Annals of Mathematics and Artificial Intelligence
https://doi.org/10.1007/s10472-021-09736-4

Classifier-based constraint acquisition

S. D. Prestwich[1] · E. C. Freuder[2] · B. O'Sullivan[2] · D. Browne[2]

Accepted: 1 March 2021Published online: 17 April 2021

Abstract
Modeling a combinatorial problem is a hard and error-prone task requiring significant expertise. Constraint acquisition methods attempt to automate this process by learning constraints from examples of solutions and (usually) non-solutions. Active methods query an oracle while passive methods do not. We propose a known but not widely-used application of machine learning to constraint acquisition: training a classifier to discriminate between solutions and non-solutions, then deriving a constraint model from the trained classifier. We discuss a wide range of possible new acquisition methods with useful properties inherited from classifiers. We also show the potential of this approach using a Naive Bayes classifier, obtaining a new passive acquisition algorithm that is considerably faster than existing methods, scalable to large constraint sets, and robust under errors.

Keywords Constraint acquisition · Classifier · Bayesian · Boolean satisfiability

Mathematics Subject Classification 2010 68T99 · 68Q32 · 68R99

1 Introduction

A *constraint satisfaction problem* (CSP) has a set of problem variables, each with a domain of possible values, and a set or network of constraints imposed on subsets of the variables. A constraint is a relationship that must be satisfied by any solution, though it can be violated

✉ S. D. Prestwich
steven.prestwich@insight-centre.org

E. C. Freuder
eugene.freuder@insight-centre.org

B. O'Sullivan
barry.osullivan@insight-centre.org

D. Browne
david.browne@insight-centre.org

[1] Insight Centre for Data Analytics, School of Computer Science & Information Technology, University College Cork, Cork, Ireland

[2] School of Computer Science & Information Technology, University College Cork, Cork, Ireland

in non-solutions. These are the ingredients of a constraint model of a problem, but in this paper the term *constraint satisfaction* is used very broadly, and models might be formulated as constraint networks, SAT formulae or mathematical programs.

In constraint-based reasoning the effort to bridge the gap between the current state of the art and human-level Artificial Intelligence (AI) has been embodied in a long-standing challenge to address the "Holy Grail" of computer science: the user simply presents a problem to the computer in a manner natural for the user, and the computer proceeds to solve it [18]. The premise has been that Constraint Programming (CP) is well-suited to address this challenge, as once a problem is expressed as a constraint satisfaction (or optimization) problem there are general-purpose algorithms that can, in principle, proceed to solve it. Furthermore, as such problems are ubiquitous within AI and have many practical applications [17], progress in this direction will have broad implications.

The field of *Constraint Acquisition* (CA) [1, 2, 5–8, 25, 32, 42, 43, 47], also called *Constraint Learning* [36] and *Constraint Synthesis* [33], addresses the challenge of automating the expression of the problem as a CSP. In CA we are given instances of solutions and non-solutions or failures (positive and negative instances respectively) and the aim is to learn a constraint model that represents them. Beside the general goal of automated problem modelling, the model might be used as an explanation of the problem, to classify partial assignments, to show that a partial assignment cannot be placed in a class, to speed up the solution of future problems, or to find instances that optimise some objective. CA has been identified as an important topic in [32].

The CA problem is defined in [36] as follows. We are given: a space X of x *instances* (assignments to variables V); a space of possible constraints C; an unknown target constraint theory $T \subseteq C$; and a dataset of training instances E, in which *positive* instances satisfy T while *negative* instances do not. The task is to find a constraint theory $H \subseteq C$ such that all positive instances in E are satisfied, and none of the negative instances. A more detailed formal definition and theoretical results are given in [6]. *Active* methods are guided by interaction with a user or other oracle, while *passive* CA methods learn automatically. Several CA systems have been devised (see Section 3) based on machine learning, inductive logic programming and other methods, with a recent survey given in [36].

We propose a two-stage approach to CA that allows us to take advantage of machine learning research: training a classifier to learn the difference between solutions and non-solutions, then deriving a constraint model from the trained classifier. This has already been done for neural network classifiers and decision trees (see Section 3) but this work is not well known in the CA literature: it is mostly reported in other areas such as deep learning and data science. Moreover, there are many classifiers with useful properties that have not yet been used in this way. We conjecture that any classifier can, at least in principle, be used to derive a constraint model, and that some will be useful for new CA applications. We call this approach *classifier-based constraint acquisition* (ClassAcq).

One aim of this paper is to link the CA and data science literatures more closely together on this issue, for greater synergy between the two fields. Another aim is to present a specific application of ClassAcq using a Naive Bayes classifier, leading to a new CA method with desirable properties. The paper is organised as follows. Section 2 presents the new Bayesian method. Section 3 discusses related work. Section 4 tests the method on examples. Section 5 concludes the paper and discusses other classifiers that might be useful for CA. The paper is an extension of [11] which is mostly contained in Sections 3 and 5.1.

2 Constraint acquisition by Naive Bayes classifier

The CLASSACQ idea of learning constraints from a trained classifier is not new. It is known that decision trees can be transformed into constraint models [10, 29, 45], as can artificial neural networks [4, 27, 29]. Some neural networks can also be transformed to integer programs [15, 37, 40]. However, these methods are not generally used for CA, and other classifiers have been neglected in this context. In this section we derive a new CA method from a Naive Bayes classifier.

2.1 Naive Bayes classifiers

Given a vector $x = \langle x_1, \ldots, x_N \rangle$ of values x_i to be classified, Naive Bayes classifiers usually select a class using the *maximum a posteriori* rule to choose the most likely class:

$$\text{argmax}_k \left(\Pr(C_k) \prod_{i=1}^{N} \Pr(x_i | C_k) \right) \tag{1}$$

This rule selects the class k that is the mode of the posterior distribution. To train the classifier we must estimate the prior class probabilities $\Pr(C_k)$, and the conditional probabilities $\Pr(x_i|C_k)$ of observing x_i in class C_k. These are estimated simply by counting values in the dataset, so training these classifiers is fast and scalable.

An assumption is that the x_i are values of independent variables or features. Although this is often unrealistic, Naive Bayes classifiers often give surprisingly good results, are provably optimal for some cases [14], and are a standard tool for some applications including spam detection. They are also robust under noise and errors, because corrupted data can be neutralised by sufficient correct data.

The properties of speed, scalability and robustness make Naive Bayes an interesting candidate for the CLASSACQ approach.

2.2 A linear model using constraints as features

At first glance the Naive Bayes assumption of independence between variables seems to make them unsuitable for learning constraints between variables. However, to learn binary constraints (involving two variables) we can combine pairs of variables into single features, which is essentially how a Pairwise Naive Bayes classifier works [3]. More generally, we could consider variable tuples of arbitrary size to learn non-binary constraints. We use this constraints-as-features idea as follows.

Suppose the dataset is a set of instances of the form $x = \langle x_1, \ldots, x_N \rangle$, where each value x_j can in principle be taken from any domain, and each instance is in class C_+ (solutions) or C_- (non-solutions). Our method requires a set of *candidates*, called the *bias*, that may or may not be constraints of the model we are trying to learn. We define binary features c_j: for any instance, $c_j = 1$ iff constraint j is violated by that instance. This transforms the dataset into a set of binary vectors, each bit or feature corresponding to a candidate constraint. Hence the dataset can be represented by a 2-dimensional array c_{ij} of bits, with rows i corresponding to instances and columns j to candidates.

