Constraint Learning: An Appetizer

Stefano Teso(✉)

KU Leuven, Leuven, Belgium
stefano.teso@cs.kuleuven.be

Abstract. Constraints are ubiquitous in artificial intelligence and operations research. They appear in logical problems like propositional satisfiability, in discrete problems like constraint satisfaction, and in full-fledged mathematical optimization tasks. Constraint learning enters the picture when the structure or the parameters of the constraint satisfaction/optimization problem to be solved are (partially) unknown and must be inferred from data. The required supervision may come from offline sources or gathered by interacting with human domain experts and decision makers. With these lecture notes, we offer a brief but self-contained introduction to the core concepts of constraint learning, while sampling from the diverse spectrum of constraint learning methods, covering classic strategies and more recent advances. We will also discuss links to other areas of AI and machine learning, including concept learning, learning from queries, structured-output prediction, (statistical) relational learning, preference elicitation, and inverse optimization.

Keywords: Machine learning · Constraint satisfaction · Constraint optimization · Interactive learning

1 Introduction

Constraint learning is the task of acquiring constraint satisfaction or optimization problems from examples of solutions and non-solutions or other types of supervision. This document overviews selected topics in constraint learning and has no ambition of completeness. More specifically, we will discuss learning of *hard* constraints, which implicitly define a satisfaction problem; learning of *soft* constraints, where competing constraints are assigned different preferences; and *interactive* learning of hard or soft constraints, where the learning algorithm obtains supervision by interacting with an oracle (e.g. a human expert, a non-expert, a measurement apparatus). For a more in-depth guide to constraint learning and related areas, we refer the interested reader to the works by O'Sullivan [35], Bessiere et al. [9], Lombardi et al. [30], and De Raedt et al. [15]. Further pointers to the literature are provided in Sect. 6.

Why Constraint Learning?

Constraints are extremely popular from a modeling perspective, and appear in all sort of satisfaction and optimization problems in both artificial intelligence and operations research. Plenty of frameworks for modelling and solving problems involving constraints exist, from propositional satisfiability (SAT) and linear programming (LP), to constraint satisfaction [42], to more sophisticated alternatives that combine combinatorial and numerical elements, like mixed-integer linear programming (MILP). Given a formal specification of a constraint satisfaction or optimization problem of interest, it is often sufficient to feed it to an appropriate solver to obtain an appropriate solution[1].

A major bottleneck of this setup is that obtaining a formal constraint theory is non-obvious: designing an appropriate, working constraint satisfaction or optimization problem requires both domain and modeling expertise. For this reason, in many cases a modeling expert is hired and has to interact with domain expert to acquire informal requirements and turn them into a valid constraint theory. This process is can be expensive and time consuming.

The idea is then to replace or assist this process by acquiring a constraint theory directly from examples. In the simplest setting, when learning hard constraints—which implicitly define constraint satisfaction models like propositional concepts [54], satisfiability modulo theory formulas [5], and general constraint theories over discrete variables [9]—one is given a data set of positive and negative configurations and searches for a theory that covers (i.e. classifies as feasible) all of the positive examples and none of the negative ones. At its core, this is a form of concept learning [9]. Several variants and extensions of this setup have been proposed, including learning of soft constraints [41], which can be violated and are assigned different degrees of importance, and full-fledged optimization problems [36]. In the following, we will briefly cover the most prominent of these settings.

Of course, in practice the learned constraint theory may be approximate or wrong. In this case, two things can occur: either more data is provided and the theory is refined accordingly, or a human expert can revise the model via inspection and debugging. Here the goal is not to replace human experts, but rather to aid them by generating a reasonable initial theory consistent with all available supervision. Notice that in some settings the model does not have to be perfect. For instance, in recommendation the goal is to acquire a soft constraint theory that knows enough about the preferences of the target user to be able to provide reasonable personalized recommendations: so long as the model manages to identify some interesting products, the goal is met [39]. In this case, approximate models are acceptable and no human intervention is needed.

Dimensions of Constraint Learning

Formalisms and approaches to constraint learning can be roughly grouped based on three criteria:

[1] One should of course keep in mind that many constrained satisfaction/optimization problems can be NP-hard, so obtaining a solution in an acceptable time may still be tricky; see below for some examples.

(a) The type of constraints being learned. One can learn hard constraints, which define pure satisfaction models, soft constraints, which implicitly define optimization models, or (in principle) both. We will consider approaches to learning hard and soft constraints in Sects. 3 and 4, respectively. There is essentially no literature on learning both hard and soft constraints, so we will skip this topic.
(b) The technique used to represent and search over the set of candidate constraints or constraint theories. We will consider both search-based and syntax-guided synthesis approaches for learning hard constraints in Sect. 3, as well as optimization-based approaches in Sects. 4 and 5.
(c) Whether learning occurs from a pre-existing data set (passive learning) or by interactively asking questions to an oracle (interactive learning). Section 5 is dedicated to this last setting.

In the next Section, we proceed by introducing a simple, general formalism for expressing hard and soft constraint theories.

2 Constraint Theories

There are a number of successful formalisms for modeling and solving constraint programming problems [42]. In this overview we will restrict ourselves to a very general but minimal notation, to avoid as much overhead as possible.

Let us start by establishing some basic notation. In the following, variables will be written in upper-case X, constants in lower-case x, and value assignments $X = x$. For simplicity, all variables will take values in the same domain $\mathcal{X} \subset \mathbb{Z}$ or $\mathcal{X} = \mathbb{R}$, unless otherwise specified. Vectors of variables will be written in upper-case bold $\boldsymbol{X} = (X_1, \ldots, X_n)$ and total value assignments $X_1 = x_1 \wedge \ldots \wedge X_n = x_n$ simply as $\boldsymbol{X} = \boldsymbol{x}$. Total assignments are also called configurations. The indicator function $\mathbb{1}\{\varphi\}$ evaluates to 1 if condition φ holds and to 0 otherwise. Throughout the paper, we will use the terms "model", "constraint theory", and "constraint satisfaction/optimization problem" interchangeably.

