Generalizing Constraint Models in Constraint Acquisition

Dimos Tsouros
Department of Computer Science, KU Leuven

Steven Prestwich
School of Computer Science and Information Technology, University College Cork

Tias Guns
Department of Computer Science, KU Leuven

Abstract

Constraint Acquisition (CA) aims to widen the use of constraint programming by helping users in the modeling process. However, most CA methods learn one ground CSP: a set of individual constraints for a specific problem instance. We focus on generalizing ground CSPs, by learning parameterized constraint models that can model multiple instances of the same problem. We propose a constraint-level classifier-based approach, where a machine learning classifier is trained to predict for any possible constraint and any possible parameterization of the problem whether the constraint belongs to the problem. A key aspect is an appropriate parameterized feature representation that allows classifiers to learn (implicit) patterns in the ground CSP. The results of our evaluation show that our approach has high accuracy in learning generalized models, and is robust to the presence of noise in the ground CSP.

2012 ACM Subject Classification Computing methodologies → Discrete space search; Computing methodologies → Supervised learning by classification; Theory of computation → Constraint and logic programming

Keywords and phrases Constraint programming, Constraint acquisition, Generalization, Parameterized constraint models

Funding This research received funding from the European Research Council (ERC) under the EU Horizon 2020 research and innovation programme (Grant No 101002802, CHAT-Opt)

1 Introduction

Constraint Programming (CP) is considered one of the main paradigms for solving combinatorial problems in AI, as it provides powerful modeling languages and solvers for constraint optimization and decision-making, with many successful applications in various domains [30, 38]. In CP, the user declaratively states the constraints over a set of decision variables, defining the feasible solutions to their problem, and then a solver is used to solve it. However, modeling a new application as a constraint problem requires expertise, which is a barrier to the wider use of CP [13, 14]. This has motivated the development of methods for assisting the user in the modeling process [11, 13, 16, 19]. This is the focus of the research area of *constraint acquisition (CA)* [8], which has been identified as an important topic [22, 11] and as progress toward the "Holy Grail" of computer science [13].

In Constraint Acquisition (CA), constraints are learned either from known solutions and (optionally) non-solutions [3, 4, 28] or through interaction with a user [5, 34, 35]. Recent advancements on both *passive* and *active* acquisition systems show significant potential, introducing passive CA methods that are robust to noise in the given examples [28, 29], as well as interactive systems that use statistical machine learning (ML) to learn how to learn during the acquisition process [33]. A recent application of interactive acquisition in a real-world scheduling problem was presented in [2].

However, one significant limitation of most (passive and active) CA systems is that they only learn ground CSPs of a fixed size, for one specific problem instance under consideration [1, 7, 8, 9, 20, 23, 25, 26, 28, 36, 37]. Although acquiring individual instances of a problem can be useful, it is common that the instance at hand will often change; because the actual constraint model depends on certain input parameters. For example, in a nurse rostering problem over changing crew compositions, or changing timeframes, the variables and constraints of the ground instances will be different, even if the parameterized constraint model is the same. In practical scenarios, it is common that a given problem is solved several times, often with different parameters [31]. For this reason, well-known constraint modeling languages like MiniZinc [21] allow the use of parameters in the original model, with separate data files setting the parameter values for a given instance. However, an instance-specific ground CSP model learned using CA does not make use of this parameterization capabilities and, hence, cannot be reused for other instances. As a result, *generalizing* the constraint models learned using CA over different instances of the same problem class has been identified as one of the key next challenges for constraint acquisition, and has been identified as a requirement for its wider use [31].

In the literature of constraint acquisition, there are only a few works that support some form of generalization to unseen instances. In interactive CA, approaches for generalizing constraints learned within the given instance were introduced [6, 10], in order to enhance the acquisition process. Although such generalizations are targeted to within-instance generalization, the found generalizations could also be used across instances.

Another approach, called *extrapolation*, was recently explored [27], in which ground constraints are first learned for different instances of the problem, and then they are extrapolated to a given instance with different parameters. An advantage of this approach is that the ground constraints can be learned using any CA method. However, it requires learning the ground constraints for a number of instances of the problem to be able to generalize. In addition, it relies on a symbolic classifier based on genetic programming, which is computationally intensive and tends to learn complex, non-interpretable expressions.

A third approach is to learn interpretable *parameterized* forms of constraints during the acquisition process, given examples from different instances of the problem along with the instance parameters [17, 18]. In [18] an inductive logic programming approach is used to learn rules of the form *condition* $\Rightarrow$ *constraint* from the examples. On the other hand, COUNT-CP [17] first learns instance-specific constraints, and then these constraints are grouped, to obtain first-order constraints that can generalize to unseen instances. However, only pre-defined partitions are considered to group constraints, and thus it is not able to capture other patterns that may exist in the CP model. Finally, if the learned constraints are not completely correct, e.g. because of noisy labels in the examples, then these methods will fail to generalize.

In this paper, we propose an ML-based generalization of constraint models that is robust to noise, using constraint-level classification utilizing any standard ML classifier. More specifically, we make use of the capability of ML to learn complex functions from labelled data. We demonstrate that through a careful feature representation, classifiers can learn to distinguish which constraint is part of the ground CSP of any target instance.

In more detail, our contributions are the following:

- We propose a *classification-based approach* to generalize the set of constraints of the problem, and we show how a generate-and-test mechanism can be used to extract the set of constraints for any target problem, independent of the type of classifier used.
- To be able to generalize across different parametrizations, we propose a parameterized feature representation of constraints, considering both relation-based and scope-based constraint features.
- We make a comprehensive experimental evaluation of the proposed approach, using different classifiers, on different problem classes and parameterizations. We also show that our generalization approach is robust to different types of noise in the ground CSPs.

2 Background

In this section, we introduce the necessary concepts used in the paper.

Constraint Satisfaction Problems

A *constraint satisfaction problem* (*CSP*) is a triple $P = (V, D, C)$, consisting of:

- a set of n decision variables $V = \{v_1, v_2, ..., v_n\}$, representing the entities of the problem,
- a set of n domains $D = \{D_{v_1}, D_{v_2}, ..., D_{v_n}\}$, with $D_{v_i} \subset \mathbb{Z}$ being the domain of $v_i \in V$,
- a constraint set (also called constraint network) $C = \{c_1, c_2, ..., c_t\}$.

