Bayesian Constraint Acquisition

Steve Prestwich 2019

(Work done partly with Barry, Gene and Dave)

Overview

Modeling a combinatorial problem is a hard and error-prone task requiring expertise.

Constraint acquisition (CA) can automate this process by learning constraints from examples of solutions and (usually) non-solutions.

I describe a new statistical approach based on sequential Bayesian hypothesis testing (**sequential analysis**) that's orders of magnitude faster than existing methods.

It's also the first **robust** CA method: it can learn constraints correctly from noisy data.

Constraint programming

Constraint Programming (CP) is a powerful approach to modelling and solving decision and optimisation problems. It draws on techniques from AI, OR, graph theory etc to provide a wide range of variable types, constraints, filtering algorithms, search strategies and specification languages.

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.

But modelling an application as a CS[O]P remains a task for experts [Freuder, Puget, O'Sullivan].

Constraint acquisition

This modelling problem, and the successes of Machine Learning at automating a wide variety of tasks, has inspired the field of CA (closely related to *Constraint Learning*, *Constraint Synthesis*, and *Empirical Model Learning*).

In CA we're given examples of solutions and non-solutions (positive and negative examples, successes and failures) and the aim is to learn a constraint model that represents them.

The goal might be *automated problem modelling*, to use the model as an *explanation* of the problem, to enable *classification of partial assignments*, to *speed up* the solution of future problems, or to find *instances that optimise some objective*.

CA has been identified as an important topic, and recognised as progress toward the “holy grail” of computing in which a user simply states a problem and the computer proceeds to solve it without further programming.

Active CA methods are guided by interaction with a user or other oracle, while **passive** methods learn automatically (I'll only talk about passive CA).

Several CA systems have been devised, many based on **version space learning** or **inductive logic programming**. They usually require a set of **candidate constraints**, also called a **bias**, that may or may not occur in the model we are trying to learn.

Short survey

(*Insight UCC is well-represented!*)

Conacq [Bessiere et al.] is based on version spaces and has passive and active versions.

QuAcq [Bessiere et al.] is an active system. MultiAcq [Addi et al.] is a related method that can learn more constraints from an example. T-QuAcq [Addit et al.] uses time-bounding to reduce runtimes. MQuAcq [Tsouros et al.] improves QuAcq and MultiAcq by reducing the number of generated queries and the complexity of each query.

ModelSeeker [Beldiceanu & Simonis] needs only a few positive instances, and finds high-level descriptions using global constraints.

The Matchmaker agent [Freuder & Wallace] interacts with a user who diagnoses why an example is not a solution.

The framework of [Vu & O'Sullivan] learns several types of constraint model by expressing CA as a constraint problem.

Tacle [Kolb et al.] learns functions and constraints from spreadsheets.

Valiant's method [Valiant] learns SAT instances from positive examples only, and has been extended to first order logic using inductive logic programming.

There's also work on learning soft constraints, preferences and SAT modulo theories.

CA via classification

Recently an alternative approach has emerged (though it's not always presented as a CA method): train a classifier to distinguish between solutions and non-solutions, then derive a constraint model from the trained classifier.

I call this *ClassAcq*. It's already been done for decision trees, SVMs and neural classifiers, but there are many other classifiers with interesting properties that might be used.

I'll show that applying the ClassAcq idea to a Naive Bayes (NB) classifier leads to a fast robust CA method. Then I'll enhance the method using sequential analysis.

CA by Naive Bayes

NB classifiers are based on an assumption of independence between variables, which at first glance seems to make them unsuitable for learning constraints between variables!

But to learn binary constraints we could combine pairs of variables into single features, which is essentially how a Pairwise NB classifier works.

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 training data is a set of instances of the form $\vec{x} = \langle x_1, \ldots, x_N \rangle$, where each variable x_i can in principle have any domain, and each instance is in class C_+ (solutions) or C_- (non-solutions).

We require a set of *candidate constraints*, also called the *bias*, that may or may not occur in the model we are trying to learn.

We derive binary features c_i: for any example $c_i = 1$ iff candidate i is violated by that example. This transforms the training data into a set of binary vectors, each bit or feature corresponding to a candidate.

example

Take a vertex colouring problem with nodes x, y, z, arcs x–y and y–z, colours $x \in \{R, G\}$, $y \in \{R, G\}$, $z \in \{G, B\}$, bias $\{x \neq y, x \neq z, y \neq z\}$, and training examples $C_+ = \{RGB, GRG, GRB\}$ and $C_- = \{RRG, GGB, RGG\}$, or in feature space $\{000, 000, 000\}$ and $\{100, 100, 001\}$.

