoming edge. Thus each token has exactly one head, but the resulting output structure is not necessarily a tree; it might not be connected and it might contain cycles. Obviously this algorithm is not computationally challenging: In the matrix of sufficient weights $w_{(head,dependent)}$, we have to find the largest entry in each row.

For this experiment, I use a subset of 10k examples, 5k for training and 5k for testing. I rely on the hash function using PoS2-tags and attachment distance proposed above and a polynomial Bag-of-Words input kernel of degree 2. I set $\varepsilon = 0.05$ and optimize $C$ in a grid search, until the training error does not improve anymore.

Interestingly, as we can see in Figure 6.5, the test loss suffers very little from this approximation. Particularly for models with larger $C$, we gain only 0.21% less loss by using an exact algorithm is, which is less than 3% better.

Next we check whether the predicted outputs are valid trees or not despite the approximate algorithm. We achieve this by checking using depth-first search, if the graph returned is connected and cycle-free. As we can see in Table 6.5, for larger values of $C$, more than 92% of the outputs are valid tree structures.

In conclusion, I believe that approximate construction is a valid alternative to exact construction. The proposed learner is capable of learning what valid output structures are from the trees provided for training. Only in scenarios, where the annotation layer of dependency grammar structure is used as input for a next processing step that strictly relies on a valid tree structures, we have to use exact construction.

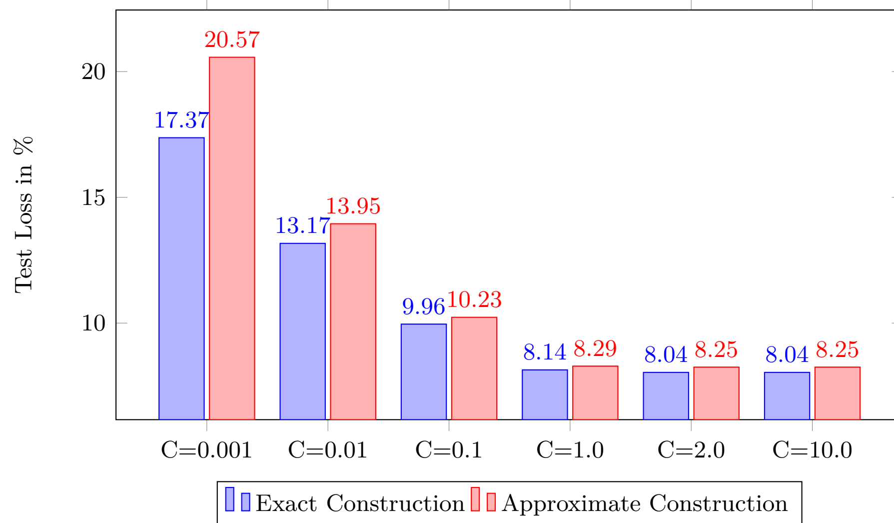



**Figure 6.5:** A comparison of exact construction and approximate construction.

| C | % of valid trees |
|---|---|
| 0.001 | 93.99% |
| 0.01 | 85.83% |
| 0.1 | 91.50% |
| 1.0 | 92.54% |
| 2.0 | 92.59% |
| 10.0 | 92.14% |

**Table 6.5:** Percentage of valid tree structures for different $C$.

### 6.3.5 Analyzing Efficiency Gains

In Section 4.4.2 I proposed optimizations that exploit the sparsity of the Atomic output kernel that should result in learning times whenever the input kernel can be cached.

In this experiment I use datasets $\mathcal{D}$ of different sizes. We test the total runtime of training a model on 50% of the data and testing on the other 50%. I report runtimes on my 2014 laptop. The hash function used in the Atomic kernel is the one proposed above based on PoS2-tags and attachment distance. The input kernel used is the n-Gram kernel for annotated sequences. I set $C = 10$ and $\varepsilon = 0.05$. Note that Joachims et al. showed that training time is linear in $C$ and inversely linear in $\varepsilon$ [26]. Thus these results do not need to be empirically verified with different settings for those parameters.