In our simple framework, a constraint theory (or constraint network [9]) is defined by a set of variables $\boldsymbol{X}$ and by a set of constraints $c_1, \ldots, c_m$. A constraint with n arguments $c_j(X_1, \ldots, X_n)$ distinguishes between feasible and infeasible configurations. For instance, the constraint $c_j(X_1, X_2) = X_1 \vee X_2$, where X_1 and X_2 are Boolean, specifies that the configurations $(\texttt{true}, \texttt{true})$, $(\texttt{true}, \texttt{false})$, and $(\texttt{false}, \texttt{true})$ are feasible, while $(\texttt{false}, \texttt{false})$ is not. A constraint with n arguments can be more formally defined as an n-ary relation [9], but we will leave such technicalities aside. Further, constraints over continuous variables are also possible, e.g., consider the linear constraint $2X_1 - 3X_2 \leq 15$ or the mixed non-linear constraints $\neg\text{Slim} \implies (H < 110) \vee (\pi \cdot R^2 \geq 1000)$, where $X_1, X_2, H, R \in \mathbb{R}$ and Slim is Boolean. The set of constraints $c_1, \ldots, c_m$ is usually implicitly conjoined, but keep in mind that this does not hold for soft constraint theories; see Sects. 4 for some counter-examples.

In this overview, we will consider different kinds of constraint theories over both discrete and continuous variables, including:

– *Propositional logic theories* (or formulas) consist of Boolean variables, $\mathcal{X} = \{\texttt{true}, \texttt{false}\}$, combined with the usual logical connectives: conjunction $\wedge$, disjunction $\vee$, and negation $\neg$. Propositional formulas are most often encountered in conjunctive or disjunctive normal form. An example may be:

$$(\text{Saturday} \vee \text{Sunday}) \wedge \text{Sunny} \wedge \neg\text{Bored} \wedge \neg\text{Sick}$$

which is a possible definition for the concept of “fun weekend”. The main inference task is satisfiability (SAT) [21], where the goal is to find a total value assignment $\boldsymbol{X} = \boldsymbol{x}$ that renders the theory true. Extensions include MAX-SAT and weighted MAX-SAT, which will be discussed later.

– *Constraint networks* [9] leverage pure propositional logic to general discrete variables and arbitrary constraints, for instance inequality relations $\leq$, $=$, $\neq$, ... and global constraints like all-different. Inference is once again a form of satisfaction.

– *Satisfiability modulo theories* (SMT for short) extend propositional logic with one or more decidable theories $\mathcal{T}$ [5]. For instance, satisfiability modulo *linear real arithmetic* (SMT($\mathcal{LRA}$)) combines logic with linear arithmetic over the reals, and therefore introduces continuous variables and arbitrary linear constraints over them. An example SMT($\mathcal{LRA}$) formula for describing a happy outdoor weekend is:

$$(\text{Saturday} \vee \text{Sunday}) \wedge (\text{Rain} + \text{SoilHumidity} \leq 2)$$

Other decidable theories include linear *integer* arithmetic, bit-vectors, and uninterpreted functions, but we will stick to $\mathcal{LRA}$ for simplicity. In this case inference is also a form of satisfaction.

– *Linear programs* (LP) include both a linear objective function of real variables and a set of implicitly conjoined linear constraints [52]. An LP in canonical form is written as:

$$\begin{aligned} &\max_{\boldsymbol{x}} \ \boldsymbol{f}^\top \boldsymbol{x} \\ &\text{s.t. } \boldsymbol{a}_j^\top \boldsymbol{x} \leq b_j \qquad \forall j = 1, \ldots, m \end{aligned}$$

Here $\boldsymbol{f}$ is a constant vector that defines the (gradient of) the objective function while $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_m$ and $b_1, \ldots, b_m$ specify m linear constraints. Mixed-integer linear programs (MILP) have the same form, but allow both continuous and integer variables.
Notice that, for both LPs and MILPs, inference is a form of optimization rather than satisfaction, that is, the model specifies not only a feasible space (like in standard satisfaction) but also a score over alternative feasible configurations.

Of course there are many other kinds of constraint theories, e.g., in database systems and spreadsheet software. In this case, variables can be strings or other objects. However, we will not consider these further, and refer the interested reader to [17,26] instead.

3 Learning Hard Constraints

Warmup: Learning k-CNF Theories

Let us start from the simplest constraint learning problem: learning a k-CNF formula. Such formulas are the conjunction of clauses (disjunctions) with at most k literals each, where a literal is either a variable or its negation. For instance, happy weekends are captured by the 2-CNF formula (Saturday $\vee$ Sunday) $\wedge$ $\neg$Rainy $\wedge$ $\neg$ImminentDeadline.

Now, let there be a hidden k-CNF theory φ^*. Given a set of example configurations labeled based on whether they are feasible with respective to φ^* (positive) or not (negative), we want to recover φ^* from the data only. Valiant's algorithm is a classic strategy to achieve this goal. The idea is simple. First, build the set of all candidate clauses of length at most k over the variables $\boldsymbol{X}$. Then, taking each positive example in turn, remove from the set of candidates all of the clauses that are inconsistent with the example. This makes sure that, upon scanning over all positive examples, the set of candidates only contains clauses consistent with the data set. Upon termination, the learned formula is retrieved by taking the conjunction of all surviving clauses.

Despite its simplicity, Valiant's algorithm is PAC (probabilistically approximately correct) algorithm, meaning that the probability over all random data sets that the algorithm works is arbitrarily high so long as there are enough examples [54]. However, in order to work, Valiant's algorithm requires two key assumptions[2]: (a) the data set must be noiseless, i.e., there must be no measurement errors on the variables and no corruption on the labels, and (b) the hidden concept φ^* must be a k-CNF formula, which is not always known in advance. This is the so-called *realizable* setting. If these assumptions are not satisfied, then the algorithm gives unreliable results.