Which candidates in the bias are constraints? $x \neq y$ and $y \neq z$ are violated by solutions but $x \neq z$ isn't, so we might conclude that those 2 are constraints.

(We used only C_+ but most methods also use C_-.)

Because the features are binary we use Bernoulli NB. It selects a class using the *maximum a posteriori* rule:

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

ie select the class k that is the mode of the posterior distribution, where $p(C)$ is a prior class probability and $p(x|C)$ is the conditional probability of observing x in class C. In our application an example is a solution iff:

$$\prod_i \frac{p(c_i = 1|C_-)}{p(c_i = 1|C_+)} < \frac{p(C_+)}{p(C_-)}$$

In general we don't know $p(C_-)$ or $p(C_+)$ because there's no guarantee that these probabilities are reflected in the training data. Eg given a tightly constrained problem we might generate training data with similar numbers of solutions and non-solutions to facilitate learning. And we rarely know how tightly-constrained an unknown constraint model is.

So we assume an *uninformed prior* $p(C_+) = p(C_-) = 1$. Then an example is classed as a solution iff

$$\prod_i \frac{p(c_i = 1|C_-)}{p(c_i = 1|C_+)} < 1 \quad \text{or} \quad \sum_i \ln\left(\frac{p(c_i = 1|C_-)}{p(c_i = 1|C_+)}\right) < 0$$

This linear constraint mimics a NB classifier given c_i values: given any previously unseen example, we can compute the c_i then test the linear constraint; if it is satisfied then the example is classified as a solution; if it is violated the example is classified as a non-solution.

The constraint can also be used to check whether a partial assignment to the c_i 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_i.

We now have a constraint model derived from NB: are we done? No!

It only has 1 big linear constraint on binary variables (c_i), plus a lot of “reification constraints” linking the c_i to the problem variables. This is not what we wanted.

Instead we'd like to learn which candidates i are in the model.

Luckily, in practice the coefficients of c_i for actual constraints are quite large positive values, while those for non-constraint candidates have positive or negative values close to 0. We can exploit this:

- Force $c_i = 0$ for candidates i with large coefficients, thus insisting that those candidates are satisfied: these are the learned constraints.
- Simply ignore all other candidates because there is insufficient evidence that they are constraints. This approximation turns out to work fine.

In fact there's no need to generate a feature-based dataset, which is fortunate as the bias might be large. We can discard NB and the c_i leaving a simple test: for each candidate i compute

$$K_i = \frac{p(\text{viol}(i)|C_-)}{p(\text{viol}(i)|C_+)}$$

where viol(i) means that candidate i is violated by an example. Then candidate i is accepted as a constraint if and only if $K_i > \kappa$ for some threshold κ.

(Conditional probabilities are estimated by counting occurrences in the data.)

The method has two parameters: an additive smoothing constant often used to avoid zeroes and infinities in Bayesian methods, and κ (I'll discard these later).

The test has a straightforward intuition: a constraint should be satisfied by all solutions (or most if we accept the possibility of error) but might be violated or satisfied by many non-solutions.

We call this CA method <u>BayesAcq</u> (cf ConAcq etc).

CA by sequential analysis

K_i can be viewed as a likelihood ratio called a **Bayes factor**, so the BayesAcq test can be viewed as an application of Bayesian hypothesis testing (BHT):

- the violations of candidate i by examples are the observed data
- non-solutionhood of an example (membership of C_-) is the null hypothesis $\mathcal{H}_0$
- solutionhood (membership of C_+) is the alternative hypothesis $\mathcal{H}_1$

The Bayes factor measures the relative plausibility of hypotheses $\mathcal{H}_0$ and $\mathcal{H}_1$ based on the observed data for candidate i. If $\mathcal{H}_0$ is sufficiently more plausible this fits one definition of a constraint: a relation that is far more likely to be violated by a non-solution than by a solution.

Seems nice but a bit academic: how can we exploit this connection?

BayesAcq calculates Bayes factors using all available examples, but this is not always necessary! In *sequential* BHT, or **sequential analysis**, the sample size is not fixed in advance, and a stopping rule can be used to accept or reject a hypothesis much earlier.

As samples arrive the Bayes factor is updated and monitored: if it becomes large enough then the null hypothesis is accepted, while if it becomes small enough the alternative hypothesis is accepted.

Early stopping has been used many times...

- Clinical trials can be halted as soon as it becomes obvious that an experimental treatment is harmful, or that one treatment is much more successful than another.

