Advanced Machine Learning **Fall 2021**

Lecture 8: Structured SVMs

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

Disclaimer: *These notes are adapted from ETH's Advanced Machine Learning Course, Bishop's "Pattern Recognition and Machine Learning" book and Joachims et al.'s paper: "Predicting Structured Objects with Support Vector Machines"*

8.1 Non-Linear SVMs

The SVM soft-margin dual formulation is the following:

$$\begin{aligned} \max_{a} \quad & \sum_{i=1}^{N} a_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} a_i a_j t_i t_j \phi(x_i)^T \phi(x_j) \\ \text{subject to} \quad & 0 \leq a_i \leq C, \; i = 1, \ldots, N \\ & \sum_{i=1}^{N} a_i t_i = 0 \end{aligned}$$

For very high dimensional space, the nonlinear transformation $\phi(x)$ might be computationally too difficult to compute, when we only require the inner product $\langle \phi(x_i), \phi(x_j) \rangle$. A way to overcome this problem is making use of **Kernel** functions.

Definition 8.1 *A function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is said to be a kernel function if and only if it is an inner product $\langle \boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{x'}) \rangle$ for some (possibly infinite dimensional) mapping $\boldsymbol{\phi}(\boldsymbol{x})$.*

Definition 8.2 *A function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is said to be symmetric positive semidefinite (PSD) if it is symmetric, i.e $k(\boldsymbol{x}, \boldsymbol{x'}) = k(\boldsymbol{x'}, \boldsymbol{x})$; For any integer $m > 0$ and any set of inputs $\boldsymbol{x_1}, ..., \boldsymbol{x_m} \in \mathbb{R}^d$, the following matrix is positive semi-definite:*

$$\boldsymbol{K} = \begin{bmatrix} k(\boldsymbol{x_1}, \boldsymbol{x_1}) & \dots & k(\boldsymbol{x_1}, \boldsymbol{x_m}) \\ \vdots & \ddots & \vdots \\ k(\boldsymbol{x_m}, \boldsymbol{x_1}) & \dots & k(\boldsymbol{x_m}, \boldsymbol{x_m}) \end{bmatrix} \succeq \boldsymbol{0}$$

*This matrix, with (i, j)-th entry equal to $k(\boldsymbol{x_i}, \boldsymbol{x_j})$, is called the **Gram matrix**.*

Theorem 8.3 *The above two definitions are equivalent. That is, k is a kernel function if and only if it is symmetric PSD.*

Proof: The "only if" part is easy to show (at least when ϕ is finite-dimensional): The inner product is certainly symmetric, and the Gram matrix can be written as $\boldsymbol{K} = \boldsymbol{\Phi}^\intercal \boldsymbol{\Phi}$, where $\boldsymbol{\Phi} \in \mathbb{R}^{\dim(\boldsymbol{\Phi}) \times m}$ contains the m feature vectors $\{\boldsymbol{\phi}(\boldsymbol{x_t})\}_{t=1}^m$ as columns. The matrix $\boldsymbol{K} = \boldsymbol{\Phi}^\intercal \boldsymbol{\Phi}$ is trivially positive semidefinite, since for any $\boldsymbol{z}$ we have $\boldsymbol{z}^\intercal \boldsymbol{\Phi}^\intercal \boldsymbol{\Phi} \boldsymbol{z} = \|\boldsymbol{\Phi} \boldsymbol{z}\|^2 \geq 0$.

The "if" part is more challenging and comes from *Mercer's Theorem.* ■

Theorem 8.4 ***Mercer's Theorem***
Recall: any positive definite matrix $\boldsymbol{K}$ can be represented using an eigendecomposition of the form $\boldsymbol{K} = \boldsymbol{U}^\intercal \boldsymbol{\Lambda} \boldsymbol{U}$, where $\boldsymbol{\Lambda}$ is a diagonal matrix of eigenvalues $\lambda_i > 0$, and $\boldsymbol{U}$ is a matrix containing the eigenvectors.
Now consider element (i, j) of $\boldsymbol{K}$:

$$k_{ij} = (\boldsymbol{\Lambda}^{\frac{1}{2}} \boldsymbol{U}_{:i})^\intercal (\boldsymbol{\Lambda}^{\frac{1}{2}} \boldsymbol{U}_{:j})$$

where $\boldsymbol{U}_{:i}$ is the i'th column of $\boldsymbol{U}$. If we define $\boldsymbol{\phi}(\boldsymbol{x_i}) = \boldsymbol{\Lambda}^{\frac{1}{2}} \boldsymbol{U}_{:i}$, then we can write:

$$k_{ij} = \boldsymbol{\phi}(\boldsymbol{x_i})^\intercal \boldsymbol{\phi}(\boldsymbol{x_j}) = \sum_m \phi_m(\boldsymbol{x_i}) \phi_m(\boldsymbol{x_j})$$