A *constraint* c is a pair $(rel(c), var(c))$, where $var(c) \subseteq V$ is the *scope* of the constraint, and $rel(c)$ is a relation over the domains of the variables in $var(c)$, that restricts their allowed value assignments. $|var(c)|$ is called the *arity* of the constraint. $const(c)$ is the set of constant values present in the constraint c. The set of solutions of a constraint set C is denoted by $sol(C)$.

Constraint Acquisition

In Constraint Acquisition (CA), the pair (V, D) is called the *vocabulary* of the problem at hand and is common knowledge shared by the user and the system. Besides the vocabulary, the learner is also given a *language* Γ consisting of a broad range of *fixed arity* constraint relations that may exist in the problem at hand. Using the vocabulary (V, D) and the constraint language Γ, the system generates the *constraint bias* B, which is the set of all expressions that are candidate constraints for the problem.

The (unknown) target constraint set C_T is a constraint set such that for every example e it holds that $e \in sol(C_T)$ iff e is a solution to the problem the user has in mind. The goal of CA is to learn a constraint set C_L that is equivalent to the unknown target constraint set C_T.

Machine Learning Classification

ML classification is a supervised learning task that involves learning a function over a given dataset. The dataset, denoted as $\mathbf{E}$, is a collection of N training examples, $\mathbf{E} = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$. Each training example is a pair (x_i, y_i), where x_i is a feature vector from the input space X and y_i is the corresponding class label from the output space Y. The feature vector $\mathbf{x}_i$ is composed of m features, $\mathbf{x}_i = (\phi_{i1}, \phi_{i2}, \ldots, \phi_{im})$, with each feature ϕ_{ij} being a quantifiable property or characteristic of training example i. In the case of (multi-class) classification, Y contains discrete values. An ML classifier aims to learn a function $f_\theta : X \rightarrow Y$, parameterized by a set of learnable parameters θ. These parameters are adjusted during the training process to minimize a loss function $L(f_\theta(x), y)$ measuring the error between the predicted and actual class label.

Rule-Based Classifiers

Rule-based classifiers represent a distinct category of machine learning classifiers, learning a function $f_\theta : X \rightarrow Y$ that is represented with if-then rules, denoted as an ordered set $R = \{r_1, r_2, ..., r_k\}$. Each rule r_i is a pair (Q_i, y_i), with Q_i being a set of conditions and y_i a class label. Each condition $q_{ij} \in Q_i$ is a function $q_{ij} : X \rightarrow \{0, 1\}$ that maps an example x to a binary value indicating whether the given example satisfies the condition. For a rule to be satisfied, all of its conditions need to be satisfied, i.e., $q_{ij}(x) = 1 \mid \forall q_{ij} \in Q_i$. When a new example x is introduced, the rules are sequentially evaluated, and the class label y_i corresponding to the first rule r_i that is satisfied is assigned as the prediction. When no rule is applicable, a default class $y_{default}$ is assigned. Depending on the algorithm used, the set of rules R might also be unordered.

3 Problem Definition

Constraint problems often depend on a set of input parameters $\mathcal{P}$, with their values defining the specific instantiation of the ground-CSP of a given instance. We showcase this with the following example, which we will also use as a running example throughout the paper.

▶ **Example 1** (Running example)**.** We consider a simplified exam timetabling problem, where we have s semesters, with cps courses per semester, and we need to schedule the examination of the courses on d available days, with each day having t timeslots. These are the (named) parameters of the problem, i.e., $\mathcal{P} = \{s, cps, d, t\}$. All courses must be scheduled in a different timeslot, while courses from the same semester must be scheduled on different days.

Different values for the given parameters will lead to instances of varying size. In this problem, the number of courses per semester cps and the number of semesters s are used to model the variables of the problem V, while the available days d and timeslots per day t are important to define the domains D of the variables. The parameters are also important to model the set of constraints C. Besides the `all_different` constraint between all the variables, the problem has the `different_day` constraints, which are defined over partitions of the variables that model courses occurring in the same semester. In addition, the `different_day` constraints need the number of timeslots t per day to determine when two courses are scheduled on the same day.

We now formally define the CSP instance generalization problem in the context of constraint acquisition. To do that, first we define the concept of *problem-instance*.

▶ **Definition 1** (Parameterized problem-instances)**.** *Consider a parameterized constraint problem characterized by a set of named parameters $\mathcal{P} = \{p_1, p_2, \ldots, p_q\}$. A problem-instance A is a specific occurrence of the problem defined by a distinct set of parameter values $\mathcal{P}_A = \{(p_1, u_{A1}), (p_2, u_{A2}), \ldots, (p_q, u_{Aq})\}$, with the tuple (p_i, u_{Ai}) containing the named parameter and its value in instance A. This instance is modeled as a CSP using a set of variables V_A, with their respective domains D_A, and a set of constraints C_A for this (V_A, D_A). We call the tuple (V_A, D_A, C_A) a ground CSP. The variables, domains, and constraints of this CSP are determined based on the values of the parameters of the problem. Hence, the problem-instance A is a tuple $(\mathcal{P}_A, V_A, D_A, C_A)$.*

We can now define the problem of generalizing constraint models from given problem-instances.

▶ **Definition 2** (Generalizing Constraint Models)**.** *Given one or more problem-instances described by a tuple $(\mathcal{P}_A, V_A, D_A, C_A)$, the objective of generalization is to construct a function F such that, for any target problem-instance with a vocabulary (V_T, D_T), defined by a unique set of parameter values $\mathcal{P}_T = \{(p_1, u_{T1}), (p_2, u_{T2}), \ldots, (p_q, u_{Tq})\}$, that $F(\mathcal{P}_T, V_T, D_T)$ will return the corresponding set of constraints C_T. That is, the aim is to extrapolate the given ground CSPs to a target instance with parameters $\mathcal{P}_T$, accurately determining the set of constraints C_T without any (additional) interaction with the user.*

4 Generalizing Constraint Models

In order to generalize the set of constraints of the problem, we propose a constraint-level classification approach, named GenCon. We build upon the method from [33], which shows how ML classifiers can detect implicit patterns within the learned constraint network during CA. However, [33] employs ML classifiers to obtain a probabilistic estimate for each constraint, estimating how likely it is to belong to the target problem or not. This is done in order to guide interactive CA for a single problem-instance, with fixed parameters. On the other hand, we are interested in generalizing what