- In manufacturing, product lots are tested for defects: lots should be accepted or rejected after as few tests as possible, to save time and costs.

- A similar approach (*Banburismus*) was developed independently by Turing for fast decryption.

Similarly, we can speed up CA by using fewer examples when testing candidates.

I use a simple algorithm invented by OR pioneer Abraham Wald in 1945: the **Sequential Probability Ratio Test** (SPRT). It implicitly uses Bayesian updates but (probably because of the primitive state of computing in the 1940s) avoids divisions via an approximation.

I'll use a manufacturing example to illustrate SPRT...

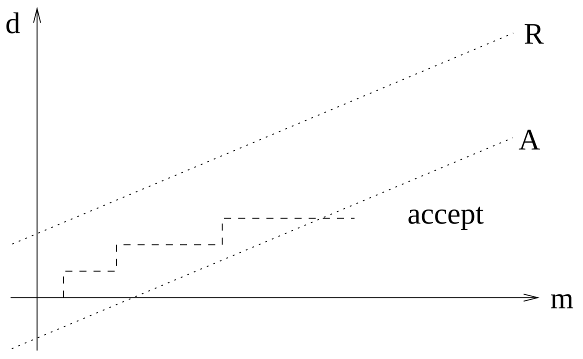
Products are sampled and tested one by one ($m = 1, 2, \ldots$), counting the number d_m of defects found so far.

If at any point $d_m < A_m$ the lot is accepted and the algorithm halts (A_m is an **acceptance number**).

But if at any point $d_m > R_m$ the lot is rejected and the algorithm halts (R_m is a **rejection number**).

Otherwise the algorithm continues indefinitely.

A_m and R_m increase with time, eg



If we cross the A-line we accept the lot, if we cross the R-line we reject it, otherwise we continue sampling.

SPRT has 4 probability parameters p_0, p_1, α, β which specify how to compute A_m, R_m:

$$A_m = \frac{\ln \frac{\beta}{1-\alpha}}{\ln \frac{p_1}{p_0} + \ln \frac{1-p_1}{1-p_0}} + m \frac{\ln \frac{1-p_0}{1-p_1}}{\ln \frac{p_1}{p_0} - \ln \frac{1-p_1}{1-p_0}}$$

$$R_m = \frac{\ln \frac{1-\beta}{\alpha}}{\ln \frac{p_1}{p_0} + \ln \frac{1-p_1}{1-p_0}} + m \frac{\ln \frac{1-p_0}{1-p_1}}{\ln \frac{p_1}{p_0} - \ln \frac{1-p_1}{1-p_0}}$$

This specifies the sampling plan.

(*Nice algorithm! Any ideas for other applications?*)

We can apply SPRT to BayesAcq to get a sequential version: <u>SeqBayesAcq</u>. It's potentially faster because it adaptively reduces the number of examples used for testing a candidate. (In particular: assuming no data errors, we can stop testing a candidate as soon as we encounter a solution that violates it.)

It has only 2 easy-to-understand parameters A, R (if we don't expect any data errors then set $R = 1$) and uses only integer arithmetic.

For each candidate i we test whether it is violated by each of a random sequence of examples...

- On observing some number A of non-solutions on which it does not hold, accept it as a constraint.

- On observing some number R of solutions in which it does not hold, reject it.

- If neither threshold is reached before the examples are exhausted, reject the candidate.

pseudocode

```
SeqBayesAcq($R$,$A$)
    for each candidate $c$ in the bias
        $r \leftarrow 0$   $a \leftarrow 0$
        repeat
            randomly choose an example $e$ without replacement
                (if impossible then reject $c$ as inconclusive)
            if $c$ is violated in $e$
                if the example is a solution
                    $r \leftarrow r + 1$
                    if $r \geq R$ reject $c$ as a constraint
                else
                    $a \leftarrow a + 1$
                    if $a \geq A$ accept $c$ as a constraint
```

We can prove that SeqBayesAcq is an instance of SPRT: any reasonable choice of A, R ($1 \leq R < A$) corresponds to at least one meaningful choice of SPRT parameters.

Inconclusive candidates

In experiments some candidates were rejected as inconclusive when they should have been learned. This was caused by an insufficient number of violations, even on datasets of several thousand examples. It occurs with candidates that are hard to violate, eg with high arity.

We modify SeqBayesAcq slightly: instead of rejecting all inconclusive candidates, we accept those for which $r = 0$ and $a > 0$ and reject others.

SPRT is often modified to handle inconclusive cases, yielding a **Truncated SPRT** that accepts or rejects them on the basis of a limited number of samples.