Thus we see that the entries in the kernel matrix can be computed by performing an inner product of some feature vectors that are implicitly defined by the eigenvectors of the kernel matrix. This idea can be generalized to apply to kernel functions, not just kernel matrices.
For example consider the **quadratic kernel** $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x'}) = \langle \boldsymbol{x}, \boldsymbol{x'} \rangle^2$. In 2d, we have:

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x'}) = (x_1 x_1' + x_2 x_2')^2 = x_1^2 (x_1')^2 + 2 x_1 x_2 x_1' x_2' + x_2^2 (x_2')^2$$

We can write this as $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x'}) = \boldsymbol{\phi}(\boldsymbol{x})^\intercal \boldsymbol{\phi}(\boldsymbol{x})$ if we define $\boldsymbol{\phi}(\boldsymbol{x_1}, \boldsymbol{x_2}) = [x_1^2, \sqrt{2} x_1 x_2, x_2^2] \in \mathbb{R}^3$. So we embed the 2d inputs $\boldsymbol{x}$ into a 3d feature space $\boldsymbol{\phi}(\boldsymbol{x})$.
Now consider the RBF kernel. In this case, the corresponding feature representation is infinite dimensional. However, by working with kernel functions, we can avoid having to deal with infinite dimensional vectors.

8.1.1 Kernel Engineering

Given two valud kernels $\mathcal{K}_1(\boldsymbol{x}, \boldsymbol{x'})$ and $\mathcal{K}_2(\boldsymbol{x}, \boldsymbol{x'})$, we can create a new kernel using any of the following methods:

- $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x'}) = c\mathcal{K}_1(\boldsymbol{x}, \boldsymbol{x'})$, for any constant $c > 0$
- $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x'}) = f(\boldsymbol{x})\mathcal{K}_1(\boldsymbol{x}, \boldsymbol{x'})f(\boldsymbol{x'})$, for any function f
- $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x'}) = q(\mathcal{K}_1(\boldsymbol{x}, \boldsymbol{x'}))$, for any function polynomial q with non-negative coefficients
- $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x'}) = \exp(\mathcal{K}_1(\boldsymbol{x}, \boldsymbol{x'}))$
- $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x'}) = \boldsymbol{x}^\intercal \boldsymbol{A} \boldsymbol{x'}$, for any PSD matrix $\boldsymbol{A}$

For example, suppose we start with the linear kernel $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x'}) = \boldsymbol{x}^\intercal \boldsymbol{x'}$. We know this is a valid Mercer kernel, since the corresponding Gram matrix is just the (scaled) covariance matrix of the data. From the above rules, we can see that the polynomial kernel $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x'}) = (\boldsymbol{x}^\intercal \boldsymbol{x'})^M$is a valid Mercer kernel. This contains all monomials of order M. For example, if M = 2 and the inputs are 2d, we have:

$$(\boldsymbol{x}^\intercal \boldsymbol{x'})^2 = (x_1 x_1' + x_2 x_2')^2 = (x_1 x_1')^2 + (x_2 x_2)^2 + (x_1 x_1')(x_2 x_2')$$

We can generalize this to contain all terms up to degree M by using the kernel $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x'}) = (\boldsymbol{x}^\intercal \boldsymbol{x'} + c)^M$. For example, if M = 2 and the inputs are 2d, we have:

$$(\boldsymbol{x}^\intercal \boldsymbol{x'} + 1)^2 = (x_1 x_1')^2 + (x_1 x_1')(x_2 x_2') + (x_1 x_1') + (x_2 x_2)(x_1 x_1') + (x_2 x_2')^2 + (x_2 x_2') + (x_1 x_1') + (x_2 x_2') + 1$$

We can also use the above rules to establish that the Gaussian kernel is a valid kernel. To see this, note that:

$$\|\boldsymbol{x} - \boldsymbol{x'}\|^2 = \boldsymbol{x}^\intercal \boldsymbol{x} + (\boldsymbol{x'})^\intercal \boldsymbol{x'} - 2\boldsymbol{x}^\intercal \boldsymbol{x'}$$

and hence

$$\mathcal{K}(\boldsymbol{x}, \boldsymbol{x'}) = \exp(-\|\boldsymbol{x} - \boldsymbol{x'}\|^2 / 2\sigma^2) = \exp(-\boldsymbol{x}^\intercal \boldsymbol{x}/2\sigma^2) \exp(\boldsymbol{x}^\intercal \boldsymbol{x'}/\sigma^2) \exp(-(\boldsymbol{x'})^\intercal \boldsymbol{x'}/2\sigma^2)$$

is a valid kernel.