Encoding and Searching the Space of Candidates

Let us focus on discrete variables only for the time being. During learning, it is convenient to sort the space of candidate theories according to the *generality relation* $\succeq_g$. A constraint c is said to be more general than a constraint c', written $c \succeq_g c'$, if and only if the feasible set of c contains the feasible set of c', that is:

$$c \succeq_g c' \quad \iff \quad \forall \boldsymbol{x} \,.\, (\boldsymbol{x} \models c') \implies (\boldsymbol{x} \models c)$$

If c is more general than c', then c' is more specific than c, and vice versa. The generality relation induces a lattice over both constraints and constraint theories, which is perhaps the most common way to structure the space of candidates.

Once the set of candidates is given, the question becomes how to efficiently find a candidate theory compatible with the observed examples. In this sense,

[2] There are other technical assumptions over the distribution of the examples, which will be ignored for simplicity.

Valiant's algorithm can be viewed as a generate-and-test algorithm: it enumerates all candidates and then discards all of the ones incompatible with the data. In doing so, it keeps track of the *most specific* candidate theory. Beyond Valiant's algorithm, there exist notable examples of generate-and-test learners, led by the impressive ModelSeeker [7] approach to acquiring global constraints. But let us briefly consider alternative search strategies too.

There are three classic approaches to searching the space of candidates, all based on the above lattice structure:

- *General-to-specific* (aka top-down) approaches start from the most general concept (namely, `true`) and gradually specialize it according to the examples by introducing extra constraints that exclude the observed negatives from the feasible space of the learned theory.
- *Specific-to-general* (aka bottom-up) approaches, unsurprisingly, do the converse: they start from a most specific theory or set of theories and incrementally generalize them. It is often the case that the most specific theory is simply the disjunction of all positive examples—which, unless the data is inconsistent, excludes all negative examples. Generalization then boils down to removing constraints or removing conditions from constraints so to enlarge the feasible space of the candidate theory, while keeping all negatives outside of it.
- *Version space* approaches keep track of the whole set of candidates at once. The version space (VS) is indeed defined as the set of all theories that are consistent with respect to the dataset [33], and it is the sub-lattice (induced by the generality relation $\succeq_g$) between a set of most-general (top) and most-specific (bottom) candidates.
  In a discrete setting, VS learners leverage incremental bi-directional search, whereby an initial estimate of the VS is incrementally refined iterating over all examples, and for each example checking whether the most general theory wrongly covers it (if it is negative) or whether the most specific candidate wrongly excludes it (if it is positive). In either case the VS is updated.
  We note in passing that version spaces are not restricted to discrete variables, and that they are used in recent algorithms for both active learning [20] and preference elicitation [12].

In more general terms, learning of hard constraints can be viewed simply as a search problem, and therefore any search algorithm can in principle be used, including stochastic local search, genetic algorithms, *etc.*

It is worth remarking that, in most cases of interest, the set of candidates is exponential in the number of variables. For instance, for k-CNF, given v variables one can build $2v$ literals, and thus $(2v)^k$ potential clauses out of them. Learning approaches use smart strategies to avoid enumerating such a humongous set. Perhaps the simplest solution is to leverage the generality relation, by automatically excluding all constraints that are more general than an already excluded one. Alternative approaches include restricting the space of hypotheses by introducing background knowledge, e.g., by restricting the value of k or the initial set of candidates. Providing constraint templates to be filled in, as in sketching [48], is also an option. A sensible alternative is to compactly represent the space of candidates using a satisfaction or optimization problem. This strategy is at the core of syntax-guided synthesis (SyGuS) [1,2], a general framework for designing programs from specifications and examples. Two notable examples of SyGuS are the celebrated constraint learning Conacq [8], which encodes the version space using a propositional formula, and Incal [27], an approach for learning SMT($\mathcal{LRA}$) theories from examples that extends SyGuS to continuous variables.

4 Learning Soft Constraints

Soft constraints are a powerful tool for dealing with conflicting requirements, uncertain inputs, and imperfect specifications. Intuitively, soft constraints introduce two new rules: first, it is not mandatory to satisfy soft constraints, and second, some soft constraints are more important than others [10,42]. Therefore, if two soft constraints are incompatible, the most preferable one should be satisfied.

Soft Constraints with Linear Preferences

In their most general form, preferences over soft constraints can be encoded as a binary relation. Although very flexible, in the worst case encoding a relation over m soft constraints requires m^2 parameters. This is very cumbersome both when manually designing the constraint theory and when learning it from examples.

For this reason, we will focus on a more agile alternative, where the absolute importance of a constraint is determined by an objective function associated to it. The overall quality of a configuration is then given by the sum of the importances of the soft constraints that it satisfies. More formally, a theory in this form includes:

- m soft constraints $s_1, \ldots, s_m$, each associated to an objective function $w_j : \mathcal{X} \to \mathbb{R}$, $j = 1, \ldots, m$, and
- k hard constraints $h_1, \ldots, h_k$ that must be satisfied.

Finding the most preferable (aka optimal or highest scoring) configuration $\boldsymbol{x}^*$ is accomplished by maximizing the total weight $f_{\boldsymbol{w}}(\boldsymbol{x})$ of the soft constraints it satisfies[3], as follows:

$$\boldsymbol{x}^* = \operatorname*{argmax}_{\boldsymbol{x}} f_{\boldsymbol{w}}(\boldsymbol{x}) := \sum_{j=1}^{m} w_j(\boldsymbol{x}) \mathbb{1}\{\boldsymbol{x} \models s_j\} \tag{1}$$

$$\text{s.t. } \boldsymbol{x} \models h_j \qquad \forall j = 1, \ldots, k \tag{2}$$

[3] Notice that the optimal configuration may not be unique, and that all optima have the same score.

The computational complexity of this *inference* problem depends on the type of constraints and objective functions appearing above, but in most cases of interest (like MAX-SAT below) it is NP-hard or beyond.

Although more general alternatives exist, such as semiring-based constraints [10], we will stick to our simple framework because: (1) it captures many prominent settings, from weighted maximum satisfiability (weighted MAX-SAT) up to optimization modulo theories [44], and (2) soft constraint theories in the above format can be easily learned with high-quality machine learning algorithms, as discussed next.