In Figure 6.6 we see, that the runtimes for executing the optimized code is only a fraction of the runtime of the code not optimized for Atomic output kernels. I want to emphasize that the optimizations do not compute an approximation but are a more efficient way to compute the same, exact model. The biggest portion of the computation is spent on updating the matrix of inner products of cutting planes $H(\hat{y}, \hat{y}')$. The other components, namely solving the Quadratic Program and computing the separation oracle, are a lot faster.

### 6.3.6 Presenting my Best Kernel Parser

In this section I want to present the best model that I was able to train using the approaches presented above. It relies on the most complex input kernel, the Annotated N-Gram kernel for n-grams up to size 6 and a decay factor $\lambda = 0.5$. For the output kernel, I use the sum of two Atomic output kernels, the first one uses a hash function based on PoS2-tags and attachment distance, and the second one uses a hash function based on tokens. Obviously, using two hash functions does not alter the inference problem substantially, the sufficient weights are merely the sum of the two individual sufficient weights.

I train the model on the first 50% of the TüBa-D/Z and test on the remaining 50%. The optimization accuracy was set to $\varepsilon = 0.05$. In Table 6.6 we can see the train- and test-loss for increasing values of $C$. We can see that it achieves a test error of 6.65%, which is comparable to the errors achieved by the dependency parser using explicit features that I have presented in Section 6.2. The train error is substantially smaller, as we have also witnessed in the other experiments shown above.

| | h | | |
|---|---|---|---|
| C | Train Error | Test Error | Δ |
| 10 | 14.29% | 15.16% | 0.87% |
| 50 | 11.90% | 13.43% | 1.53% |
| 75 | 10.34% | 12.33% | 1.99% |
| 100 | 9.40% | 11.72% | 2.32% |
| 200 | 6.32% | 9.85% | 3.53% |
| 500 | 1.86% | 7.40% | 5.54% |
| 1000 | 0.61% | 6.65% | 6.04% |

**Table 6.6:** Empirical loss for different values of C. Training and testing were performed using 50% of the data respectively. $\varepsilon = 0.05$.