8.1.2 Combining kernels by addition and multiplication

We can also combine kernels using addition or multiplication:

- $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x'}) = \mathcal{K}_1(\boldsymbol{x}, \boldsymbol{x'}) + \mathcal{K}_2(\boldsymbol{x}, \boldsymbol{x'})$
- $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x'}) = \mathcal{K}_1(\boldsymbol{x}, \boldsymbol{x'}) \times \mathcal{K}_2(\boldsymbol{x}, \boldsymbol{x'})$

Multiplying two positive-definite kernels together always results in another positive definite kernel. This is a way to get a conjunction of the individual properties of each kernel. In addition, adding two positive-definite kernels together always results in another positive definite kernel. This is a way to get a disjunction of the individual properties of each kernel.

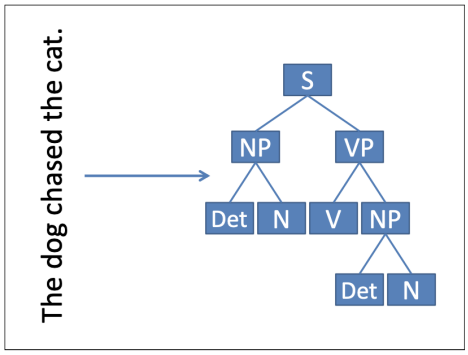
8.1.3 Kernels for structured inputs

Kernels are particularly useful when the inputs are structured objects, such as strings and graphs, since it is often hard to "featurize" variable-sized inputs. For example, we can define a string kernel which compares strings in terms of the number of n-grams they have in common. We can also define kernels on graphs . For example, the random walk kernel conceptually performs random walks on two graphs simultaneously, and then counts the number of paths that were produced by both walks.

8.2 Structured SVMs

Consider the problem of natural language parsing illustrated in Figure 8.2. A parser takes as input a natural language sentence, and the desired output is the parse tree decomposing the sentence into its constituents. How can we take, say, an SVM and learn a rule for predicting trees?

Figure 8.1: Predicting trees in natural language parsing.

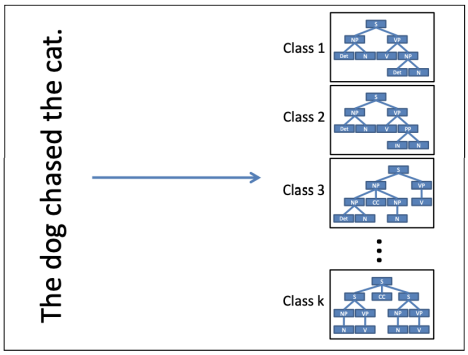


Obviously, this question arises not only for learning to predict trees, but similarly for a variety of other structured and complex outputs. Structured output prediction is the name for such learning tasks, where one aims at learning a function $h : X \to Y$ mapping inputs $x \in X$ to complex and structured outputs $y \in Y$.

On an abstract level, a structured prediction task is much like a multi-class learning task. Each possible structure $y \in Y$ (e.g. parse tree) corresponds to one class (see Figure 8.3), and classifying a new example x amounts to predicting its correct "class".

While the following derivation of structural SVMs starts from multi-class SVMs, there are three key problems that need to be overcome. **All of these problems arise from the huge number $|\mathbf{Y}|$ of classes**.

Figure 8.2: Structured output prediction as a multiclass problem.



8.2.1 Problem 1: Structural SVM Formulation

We start the derivation of the structural SVM from the multi-class SVM. These multi-class SVMs use one weight vector $\mathbf{w_y}$ for each class $\mathbf{y}$. Each input $\mathbf{x}$ now has a score for each class $\mathbf{y}$ via $\mathbf{f(x, y)} = \mathbf{w_y}\phi(\mathbf{x})$. Here $\phi(\mathbf{x})$ is a vector of binary or numeric features extracted from $\mathbf{x}$. Thus, every feature will have an additively weighted influence in the modeled compatibility between inputs $\mathbf{x}$ and classes $\mathbf{y}$. To classify $\mathbf{x}$, the prediction rule $\mathbf{h(x)}$ then simply chooses the highest-scoring class:

$$h(x) = \max_{y \in Y} f(x, y) \tag{8.1}$$

as the predicted output. This will result in the correct prediction $\mathbf{y}$ for input $\mathbf{x}$ provided the weights $\mathbf{w} = (\mathbf{w_1}, ..., \mathbf{w_k})$ have been chosen such that the inequalities $\mathbf{f(x, y_i)} < \mathbf{f(x, y)}$ hold for all incorrect outputs $\mathbf{y_i} \neq \mathbf{y}$.