Because the features are binary we use the Bernoulli Naive Bayes classifier. Following the CLASSACQ approach we train the classifier on the transformed data, then derive a con-

straint model from the trained classifier. In our application the classes are $k \in \{+, -\}$, and an instance is classed as a solution if and only if:

$$\prod_j \frac{\Pr(c_j = 1|C_-)}{\Pr(c_j = 1|C_+)} < \frac{\Pr(C_+)}{\Pr(C_-)}$$

In general we do not know $\Pr(C_-)$ or $\Pr(C_+)$ because there is no guarantee that these probabilities are reflected in the training data. For example, given a tightly constrained problem we might generate training data with 1000 solutions and 1000 non-solutions to facilitate learning. Moreover, we rarely know how tightly-constrained an unknown constraint model is. We therefore assume an *uninformed prior* $\Pr(C_+) = \Pr(C_-)$ so that an instance is classed as a solution if and only if:

$$\prod_j \frac{\Pr(c_j = 1|C_-)}{\Pr(c_j = 1|C_+)} < 1 \quad \text{or} \quad \sum_j \ln\left(\frac{\Pr(c_j = 1|C_-)}{\Pr(c_j = 1|C_+)}\right) < 0$$

From this we can derive a linear constraint

$$\sum_j c_j \ln\left(\frac{\Pr(c_j = 1|C_-)}{\Pr(c_j = 1|C_+)}\right) < 0 \tag{2}$$

that mimics a Naive Bayes classifier given c_j values: given any previously unseen instance, we can compute the c_j then test the linear constraint; if it is satisfied then the instance is classified as a solution; if it is violated the instance is classified as a non-solution. However, the constraint can also be used to check whether a partial assignment to the c_j can be completed to obtain a solution, or to find an assignment that optimises some objective, by enumerating combinations of values for the unassigned c_j.

2.3 A simplifying approximation

We have used Naive Bayes to learn a linear model. However, a single linear constraint on binary variables is not the constraint model we desire. Instead we would like to learn which candidates j (expressed on the original problem variables) should be in the model. Fortunately, in practice the coefficients (in constraint (2)) of c_j for actual constraints are quite large positive values, while those for non-constraint candidates have positive or negative values close to 0. We therefore force $c_j = 0$ for candidates j with large coefficients, thus insisting that they are satisfied. We also adopt the simple approximation of ignoring all other candidates j because there is insufficient evidence that they are constraints.

We can now discard Naive Bayes and the c_j leaving a simple CA method: for each candidate j compute $K_j = \Pr(\text{viol}(j)|C_-)/\Pr(\text{viol}(j)|C_+)$ where $\text{viol}(j)$ denotes violation of candidate j. If K_j is greater than some threshold κ then learn candidate j as a constraint, otherwise ignore it. Discarding the c_j can save a great deal of memory, because we no longer need to store a number for every candidate and instance. Instead we need only store the original dataset and test violations on the fly. This becomes important in Section 4.4 in which the largest example has 1000 problem variables and over a billion candidates (features).

2.4 Probability estimation

To compute K_j we use a standard definition of conditional probability:

$$K_j = \frac{\Pr(\text{viol}(j)|C_-)}{\Pr(\text{viol}(j)|C_+)} = \frac{\Pr(\text{viol}(j), C_-)/\Pr(C_-)}{\Pr(\text{viol}(j), C_+)/\Pr(C_+)}$$

which under the uninformed prior reduces to $\Pr(\text{viol}(j), C_-)/\Pr(\text{viol}(j), C_+)$. The conditional probabilities can be estimated by counting occurrences in the training data: $\mathbf{n}(\text{viol}(j), C_-)/\mathbf{n}(\text{viol}(j), C_+)$ where $\mathbf{n}(\text{viol}(j), C)$ denotes the number of instances in class C that violate candidate $\text{viol}(j)$.

To avoid infinities we prevent zero probabilities via *additive smoothing* (also called *Lidstone* or *Laplace smoothing*) which modifies the ratios slightly: this technique is commonly used with Naive Bayes and in statistical language modelling [30]. Suppose we have n instances, m of which have some property. Instead of estimating the probability of this property as m/n we use $(m + \alpha)/(n + \alpha d)$ where the *pseudocount* $\alpha > 0$ is a smoothing parameter. The estimated probability will then be between m/n and $1/d$. In our application the denominators $n + \alpha d$ cancel out leaving:

$$K_j = \frac{\mathbf{n}(\text{viol}(j), C_-) + \alpha}{\mathbf{n}(\text{viol}(j), C_+) + \alpha}$$

2.5 Bayesian hypothesis testing interpretation

We now provide an alternative derivation of the method, by characterising it as Bayesian hypothesis testing (BHT). Recall that K_j is the ratio of (i) the probability $\Pr(\text{viol}(j)|C_-)$ that candidate j is violated by a non-solution, and (ii) the probability $\Pr(\text{viol}(j)|C_+)$ that candidate j is violated by a solution. This corresponds to a Bayes factor, which plays a similar role in BHT to a p-value in frequentist hypothesis testing (see for example [23]). In our application the violation of candidate j is the observed data, non-solutionhood of an instance is the null hypothesis $\mathcal{H}_0$, and solutionhood is the alternative hypothesis $\mathcal{H}_1$. So K_j measures the relative plausibility of hypotheses $\mathcal{H}_0$ and $\mathcal{H}_1$ on observing the violation of candidate j. If $\mathcal{H}_0$ is much more plausible, this fits the definition of a constraint: a relation that is true in any solution, so that its violation implies non-solutionhood.

An advantage of BHT over the frequentist approach is that it can find evidence in favour of the null hypothesis, as well as against it. However, the data might provide stronger evidence for one hypothesis than the other, leaving the latter inconclusive. Our application has the same asymmetry: the satisfaction of a single constraint is very weak evidence that an instance is a solution, but a single constraint violation is strong evidence of non-solutionhood. So we learn that candidate j is a constraint if K_j is large, otherwise the evidence is inconclusive so we learn nothing about candidate j.

The logarithm of a Bayes factor is sometimes called *weight of evidence*, and Bayes factors can be measured in *decibans* (following Turing [21]) by taking $10 \log_{10}(K_j)$ with accepted characterisations of ranges of values shown in Table 1 (there are also other versions of this table). From now on we shall measure K_j in decibans.

Table 1 Weight of evidence

K_j	Decibans	Weight of evidence
$< 10^0$	0	Negative
10^0–$10^{1/2}$	0–5	Barely worth mentioning
$10^{1/2}$–10^1	5–10	Substantial
10^1–$10^{3/2}$	10–15	Strong
$10^{3/2}$–10^2	15–20	Very strong
$> 10^2$	>20	Decisive

In summary, we compute K_j for each candidate j and accept j as a constraint if and only if K_j exceeds a deciban threshold κ. We call this method BAYESACQ. It has two parameters: α and κ. For our experiments we use $\alpha = 0.01$ and choose κ empirically (but see Section 4.6 for a discussion on choosing parameters).