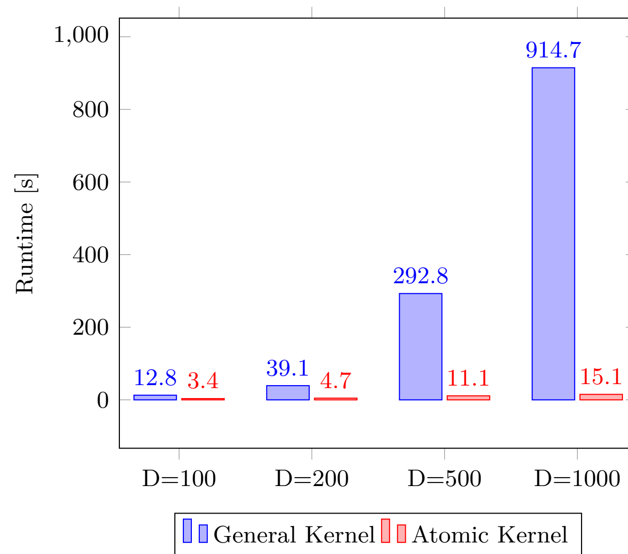



**Figure 6.6:** Runtime Comparison of Kernelized SVMs without and with optimizations for atomic kernels.

# 7 | Discussion and Outlook

In this thesis I have addressed the Machine Learning task of Structured Output Learning. I have focused on discrete structures, which first have been defined precisely. Based on the Structural Support Vector Machine which uses expert-defined, explicit feature maps, I have investigated the use of implicit feature maps defined by a Joint Tensor Kernel. Joint kernels combine a kernel for the input structures with a kernel for the output structures. This way I can utilize kernels previously proposed for Machine Learning tasks with structured inputs and can thereby reduce the amount of feature engineering required for Structured Output Prediction. When using a joint kernel, the inference problem becomes a Pre-Image problem. I have shown that the structure of this inference problem depends only on the output kernel. Thus we can say that the output kernel determines how hard it is to solve the inference problem. In turn, this means that we can choose an arbitrarily 'complicated' input kernel. I have proposed the Atomic Output Kernel which makes this problem tractable for a number of different output spaces. The Atomic Output Kernel implicitly uses a sparse feature map. By exploiting this sparseness I was able to achieve large decreases in training time.

I have applied Structured Output Learning with Joint Kernels to the learning task of Dependency Parser Training. My method achieves parsing performances comparable to state of the art methods based on millions of explicitly-defined features measured on the TüBa-D/Z treebank for dependency parsing in the German language. I have demonstrated that using Joint Tensor Kernels substantially reduces the amount of feature engineering. Instead of defining a high dimensional feature map $\phi(x, y)$ we specify only one hash-function and an input kernel. However, in order to solve the inference problem, we still have to provide a construction algorithm that returns valid output structured for the output domain. One small experiment suggests that approximate construction might be feasible.

I believe that the prediction performance can be improved by using more complex output kernels that go beyond the comparison of single atomic symbols but also

consider larger substructures. In the case of trees, such substructures could be limited to paths, e.g. all paths from root to leaf, or sets of neighboring edges. One might even consider all substructures that two trees share, as done by the convolution tree kernel introduced in Chapter 3. Obviously the question is: How can we still solve the pre-image problem efficiently? I leave this question open for future research.

# Bibliography


[1] B Bohnet. Very high accuracy and fast dependency parsing is not a contradiction. *Proceedings of the 23rd International Conference on Computational Linguistics (COLING '10)*, (August):89–97, 2010.

[2] Sabine Buchholz and Erwin Marsi. CoNLL-X shared task on multilingual dependency parsing. *Proceedings of the Tenth Conference on Computational Natural Language Learning*, (June):149–164, 2006.

[3] Sebastian Buschjäger, Lukas Pfahler, and Katharina Morik. Discovering Subtle Word Relations in Large German Corpora. *Proceedings of the 3rd Workshop on Challenges in the Management of Large Corpora (CMLC-3)*, pages 11–14, 2015.

[4] Danqi Chen and Christopher D. Manning. A Fast and Accurate Dependency Parser using Neural Networks. *EMNLP*, (i), 2014.

[5] Jianlin Cheng, Allison N. Tegge, and Pierre Baldi. Machine learning methods for protein structure prediction. *IEEE reviews in biomedical engineering*, 1:41–49, 2008.

[6] Yoeng-Jin Chu and Tseng-Hong Liu. On shortest arborescence of a directed graph. *Scientia Sinica*, 14(10):1396, 1965.

[7] Michael Collins and Nigel Duffy. Convolution Kernels for Natural Language. In *Advances in Neural Information Processing Systems (NIPS)*, 2001.

[8] Michael Collins and Nigel Duffy. New ranking algorithms for parsing and tagging: Kernels over Discrete Structures, and the Voted Perceptron. *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02*, (July):263, 2001.

[9] Michael Collins and Nigel Duffy. Parsing with a single Neuron: Convolution Kernels for Natural Language Problems. 2001.

[10] Thomas H. Cormen, Clifford Stein, Charles E. Leiserson, and Rivest Ronald L. *Introduction to Algorithms*. Third edition, 2009.

[11] C. Cortes, M. Mohri, and J. Weston. A general regression technique for learning transductions. *Proceedings of the 22nd International Conference on Machine Learning*, pages 153–160, 2005.

[12] Koby Crammer and Yoram Singer. Ultraconservative Online Algorithms for Multiclass Problems. *Journal of Machine Learning Research*, 3:951–991, 2003.