Learning Weighted MAX-SAT from Annotated Configurations

In weighted MAX-SAT, the soft constraints $s_1, \ldots, s_m$ are arbitrary logic formulas and the per-constraint objective functions are constants $w_j(\boldsymbol{x}) \equiv w_j \in \mathbb{R}$. In the simplest case, no hard constraints are present. Inference boils down to finding a configuration $\boldsymbol{x}$ that maximizes the total weight of the satisfied formulas:

$$\max_{\boldsymbol{x}} \ f_{\boldsymbol{x}}(\boldsymbol{x}) := \sum_{j=1}^{m} w_j \mathbb{1}\{\boldsymbol{x} \models s_j\} \tag{3}$$

This problem is notoriously NP-complete, but in can be solved efficiently in many practical cases [21].

Let us now consider perhaps the simplest possible learning scenario. We assume that there is a latent, unknown weighted MAX-SAT problem with parameters $\boldsymbol{w}^* \in \mathbb{R}^m$. We cannot observe this latent model, but we do know the dictionary of soft constraints $s_1, \ldots, s_m$. Further, we are given a data set of example configurations $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ annotated with their own scores according to the unknown theory, i.e., $y_i = \sum_j w_j^* \mathbb{1}\{\boldsymbol{x}_i \models s_j\}$ for all $i = 1, \ldots, n$. The configurations $\boldsymbol{x}_i$ are assumed to be sampled at random according to some underlying distribution, and no implicit guarantee is given as for their quality.

The goal of learning is to induce a model $\boldsymbol{w}$ that behaves similarly to the latent one $\boldsymbol{w}^*$. For the time being, we will consider a model good so long as the estimated parameter vector $\boldsymbol{w}$ scores the examples similarly to the hidden model. Since we have access to the value of the true objective function y_i for all example configurations, finding an appropriate parameter vector can be cast as a regression problem, namely:

$$\hat{\boldsymbol{w}} = \operatorname{argmin}_{\boldsymbol{w} \in \mathbb{R}^m} \ \textstyle\sum_{k=1}^{n} (y_i - f_{\boldsymbol{w}}(\boldsymbol{x}_i))^2 \tag{4}$$

Now, notice that the function $f_{\boldsymbol{w}}(\boldsymbol{x})$ is linear with respect to the basis defined by the indicator functions $\{\mathbb{1}\{\boldsymbol{x} \models s_j\} : j = 1, \ldots, m\}$. This means that Eq. 4 can be cast as linear regression and solved using standard regression techniques.

A very nice property of this setup is that linear regression works well even if the supervision y_i is moderately corrupted, and it provides a bridge to robust regression techniques for different kinds of noise.

One should keep in mind, however, that supervision on the per-instance scores y_i may not be readily available. This happens for instance when eliciting preferences from decision makers, who may be unable to state a numerical score. For this reason, we consider two alternative forms of supervision, namely pairwise rankings and input-output pairs, and show how to learn soft constraint theories from them.

Learning from Rankings

Let us start from pairwise rankings. In this case, we are given n pairs of configurations $\{(\boldsymbol{x}_i, \boldsymbol{x}'_i) : i = 1, \ldots, n\}$, where each pair is implicitly ranked according to the preference relation $\boldsymbol{x}_i \succeq \boldsymbol{x}'_i \Leftrightarrow f_{\boldsymbol{w}^*}(\boldsymbol{x}_i) \geq f_{\boldsymbol{w}^*}(\boldsymbol{x}'_i)$. The goal of learning is then to find a parameter vector $\boldsymbol{w}$ that ranks all of the example pairs correctly, or more formally:

$$\text{find } \boldsymbol{w} \tag{5}$$

$$\text{s.t. } f_{\boldsymbol{w}}(\boldsymbol{x}_i) - f_{\boldsymbol{w}}(\boldsymbol{x}'_i) \geq 0 \qquad \forall i = 1, \ldots, n \tag{6}$$

The above constraint can be shown to be linear (in the indicators) by rewriting it as $\sum_{j=1}^m w_j(\mathbb{1}\{\boldsymbol{x}_i \models s_j\} - \mathbb{1}\{\boldsymbol{x}'_i \models s_j\}) \geq 0$. Notice that, even though the model is acquired from ranking data, we use it to compute high-scoring configurations, as per Eq. 3.

One issue with the above formulation is that simply conforming to the supervision does not guarantee that the learned vector $\boldsymbol{w}$ generalizes well to unseen pairs. This means that the learned function $f_{\boldsymbol{w}}$ may fail to rank the true optima above all other configurations. In order to address this, following principles from statistical learning theory [43,55], it is customary to look for a vector $\boldsymbol{w}$ that correctly ranks all pairs *by the largest possible margin*, that is:

$$\max_{\boldsymbol{w}, \mu \geq 0} \ \mu \tag{7}$$

$$\text{s.t. } \mu \leq f_{\boldsymbol{w}}(\boldsymbol{x}_i) - f_{\boldsymbol{w}}(\boldsymbol{x}'_i) \qquad \forall i = 1, \ldots, n \tag{8}$$

where $\mu \in \mathbb{R}$ measures the margin. It turns out that maximizing μ is geometrically equivalent to minimizing the (squared) Euclidean norm of $\boldsymbol{w}$ ([43], Chap. 1), and so the above can be rewritten as the following quadratic convex optimization problem:

$$\min_{\boldsymbol{w}} \ \frac{1}{2}\|\boldsymbol{w}\|_2^2 \tag{9}$$

$$\text{s.t. } f_{\boldsymbol{w}}(\boldsymbol{x}_i) - f_{\boldsymbol{w}}(\boldsymbol{x}'_i) \geq 0 \qquad \forall i = 1, \ldots, n \tag{10}$$

Finally, if the observations $\boldsymbol{x}_i$ are noisy or their rankings are inconsistent, as it the case when the examples define cycles like $\boldsymbol{x}_1 \succeq \boldsymbol{x}_2 \succeq \ldots \succeq \boldsymbol{x}_1$, then it may be impossible to find a non-zero vector $\boldsymbol{w}$ that simultaneously satisfies Eq. 10 for all examples. A common solution is to introduce slack variables $\xi_i \in \mathbb{R}$ that measure the "degree of violation" for every example $k = 1, \ldots, n$:

$$\min_{\boldsymbol{w}, \boldsymbol{\xi}} \frac{1}{2}\|\boldsymbol{w}\|_2^2 + \frac{\lambda}{2}\sum_{k=1}^{s} \xi_i \tag{11}$$

$$\text{s.t. } f_{\boldsymbol{w}}(\boldsymbol{x}_i) - f_{\boldsymbol{w}}(\boldsymbol{x}_i') \geq \xi_i \qquad \forall i = 1, \ldots, n \tag{12}$$

This is the well-known formulation of ranking support vector machine (SVM), a now classical machine learning algorithm for learning to rank, originally conceived for ranking results in search engines [23]. The constant $\lambda \geq 0$ controls the trade-off between generalization (first term) and error on the training set (second term), and it is assumed to be given.