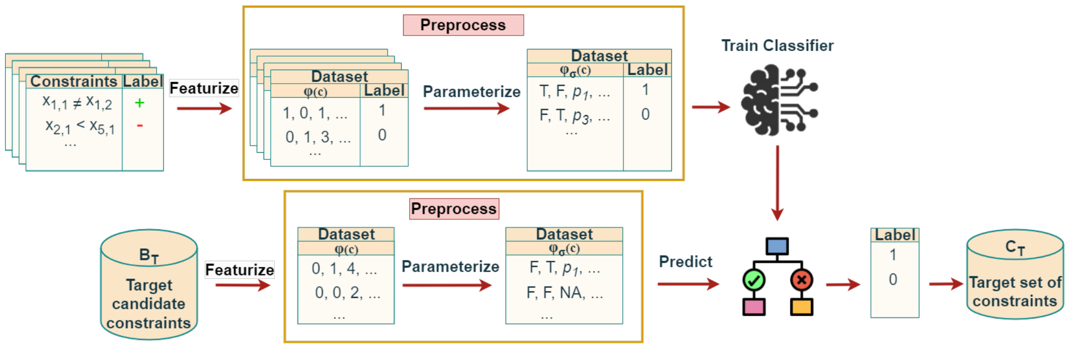


Figure 1 GenCon: Generalizing constraint models through constraint classification, using a parameterized feature representation of constraints

the classifier learns for any given instance of the problem. Hence, we extend this paradigm, proposing a generic (parameterized) feature representation of constraints.

Our framework for generalizing is shown in Figure 1. The given set of ground constraints in the input instance(s) are used in order to *train a classifier*, learning a function that can predict for any constraint whether it belongs to the set of constraints of any target instance of the problem. For this, we use a (parameterized) feature representation of constraints. In a given target instance, a generate-and-test approach is then used: a set of candidate constraints is generated using the vocabulary (V_T, D_T) and the parameters P_T of the target instance provided by the user. The classifier then predicts the label of (a feature encoding of) every constraint, extracting those classified as *true*.

4.1 Feature representation of constraints

An ML classifier expects a fixed-size list of features x as input. Hence, in order to use constraint-level classifiers, we have to devise a feature representation $\phi(c)$ applicable to any constraint c. As shown in the middle of Figure 1, this feature representation $\phi(c)$ is then transformed to a parameterized version, notated as ϕ_σ. We now explain this in more detail.

Feature representation

For our feature representation $\phi(c)$, we use information regarding the constraints' relation and its ordered list of arguments (i.e., variables and constants). We construct a feature representation with 3 feature groups. An overview can be seen in Table 1.

Basic constraint features We first use some basic features that capture general characteristic of each constraint. The first feature represents the constraint relation, such as equality, inequality, or a specific function name. The second feature indicates whether at least one constant value is present in the constraint. We also encode features for the numeric values of the constants. As the list of features in ML needs to be of fixed size, we encode as many features for the constants as the largest number of constants in the input set of constraints, using NaN values when a constraint has fewer constants.

Index-of-variables features In many cases, the given variables V are organized in the form of matrices or tensors. The dimensions of such a tensor, and the index of each variable in them, often play a crucial role in the modeling of the problem. For example, a constraint may be defined on variables in the same row or column of a matrix. Thus, in the feature representation of the constraints, we include information about the indices of its variables. We include as many index dimensions as the largest number of dimensions over all variables, again using NaN values if the variables in a constraint have fewer dimensions. For each relevant dimension i, the first feature captures if the index

Table 1 Feature representation of constraints

ID	Name	Description
Basic constraint features		
1	Relation	Constraint relation
2	Has_constant	If a constant value is present
3	Constant(s)	The constant value(s)
Index-of-variables features		
4	dim$_i$_same	If the index in dimension i is the same in all variables
5	dim$_i$_avg_dist	Average difference of the indices in dimension i among variables
Latent dimension features		
6	dim$_i$_ldim$_j$_same	If the index in latent dimension j of a dimension i is the same in all variables
7	dim$_i$_ldim$_j$_avg_dist	Average difference of the indices in dimension's i latent dimension j among variables

in dimension i is the same in all variables, while the second encodes the average difference of the indices in dimension i among variables.

Latent dimension features Besides using features over the indices of the variables, we also aim to identify "latent dimensions". These may not be explicitly present in the given matrix or tensor of the variables, but can be indirectly derived from the problem parameters $\mathcal{P}$. This is inspired by the `scheme` sequence generator of Modelseeker [3]. Focusing again on Example 1, if the variables were not provided in a 2-D matrix but in a vector of size $c \times s$, then the Index-of-variables features will not be able to capture the `different_day` constraints. However, we can consider c and s as latent dimensions, incorporating them into the feature representation of the constraints. To generate these latent dimensions, we examine possible divisors of the variables' tensor dimension sizes. More specifically, we consider each named parameter of the problem as a possible divisor.We add as many latent dimensions as there are valid divisors for the different dimensions in the input problem instances. For each latent dimension, we use the same two features as used for the known dimensions.

▶ **Example 2.** Recall the exam timetabling problem from Example 1. The variables representing the courses can be structured in a 2-D matrix with s rows and cps columns. In this problem, the index of each variable in each dimension is important for the constraints, e.g. the constraint defining that courses from the same semester must be scheduled on different days means that it applies to variables in the same row. For this example, assume we have an instance of this problem with named parameters $\mathcal{P} = \{(s, 4), (cps, 6), (d, 10), (t, 3)\}$.

The feature representation of each constraint in this problem will be of size 9: the 3 basic constraint features, 4 index-of-variable features, i.e. for the 2 dimensions, and 1 latent dimension with 2 features.The latent dimension is the division of the second dimension ($cps = 6$) by parameter $t = 3$. It splits the cps columns in two separate blocks; this was the only possible dimension divisor. The constraint c = `different_day(courses[2][1], courses[2][4])` will have the following feature representation:

$$\phi(c_i) = [\text{“}\texttt{different\_day}\text{”}, False, NaN, True, 0, False, 3, False, 1]$$

Let us now assume that the variables in the exam timetabling problem were not provided in a 2-D matrix, but in a vector of size $c \times s$, i.e. in a vector of size 24 in the current instance. In this case, the Index-of-variables features will not be able to capture the `different_day` constraints. However, our approach uses the parameters c and s to create latent dimensions, which would effectively reconstruct the rows of the matrix representation. This latent dimension will then be able to capture the `different_day` constraints in them.