2.6 Illustrative example

To illustrate BAYESACQ consider the following training data:

wealth	gender	height	solution?
rich	male	medium	yes
rich	male	tall	no
rich	female	medium	yes
comfortable	female	short	yes
poor	female	medium	no
comfortable	male	short	yes
comfortable	female	medium	yes

with variables wealth, gender and height, and domains:

variable	domain
wealth	poor, comfortable, rich
gender	male, female
height	short, medium, tall

As candidates we test all possible non-assignments (unary nogoods): wealth≠poor, wealth≠comfortable, wealth≠rich, gender≠male, gender≠female, height≠short, height≠medium and height≠tall: these are our bias. Estimating probabilities via additive smoothing we obtain:

constraint	K_j	characterisation
wealth≠poor	20	decisive
wealth≠comfortable	–25	negative
wealth≠rich	–3	negative
gender≠male	–3	negative
gender≠female	–5	negative
height≠short	–23	negative
height≠medium	–5	negative
height≠tall	20	decisive

For example the K_j value for candidate wealth≠poor is calculated by noting that this candidate is violated in one non-solution and no solution, so $K_j = (1 + 0.01)/(0 + 0.01) = 101$ or $10\log_{10}(101) \approx 20$ decibans. Violation by at least one non-solution, combined with a lack of violations by solutions, yields a *decisive* number of decibans even on this small dataset, because of the small α value. On the other hand, any candidate that is violated by solutions as well as non-solutions is unlikely to be learned as a constraint, because it yields a number of decibans that is *barely worth mentioning* or even

negative: for example wealth≠rich is violated by two solutions and one non-solution, so $K_j = (1 + 0.01)/(2 + 0.01) \approx 0.5$ or approximately -3 decibans. Hence there is a clear gap between two candidates (wealth≠poor and height≠tall) and the rest, and we learn these two as constraints. This is consistent with the training data: there are no poor or tall people in the positive class but all other values are represented.

Note that there are two K_j values that are much less than zero, corresponding to candidate constraints that are violated only in solutions. This is an artefact of the small dataset: if more data were added we would expect these candidates to be violated by at least one non-solution. This is illustrated in Section 4.6.

We can also test all literals (single variable assignments) to learn backbone values (assignments that must be true in all solutions). The above example has no backbone, but let us introduce a new attribute "age" with domain {young, middle, old}. Suppose that in all 5 solutions age=young, with middle occurring in 1 non-solution and old in the other. Then age=young is *decisive* while all others are *negative*. Again this is consistent with the data: all solutions are young while non-solutions have other ages, so it is reasonable to assume this literal is in the backbone.

2.7 Robustness under errors

Four types of error might occur in a training dataset: labeling a solution as a non-solution, labeling a non-solution as a solution, recording a violation as a satisfaction, and recording a satisfaction as a violation. A dataset might contain errors of each type. Current CA methods are not designed to be robust under even one such error.

We now show that BayesAcq is robust under errors in the following sense: any incorrect data can be overwhelmed by sufficient correct data, whatever parameters α and κ are used. (In Section 4.7 we show empirically that BayesAcq is robust in a stronger sense.) Recall that

$$K_j = \frac{\mathbf{n}(\text{viol}(j), C_-) + \alpha}{\mathbf{n}(\text{viol}(j), C_+) + \alpha}$$

and that candidate j is accepted as a constraint iff $K_j > \kappa$. Suppose the dataset is the union of some "good" data $C^g = C^g_+ \cup C^g_-$ containing no errors, and some "bad" data $C^b = C^b_+ \cup C^b_-$ containing an unknown number of errors. Then

$$K_j = \frac{\mathbf{n}(\text{viol}(j), C^g_- \cup C^b_-) + \alpha}{\mathbf{n}(\text{viol}(j), C^g_+ \cup C^b_+) + \alpha} = \frac{\mathbf{n}(\text{viol}(j), C^g_-) + \mathbf{n}(\text{viol}(j), C^g_-) + \alpha}{\mathbf{n}(\text{viol}(j), C^g_+) + \mathbf{n}(\text{viol}(j), C^b_+) + \alpha}$$

If we can obtain sufficient good data so that $|C^g_-| \gg |G^b_-|$ and $|C^g_+| \gg |G^b_+|$ then $\mathbf{n}(\text{viol}(j), C^g_-) \gg \mathbf{n}(\text{viol}(j), C^g_-)$ and $\mathbf{n}(\text{viol}(j), C^g_+) \gg \mathbf{n}(\text{viol}(j), C^b_+)$, hence the effect of C^b on K_j can be made arbitrarily small.

2.8 Discussion

BayesAcq is a simple CA method with several desirable properties. Firstly, it is robust under noise (as shown in Section 2.7; see also Section 4.7). Secondly, its memory requirement is independent of the bias size (as noted in Section 2.3): we need only store the training data in memory as each candidate can be generated and tested on the fly, so it is scalable to large biases. Thirdly, it is fast (see Section 4).

An interesting aspect of BayesAcq is that it requires a constraint to be violated in at least one non-solution. A candidate that is satisfied in *all* instances, both solutions and

non-solutions, will not be learned as a constraint because $\mathbf{n}(\text{viol}(j), C_{\_}) = \alpha$ for such candidates. This feature was not designed but emerged naturally from Naive Bayes. Very weak constraints, which are unlikely to be violated unless we have a vast amount of data, will not be learned. We consider this an advantage for the following reason. Suppose we implement a CA system for use in the real world, and build into the bias a wide variety of constraints, some of which are very weak. BAYESACQ would not learn these unless there is sufficient data to observe violations. They would probably not exclude many solutions in practice, and they would needlessly complicate the constraint model. Moreover, users would notice that the system always learns the same list of spurious constraints that have nothing to do with the application, leading to distrust of the method.

Another aspect of BAYESACQ is that it does not require each non-solution to violate at least one learned constraint. Version space methods (see Section 3) do have this requirement, and their learned constraints explain every label in the dataset. Hence those methods are solving a more difficult problem than ours, requiring the solution to a coNP-hard subproblem [6], which no doubt contributes to the difference in speed (see Section 4). Note that it would be easy to extend BAYESACQ to output the set of non-solutions that violate no learned constraint, indicating that the bias should be expanded.

3 Related work

A number of methods are reported in the CA literature. The ModelSeeker system [5] requires only a few positive instances, and finds high-level descriptions in terms of global constraints. Tacle [25] learns functions and constraints from spreadsheets. CONACQ [6, 7] is based on version spaces and has passive and active versions. QUACQ [8, 9] is an active system, extended to MULTIACQ [2]. T-QUACQ [1] uses time-bounding to reduce QUACQ runtimes. MQUACQ [42] greatly improves QUACQ and MULTIACQ by reducing the number of generated queries and the complexity of each query. Valiant's method [44] learns SAT formulae from instances and requires no non-solutions. It has been extended to first order logic using inductive logic programming [34, 35], which was also used by [26]. The framework of [47] learns several types of CP model by expressing CA as a constraint problem. The Matchmaker agent [19] interacts with a user who diagnoses why an instance is not a solution. There is also work on learning soft constraints, preferences and SAT modulo theories.