[13] Hal Daumé, John Langford, and Daniel Marcu. Search-based structured prediction. *Machine Learning*, 75(3):297–325, 2009.

[14] Janardhan Rao Doppa, Alan Fern, and Prasad Tadepalli. Structured Prediction via Output Space Search. *Journal of Machine Learning Research*, 2014.

[15] Jack Edmonds. Optimum branchings, 1967.

[16] Jason M. Eisner. Three new probabilistic models for dependency parsing. *Proceedings of the 16th Conference on Computational linguistics*, 1996.

[17] Kilian Foth. Eine Umfassende Dependenzgrammatik des Deutschen. 2006.

[18] Thomas Gärtner. *Kernels for Structured Data*. World Scientific, Singapore, 2008.

[19] Thomas Gärtner and Shankar Vembu. On Structured Output Training: Hard Cases and an Efficient Alternative. *Machine Learning*, 76(2-3):227–242, 2009.

[20] Sébastien Giguère, François Laviolette, Mario Marchand, Denise Tremblay, Sylvain Moineau, Xinxia Liang, Éric Biron, and Jacques Corbeil. Machine Learning Assisted Design of Highly Active Peptides for Drug Discovery. *PLOS Computational Biology*, 11(4):e1004074, 2015.

[21] Sébastien Giguère, Mario Marchand, François Laviolette, Alexandre Drouin, and Jacques Corbeil. Learning a peptide-protein binding affinity predictor with kernel ridge regression. *BMC bioinformatics*, 14:82, 2013.

[22] Sébastien Giguère, Amélie Rolland, François Laviolette, and Mario Marchand. Algorithms for the Hard Pre-Image Problem of String Kernels and the General Problem of String Prediction. In *Proceedings of The 32nd International Conference on Machine Learning*, 2015.

[23] Thomas Hofmann, Bernhard Schölkopf, and Alexander J. Smola. Kernel Methods in Machine Learning. *Annals of Statistics*, 36(3):1171–1220, 2008.

[24] Liang Huang and David Chiang. Better k -best Parsing. In *Proceedings of the 9th International Workshop on Parsing Technologies*, 2005.

[25] Thorsten Joachims. *Learning to Classify Text using Support Vector Machines*, volume 668 of *Kluwer International Series in Engineering and Computer Science*. Kluwer, 2002.

[26] Thorsten Joachims, Thomas Finley, and Chun Nam John Yu. Cutting-plane Training of Structural SVMs. *Machine Learning*, 77:27–59, 2009.

[27] Michael Jünger, Thomas Liebling, Denis Naddef, George Nemhauser, William Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence Wolsey. *50 Years of Integer Programming 1958-2008*. Springer Verlag, Berlin, Heidelberg.

[28] Terry Koo. *Advances in Discriminative Dependency Parsing*. PhD thesis, 2010.

[29] Terry Koo and Michael Collins. Efficient third-order dependency parsers. *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL '10)*, (July):1–11, 2010.

[30] Sandra Kübler, Ryan McDonald, and Joakim Nivre. *Dependency Parsing*. Morgan & Claypool, 2009.

[31] James Tin Yau Kwok and Ivor Wai Hung Tsang. The pre-image problem in kernel methods. *IEEE Transactions on Neural Networks*, 15(6):1517–1525, 2004.

[32] Simon Lacoste-Julien, Martin Jaggi, Mark Schmidt, and Patrick Pletscher. Block-Coordinate Frank-Wolfe Optimization for Structural SVMs. *ICML*, 28:9, 2013.

[33] Christoph H. Lampert and Matthew B. Blaschko. Structured prediction by joint kernel support estimation. *Machine Learning*, 77(2-3):249–269, 2009.

[34] Huma Lodhi, Craig Saunders, John Shawe-Taylor, Nello Cristianini, and Chris Watkins. Text Classification using String Kernels. *Journal of Machine Learning Research*, 2:419–444, 2002.