The feature representation of each constraint in this case will be of size 11: the 3 basic constraint features, 2 index-of-variable features for the 1 dimension of the vector, and 6 latent dimension features for 3 possible latent dimensions. The latent dimensions divide the vector size by respectively the named parameters $s = 4$, $cps = 6$, and $t = 3$. The constraint $c =$ `different_day(courses[7], courses[16])` then has the following feature values:

$$\phi(c_i) = [\text{“}\texttt{different\_day}\text{”}, False, NaN, False, 9, False, 1, True, 0, False, 1]$$

From numeric features to categorical features over the named parameters

As the goal is to generalize beyond a single fixed problem-instance, the feature representation of the constraints should capture the characteristics of the constraints in a generic, parameterized way. Numeric features will typically depend on parameters of the problem and not be static, and that is what we want the classifier to capture. In this step, we will hence replace the numeric features, ID 3 (constants), and 5 and 7 (average distance), by categorical features over the named parameters.

▶ **Example 3.** Recall the instance of the exam timetabling problem from Example 2. The `different_day` constraints actually have to use the value of the timeslots parameter t to find the day of each course, as the domain values of each course are the available timeslots for all days d, i.e. $D_i = \{1, ..., t*d\}$. Thus, in this case we have a constant involved, associated with parameter t. Hence, the feature representation of $c =$ `different_day(courses[2][1], courses[2][4], 3)` will involve a constant, in this case 3, and will be:

$$\phi(c_i) = [\text{“}\texttt{different\_day}\text{”}, True, 3, True, 0, False, 3, False, 1]$$

The classifier would then recognize that for constant values equal to 3, such a constraint is part of the problem. However, this is not true for all parameterizations of the problem, e.g. if the target instance has a number of timeslots $t = 5$.

To capture that, and allow our feature representation to adapt to different problem-instances, we transform all numeric features to categorical ones. There are $|\mathcal{P}| + 1$ categories: one for every named parameter, plus a NaN value in case none of the parameters match. For this, we use a numeric-to-categorical parameter mapping function $\sigma : \mathbb{R} \to \{\text{“NaN”}\} \cup \{p_i \in \mathcal{P}\}$. The function is defined as $\sigma(v, \mathcal{P}) = p_i$ if the value v corresponds to the value u_i of the named parameter p_i in the given instance(s), and $\sigma(v) =$ “NaN” otherwise.

$$\sigma(v, \mathcal{P}) = \begin{cases} p_i & v = u_i \mid (p_i, u_i) \in \mathcal{P} \\ \text{“NaN”} & \text{otherwise.} \end{cases} \tag{1}$$

Note that, in a constraint model sometimes trivial adaptations of the parameters are used, e.g. $p - 1$, $p + 1$ or the multiplication of two parameters. In order to capture such adaptations, we extend the set of parameters $\mathcal{P}$ with such trivial adaptations, along with the basic constants 0,1 as in COUNT-CP [17].

This mapping function is applied to all numeric features, replacing them by a categorical value. So $\phi(c_i)$ becomes the parameterized

$$\phi_\sigma(c_i) = [\text{“}\texttt{different\_day}\text{”}, True, ”t”, True, ”zero”, False, ”t”, False, ”one”]$$

4.2 Training a classifier for constraint classification

In order to exploit the learning capabilities of ML classifiers for this task, we need to work with a dataset at the constraint-level, i.e., such that that its examples correspond to individual constraints.

Given a set of constraints C_A for a problem-instance A, and a distinct set of constraints C_A^-, consisting of constraints that are not part of the model, we define a dataset $\mathbf{E}$, with each training example being represented as a tuple $(\mathbf{x}_i, y_i)$, corresponding to a constraint $c_i \in C$. For each training example $(\mathbf{x}_i, y_i)$, we have $\mathbf{x}_i = \phi_\sigma(c_i, \mathcal{P})$, i.e., a parameterized feature representation of constraint c_i, and $y_i = [c_i \in C_A]$, i.e., a Boolean label that indicates whether c_i is part of the set of constraints or not.

$$\mathbf{E} = \{(\mathbf{x}_i, y_i) \mid \mathbf{x}_i = \phi_\sigma(c_i, \mathcal{P}) \wedge y_i = [c_i \in C_A], \forall\, c_i \in \{C_A \cup C_A^-\}\}, \tag{2}$$

Note that, when parameterizing the feature representation of a constraint, a single numeric value in a numeric feature can be mapped to multiple parameters. When this occurs, additional examples are included in the dataset for each feature representation corresponding to the different matching parameters. Although this can add noise in the dataset, due to the examples with the wrong parameter, it ensures that the correct feature representations will be included.

Given C_A and C_A^-, the top part of our framework from Figure 1 shows how they can be used to train a classifier, learning a function f_θ that can identify constraints that are present in a given instance of the problem. First, the dataset $\mathbf{E}$ is constructed, by using a feature representation for the constraints with their corresponding label. The next step of our approach is to use the parameters of the problem to transform the feature representation into a parameterized version, replacing numeric values in the features with the named parameter corresponding to them. Hence, the classifier trained using the new dataset with the parameterized feature representation, will learn a function $f_\theta : X \rightarrow Y$ that will not depend on the problem-instance given, as the input space X represents the constraints w.r.t. the problem class. Any detected patterns present in the instance at hand will be generalizable based on the parameters of each instance.

For each problem instance, we have as input a set of constraints C_A, and we know that all constraints $c \in C_A$ will have a positive label. However, we also need a set of constraints C_A^- consisting of constraints with a negative label, to let the classifier learn how to distinguish between classes. If we do not have such a set of constraints, e.g. based on the set of candidate constraints B_A used in the constraint acquisition system utilized to learn C_A, we can generate a B_A for the given instance based on the learned set of constraints. That is, we can generate a set of constraints B_A using the set of relations detected in the given set of constraints C_A as a language Γ.

$$\Gamma = \{rel(c) \mid c \in C_A\} \tag{3}$$

The bias B_A is created applying the relations in Γ to all combinations of variables in V_A. The set of constraints that will be used as the negative examples of the dataset, is defined as $C_A^- = B_A \setminus C_A$.