Some existing work is related to CLASSACQ. In [4] hard-to-describe parts of problems are modelled via neuron global constraints, embedded in a larger model designed by an expert. In [27] decision trees and neural networks (NNs) are transformed into solvers called "consistency checking classifiers" which build propagators and answer partial queries. In [10, 29] decision trees and NNs are embedded into CP, integer programs and other optimisation models. In [15, 40] NNs are mapped to integer programs to find inputs that optimise some objective, such as finding optimal adversarial examples or proving that none exist. In [37] NNs are mapped to integer programs to solve planning problems with continuous action spaces. Thus decision trees and NNs have been mapped several times to various optimisation models, but usually not for CA. In [28] the use of machine learning methods to boost combinatorial problem modelling is surveyed, including the representation of such methods within optimisation problems. They view CA as an extreme case in which a machine learning model completely replaces an optimisation model. In our view, as the aim of CA

is to learn a constraint model that is compatible with given data, using classifiers to learn a complete constraint model is a form of passive CA.

A particularly interesting work from our point of view is that of [33], who learn models from noisy training instances, containing linear, quadratic and trigonometric constraints. Their *constraint synthesis* method models and solves the CA problem as mathematical programs, using parameters to control features such as the number of allowed constraints, and their models are human-readable. They discuss the possibility of an approach similar to ClassAcq, and mention that converting a trained neural network to a mathematical programming model requires the introduction of auxiliary variables and additional constraints, but they criticise this idea in two ways. Firstly, they mention a curse of dimensionality associated with CA: that the number of required instances grows exponentially with the number of variables in the instances. However, some classifiers are explicitly designed for very small datasets (as discussed in Section 5.1.1). Secondly, they mention the possibility of using classifiers such as SVMs and Naive Bayes to generate constraints representing decision boundaries between positive and negative instances, but criticise this approach on the grounds of transparency. We argue that constraint models need not be transparent for all applications, for example for testing whether a partial assignment can be extended to a positive instance, for finding optimal adversarial instances, or for verifying classifier properties. We also avoid the problem of a hard-to-understand linear constraint by learning constraints of whatever form the user provides in the bias, via the approximation in Section 2.3.

BayesAcq applied to SAT is similar to Valiant's method [44] in the sense that it is a generate-and-test algorithm: it generates all possible clauses of permitted length, and tests each against the training data. However, there are important differences. BayesAcq has the advantage of robustness under noise, while Valiant's method has the advantage of not requiring negative instances. Also, whereas BayesAcq tests each candidate in isolation, Valiant's algorithm first generates the set of all candidates then prunes them using each training instance in turn. This makes Valiant's method impractical when the bias is very large. A final difference is that Valiant's method will learn any clause that does not contradict the training data. In contrast BayesAcq does not learn clauses (or constraints) that are satisfied by all instances. Hence Valiant's method learns the *most specific* model while BayesAcq is less specific: we discuss a consequence of this in Section 2.8.

4 Experiments

We now test BayesAcq on examples.[1] Unless stated otherwise we use a bias of $\{\leq, \neq, \geq\}$ constraints as in [7]. Note that these subsume $\{<, =, >\}$-constraints: if we learn $x \leq y$ and $x \geq y$ we can deduce $x = y$, while if we learn $x \leq y$ and $x \neq y$ we can deduce $x < y$. The runtimes shown are not averaged over multiple runs because BayesAcq is a deterministic algorithm. It is implemented in the C programming language and executed on a 2.8 GHz Pentium 4 with 512 MB RAM. We use $\alpha = 0.01$ and $\kappa = 20$ for all experiments. For most problems we used 10,000 instances which is typical: several CA methods used 6,000–21,000 in [27].

We shall cite runtimes from other papers using different machines, so they are not directly comparable to ours. However, the machines have similar clock rates: [1] used an Intel(R) Xeon(R) @ 3.40 GHz, [42] used an Intel(R) Core(TM) i5-4690K CPU @ 3.50 GHz with

[1] Source code and examples are available on request from one of the authors (Prestwich).

8Gb of RAM, [6] used an Intel Core i7 @ 2.9 GHz with 8 Gb of RAM, and [2] used a 1.6 GHz Intel Core i5 with 4GB of RAM. Crucially, the improvement in speed due to BAYESACQ is significantly larger than any likely difference in machine performance.

4.1 Latin squares

This example was used in [1, 2, 8, 42]. An $N \times N$ Latin square has variables $x_{i,j}$ ($0 \leq i, j \leq N$) each with domain $\{0, \ldots, N-1\}$ and disequality ($\neq$) constraints between each pair of variables in the same row or column. For $N = 3, \ldots, 10$ we used 5,000 solutions and 5,000 non-solutions. Solutions were generated by randomly permuting the rows, columns and values of a single solution, while non-solutions were generated by randomly choosing a value for each variable. Rejection sampling was implemented to filter out any of the latter instances that were solutions, but this is very unlikely and did not occur in our experiments.

In each case the correct disequalities were found to be *decisive* while all other candidates were less than *substantial*. Runtimes are shown in Table 2. Runtimes for QUACQ were not given in [8] but were reportedly a few milliseconds for $N = 5$, and MULTIACQ was faster in [2] (total runtimes were not given). An improved version of QUACQ called T-QUACQ [1] took 120 seconds to learn a 10×10 Latin square, compared to 7,200 seconds for QUACQ. A comparison of six CA methods was made in [42], and the fastest was MQUACA+FINDSCOPE 2 MAX_B which took 114 seconds with $N = 10$. BAYESACQ is approximately two orders of magnitude faster than any reported method on the largest instance.

4.2 Sudoku

This example was used in [1, 2, 7, 8, 42]. Sudoku is similar to a Latin square, but with the additional constraints that smaller squares of variables must also take all different values. We used 5,000 solutions and 5,000 non-solutions, generated in the same way as Latin squares (except that not all rows and columns can be permuted).

For the 4×4 puzzle BAYESACQ tests 360 candidates and learns the correct 56 disequalities in 0.03 seconds, and for the 9×9 puzzle it tests 9,720 candidates and learns the correct 810 disequalities in 0.4 seconds. On the 9×9 puzzle passive CONACQ took 15.6 seconds to generate background knowledge and approximately 2 seconds for acquisition [6]. QUACQ took approximately 800 seconds and MULTIACQ approximately 900

Table 2 Results for 10,000 $N \times N$ Latin square examples

	Constraints		
N	Tested	Learned	Seconds
3	108	18	0.02
4	360	48	0.03
5	900	100	0.06
6	1,890	180	0.10
7	3,528	294	0.16
8	6,048	448	0.25
9	9,720	648	0.37
10	14,850	900	0.54

Table 3 Results for 1,000 random 3-SAT instances with V variables and 5 clauses

	Tested clauses with ℓ literals			
V	$\ell = 3$	$\ell = 2$	$\ell = 1$	Seconds
50	156,800	4,900	100	1.8
100	1,293,600	19,800	200	16
150	4,410,400	44,700	300	56
200	10,507,200	79,600	400	123
250	20,584,000	124,500	500	243

seconds [2]. In [1] QUACQ took 2,810 seconds and T-QUACQ 69 seconds, while in [42] MQUACA+FINDSCOPE 2 MAX_B took 85 seconds and beat five other methods. On the larger puzzle BAYESACQ is approximately two orders of magnitude faster than any reported method.

4.3 Golomb rulers

This example was used in [1, 2, 6]. A Golomb ruler is a set of N marks at integer positions along an imaginary ruler such that no two pairs of marks are the same distance apart. The smallest number is 0 and the largest is the ruler length. A problem description and constraint models are described in [38]. We generated 5,000 solutions and 5,000 non-solutions in the same way as Latin squares and Sudoku, except that solutions were generated by permuting several known optimal rulers. The problem is modelled by integer variables $x_1, \ldots, x_{N-1}$ with domains $\{0, \ldots, L\}$. To our usual bias we add quaternary constraints $|x_i - x_j| \neq |x_{i'} - x_{j'}|$ $(i < j, i' < j', i < i', j \neq j')$.