$$\begin{array}{ll} \min\limits_{\mathbf{w}} & \frac{1}{2}\|w\|^2 \\ \text{subject to} & f(x_i, y_i) - f(x_i, y) = (w_{y_i} - w_y) \cdot \phi(x_i) \geq 1, \ \forall i, y \neq y_i \end{array} \tag{8.2}$$

The first challenge in using (8.8) for structured outputs is that, while there is generalization across inputs $\mathbf{x}$, there is no generalization across outputs. This is due to having a separate weight vector $\mathbf{w_y}$ for each class $\mathbf{y}$. Furthermore, since the number of possible outputs can become very large (or infinite), naively reducing structured output prediction to multi-class classification leads to an undesirable blow-up in the overall number of parameters and in the overall number of inequalities, which is $\mathbf{n(k-1)}$.

The key idea in overcoming these problems is to extract features from input-output pairs using a so-called joint feature map $\mathbf{\Psi(x, y)}$ instead of $\mathbf{\Phi(x)}$. These joint features will allow us to generalize across outputs and to define meaningful scores even for outputs that were never actually observed in the training data. At the same time, since we will define compatibility functions via $\mathbf{f(x, y)} = \mathbf{w} \cdot \mathbf{\Psi(x, y)}$, the number of parameters will simply equal the number of features extracted via $\mathbf{\Psi}$, which may not depend on $|\mathbf{Y}|$. One can then use the formulation in (8.8) with the more flexible definition of $\mathbf{f}$ via $\mathbf{\Psi}$ to arrive at the following (hard-margin) optimization problem:

$$\begin{aligned} \min_{\mathbf{w}} \quad & \frac{1}{2}\|w\|^2 \\ \text{subject to} \quad & w \cdot \Psi(x_i, y_i) - w \cdot \Psi(x_i, y) \geq 1, \ \forall i, y \neq y_i \end{aligned}$$

In other words, find a weight vector $\mathbf{w}$ of an input-output compatibility function $\mathbf{f}$ that is linear in some joint feature map $\mathbf{\Psi}$ so that on each training example it scores the correct output higher by a fixed margin than every alternative output, while having low complexity (i.e. small norm $\|w\|$). Note that the number of linear constraints is still $n(|Y|-1)$. The design of the features $\mathbf{\Psi}$ is problem-specific, and it is a strength of the developed methods to allow for a great deal of flexibility in how to choose it.

8.2.2 Problem 2: Inconsistent Training Data

So far we have tacitly assumed that the optimization problem has a solution, i.e. there exists a weight vector that simultaneously fulfills all margin constraints. In practice this may not be the case, either because the training data is inconsistent or because our model class is not powerful enough. If we allow for mistakes, though, we must be able to quantify the degree of mismatch between a prediction and the correct output, since usually different incorrect predictions vary in quality. This is exactly the role played by a loss function, formally $\Delta : Y \times Y \to \mathbb{R}$, where $\mathbf{\Delta}(\mathbf{y_i}, \mathbf{y})$ is the loss (or cost) for predicting $\mathbf{y}$, when the correct output is $\mathbf{y_i}$. Like the choice of $\mathbf{\Psi}$, defining $\mathbf{\Delta}$ is problem-specific. One can convert this back into a quadratic program as follows:

$$\begin{aligned} \min_{\mathbf{w}, \xi_i \geq 0} \quad & \frac{1}{2}\|w\|^2 + \frac{C}{n}\sum_{i=1}^{n}\xi_i \\ \text{subject to} \quad & w \cdot \Psi(x_i, y_i) - w \cdot \Psi(x_i, y) \geq \Delta(y_i, y) - \xi_i, \ \forall i, y \neq y_i \end{aligned}$$

Note that we added the $\frac{\mathbf{1}}{\mathbf{n}}$ term, this is a guarantee that we are minimizing the empirical risk:

Theorem 8.5 *If w^*, ξ^* are optimal, then the empirical risk of w^* with respect to $\mathbf{\Delta}$:*

$$\frac{1}{n}\sum_{i=1}^{n}\Delta(y_i, h_{w^*}(x_i)) \leq \frac{1}{n}\sum_{i=1}^{n}\xi_i^*$$

Proof: Suffices to prove that $\Delta(y_i, h_{w^*}(x_i)) \leq \xi_i^*, \ \forall i$