Although earlier works focused on solving the above optimization problem (OP) in the dual (cf. [40]), current state-of-the-art approaches rely on gradient-based optimization in the primal [46]. Practitioners need not worry about these details, since efficient (ranking) SVM solvers are included by default in most machine learning libraries, like scikit-learn [37] and Weka [19].

Learning from Input-Output Pairs

Input-output pairs are another popular form of supervision. Let $\boldsymbol{U}$ and $\boldsymbol{V}$ partition the set of variables (i.e., $\boldsymbol{X} = \boldsymbol{U} \cup \boldsymbol{V}$, $\boldsymbol{U} \cap \boldsymbol{V} = \varnothing$). The intuition is that the variables $\boldsymbol{U}$ act as inputs and thus are always known and fixed, while $\boldsymbol{V}$ are outputs and can be optimized over.

The data set, in this case, consists of n input-output pairs $\{(\boldsymbol{u}_i, \boldsymbol{v}_i) : i = 1, \ldots, n\}$, where for any partial assignment $\boldsymbol{u}_i$, the output $\boldsymbol{v}_i$ is chosen optimally w.r.t. the latent parameters $\boldsymbol{w}^*$, that is:

$$\boldsymbol{v}_i = \operatorname*{argmax}_{\boldsymbol{v}} \; f_{\boldsymbol{w}^*}(\boldsymbol{u}_i \circ \boldsymbol{v}_i) \tag{13}$$

Here $\circ$ indicates vector concatenation. This kind of supervision is common in machine learning tasks that require to learn a map from structured inputs to structured outputs, like text parsing (where a sentence $\boldsymbol{u}$ is mapped to a parse tree $\boldsymbol{v}$) or image segmentation (an image $\boldsymbol{u}$ is mapped to a set of labeled segments $\boldsymbol{v}$), but it is also used in constraint learning, where the distinction between input and output variables is less well-defined [51].

In order to learn from input-output pairs, we adapt the training procedure of structured-output support vector machines (SSVM) [53], see [24] for a gentler introduction. Given an example $(\boldsymbol{u}_i, \boldsymbol{v}_i)$ and an arbitrary vector $\boldsymbol{w}$, let $\boldsymbol{v}'$ be the output of Eq. 4 when using $\boldsymbol{u}_i$ as input and $\boldsymbol{w}$ as parameters. Also, let $\Delta(\boldsymbol{v}_i, \boldsymbol{v}')$ be a distortion function[4] that measures the difference between the correct output $\boldsymbol{v}_i$ and the predicted one $\boldsymbol{v}'$.

[4] For technical reasons, the distortion is often assumed to lie in the range $[0, 1]$, see [32].

The intuition behind structured-output SVMs is that the vector $\boldsymbol{w}$ should be chosen so that, for any example, the predicted output has low distortion. This can be formalized as follows[5]:

$$\max_{\boldsymbol{w},\boldsymbol{\xi}} \frac{1}{2}\|\boldsymbol{w}\|_2^2 + \frac{\lambda}{2}\sum_{k=1}^{s} \xi_i \tag{14}$$

$$\text{s.t. } f_{\boldsymbol{w}}(\boldsymbol{u}_i \circ \boldsymbol{v}_i) - f_{\boldsymbol{w}}(\boldsymbol{u}_i \circ \boldsymbol{v}'') \geq \Delta(\boldsymbol{v}_i, \boldsymbol{v}'') - \xi_i \quad \forall \boldsymbol{v}'' \neq \boldsymbol{v}_i \, \forall i = 1, \ldots, n \tag{15}$$

The objective function is the same as for ranking data. The constraint, on the other hand, is much more complex: it requires the correct output $\boldsymbol{v}_i$ to be scored higher than any alternative output $\boldsymbol{v}'' \neq \boldsymbol{v}_i$ by a margin proportional to the distortion. This takes care of enforcing an appropriate margin between $\boldsymbol{v}_i$ and the predicted output $\boldsymbol{v}'$ too. The per-example slacks ξ_i allow for scoring mistakes in noisy data sets.

The similarity to learning to rank is striking, but solving this optimization problem is trickier, because the number of alternative outputs $\boldsymbol{v}'$ can be very (exponentially) large. This means that Eq. 15 has to be enforced over an enormous amount of configurations. In order to solve this OP, the most straightforward approach is to employ cutting planes, which we will not discuss. We refer the interested reader to [24] instead.

Learning more General Constraint Theories

A striking property of the OPs described in the previous sections is that they are relatively agnostic to the particular choice of constraints and per-constraint objective functions. Indeed, regression-based learning has been used to learn arbitrary weighted CSPs [41], and SSVM-based learning for learning Optimization Modulo Theories [51]. In this last case, the restriction that the per-constraint objective functions $w_j(\boldsymbol{x})$ are constant is lifted—although the learning algorithm remains essentially unchanged.

5 Interactive Learning

Applications where supervision is scarce and expensive are not well suited for offline learning, because there are often too few examples to learn a reasonably accurate model. In this case, a sensible thing to do is to acquire supervision directly from an oracle by asking informative questions. This allows the learning algorithm to optimize the performance/example ratio, and thus to acquire good models at a small cost.