For $N = 12$ BAYESACQ tests 198 binary and 1,485 quaternary candidates, and correctly learns 66 disequalities and all the quaternaries as constraints in 0.07 seconds. In [2] QUACQ took 2,257 seconds while MULTIACQ took 2,335 seconds. In [6] CONACQ took 2,193 seconds on a smaller instance ($N = 8$). In [1] QUACQ took 11,972 seconds while T-QUACQ took 1,184 seconds. BAYESACQ is more than four orders of magnitude faster than any reported method.

4.4 Random 3-SAT

The benchmarks used above were learned very quickly by BAYESACQ so we now tackle a larger problem. We generated random 3-SAT instances with V variables, C clauses and E randomly-generated instances. We use several values of V and choose $E = 1000$, and $C = 5$ so that approximately half of the instances are solutions. The bias is the set of all possible clauses with 1, 2 or 3 literals. The results are shown in Table 3 with the number of clauses tested in the *tested* column and the learning runtime in the *seconds* column. For the largest instance the bias contains over 20 million clauses, which is considerably larger than biases used in other CA papers.

BAYESACQ learns the correct clauses, but sometimes it also learns a few additional clauses. This is not an error: if there exists a 1-, 2- or 3-literal clause that can be derived by resolution from the original C clauses then it will also be learned. This is the SAT equivalent of an implied constraint, and BAYESACQ tries to learn learn any candidate that is a valid constraint, whether implied or part of the original model.

We also generated a 1000-variable random 3-SAT example with a bias of 1.3 billion clauses and 1000 instances. We increased the size of the target to 50 clauses, obtaining an approximately balanced dataset via rejection sampling (only accepting non-solutions with probability 0.0013). BAYESACQ took 16,259 seconds to learn the correct clauses. This shows that it can learn from biases that are much larger than those used in most CA papers.

4.5 Redundant and implied constraints

For CA systems based on version spaces, redundant constraints can prevent learning. We take a small example from [7] which was designed to illustrate the problem. The training data is:

x_1	x_2	x_3	solution?
4	3	1	yes
2	3	1	no
3	1	2	no

For this example we use a bias of $\{<, >, \leq, \geq, \neq\}$-constraints. Some of these are made redundant by others, for example $x_1 < x_2$ makes $x_1 \neq x_2$ redundant. This prevented CONACQ from eliminating some hypotheses such as $x_1 \neq x_2$, and a special technique (redundancy rules) was added to handle such cases. The target constraints are $x_1 > x_2$, $x_1 > x_3$ and $x_2 > x_3$. This example causes no problems for BAYESACQ, which learns the constraints $x_1 > x_2$, $x_2 > x_3$, $x_1 \geq x_2$ and $x_2 \geq x_3$. It does not learn $x_1 > x_3$ because in this small dataset it is not violated anywhere, but it is made redundant by $x_1 > x_2$ and $x_2 > x_3$. Those also make $x_1 \geq x_2$ and $x_2 \geq x_3$ redundant. If we eliminate these we have a minimal set of constraints $x_1 > x_2$ and $x_2 > x_3$. If more instances were added to the training data we would also expect to learn $x_1 > x_3$ (and $x_1 \geq x_3$).

In [7] a further example is designed to show that redundancy rules do not always work, necessitating a more complex technique based on higher-order redundancies. The training data is:

x_1	x_2	x_3	solution?
2	2	2	yes
3	3	4	no
1	3	3	no

This time the target constraints are $x_1 = x_2$, $x_1 = x_3$ and $x_2 = x_3$. BAYESACQ learns these plus $x_1 \geq x_2$, $x_1 \geq x_3$ and $x_2 \geq x_3$, which are redundant and can be eliminated. In future work we aim to address this issue, but it is orthogonal to the CLASSACQ approach.

4.6 Choosing parameters

When applying BAYESACQ we must choose values for the additive smoothing parameter α and threshold κ. Setting $\alpha = 1$ is reasonable and can be justified using Laplace's *rule of succession* and the *principle of indifference*, in which case additive smoothing corresponds to *add-one smoothing*. A value of $\frac{1}{2}$ corresponds to the *Jeffries prior*. But a lower value

such as 0.01 is often chosen, and it has the advantage of yielding a significant weight of evidence, even with only one observed non-solution violation. We therefore use a small α value, though if the data is noisy then it might be safer to use a larger value, thus requiring several non-solution violations before accepting that a candidate is indeed a constraint.

Choosing an appropriate κ is critical. A threshold of (say) *very strong* should work for most applications given sufficient data, but we would like BAYESACQ to yield reasonable results even on small datasets. Fortunately we can exploit the fact that weights of evidence tend to be clustered around 0 for non-constraint candidates, and around some higher value for constraints. To illustrate this property, we generate scatterplots of the K_j values for the $\neq$ version of the Latin square problem in Section 4.1, using datasets of different size: 10 samples per class in Fig. 1, 100 in Fig. 2 and 1000 in Fig. 3. It can also be seen that the more instances are used, the clearer the separation between constraints and non-constraints. Taking more samples also increases confidence in a learned constraint: with 10 samples per class we find the highest K_j values are around 20–23; with 100 they are around 20–30, and with 1000 they are around 33–37. Taking more samples also reduces error: with 1000 instances per class all 100 constraints have high K_j, but the number of learned constraints reduces with fewer samples. The graphs also illustrate a point made in Section 2.6: that with small datasets we observe some large negative K_j values, corresponding to non-constraints that are violated only in non-solutions, but these vanish as the data grows because non-constraints tend to be violated in solutions.

Based on these observations, a reasonable approach would be to assume that the K_j-values form a Gaussian mixture model with means at 0 (for non-constraints) and some unknown higher value (for constraints). We could then use standard statistical techniques to choose κ between the two means. An alternative would be a clustering algorithm such as k-means (with k=2). However, a fast and simple method is sufficient for all our examples: choose κ to be half the value of the greatest K_j value $\hat{K}$ in the dataset, that is $\kappa = \hat{K}/2$.

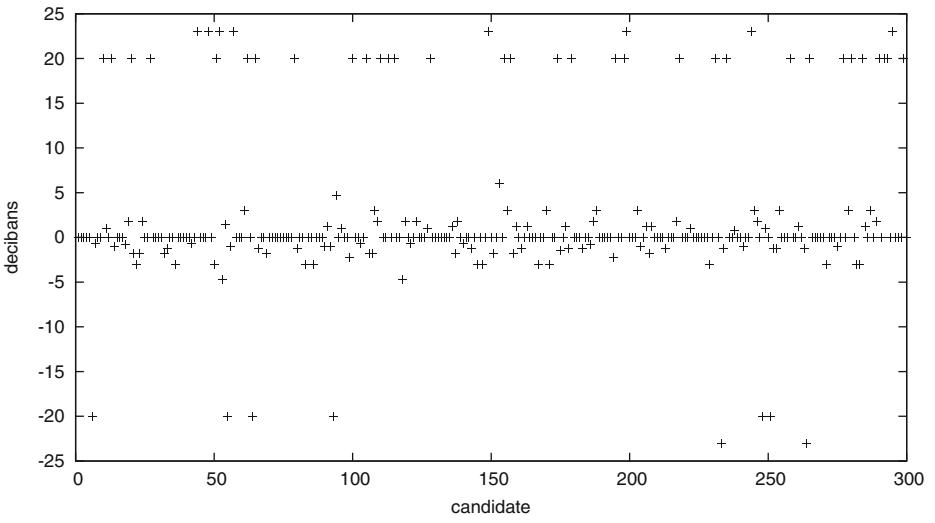

Fig. 1 Latin square disequality K_j values (10 instances per class)