4.3 Generate-and-test extraction of the target model

Given a function f_θ learned using any learning paradigm, we can use it to classify new constraints for any given instance of the problem. We now describe a *generate-and-test* approach for the extraction of the target set of constraints. In this approach, a set of candidate constraints B_T for the target problem T is generated, and the learned function f_θ is used to identify which constraints are true for the given instance. This procedure is shown in the bottom part of Figure 1. For the generation of the set of candidate constraints B_T for the target instance T, the system uses the language Γ as described above. For relations including constant values, the set $\mathcal{P}$ is used as the candidate constant values. The constraints in B_T are taken one by one, and are featurized based on the same feature representation used as in the training process, transformed to the parameterized version. The function f_θ learned using the classifier is then utilized to each one, predicting if the corresponding constraint is part of the target constraint model. The constraints that have a true predicted label are added to our constraint model:

$$C_T = \{c \mid c \in B \wedge f_\theta(\phi_\sigma(c, \mathcal{P})) = True\} \tag{4}$$

5 Experimental Evaluation

We now experimentally evaluate our proposed approach for generalizing constraint models. We evaluate our method in 4 benchmarks with a variety of patterns that need to be generalized. The evaluation takes place on different problem instances with their ground CSPs obtained using interactive CA.

However, as the input sets of constraints for the given problem instances can be learned through CA systems, they can be noisy. This can occur if the training examples used in a passive CA system are too few, or if there are errors in the labels. Hence we also evaluate the impact of noise in its performance. We recognise two different types of noise in our setting:

1. False positive (FP) noise: The input set of constraints is not sound, i.e., it also includes wrong constraints.
2. False negative (FN) noise: The input set of constraints is not complete, i.e. it does not include all constraints of the problem.

Based on these definitions, we aim to answer the following research questions:

(Q1) To what extent does our proposed method effectively generalize ground CSPs?

(Q2) What is the impact of FP noise, i.e., when the input set of constraints includes also wrong constraints, in the performance of our method?

(Q3) What is the impact of FN noise, i.e., when the input set of constraints does not include all correct constraints, in the performance of our method?

5.1 Experimental setup

We now discuss the details of our experimental setup for answering the experiment questions.

Comparison and evaluation metrics

In our proposed method any classifier can be utilized. Thus, in our experiments, we compared a variety of classifiers. Specifically, we used the following: Decision Trees (DT), Random Forests (RF), Naive Bayes (NB), Multi-layer Perceptron (MLP), K-Nearest Neighbours (KNN) and CN2. We used CN2, DT, RF, and NB in their default settings, while we tuned the most important hyperparameters for MLP and KNN.[1]

We also compare our approach with the generalization approach followed by Count-CP [17]. Count-CP consists of two steps: learning a ground-CSP for a specific problem-instance and generalizing by learning first-order constraints. In the available implementation of Count-CP[2] the two steps are mixed, so we re-implemented its generalization method.

We evaluate each method in terms of how accurate the predicted models are. This is done through assessing how many of the constraints found are correct, and how many constraints are missing from the model. We define as True Positives (TP) the correctly identified constraints, as False Positives (FP) the incorrectly identified constraints, and as False Negatives (FN) the missed constraints. Our evaluation is based on the following metrics that are common in ML:

Precision (Pr): Precision is a measure of the accuracy of the identified constraints in the target instance. A high precision score signifies a low rate of false positive errors, indicating that when the method identifies a constraint, it is likely to be correct. When the precision score is 100%, the predicted set of constraints is sound. It is calculated as $Pr = TP/(TP + FP)$.

Recall (Re): Recall is a measure of the method's ability to identify all relevant constraints. A high recall score indicates a low rate of false negative errors, indicating that the method is effective

[1] Tuning details can be found in the appendix

[2] https://github.com/ML-KULeuven/COUNT-CP

in identifying all relevant constraints and has a low rate of false negatives. When the recall score is 100%, the predicted set of constraints is complete. It is calculated as $Re = TP/(TP + FN)$.

Benchmarks

In our experimental evaluation, we used well-known benchmarks commonly used in CA, consisting of different patterns on their constraints. In each benchmark we used 3 instances of different sizes, evaluating the performance of the compared methods to generalize from the smaller to the larger ones. This means that we evaluate 3 different generalizations for each benchmark, and we present the average amongst them. We used the following benchmarks:

Sudoku The Sudoku puzzle is a $n \times n$ grid, which must be completed in such a way that all the rows, columns, and n non-overlapping $b \times b$ (with $n = b^2$) squares contain distinct numbers. The variables are the grid cells, arranged in a 2D matrix with domains the different values they can take. We considered as parameter the block size and the grid size, i.e., $\mathcal{P} = \{b, n\}$. We used instances with parameters $\mathcal{P}_1 = \{b = 3, n = 9\}$, $\mathcal{P}_2 = \{b = 4, n = 16\}$, and $\mathcal{P}_3 = \{b = 5, n = 25\}$.

Golomb rulers. The problem is to find a ruler with m marks, where the distance between any two marks is different from that between any other two marks. The variables are the marks, arranged in a vector. We considered as parameter the amount of marks, i.e., $\mathcal{P} = \{m\}$. We used instances with $\mathcal{P}_1 = \{m = 8\}$, $\mathcal{P}_2 = \{m = 12\}$, and $\mathcal{P}_3 = \{m = 14\}$.

Exam Timetabling There are s semesters, each containing $cps($ courses, and we want to schedule the exams of the courses in a period of d days, On each day we have t available timeslots. The variables are the courses ($|V| = ns \cdot cps$), arranged in a 2D matrix, having as domains the timeslots they can be assigned on. The constraints define that all courses need to be scheduled in a different timeslot, while exams of courses from the same semester must be scheduled on different days. The latter is expressed by constraints of the type $\lfloor v_1/spd \rfloor \neq \lfloor v_2/spd \rfloor$. The parameters of the problem are $\mathcal{P} = \{s, cps, d, t\}$. We used instances with parameters $\mathcal{P}_1 = \{s = 4, cps = 5, d = 8, t = 6\}$, $\mathcal{P}_2 = \{s = 8, cps = 6, d = 10, t = 9\}$, and $\mathcal{P}_3 = \{s = 10, cps = 6, d = 15, t = 15\}$.