If $h_{w^*}(x_i) = y_i$, then $\Delta(y_i, h_{w^*}(x_i)) = 0 \leq \xi_i^*$

If $h_{w^*}(x_i) = y \neq y_i$, then:

$$\begin{aligned} w^{*T} \cdot \Psi(x_i, y_i) < w^{*T} \cdot \Psi(x_i, y) \implies w^{*T} \cdot \Psi(x_i, y_i) - w^{*T} \cdot \Psi(x_i, y) = \delta < 0 \\ \delta + \xi_i^* \geq \Delta(y_i, y) \implies \xi_i^* \geq \Delta(y_i, y), \ \forall i \text{ (from the optimization constraint)} \end{aligned}$$

■

8.2.3 Problem 3: Efficient Training

Last, but not least, we need a training algorithm that finds the optimal $\mathbf{w}$ solving the quadratic program. Since there is a constraint for every incorrect label $\mathbf{y}$, we cannot enumerate all constraints and simply give the optimization problem to a standard QP solver. Instead, we propose to use the cutting-plane Algorithm 1. The key idea is to iteratively construct a working set of constraints $\mathbf{W}$ that is equivalent to the full set of constraints up to a specified precision ϵ.

Algorithm 1 for training structural SVMs (margin-rescaling).

1: Input: $S = ((\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n))$, C, ϵ
2: $\mathcal{W} \leftarrow \emptyset$, $\boldsymbol{w} = \mathbf{0}$, $\xi_i \leftarrow 0$ for all $i = 1, ..., n$
3: **repeat**
4: **for** i=1,...,n **do**
5: $\hat{\mathbf{y}} \leftarrow \operatorname{argmax}_{\hat{\mathbf{y}} \in \mathcal{Y}} \{\Delta(\mathbf{y}_i, \hat{\mathbf{y}}) + \boldsymbol{w} \cdot \Psi(\mathbf{x}_i, \hat{\mathbf{y}})\}$
6: **if** $\boldsymbol{w} \cdot [\Psi(\mathbf{x}_i, \mathbf{y}_i) - \Psi(\mathbf{x}_i, \hat{\mathbf{y}})] < \Delta(\mathbf{y}_i, \hat{\mathbf{y}}) - \xi_i - \epsilon$ **then**
7: $\mathcal{W} \leftarrow \mathcal{W} \cup \{\boldsymbol{w} \cdot [\Psi(\mathbf{x}_i, \mathbf{y}_i) - \Psi(\mathbf{x}_i, \hat{\mathbf{y}})] \geq \Delta(\mathbf{y}_i, \hat{\mathbf{y}}) - \xi_i\}$
8: $(\boldsymbol{w}, \boldsymbol{\xi}) \leftarrow \operatorname{argmin}_{\boldsymbol{w}, \boldsymbol{\xi} \geq \mathbf{0}} \frac{1}{2} \boldsymbol{w} \cdot \boldsymbol{w} + \frac{C}{n} \sum_{i=1}^{n} \xi_i$ s.t. $\mathcal{W}$
9: **end if**
10: **end for**
11: **until** $\mathcal{W}$ has not changed during iteration
12: return($\boldsymbol{w}$,$\boldsymbol{\xi}$)

Starting with an empty $\mathbf{W}$ and $\mathbf{w} = \mathbf{0}$, Algorithm 1 iterates through the training examples. For each example, the **argmax** in Line 5 finds the most violated constraint of the quadratic program. If this constraint is violated by more than ϵ (Line 6), it is added to the working set $\mathbf{W}$ in Line 7 and a new $\mathbf{w}$ is computed by solving the quadratic program over the new $\mathbf{W}$ (Line 8). The algorithm stops and returns the current $\mathbf{w}$ if $\mathbf{W}$ did not change between iterations.

But how long does it take to terminate? It can be shown that Algorithm 1 always terminates in a polynomial number of iterations that is independent of the cardinality of the output space $|Y|$. In fact, a refined version of Algorithm 1 always terminates after adding at most $O(C\epsilon^{-1})$ constraints to W. Note that the number of constraints is not only independent of $|Y|$, but also independent of the number of training examples n, which makes it an attractive training algorithm even for conventional SVMs.
While the number of iterations is small, the **argmax** in Line 5 might be expensive to compute. In general, this is true, but note that this **argmax** is closely related to the **argmax** for computing a prediction $\mathbf{h}(\mathbf{x})$. It is therefore called the "loss-augmented"inference problem, and often the prediction algorithm can be adapted to efficiently solve the loss-augmented inference problem as well.