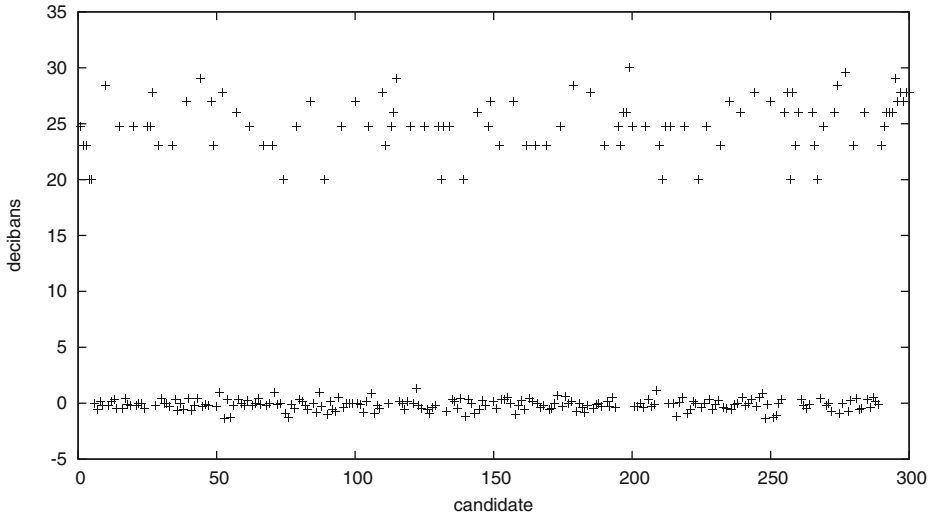

Fig. 2 Latin square disequality K_j values (100 instances per class)

4.7 Robustness experiments

The ability to recover from a small number of errors (see Section 2.7) is an advance in CA, but this is a weak form of robustness. If the data source generating the instances has a constant error rate then there might never be sufficient good data to overwhelm the bad.

We now perform experiments to test the ability of BAYESACQ to learn under constant error rates. We corrupted the 250-variable SAT example by deliberately misclassifying a

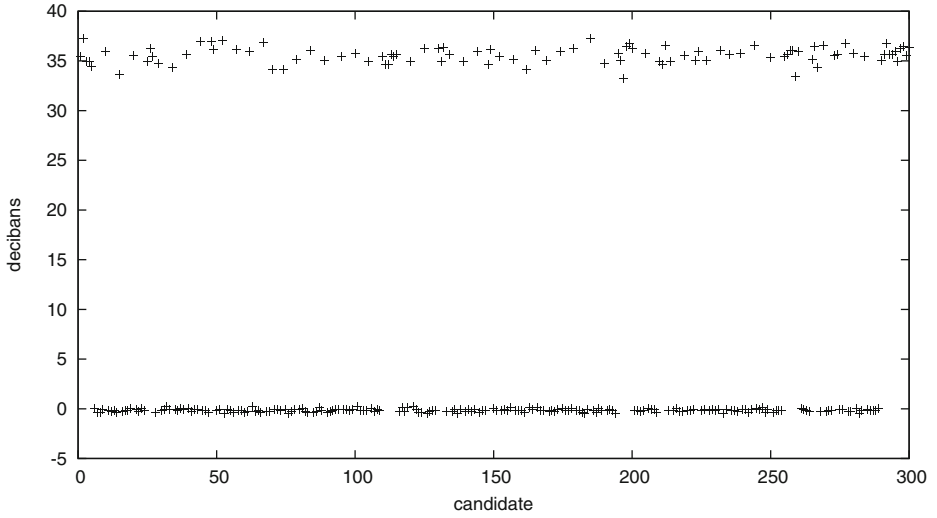

Fig. 3 Latin square disequality K_j values (1000 instances per class)

fixed percentage of the instances, randomly selected. With no misclassification the clauses have K-values of approximately 40 decibans with other candidates are below 10, so our default threshold $\kappa = 20$ works well. Adding errors and keeping the default value $\alpha = 0.01$:

- With 1% misclassification the clauses have K-values of 18–41 decibans, while other candidates have values as high as 9. The errors reduce the distinction between true clauses and other candidates, but by choosing κ more carefully we can still learn the instance.
- With 3% misclassification the clauses have K-values of 15–41, while other candidates are again as high as 9. The distinction is more blurred but still significant.
- With 10% misclassification the K-values of clauses have a reduced range of approximately 8–12, while other candidates have values as high as 7. We still learn the instance by choosing $\kappa = 8$ but the distinction is becoming very blurred, and we do not expect all problems to be learned correctly.
- With 20% misclassification no value of κ learns the instance correctly.

We repeated the experiment with the 9×9 Sudoku problem. With no misclassification any κ greater than 1 deciban learns the correct constraints. With 1% misclassification the range reduces to 2–6 decibans, with 3% the range is 2–4, with 10% only $\kappa = 2$ works, and with 20% no κ value works.

These results show empirically that BAYESACQ can learn constraint models correctly even when there is significant noise in the data. However, as the error rate increases more care must be taken when choosing parameter values, thresholds classed as *barely worth mentioning* become necessary, and too many errors defeat the method. (Note that changing κ or α does not affect runtimes, so learning from noisy data does not take longer.)

5 Conclusion and future work

We proposed a general approach to CA called CLASSACQ based on classifiers. The motivation is to exploit existing classifiers to generate new CA methods with interesting properties. Given the large number of classifiers, constraint models, and methods (existing or yet to be found) for transforming classifiers into constraint models, CLASSACQ yields a rich landscape of new CA methods to explore. Only a small number of these have been explored so far, mainly using decision trees and neural networks. In future work we shall explore other combinations. We might even envision putting them into a portfolio for automating the selection of a combination for specific tasks. We view all this as progress toward the Holy Grail of computer science mentioned in Section 1.

As a first illustration of the potential of CLASSACQ we derived a new CA method called BAYESACQ from a Bernoulli Naive Bayes classifier, which is applicable to SAT and finite-domain CSPs. It inherits the simplicity, scalability and robustness of Naive Bayes. In experiments it achieves state-of-the-art speed on several benchmarks, its low memory requirement allows it to scale up to biases containing over a billion candidates, and it is able to learn redundant constraints that cause problems for version space methods. The Bayesian hypothesis testing connection mentioned in Section 2.5 has since been exploited to obtain an even faster CA method, using the (non-Bayesian but related) Sequential Probability Ratio Test [31].

5.1 Future work

There are many other classifiers that have not yet been used for CA. For example, instead of Naive Bayes we could use a classifier based on a less trivial Bayesian network. This might give more accurate results while retaining robustness, though sacrificing some speed.