[35] André F. T. Martins. The Geometry of Constrained Structured Prediction: Applications to Inference and Learning of Natural Language Syntax. *PhD thesis*, 2012.

[36] André F. T. Martins, Mario A. T. Figueiredo, Pedro M. Q. Aguiar, Noah A. Smith, and Eric P. Xing. Online Multiple Kernel Learning for Structured Prediction. *Stat*, pages 1–21, 2010.

[37] André F.T. Martins, Noah A. Smith, and Eric P. Xing. Concise Integer Linear Programming Formulations for Dependency Parsing. 2007.

[38] Ryan McDonald. Discriminative Learning and Spanning Tree Algorithms for Dependency Parsing. *PhD Thesis*, pages 1–240, 2006.

[39] Ryan McDonald, Koby Crammer, and Fernando Pereira. Online Large-Margin Training of Dependency Parsers. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, volume 32, pages 91–98, 2005.

[40] Ryan McDonald, Kevin Lerman, and Fernando Pereira. Multilingual dependency analysis with a two-stage discriminative parser. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, pages 216–220. Association for Computational Linguistics, 2006.

[41] Ryan Mcdonald and Fernando Pereira. Online Learning of Approximate Dependency Parsing Algorithms. *In Proc. of EACL*, pages 81–88, 2006.

[42] Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. Non-projective dependency parsing using spanning tree algorithms. *HLT '05: Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, 18:523–530, 2005.

[43] Sebastian Mika, Bernhard Schölkopf, Alex Smola, Klaus-Robert Müller, Matthias Scholz, and Gunnar Rätsch. Kernel PCA and De-Noising in Feature Spaces. *Analysis*, 11(i):536–542, 1999.

[44] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic Regularities in Continuous Space Word Representations. In *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 9-14, 2013, Westin Peachtree Plaza Hotel, Atlanta, Georgia,* {*USA*}, pages 746–751, 2013.

[45] Priti Mishra and Margaret H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, 1992.

[46] Joakim Nivre, Johan Hall, and Jens Nilsson. MaltParser : A Data-Driven Parser-Generator for Dependency Parsing. *Proceedings of the fifth international conference on Language Resources and Evaluation (LREC2006)*, pages 2216–2219, 2006.

[47] Wenzhe Pei, Tao Ge, and Baobao Chang. An Effective Neural Network Model for Graph-based Dependency Parsing. *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 313–322, 2015.

[48] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. *Advances in Neural Information Processing Systems (NIPS)*, 2007.

[49] Ali Rahimi and Benjamin Recht. Uniform approximation of functions with random bases. *46th Annual Allerton Conference on Communication, Control, and Computing*, pages 555–561, 2008.

[50] Ali Rahimi and Benjamin Recht. Weighted sums of random kitchen sinks: Replacing minimization with randomization in learning. *Advances in Neural Information Processing Systems (NIPS)*, 2009.

[51] Stephane Ross and J. Andrew Bagnell. Reinforcement and Imitation Learning via Interactive No-Regret Learning. page 14, 2014.

[52] Alexander M. Rush, David Sontag, Michael Collins, and Tommi Jaakkola. On Dual Decomposition and Linear Programming Relaxations for Natural Language Processing. *EMNLP*, (October):1–11, 2010.

[53] Bernhard Scholkopf, Sebastian Mika, Chris J. C. Burges, Phillip Knirsch, Klaus-Robert Müller, Gunnar Rätsch, and Alexander J. Smola. Input space versus feature space in kernel-based methods. *IEEE Transactions on Neural Networks*, 10(5):1000–1017, 1999.

[54] Bernhard Scholkopf and Alexander J. Smola. *Learning with Kernels*. PhD thesis, 2002.

[55] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, and Andrew Cotter. Pegasos: Primal estimated sub-gradient solver for SVM. *Mathematical Programming*, 127:3–30, 2011.