Required datasets

Different CA methods require datasets with different characteristics (eg ModelSeeker only needs a few solutions).

SeqBayesAcq works best on datasets that are large (like most other methods) and *balanced* (or nearly so): they have a similar number of solutions and non-solutions.

But does it work? We incorrectly assumed feature independence in NB, discarded inconclusive candidates, and used Wald's approximation: can it possibly be accurate after all this fudging?

Experiments

I'll test BayesAcq and SeqBayesAcq on standard and new benchmarks with mostly default parameter settings. The bias is usually all possible $\{\leq, \neq, \geq\}$ constraints on the variables.

They're implemented in C and executed on a 2.8 GHz Pentium 4. I'll compare times with published results on machines with similar speed: <u>not generally recommended</u> but the differences in speed dwarf any likely differences in machine performance!

9×9 Sudoku

In one paper QuAcq took 2810s, and a time-bounded version called T-QuAcq took 69s.

In another paper QuAcq took approximately 800s and MultiAcq approximately 900s.

MquAca+FindScope 2 max_{B} took 85s and beat 5 other methods.

Conacq took 16s to generate background knowledge and approximately 2s for acquisition.

BayesAcq took 0.4s and SeqBayesAcq 0.05s.

10×10 Latin square

QuAcq took 7200s and T-QuAcq 120s.

In a comparison of 6 methods the fastest was MquAca+ FindScope 2 $\max_{B}$ with 114s.

BayesAcq took 0.6s and SeqBayesAcq 0.06s.

(We used a 20×20 Latin square to further compare the two Bayesian methods: BayesAcq took 19s and SeqBayesAcq 0.3s.)

Golomb rulers

The largest case usually tested is $N = 12$. QuAcq took 11972s and T-QuAcq 1184s.

In another paper QuAcq took 2257s and MultiAcq took 2335s.

BayesAcq took 0.07s and SeqBayesAcq 0.05s.

On a smaller instance ($N = 8$) Conacq took 2193s.

T-QuAcq was also tested on larger Golomb rulers and failed to converge when $N = 20$. But for $N = 27$ both SeqBayesAcq and BayesAcq took 3s (on this dataset all quaternary constraints are "inconclusive").

Bandwidth vertex colouring

A popular benchmark is the RLFAP. With 25 variables & 25 values MultiAcq took 1441s, MAcq-co took 142s, QuAcq took 35s.

Another paper improved QuAcq from 1653 to 151s.

Another paper tested 4 variants of QuAcq on a larger example with 50 variables & 40 values, all taking over 200s.

We use an almost identical but larger problem: *band-width colouring* with 100 variables & 75 values. BayesAcq took 0.24s and SeqBayesAcq 0.023s.

Large random 3-SAT

The benchmarks are too small for a real comparison between SeqBayesAcq and BayesAcq, so we compare them on bigger problems: random 3-SAT with 5 clauses and 1000 examples.

		learning time (seconds)	
V	bias size	BayesAcq	SeqBayesAcq
50	1.6×10^5	1.8	0.02
100	1.3×10^6	16	0.1
150	4.5×10^6	56	0.5
200	1.1×10^7	123	0.9
250	2.1×10^7	243	1.6

Both can handle large biases, and SeqBayesAcq scales better.

Even larger random 3-SAT

1000 variables, 50 clauses, and a bias of 1.3×10^9. BayesAcq took 16259s while SeqBayesSeq took 78s (about 200$\times$ faster).

This further illustrates the improved performance of SeqBayesAcq over BayesAcq.

It also shows that both can handle biases that are much larger than those used in most CA papers (usually at most tens of thousands).

Robust CA

Current CA systems are not **robust** under errors. For systems based on version space learning, if training examples are misclassified they may become inconsistent, causing the version space to collapse. (*Rough version spaces* are designed to be robust but do not seem to have been applied to CA.)

Statistical approaches seem particularly appropriate for noisy data! On the 20×20 Latin square and the 250-variable random 3-SAT example we deliberately misclassified 10% of the examples: SeqBayesAcq learned the correct constraint model for a range of R values.

Conclusion

SeqBayesAcq is an application of sequential analysis to CA. In experiments it learns several examples accurately, is orders of magnitude faster than existing methods, and is the first to handle noisy data sources.

It's amenable to parallelisation: candidates are tested independently, so we could partition the bias into disjoint subsets and test them on (say) a GPU.

In future work I'd like to try using other classifiers for CA, eg based on few-shot learning.

THE END