We now mention some practical problems arising in CA, and suggest classifiers that might be used to derive new CA methods to tackle these problems. Though these are only suggestions, we see this as a mapping out of an interesting programme of work, and as a contribution of this paper. The derivation of BAYESACQ from a Naive Bayes classifier, along with existing results from deep learning and decision trees, indicates the fruitfulness of the idea.

5.1.1 Small datasets

In [33] the possibility of using classifiers to generate constraint models is discussed, but criticised on the grounds that the number of required instances grows exponentially with the number of variables in the instances. However, some classifiers are explicitly designed for small datasets [39]. In *few-shot learning* the training dataset has only a small number of instances from each class, or just one in the case of *one-shot learning*. Recent examples of such classifiers are *Prototypical Networks* [39] and *Matching Networks* [46] which are based on nearest-neighbour algorithms. Nearest-neighbour classifiers have been applied to problems with both large and small datasets, including image classification, recommender systems, document classification, medical diagnosis, facial recognition and theft prevention.

We can derive constraint models for such classifiers. A particularly simple example is the basic nearest-neighbour algorithm. If we have just one solution and one failure, the classifier reduces to a single constraint stating that an instance is closer to the solution than to the failure. Depending on the distance metric, this might be expressed using a global distance constraint. It can be generalised to multiple instances and weighted k-nearest neighbours.

5.1.2 Imbalanced datasets

In some CA applications it is impractical to obtain a large dataset of negative instances. For example we might collect solutions automatically, but have no idea what failures look like. For such problems we can adapt *one-class classifiers* which are surveyed in [24] and can be used when the negative class is absent, poorly sampled or ill-defined. The aim of one-class classification is to recognise instances from a class, rather than to discriminate between classes. Its many applications include the detection of abnormal machine behaviour, automatic medical diagnosis and authorship verification [24]. Proposed approaches include a form of SVM, neural networks, decision trees, nearest neighbours, genetic algorithms and Bayesian methods.

A simple example is the method of [12], which computes the convex hull of the training data (actually a computationally cheaper approximation based on random projections). A convex hull is a convex polytope which can be modelled exactly using a linear program. If the data is integral then integrality constraints can be added to obtain an integer program or finite-domain CSP.

The ModelSeeker CA system [5] also requires only positive instances, and has successfully found global constraint models for several applications. However, applications such as those above might not have a deep constraint structure to be discovered, and for these a model based on a one-class classifier is an interesting alternative. Both are ideal

for historical data, but ModelSeeker requires a heuristic step to choose between alternative models.

5.1.3 Mixed data types

CA systems typically generate discrete constraint models. However, many classifiers work on continuous variables, for example support vector machines (SVMs) and deep learning classifiers. They can also be applied to categorical variables via *one-hot encoding* (also called *binarisation* or *reification*).

Decision trees and random forests handle combinations of categorical, discrete and continuous variables in a natural way. There are at least three known ways of transforming a decision tree to a CP: a rule-based method using Boolean meta-constraints, table constraints, and decision diagram-based global constraints [10]. In [29, 45] a method similar to the rule-based model is used for CP and integer programming. However, these are not generally thought of as CA methods, and their usual aim is to speed up solution methods by learning part of a model.

5.1.4 Overfitting

Overfitting is a major problem in supervised learning, and occurs when a learner interprets errors or noise as data, or places too much emphasis on errors. Current CA methods are not robust in this sense: for example the algorithm of Valiant [44] is highly vulnerable to outliers. In contrast, many classifiers are designed to resist overfitting. For example soft-margin SVM explicitly allows a small number of exceptions. Bayesian classifiers are particularly robust as they are probabilistic in nature. Deep learning classifiers use the *dropout* technique to reduce overfitting by introducing noise, and often use a validation dataset to detect its occurrence.

As an example we consider SVMs which are state-of-the-art for a vast range of applications, including medical diagnosis, fault detection and satellite data. The simplest version learns a maximum-margin hyperplane, and we can impose a single constraint stating that an instance lies on its positive side. This can be generalised to soft margins, and adding a kernel leads to a nonlinear constraint. This possibility is mentioned in [33] but criticised on the grounds that the resulting models are hard for humans to understand. We discuss this point in Section 3.

5.1.5 Auxiliary variables and channeling constraints

In the CA literature the learned model is usually a set of constraints on the given variables. However, it is well known in CP that better models are sometimes obtained by defining extra *auxiliary variables*, on which it might be easier or more powerful to express certain constraints. Auxiliary and given variables are connected to each other by adding *channeling constraints* to the model [13]. However, improved filtering is not the only motivation for creating auxiliary variables: for some problems they greatly reduce the size of the constraint model, for example the problem of finding *covering arrays* [22] (see [20] chapter 11.8 for a discussion of similar techniques for other problems). A similar result holds in SAT, where auxiliary variables can be used to obtain *Tseitin encodings* that are exponentially smaller than "flat" encodings [41].

Although auxiliary variables are an important modeling technique, their automatic discovery has not been addressed in the CA literature. Constraint models derived from decision

trees contain auxiliary variables, but they do not resemble the models generated by human experts and are not introduced explicitly to reduce model size. Ideally we require a method for discovering *useful* auxiliary variables, arranged in layers and connected by channeling constraints, which lead to compact constraint models.

For this we propose Deep Learning (DL) classifiers which have recently swept the field in many areas, including image and video analysis, bioinformatics and malware detection. It is known in DL that, although feedforward networks with a single hidden layer are *universal approximators* that can model any function with arbitrary accuracy, deep networks can be much more compact. DL also has techniques for reducing network size [11, 16] which in ClassAcq leads directly to smaller constraint models. We therefore conjecture that CA will be particularly powerful when based on deep learning to learn what we might call *deep constraint models*. Compiling neural networks to optimisation models is well-known (see Section 3) but its connection to CA and auxiliary variables has not been previously pointed out, nor the use of network compression techniques to reduce model size.

Acknowledgements This material is based upon works supported by the Science Foundation Ireland under Grant No. 12/RC/2289-P2 which is co-funded under the European Regional Development Fund.

Funding Open Access funding provided by the IReL Consortium.

References

1. Asdi, H.A., Bessiere, C., Ezzahir, R., Lazaar, N.: Time-bounded query generator for constraint acquisition. In: Proceedings of the 15th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research, Lecture Notes in Computer Science, vol. 10848, pp. 1–17 (2018)
2. Arcangioli, R., Bessiere, C., Lazaar, N.: Multiple constraint acquisition. In: Proceedings of the 25th International Joint Conference on Artificial Intelligence (2016)
3. Asafu-Adjei, J.K., Betensky, R.A.: A pairwise Naïve Bayes approach to Bayesian classification. Intern. J. Pattern Recognit. Artif. Intell. **29**(7) (2015)
4. Bartolini, A., Lombardi, M., Milano, M., Benini, L.: Neuron constraints to model complex real-world problems. In: Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming Lecture Notes in Computer Science, vol. 6876, pp. 115–129 (2011)
5. Beldiceanu, N., Simonis, H.: Modelseeker: Extracting global constraint models from positive examples. In: Data Mining and Constraint Programming, Lecture Notes in Computer Science, vol. 10101, pp. 77–95. Springer (2016)
6. Bessiere, C., Koriche, F., Lazaara, N., O'Sullivan, B.: Constraint acquisition. Artif. Intell. **244**, 315–342 (2017)
7. Bessiere, C., Coletta, R., Freuder, E.C., O'Sullivan, B.: Leveraging the learning power of examples in automated constraint acquisition. In: Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming Lecture Notes in Computer Science, vol. 3258, pp. 123–137 (2004)