Notice that the oracle may be a human subject, like when eliciting constraints (knowledge) from a domain expert [41] or preferences from an end-user [39], or

[5] There exist several variants of structured-output SVM, here we opt for the simpler one; see the references for more details.

a full-fledged scientific apparatus, as in automated scientific experiments [25]. In the first case, the goal is to extract a (soft or hard) constraint theory from a domain expert, who is otherwise unable to formalize and model her knowledge upfront. The second case captures applications like interactive recommender systems, where very little expertise (or patience!) can be expected of the human counterpart, and any request for supervision has to be designed so to be easy to understand and answer.

The main questions that arise when designing an interactive learning algorithm are of course what *kind* of questions should be asked to the oracle, and how to pick good questions. The answer to both questions is very application- and oracle-specific, as we will see in the following.

Interactive Learning of Hard Constraints

The most straightforward approach to learning hard constraints is to trivially select each candidate constraint h_j in turn, $j = 1, \ldots, m$, and ask the oracle whether it appears in the latent constraint theory. Unfortunately, this naive approach is not very useful, as it requires to consider all constraints, which can be exponentially many in the number of variables. This procedure is therefore unfeasible even for theories of modest size, especially if the oracle is a human being.

A more appropriate procedure is to use *membership constraints*. In this setting, the learner chooses an instance $\boldsymbol{x}$ and asks the user whether it satisfies the hidden theory or not. This setup stands at the core of query-based learning [4], a venerable and sound approach to learning. Alternative query types will be considered later on.

Now, what is the best way to choose the query configuration $\boldsymbol{x}$? Before proceeding, let us assume a realizable scenario, i.e., that (a) the set of hypotheses includes the latent concept, and (b) that the oracle always answers correctly. This is the case, for instance, if the learning problem is well engineered and the oracle is a human domain expert, e.g., an employee who has a vested interest in answering the questions asked by the algorithm. Under these (rather strong) assumptions, one can resort to halving approaches, which roughly work as follows.

Learning is interactive. At all iterations, the learner keeps track of the set of candidate theories that are consistent with respect to all answers observed so far—i.e., the version space. The intuition is that, in each iteration, the algorithm chooses a configuration $\boldsymbol{x}$ so that, regardless of the answer to the membership query, the version space is reduced as much as possible. In the best possible scenario, the version space halves at each iteration, and therefore ideally the number of queries necessary to find the correct concept is approximately $\log_2 |\mathcal{H}|$, where $\mathcal{H}$ is the set of candidate hypotheses.

Unfortunately, this theoretically appealing approach has several flaws. First, keeping track of the version space can be quite complicated. The best approaches to date make use of rather convoluted schemas [8] or only store the most general and most specific candidate theories in the version space [9]. Second, it turns out that in many interesting cases, choosing the (approximately) optimal instance $\boldsymbol{x}$ is computationally intractable [31]. A simple approximation to this schema is to choose an instance $\boldsymbol{x}$ that reduces the version space by *some* amount. This is accomplished in [9] by choosing an instance whose feasibility the most general and the most specific theories in the current version space disagree on. Therefore, if it turns out that $\boldsymbol{x}$ is feasible, the most specific hypothesis is wrong and it must be generalized. On the other hand, if $\boldsymbol{x}$ is actually unfeasbile, the converse is true and the most general hypothesis must be specialized. Thus, this strategy guarantees that the version space reduces at each iteration, and that it eventually contains only concepts that match all examples.

Interactive Learning of Soft Constraints

Consider the following toy application: A user wishes to buy a custom PC. The PC is assembled from individual components: CPU, HDD, RAM, etc. Valid PC configurations must satisfy constraints, e.g. CPUs only work with compatible motherboards [50]. In this setting, one is tasked with constructing a PC configuration $\boldsymbol{x}$ that is both palatable to the customer and compatible with any hard constraints, e.g., that the CPU and the motherboard should be compatible.

As above, we cast this kind of problems as learning a weighted CSP that captures the preferences of the customer, and that can be used to generate high-scoring

In the following, we will consider three queries of three common kinds:

- *Scoring queries.* In this case, the oracle is asked to provide the true numerical score of configuration $\boldsymbol{x}$ chosen by the learner.
  Queries of this type make sense when interacting with very precise oracles, such as measurement devices in automated scientific experiments [25], but not as much when interacting with human oracles. Indeed, it is very difficult—even for domain experts—to provide precise or approximate numerical scores. This is why most scoring systems use discrete ratings, such as star ratings. Regardless, depending on the application, other query types may be easier to answer.
- *Ranking queries.* In this case, the query consists of two unlabeled configurations $\boldsymbol{x}$ and $\boldsymbol{x}'$, and the oracle is tasked with indicating the most preferable of the two, i.e., whether $f_{\boldsymbol{w}^*}(\boldsymbol{x}) \geq f_{\boldsymbol{w}^*}(\boldsymbol{x}')$ holds.
  These queries have found ample application in interactive preference elicitation and recommendation tools [39].
- *Improvement queries.* In this case, the algorithm chooses a single configuration $\boldsymbol{x}$ and asks the oracle to provide an improved version $\bar{\boldsymbol{x}}$. Notice that the improvement is allowed to be small or partial [47].
  It is easy to see that the pair $(\bar{\boldsymbol{x}}, \boldsymbol{x})$ implicitly defines a pairwise ranking of the form $\bar{\boldsymbol{x}} \succeq \boldsymbol{x}$, and so the supervision is the same as for ranking queries. It is important to notice that, however, the interaction itself is different. Indeed, improvement queries only require the human oracle to observe and analyze a single configuration (rather than two), and synergize very well with direct manipulation interfaces like those used in computer-assisted design.

Notice that scoring queries lead to collecting a large data set of scoring information, and thus learning can be cast in terms of linear regression. In the other two cases, the collected data set contains pairwise ranking information, and therefore learning boils down to learning to rank. Thus the techniques discussed in the previous section can be immediately applied to the interactive case: it is just a matter of solving a regression, ranking, or structured-output learning every time a new answer is achieved. Although not entirely principled, this approach tends to work well in practice.