[56] Robert Endre Tarjan. Finding optimum branchings. *Networks*, 7:25–35, 1977.

[57] Ben Taskar, Dan Klein, Michael Collins, Daphne Koller, and Christopher D. Manning. Max-Margin Parsing. *EMNLP*, 2004.

[58] Heike Telljohann, Erhard W Hinrichs, Sandra Kübler, Heike Zinsmeister, and Kathrin Beck. Stylebook for the Tübingen treebank of written German (TüBa-D/Z).

[59] Choon Hui Teo, Alex Smola, S. V.N. Vishwanathan, and Quoc Viet Le. A Scalable Modular Convex Solver for Regularized Risk Minimization. *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '07*, pages 727–736, 2007.

[60] Ioannis Tsochantaridis, Thorsten Joachims, Thomas Hofmann, and Yasemin Altun. Large Margin Methods for Structured and Interdependent Output Variables. *Journal of Machine Learning Research (JMLR)*, 6:1453–1484, 2005.

[61] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory.* Springer, second edition, 2000.

[62] Andreas Vlachos. An investigation of imitation learning algorithms for structured prediction. *Workshop on Reinforcement Learning*, pages 1–10, 2012.

[63] David J. Weiss and Benjamin Taskar. Structured Prediction Cascades. *Aistats*, pages 916–923, 2010.

[64] Jason Weston, Gökhan Bakır, Bernhard Schölkopf, Oliver Bousquet, Tobias Mann, and William Stafford Noble. Joint Kernel Maps. In *Predicting Structured Data*, page 67ff. 2007.

[65] Jason Weston, Olivier Chapelle, André Elisseeff, Bernhard Sch, and Vladimir Vapnik. Kernel Dependency Estimation. *Advances in Neural Information Processing Systems (NIPS)*, pages 873 – 880, 2002.

[66] R.C. Williamson, A.J. Smola, and B. Scholkopf. Generalization performance of regularization networks and support vector machines via entropy numbers of compact operators. *IEEE Transactions on Information Theory*, 47(6):2516–2532, 2001.

[67] Hiroyasu Yamada and Yuji Matsumoto. Statistical Dependency Analysis with Support Vector Machines. *In Proceedings of IWPT*, pages 195–206, 2003.

[68] Chun-Nam John Yu and Thorsten Joachims. Training Structural SVMs with Kernels using Sampled Cuts. *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD 08*, page 794, 2008.

# Appendices

# A Glossary

| Expression | Notation | Description |
|---|---|---|
| Atomic Kernel | | See Section 4.2.2 |
| Construction algorithm | | Algorithm that computes an output structure from given sufficient weights |
| Cutting Plane | $\bar{y} \in \mathcal{W}$ | A tuple of $n$ output structures, See (2.12) in Section 2.2.2. |
| Discrete Output Structure | | See Definition 2.1.2 on page 7 |
| Discrete Structure | | See Definition 2.1.1 on page 6 |
| Feature Map | $\phi(x, y)$ | See Joint Feature Map |
| Indicator Function | $1_{expr}$, e.g. $1_{x=x'}$ | Function that is 1 if *expr* is true and 0 otherwise |
| Inference Problem | $\hat{y}(x)$ | The problem of computing a prediction given a model. In the explicit case: $\arg\max_y \langle w, \phi(x, y) \rangle$, for the implicit case see (4.9) and (4.14) |
| Input Kernel | $k_{\mathcal{X}}(x, x')$ | A kernel $k_{\mathcal{X}} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ |
| Joint Feature Map | $\phi(x, y)$ | A function $\phi : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^d$ that maps an input/output pair to a feature vector, see Section 2.2.1. |
| Joint Tensor Kernel | $k((x, y), (x', y'))$ | Kernel $k((x, y), (x', y')) = k_{\mathcal{X}}(x, x') \cdot k_{\mathcal{Y}}(y, y')$ that combines an Input Kernel with an Output Kernel, see Section 4.2.1. |
| Loss Function | $l(x, y; \hat{y}(\cdot))$ or $l(x, y; w)$ | Loss function used for risk minimization that quantifies the error a model $\hat{y}(\cdot)$ or $w$ produces given an input $x$ with true output $y$, see Section 2.1.3 |
| Loss Function | $L(y, y')$ | Function $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$ that computes the difference of two output structures, see Section 2.2.1 |
| Most-violated constraint | | Structure returned by Separation Oracle. |