Nurse rostering There are n nurses, s shifts per day, ns nurses per shift, and d days. The goal is to create a schedule, assigning a nurse to all existing shifts. The variables are the shifts, and there are a total of $d \cdot s \cdot ns$ shifts. The variables are modeled in a 3D matrix. The domains of the variables are the nurses. Each shift in a day must be assigned to a different nurse and the last shift of a day must be assigned to a different nurse than the first shift of the next day. The parameters of the problem are $\mathcal{P} = \{n, s, ns, d\}$. We used instances with parameters $\mathcal{P}_1 = \{n = 7, s = 3, ns = 3, d = 15\}$, $\mathcal{P}_2 = \{n = 14, s = 3, ns = 5, d = 18\}$, and $\mathcal{P}_3 = \{n = 30, s = 3, ns = 6, d = 20\}$.

Implementation and hardware details

- All the experiments were conducted on a system carrying an Intel(R) Core(TM) i7-2600 CPU, 3.40GHz clock speed, with 16 GB of RAM.
- All methods and benchmarks were implemented[3] in Python. We used the CPMpy constraint programming and modeling library [15] for constraint modeling. The implementation of the ML classifiers was carried out using the Scikit-Learn library [24], except CN2, for which the Orange library [12] was used. The GrowAcq interactive CA algorithm [32] was used to extract the sets of constraints for the given benchmark instances.
- The results presented in each benchmark, for each method, are the means of the results for all problem-instances, where each method was run 10 times for each problem-instance.

[3] Our code will be available online upon acceptance

5.2 Results

Q1: To what extent does our proposed method effectively generalize ground-CSPs?

Table 2 shows the results comparing different classifiers in our method, and the Count-CP generalization approach. As we can see, the DT, CN2, RF and MLP classifiers achieve 100% precision and recall in all benchmarks. The classifiers presenting inferior performance are NB and KNN. NB presents worse performance in Nurse and Exam benchmarks. In Exam, although all the constraints found were correct, not all of them were found, as NB fails to detect the `different_day` constraints on courses of the same semester. That is because it failed to recognise both the different partitioning, the relation used and the constant parameter. On the other hand, in Nurse it missed some constraints (lower recall) but also some of the predicted ones were false positives. That is because Nurse contains constraints in a pattern that consist of combinations of partitions and sequence conditions.Due to the feature independency assumption of NB, it struggles to capture such patterns. Nurse was also the benchmark on which KNN presented bad results, in both precision and recall, because of the same pattern of constraints.

Table 2 Results with comparing our method (using different classifiers) with the Count-CP generalization

Classifier	Nurse		Exam		Golomb		Sudoku	
	Pr	Re	Pr	Re	Pr	Re	Pr	Re
NB	76%	84.6%	100%	91.6%	100%	100%	100%	100%
{DT, CN2, RF, MLP}	100%	100%	100%	100%	100%	100%	100%	100%
KNN	60%	84%	100%	100%	100%	100%	100%	100%
Count-CP	100%	81.6%	100%	100%	100%	100%	100%	100% (75.6%)

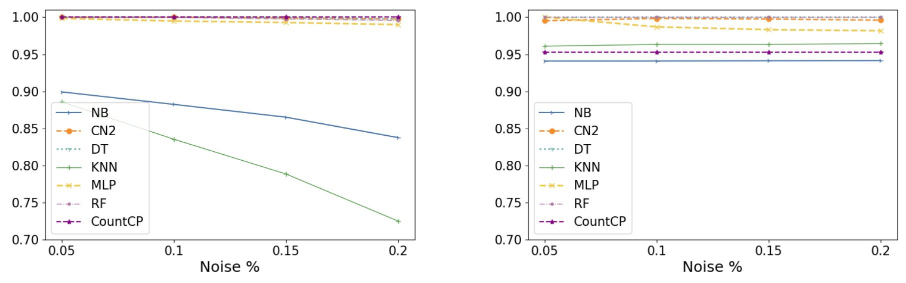
Regarding COUNT-CP, it presents very good performance on all benchmarks except Nurse, where again the pattern regarding the consecutive shifts cannot be captured. That is because it does not include sequence conditions in its generalization approach, and thus it is only searching for patterns applying in all sequences of predefined partitions. In addition, in Sudoku, we manually gave the custom partitions of the blocks as input, and thus it is able to generalize such patterns. However, as shown in brackets, the method presents lower recall results when the custom partitions for the blocks are not explicitly given by the user.

The rest of the classifiers learn all the correct constraints in all benchmarks, presenting 100% scores in both recall and precision. This shows the generalization capabilities of our approach, on the different patterns on constraints that are present in the benchmarks, some of which are complex. Notably, the results for each benchmark represent the average of results obtained using different instances as input and target models. This demonstrates that GENCON is capable of generalizing from smaller inputs to significantly larger target instances.

Q2: What is the impact of FP noise in the performance of our method?

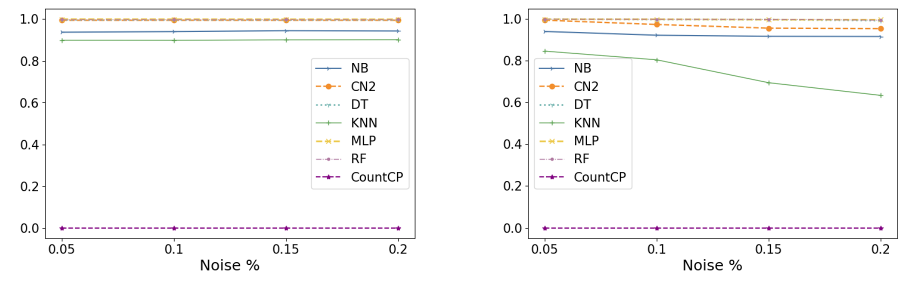
To answer this question, we altered the ground CSPs for each input instance, injecting additional (wrong) constraints in it. We evaluated our method on 4 levels of noise ($\{5\%, 10\%, 15\%, 20\%\}$) by randomly adding the respective percentage of constraints from the set of candidate constraints C_A^- in the input set of constraints C_A. Due to space limitations, we present the average results over all benchmarks for each method.[4] The results are shown in Figure 2.