The question is, again, how to pick the right query. Ideally, the most informative question should be chosen. Unfortunately, evaluating the true informativeness of a query based on the information available during learning is tricky. A very simple solution is to choose a configuration $\boldsymbol{x}$ (or a pair of configurations) that is maximally *uncertain* according to the model. More specifically, an instance is uncertain if the entropy of the response variable (e.g. of the score, for scoring queries) is large. Measures based on the margin are also common [45]. The major down-side of uncertainty sampling is that the uncertainty provided by the model may be misleading, e.g., for instance if the learned theory is "overconfident". It is therefore customary to combine uncertainty sampling with other strategies that lessen its reliance on the model's estimates [45].

6 Further Reading

Learning and Optimization. The interplay between machine learning and satisfaction/optimization is not limited to constraint learning. In most of machine learning (excluding a large chunk of its Bayesian side), learning is framed as the task of optimizing some regularized loss function with respect to the data set [55]—hence the centrality of optimization in ML. Nowadays, the most popular solution approach are gradient descent techniques [29], but one should not forget that there are a plethora of valid alternatives, cf. [11,49]. Some learning frameworks go one step further, and make use of declarative constraint programming to model and solve learning tasks like program synthesis and pattern mining [2,18]. Another link between ML and optimization occurs in probabilistic graphical models, structured-output prediction, and constraint learning, where computing a prediction for an unseen instance is itself an optimization problem. There is also abundant literature on speed-up learning, i.e., on leveraging machine learning techniques for improving the run-time of satisfaction and optimization solvers; see for instance [22] for a method to accelerate branch & bound in mixed-integer linear programming solvers. It is easy to spot a recursion here: machine learning could in principle be used to accelerate an optimization step which is itself part of a learning problem. While true, diminishing returns make it difficult to exploit this loop.

Learning Hard Constraints. The task of acquiring hard constraints is surprisingly close to concept learning [54] and binary classification: in all of these, the learner has to acquire a hidden concept from examples. The main difference is whether the model in question is a constraint theory, and whether it is used in a purely predictive manner or also for inspection, interpretation, debugging, *etc.* A major advantage of constraints is that they can be verified by domain experts either manually or with appropriate tools—and modified, if necessary. Verification of models learned from data is often a required to guarantee the proper functioning of the model [3], and indeed verification techniques are being actively researched for non constraint-based models like neural networks [13].

From a search perspective, learning hard constraints is also closely related to feature selection, pattern mining, and structure learning of probabilistic graphical models [28]—e.g. Bayesian networks and Markov networks. All these tasks revolve around efficiently encoding and searching a very large set of candidates, and often employ similar techniques, e.g., version spaces, variants of grafting [38], and syntax-guided synthesis [6], albeit often under different names. Inductive logic programming, where the task is to learn first-order theories [34], can be handled analogously, but it involves even larger search spaces.

Learning soft constraints. As for soft constraints, we focused on learning techniques rooted in statistical learning theory and empirical risk minimization [55] and max-margin methods like support vector machines [43]. Learning from input-output examples stands at the core of structured-output prediction [24] and learning to search [14]. One link that is seldom made is to inverse (combinatorial) optimization, where the goal is to adjust a pre-existing (combinatorial) optimization model such that it adheres to a set of known optimal solutions. Unsurprisingly, this field has slowly been drifting closer to the structured-output setting, and these two problems have recently been tackled using similar strategies, see for instance [47] and [16].

Acknowledgments. The author is grateful to Luc De Raedt and Andrea Passerini for many insightful discussions. These lecture notes are partially based on material co-developed by LDR, AP and the author. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. [694980] SYNTH: Synthesising Inductive Data Models).

References

1. Alur, R., et al.: Syntax-guided synthesis. In: 2013 Formal Methods in Computer-Aided Design, pp. 1–8. IEEE (2013)
2. Alur, R., Singh, R., Fisman, D., Solar-Lezama, A.: Search-based program synthesis. Commun. ACM **61**(12), 84–93 (2018)
3. Andrews, R., Diederich, J., Tickle, A.B.: Survey and critique of techniques for extracting rules from trained artificial neural networks. Knowl.-Based Syst. **8**(6), 373–389 (1995)
4. Angluin, D.: Queries and concept learning. Mach. Learn. **2**(4), 319–342 (1988)
5. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. Handb. Satisf. **185**, 825–885 (2009)

6. Bartlett, M., Cussens, J.: Integer linear programming for the Bayesian network structure learning problem. Artif. Intell. **244**, 258–271 (2017)
7. Beldiceanu, N., Simonis, H.: A model seeker: extracting global constraint models from positive examples. In: Milano, M. (ed.) CP 2012. LNCS, pp. 141–157. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33558-7_13
8. Bessiere, C., Coletta, R., Koriche, F., O'Sullivan, B.: A SAT-based version space algorithm for acquiring constraint satisfaction problems. In: Gama, J., Camacho, R., Brazdil, P.B., Jorge, A.M., Torgo, L. (eds.) ECML 2005. LNCS (LNAI), vol. 3720, pp. 23–34. Springer, Heidelberg (2005). https://doi.org/10.1007/11564096_8
9. Bessiere, C., et al.: New approaches to constraint acquisition. In: Bessiere, C., De Raedt, L., Kotthoff, L., Nijssen, S., O'Sullivan, B., Pedreschi, D. (eds.) Data Mining and Constraint Programming. LNCS (LNAI), vol. 10101, pp. 51–76. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50137-6_3
10. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint logic programming: syntax and semantics. ACM Trans. Program. Lang. Syst. (TOPLAS) **23**(1), 1–29 (2001)
11. Bottou, L., Curtis, F.E., Nocedal, J.: Optimization methods for large-scale machine learning. Siam Rev. **60**(2), 223–311 (2018)
12. Boutilier, C., Regan, K., Viappiani, P.: Simultaneous elicitation of preference features and utility. In: Twenty-Fourth AAAI Conference on Artificial Intelligence (2010)
13. Bunel, R.R., Turkaslan, I., Torr, P., Kohli, P., Mudigonda, P.K.: A unified view of piecewise linear neural network verification. In: Advances in Neural Information Processing Systems, pp. 4790–4799 (2018)
14. Daumé III, H., Marcu, D.: Learning as search optimization: approximate large margin methods for structured prediction. In: Proceedings of the 22nd International Conference on Machine Learning, pp. 169–176. ACM (2005)
15. De Raedt, L., Passerini, A., Teso, S.: Learning constraints from examples. In: Thirty-Second AAAI Conference on Artificial Intelligence (2018)
16. Dong, C., Chen, Y., Zeng, B.: Generalized inverse optimization through online learning. In: Advances in Neural Information Processing Systems, pp. 86–95 (2018)
17. Gulwani, S., Hernandez-Orallo, J., Kitzelmann, E., Muggleton, S.H., Schmid, U., Zorn, B.: Inductive programming meets the real world. Commun. ACM **58**(11), 90–99 (2015)
18. Guns, T., Dries, A., Tack, G., Nijssen, S., De Raedt, L.: Miningzinc: a modeling language for constraint-based mining. In: Twenty-Third International Joint Conference on Artificial Intelligence (2013)
19. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: an update. ACM SIGKDD Explor. Newsl. **11**(1), 10–18 (2009)
20. Hanneke, S., et al.: Theory of disagreement-based active learning. Found. Trends® Mach. Learn. **7**(2–3), 131–309 (2014)
21. Hansen, P., Jaumard, B.: Algorithms for the maximum satisfiability problem. Computing **44**(4), 279–303 (1990)
22. He, H., Daume III, H., Eisner, J.M.: Learning to search in branch and bound algorithms. In: Advances in Neural Information Processing Systems, pp. 3293–3301 (2014)
23. Joachims, T.: Optimizing search engines using clickthrough data. In: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 133–142. ACM (2002)