| | | |
|---|---|---|
| Output Kernel | $k_{\mathcal{Y}}(y, y')$ | A kernel $k_{\mathcal{Y}} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ |
| Pre-Image Problem | | The inference problem for kernelized models, see Sections 4.1.2 and 4.2.3 |
| Separation Oracle | | Explicit case: Function that computes $\arg\max_y' \langle w, \phi(x, y') \rangle + L(y, y')$ where $y$ is the true, annotated output for $x$. Implicit case see Section 4.3. |
| Sufficient Weights | $w_\sigma$ | See Section 4.2.3. |

# B Implementation Details

As part of this thesis, a Java 8 implementation of Structural Support Vector Machines with explicit features –Algorithms 2 – and with implicit features – Algorithm 3 – was developed. The source code is available at

`https://bitbucket.org/Whadup/structuredoutputprediction/`,

a jar file is available at `http://www-ai.cs.uni-dortmund.de/auto?expr=Software`. This section describes how it can be adapted and used for structured output learning tasks.

## B.1 Java Library

Both the algorithm for explicit and implicit features have been implemented in Java 8. It can use two different Quadratic Program solvers, the commercial solver Gurobi[1] which offers free licenses for academic use, and the free solver available in the oj! Algorithms Java library[2].

The implementation is heavily parallelized using the new Java 8 stream processing API[3] that allows to process data in parallel in a map-reduce-like fashion. This way, critical parts of the algorithm like computing the separation oracle and updating the inner product matrix $H$ are divided into a number of threads equal to the number of available processors.

We have seen that both algorithms require the user to further specify parts of the algorithm like the $\phi(x, y)$ mapping used in the explicit version or the kernels used in the implicit version. In the following sections, I will discuss the interfaces used to specify the learning tasks. First, I will introduce some common types used. Most importantly, all input and output types must implement the empty `Structure` interface. An input/output pair is stored in an object of the class `Example`. All sparse feature vectors have type `SparseVector`, which extends a `HashMap<String,Double>` with some algebraic operations. Dense vectors like the model are stored in a object of type `DenseVector` which extends a double array with algebraic operations.

[1] `http://gurobi.com/`

[2] `http://ojalgo.org/`

[3] `http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html`

### B.1.1 Explicit Features

The class `SVMStruct` implements Algorithm 2, which learns a model for structured output prediction. As seen in Section 2.2, we have to provide a feature mapping, a loss function, an inference algorithm and a separation oracle. To do so, we must extend the abstract class `AbstractSVMStructInstance`. We have to implement the methods

- `public SparseVector phi(X x, Y y)`
  which returns the feature mapping of an input/output pair
- `public Y predict(X x, SVMStruct svm)`
  which solves the inference problem for an input using the current model `svm`. The current weight vector can be accessed via `svm.w`.
- `public Y marginRescaling(X x, Y y, SVMStruct svm)`
  which solves the separation oracle problem for an input using the current model.
- `public double loss(Y y, Y predY)`
  which computes a positive loss for a pair of outputs.

where `X` and `Y` both implement the `Structure` interface. When we have implemented this interface, we can create a new instance of `SVMStruct svm`, specify which $C$ and $\varepsilon$ to use and train a model using the `svm.train(List<Example> data)` method. The train algorithm solves the QP problem in its dual form, as recommended by Joachims et al. [26]. This reduces the size of the problem, as we only have one variable per cutting plane and only one constraint. Also, the method applies a pruning heuristic that deletes cutting planes that have not been used in 50 iterations.