8. Bessiere, C., Coletta, R., Daoudi, A., Lazaar, N., Bouyakhf, E.H.: Boosting constraint acquisition via generalization queries. In: Proceedings of the 21st European Conference on Artificial Intelligence, pp. 99–104 (2014)
9. Bessiere, C., Coletta, R., Hebrard, E., Katsirelos, G., Lazaar, N., Narodytska, N., Quimper, C.-G., Walsh, T.: Constraint acquisition via partial queries. In: Proceedings of the 23rd International Joint Conference on Artificial Intelligence, pp. 475–481. AAAI Press (2013)
10. Bonfietti, A., Lombardi, M., Milano, M.: Embedding decision trees and random forests in constraint programming. In: Proceedings of the International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Lecture Notes in Computer Science, vol. 9075, pp. 74–90. Springer (2015)
11. Browne, D., Giering, M., Prestwich, S.D.: Pulse-net: Dynamic compression of convolutional neural networks. In: Proceedings of the IEEE 5th World Forum on Internet of Things (2019)
12. Casale, P., Pujol, O., Radeva, P.: Approximate convex hulls family for One-Class classification. In: proceedings of the International Workshop on Multiple Classifier Systems Lecture in Notes Computer Sci, vol. 6713, pp. 106–115 (2011)
13. Cheng, B.M.W., Choi, K.M.F., Lee, H.H.M., Wu, J.C.K.: Increasing constraint propagation by redundant modeling: An experience report. Constraints **4**, 167–192 (1999)
14. Domingos, P., Pazzani, M.: On the optimality of the simple bayesian classifier under Zero-One loss. Mach. Learn. **29**, 103–130 (1997)
15. Fischetti, M., Jo, J.: Deep neural networks as 0-1 mixed integer linear programs: A feasibility study. Constraints **23**(3), 296–309 (2018)
16. Frankle, J., Carbin, M.: The lottery ticket hypothesis: Finding sparse, trainable neural networks. In: Proceedings of the International Conference on Learning Representations (2019). to appear
17. Freuder, E.C.: Constraints: The ties that bind. In: Proceedings of the 21st National Conference on Artificial Intelligence, pp. 1520–1523. AAAI Press (2006)
18. Freuder, E.C.: Progress towards the holy grail. Constraints **23**, 158–171 (2018)
19. Freuder, E.C., Wallace, R.J.: Suggestion strategies for Constraint-Based matchmaker agents. Int. J. Artif. Intell. Tools **11**(1), 3–18 (2002)
20. Gent, I.P., Petrie, K.E., Puget, J.-F.: Handbook of Constraint Programming. Elsevier, Amsterdam (2006)
21. Good, I.J.: Turing's anticipation of empirical Bayes in connection with the cryptanalysis of the naval enigma. J. Stat. Comput. Simul. **66**(2), 101–111 (2000)
22. Hnich, B., Prestwich, S.D., Selensky, E., Smith, B.M.: Constraint models for the covering test problem. Constraints **11**(3), 199–219 (2006)
23. Kass, R.E., Raftery, A.E.: Bayes Factors. J. Amer. Stat. Assoc. **90**(430), 773–795 (1995)
24. Khan, S., Madden, M.: One-Class Classification: Taxonomy of study and review of techniques. Knowl. Eng. Rev. **29**(3), 345–374 (2014)
25. Kolb, S., Paramonov, S., Guns, T., De Raedt, L.: Learning constraints in spreadsheets and tabular data. Mach. Learn. **106**, 1441–1468 (2017)
26. Lallouet, A., Lopez, M., Martin, L., Vrain, C.: On learning constraint problems. In: Proceedings of the IEEE International Conference on Tools With Artificial Intelligence, pp. 45–52 (2010)
27. Lallouet, A., Legtchenko, A.: Two contributions of constraint programming to machine learning. In: Proceedings of the European Conference on Machine Learning Lecture Notes in Artificial Intelligence, vol. 3720, pp. 617–624. Springer (2005)
28. 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 (2018)
29. Lombardi, M., Milano, M., Bartolini, A.: Empirical decision model learning. Artif. Intell. **244**(Supplement C), 343–367 (2017)
30. Manning, C.D., Raghavan, P., Schütze, M.: Introduction to information retrieval. Cambridge University Press, Cambridge (2008)
31. Prestwich, S.D.: Robust constraint acquisition by sequential analysis. In: Proceedings of the 24th European Conference on Artificial Intelligence, Frontiers in Artificial Intelligence and Applications, vol. 325, pp. 355–362. IOS Press (2020)
32. O'Sullivan, B.: Automated modelling and solving in constraint programming. In: Proceedings of the 24th AAAI Conference on Artificial Intelligence, pp. 1493–1497 (2010)
33. 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)
34. De Raedt, L., Dehaspe, L.: Clausal discovery. Mach. Learn. **26**, 99–146 (1997)
35. De Raedt, L., Džeroski, S.: First Order jk-clausal Theories are PAC-learnable. Artif. Intell. **70**, 375–392 (1994)

36. De Raedt, L., Passerini, A., Reso, S.: Learning constraints from examples. In: Proceedings of the 32nd AAAI Conference on Artificial Intelligence, pp. 7965–7970 (2018)
37. Say, B., Wu, G., Zhou, Y.Q., Sanner, S.: Nonlinear hybrid planning with deep net learned transition models and mixed-integer linear programs. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence, pp. 750–756 (2017)
38. Smith, B.M., Stergiou, K., Walsh, T.: Modelling the Golomb Ruler Problem. In: Proceedings of the 16th International Joint Conference on Artificial Intelligence (1999)
39. Snell, J., Swersky, K., Zemel, R.: Prototypical networks for few-shot learning. In: Proceedings of the 31st Conference on Neural Information Processing Systems (2017)
40. Tjeng, V., Tedrake, R.: Verifying neural networks with mixed integer programming. coRR (2017)
41. Tseitin, G.: On the complexity of derivation in propositional calculus. In: Siekmann, J., Wrightson, G. (eds.) Automation of Reasoning: Classical Papers in Computational Logic, vol. 2, pp. 466–483. Springer (1983)
42. Tsouros, D.C., Stergiou, K., Sarigiannidis, P.G.: Efficient methods for constraint acquisition. In: Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science, vol. 11008, pp. 373–388 (2018)
43. Tsouros, D.C., Stergiou, K., Bessiere, C.: Structure-driven multiple constraint acquisition. In: 25th International Conference on Principles and Practice of Constraint Programming Lecture Notes in Computer Science, vol. 11802, pp. 709–725 (2019)
44. Valiant, L.G.: A theory of the learnable. Commun. ACM **27**(11), 1134–1142 (1984)
45. Verwer, S., Zhang, Y., Ye, Q.C.: Auction optimization using regression trees and linear models as integer programs. Artif. Intell. **244**, 368–395 (2017)
46. Vinyals, O., Blundell, C., Lillicrap, T., Kavukcuoglu, K., Wierstra, D.: Matching networks for one shot learning. In: Proceedings of the 30th Conference on Neural Information Processing Systems, pp. 3637–3645 (2016)
47. Vu, X.-H., O'Sullivan, B.: A unifying framework for generalized constraint acquisition. Int. J. Artif. Intell. Tools **17**(5), 803–833 (2008)