[4] Detailed results per benchmark can be found in the appendix



(a) Precision **(b)** Recall

Figure 2 Results with the presence of FP noise in the input constraint model, i.e., having additional (wrong) constraints in the input ground CSPs



(a) Precision **(b)** Recall

Figure 3 Results with the presence of FN noise in the input constraint model, with correct constraints missing in the input ground CSPs

We can observe that the KNN classifier is the most sensitive to the noise, presenting an increasingly worse average precision score while noise increases, down to 72% in the case of 20% of noise, while its recall score stays high. This means that although it predicts correctly most of the constraints of the target instances, as in the results with no noise, it also predicts wrong constraints as part of the target model. A smaller, but still significant, decrease in precision is also shown with the NB classifier, with its recall staying at the same level as in the initial results without noise. When any of the other classifiers are used in GenCon, the results stay very high, close to 100% for most classifiers, even when the noise percentage reaches 20%.

Similarly, the COUNT-CP generalization approach keeps the same performance as in the original results without noise. That is because it only searches for specific partition patterns and symbolic expression bounds, and the randomly inserted constraints do not follow any of the searched patterns, and thus are directly disregarded. Its average recall in the no-noise case is around 95% though, which is worse than all classifiers except NB.

Q3: What is the impact of FN noise on the performance of our method?

To answer this question, we need to change the ground CSPs for each instance when it is given as input, so that it does not include all the constraints, i.e. some constraints are missing. We evaluated our method on 4 different levels of noise ($5\%, 10\%, 15\%, 20\%$) by randomly removing the respective percentage of constraints from the input set of constraints C_A. As in Q2, we present the average results over all benchmarks for each method. The results are shown in Figure 3.

We can observe that the most sensitive to this type of noise is again the KNN classifier, this time presenting worse recall results. This means that it cannot find all the constraints of the target instance, due to not having all of the correct constraints in the input instance. When any of the other classifiers is utilized in GenCon, the results stay very high: close to 100% for most classifiers even for up to 20% noise, except NB whose recall is lower in the initial results without noise, but does not reduce further when noise is injected. The results demonstrate the ability of our classification-based approach to generalize even in the presence of high percentages of noise.

On the other hand, the recent COUNT-CP generalization approach completely fails to detect any patterns and does not predict any constraints in the target instance, as it needs to find an exact partition in which all sequences of variables share a given constraint. When some constraints are randomly removed from the set of constraints, e.g. if they are not learned by a CA system due to noise in the labeling of the solutions and non-solutions given, then the COUNT-CP generalization approach fails to detect any patterns, even in the presence of only 5% of noise.

6 Conclusions

CP problems are typically expressed as parameterized problems. However, most CA methods learn one ground CSP for a specific problem instance. We focused on alleviating this limitation by generalizing ground CSPs to parameterized constraint models that can model different instances of the same problem. We proposed a constraint-level classification paradigm, where any standard ML classifier can be used. We also introduced a parameterized feature representation to let the classifiers learn which constraints are part of the ground CSP of any target instance.

Our evaluation shows that this approach presents high accuracy and is robust even in the presence of varying amounts of noise. Importantly, it can generalize well both when the input includes wrong constraints, and when it does not include all the correct ones. This shows significant potential for using ML-based techniques for generalizing constraint models, given a carefully constructed feature representation of constraints. Our proposed one can capture many common patterns in CP. However, for problems containing more complex patterns, the feature representation may need to be extended, to be able to capture more complex conditions.

Future work can look into representation learning techniques that could reduce the dependency on manually crafted features. Another promising avenue is to investigate exploiting generalization during the learning process of CA algorithms. This could reduce the amount of queries in interactive CA, and in enhance their robustness, given the noise robustness shown in this paper.

References

1 R. Arcangioli, C. Bessiere, and N. Lazaar. Multiple constraint acquisition. In *25th International Joint Conference on Artificial Intelligence*, 2016.

2 Hugo Barral, Mohamed Gaha, Amira Dems, Alain Côté, Franklin Nguewouo, and Quentin Cappart. Acquiring constraints for a non-linear transmission maintenance scheduling problem. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer, 2024.

3 Nicolas Beldiceanu and Helmut Simonis. A model seeker: Extracting global constraint models from positive examples. In *Principles and practice of constraint programming*, pages 141–157. Springer, 2012.

4 Senne Berden, Mohit Kumar, Samuel Kolb, and Tias Guns. Learning max-sat models from examples using genetic algorithms and knowledge compilation. In *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, 2022.

5 Christian Bessiere, Clement Carbonnel, Anton Dries, Emmanuel Hebrard, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, Kostas Stergiou, Dimosthenis C Tsouros, and Toby Walsh. Learning constraints through partial queries. *Artificial Intelligence*, 319:103896, 2023.

6 Christian Bessiere, Remi Coletta, Abderrazak Daoudi, Nadjib Lazaar, Younes Mechqrane, and El-Houssine Bouyakhf. Boosting constraint acquisition via generalization queries. In *ECAI*, pages 99–104, 2014.

7 Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, Toby Walsh, et al. Constraint acquisition via partial queries. In *IJCAI*, volume 13, pages 475–481, 2013.

8 Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. Constraint acquisition. *Artificial Intelligence*, 244:315–342, 2017.

9 A. Bonfietti, M. Lombardi, and M. Milano. Embedding decision trees and random forests in constraint programming. In *12th International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Lecture Notes in Computer Science vol. 9075*, pages 74–90, 2015.

10 Abderrazak Daoudi, Nadjib Lazaar, Younes Mechqrane, Christian Bessiere, and El Houssine Bouyakhf. Detecting types of variables for generalization in constraint acquisition. In *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 413–420. IEEE, 2015.

11 Luc De Raedt, Andrea Passerini, and Stefano Teso. Learning constraints from examples. In *Proceedings in Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

12 Janez Demšar, Tomaž Curk, Aleš Erjavec, Črt Gorup, Tomaž Hočevar, Mitar Milutinovič, Martin Možina, Matija Polajnar, Marko Toplak, Anže Starič, et al. Orange: data mining toolbox in python. *the Journal of machine Learning research*, 14(1):2349–2353, 2013.

13 Eugene C Freuder. Progress towards the holy grail. *Constraints*, 23(2):158–171, 2018.

14 Eugene C Freuder and Barry O'Sullivan. Grand challenges for constraint programming. *Constraints*, 19(2):150–162, 2014.

15 Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

16 Samuel M Kolb. Learning constraints and optimization criteria. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

17 Mohit Kumar, Samuel Kolb, and Tias Guns. Learning constraint programming models from data using generate-and-aggregate. In *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

18 A. Lallouet, M. Lopez, L. Martin, and C. Vrain. On learning constraint problems. In *Proceedings of the IEEE International Conference on Tools With Artificial Intelligence*, page 45–52, 2010.