### B.1.2 Implicit Features

The class `SVMStructKernel` implements Algorithm 3. Similar to the implicit case, we have to supply a `SVMStructKernelInstance` that specifies which input kernel, output kernel and loss function to use. Recall that we can use an arbitrary input kernel that implements the interface `Kernel`. The output kernel has to implement the interface `OutputKernel`, which provides methods that solve the inference and separation oracle problem:

- `public Y preImage(Map<Y,Double> alphaY,X x)`
  solves the inference pre-image problem given a linear combination of kernel computations `alphaY` and an input `x`.
- `public Y preImagePenelized(Map<Y,Double> alphaY,Y y,X x)`
  Solves the separation oracle pre-image problem that also takes into account the loss induced by `y`.

We can specify an arbitrary loss function, as long as it fits the separation oracle pre-image problem specified by the output kernel. I encourage to implement the loss function discussed in Section 4.3.

If we want to benefit from the speedup techniques discussed in Section 4.4.2, the output kernel has to implement the interface `AtomicKernel`. This means that it has to provide a method `public SparseVector featureMap(X x, Y y)` that makes the feature map used in the output kernel explicit.

In order to train a model, we have to create a new instance of the `SVMStructKernel svm`, specify $C$ and $\varepsilon$, set the kernel instance and train the model using the `train` method as discussed above.

## B.2 RapidMiner Integration for Dependency Parsing

To demonstrate how the Structural SVM could be integrated into the RapidMiner[4] environment, I developed a small extension that offers Dependency Parser training discussed as in Chapter 6 in RapidMiner. At its core is the **Train Parser** operator, which allows to train a parser with different parameters for the Structural SVM. Furthermore I have implemented operators for loading dependency treebanks in the `.conll` format and for measuring the performance of a dependency parser using the unlabeled attachment score (UAS) loss function.

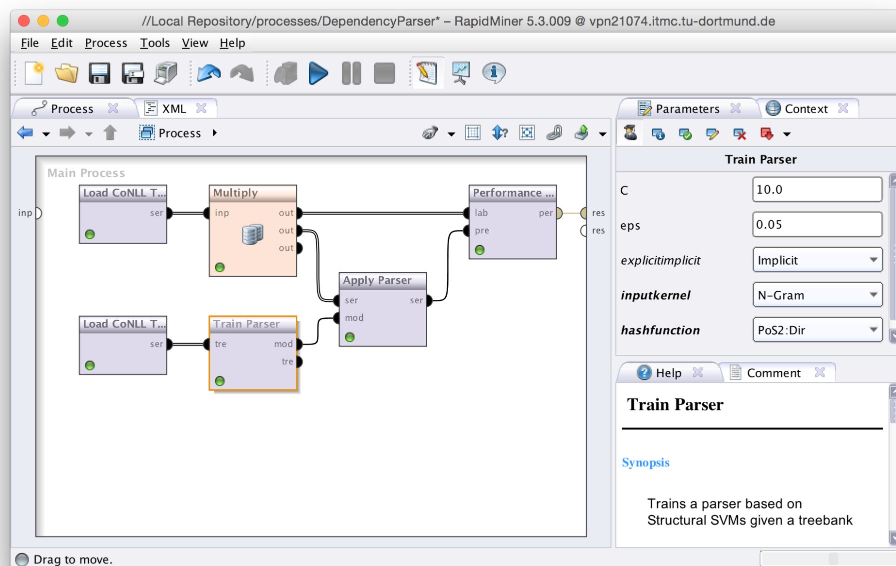



**Figure B.1:** RapidMiner Process for Training and Testing Dependency Parsers

[4] `http://rapidminer.com`