24. Joachims, T., Hofmann, T., Yue, Y., Yu, C.N.: Predicting structured objects with support vector machines. Commun. ACM **52**(11), 97 (2009)
25. King, R.D., et al.: The automation of science. Science **324**(5923), 85–89 (2009)
26. Kolb, S., Paramonov, S., Guns, T., De Raedt, L.: Learning constraints in spreadsheets and tabular data. Mach. Learn. **106**, 1–28 (2017)
27. Kolb, S., Teso, S., Passerini, A., De Raedt, L.: Learning SMT (LRA) constraints using SMT solvers. In: IJCAI, pp. 2333–2340 (2018)
28. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. MIT Press, Cambridge (2009)
29. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436 (2015)
30. Lombardi, M., Milano, M.: Boosting combinatorial problem modeling with machine learning. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence, pp. 5472–5478. AAAI Press (2018)
31. Louche, U., Ralaivola, L.: From cutting planes algorithms to compression schemes and active learning. In: 2015 International Joint Conference on Neural Networks (IJCNN), pp. 1–8. IEEE (2015)
32. McAllester, D.: Generalization bounds and consistency. In: Predicting Structured Data, pp. 247–261 (2007)
33. Mitchell, T.M.: Generalization as search. Artif. Intell. **18**(2), 203–226 (1982)
34. Muggleton, S., De Raedt, L.: Inductive logic programming: theory and methods. J. Logic Program. **19/20**, 629–679 (1994)
35. O'Sullivan, B.: Automated modelling and solving in constraint programming. In: Twenty-Fourth AAAI Conference on Artificial Intelligence (2010)
36. Pawlak, T.P., Krawiec, K.: Automatic synthesis of constraints from examples using mixed integer linear programming. Eur. J. Oper. Res. **261**(3), 1141–1157 (2017)
37. Pedregosa, F., et al.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
38. Perkins, S., Lacker, K., Theiler, J.: Grafting: fast, incremental feature selection by gradient descent in function space. J. Mach. Learn. Res. **3**(03), 1333–1356 (2003)
39. Pigozzi, G., Tsoukias, A., Viappiani, P.: Preferences in artificial intelligence. Ann. Math. Artif. Intell. **77**(3–4), 361–401 (2016)
40. Platt, J.: Sequential minimal optimization: a fast algorithm for training support vector machines (1998)
41. Rossi, F., Sperduti, A.: Acquiring both constraint and solution preferences in interactive constraint systems. Constraints **9**(4), 311–332 (2004)
42. Rossi, F., Van Beek, P., Walsh, T.: Handbook of Constraint Programming. Elsevier, Amsterdam (2006)
43. Scholkopf, B., Smola, A.J.: Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond. MIT Press, Cambridge (2001)
44. Sebastiani, R., Tomasi, S.: Optimization modulo theories with linear rational costs. ACM Trans. Comput. Log. (TOCL) **16**(2), 12 (2015)
45. Settles, B.: Active learning. Synth. Lect. Artif. Intell. Mach. Learn. **6**(1), 1–114 (2012)
46. Shalev-Shwartz, S., Singer, Y., Srebro, N., Cotter, A.: Pegasos: primal estimated sub-gradient solver for svm. Math. Program. **127**(1), 3–30 (2011)
47. Shivaswamy, P., Joachims, T.: Coactive learning. J. Artif. Intell. Res. (JAIR) **53**, 1–40 (2015)
48. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. ACM Sigplan Not. **41**(11), 404–415 (2006)
49. Sra, S., Nowozin, S., Wright, S.J.: Optimization for Machine Learning. MIT Press, Cambridge (2012)

50. Teso, S., Dragone, P., Passerini, A.: Coactive critiquing: elicitation of preferences and features. In: AAAI (2017)
51. Teso, S., Sebastiani, R., Passerini, A.: Structured learning modulo theories. Artif. Intell. **244**, 166–187 (2017)
52. Todd, M.J.: The many facets of linear programming. Math. Program. **91**(3), 417–436 (2002)
53. Tsochantaridis, I., Hofmann, T., Joachims, T., Altun, Y.: Support vector machine learning for interdependent and structured output spaces. In: Proceedings of the Twenty-first International Conference on Machine Learning, p. 104. ACM (2004)
54. Valiant, L.: A theory of the learnable. Commun. ACM **27**, 1134–1142 (1984)
55. Vapnik, V.: An overview of statistical learning theory. IEEE Trans. Neural Netw. **10**(5), 988–999 (1999)