19 Michele Lombardi and Michela Milano. Boosting combinatorial problem modeling with machine learning. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 5472–5478, 2018.

20 Michele Lombardi, Michela Milano, and Andrea Bartolini. Empirical decision model learning. *Artificial Intelligence*, 244:343–367, 2017.

21 Nicholas Nethercote, Peter James Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, 2007. URL: `https://api.semanticscholar.org/CorpusID:1814073`.

22 Barry O'Sullivan. Automated modelling and solving in constraint programming. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, pages 1493–1497. AAAI Press, 2010. URL: `https://doi.org/10.1609/aaai.v24i1.7530`, `doi:10.1609/AAAI.V24I1.7530`.

23 Tomasz P Pawlak and Krzysztof Krawiec. Automatic synthesis of constraints from examples using mixed integer linear programming. *European Journal of Operational Research*, 261(3):1141–1157, 2017.

24 Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, 2011. URL: `https://dl.acm.org/doi/10.5555/1953048.2078195`, `doi:10.5555/1953048.2078195`.

25 Steve Prestwich. Robust constraint acquisition by sequential analysis. In *24th European Conference on Artificial Intelligence, Frontiers in Artificial Intelligence and Applications vol 325, IOS Press*, pages 355–362, 2020.

26 Steve Prestwich. Unsupervised constraint acquisition. In *33rd International Conference on Tools with Artificial Intelligence*, 2021.

27 Steve Prestwich. Extrapolating constraint networks by symbolic classification. In *5th IJCAI Workshop on Data Science Meets Optimization*, 2022.

28 Steven D Prestwich, Eugene C Freuder, Barry O'Sullivan, and David Browne. Classifier-based constraint acquisition. *Annals of Mathematics and Artificial Intelligence*, pages 1–20, 2021.

29 Steven D Prestwich and Nic Wilson. A statistical approach to learning constraints. *International Journal of Approximate Reasoning, Special Issue on Synergies Between Machine Learning and Reasoning*, 171, 2024. To appear.

30 Helmut Simonis. Building industrial applications with constraint programming. In *International Summer School on Constraints in Computational Logics*, pages 271–309. Springer, 1999.

31 Helmut Simonis. Requirements for practical constraint acquisition. In *AAAI 2023 Bridge on Constraint Programming and Machine Learning*, 2023.

32 Dimosthenis Tsouros, Senne Berden, and Tias Guns. Guided bottom-up interactive constraint acquisition. In *International Conference on Principles and Practice of Constraint Programming*, 2023.

33 Dimosthenis C. Tsouros, Senne Berden, and Tias Guns. Learning to learn in interactive constraint acquisition. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan, editors, *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, February 20-27, 2024, Vancouver, Canada*, pages 8154–8162. AAAI Press, 2024. URL: `https://doi.org/10.1609/aaai.v38i8.28655`, `doi:10.1609/AAAI.V38I8.28655`.

34 Dimosthenis C Tsouros and Kostas Stergiou. Efficient multiple constraint acquisition. *Constraints*, 25(3):180–225, 2020.

35 Dimosthenis C Tsouros and Kostas Stergiou. Learning max-csps via active constraint acquisition. In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

36 Dimosthenis C Tsouros, Kostas Stergiou, and Panagiotis G. Sarigiannidis. Efficient methods for constraint acquisition. In *24th International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science vol. 11008*, pages 373–388, 2018.

37 Xuan-Ha Vu and Barry O'Sullivan. A unifying framework for generalized constraint acquisition. *Int. J. Artif. Intell. Tools*, 17(5):803–833, 2008. `doi:10.1142/S0218213008004175`.

38 Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1-2):139–168, 1996.

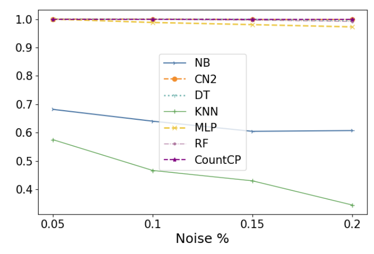
A Tuning details

We tuned the most important hyperparameters for MLP and KNN. A grid search, was conducted, using all different generalizations on all benchmarks utilized in the experiments. We used balanced accuracy as the tuning metric to address class imbalance.

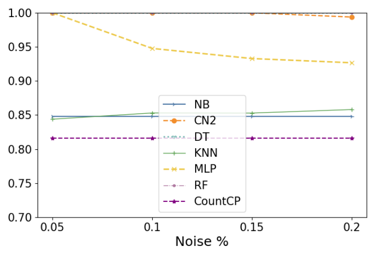
Regarding KNN, we explored different distance metrics (euclidean, manhattan, and minkowski), weights (uniform and distance) and number of neighbours (1-21). The optimal configuration used euclidean distance metric, uniform weights and 2 neighbours.

For the MLPs, we focused on tuning the number of hidden layers (1-4), the number of neurons per layer ([8, 16, 32, 64]), the optimizer (sgd and adam) and the learning rate ([0.001, 0.01, 0.1, 1]). The optimal configuration was identified as an MLP with a single hidden layer having 64 neurons, using adam as the optimizer and with a learning rate of 0.01, using ReLu as the activation function.

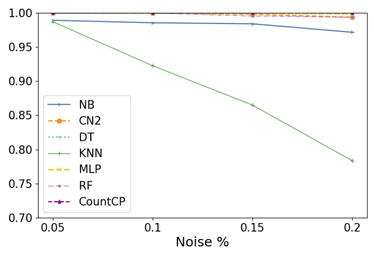
B Q2: Impact of FP noise per benchmark



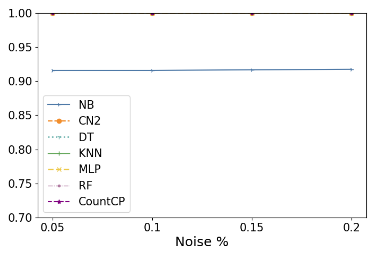
(a) Precision on Nurse Rostering



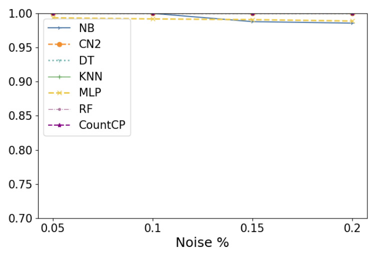
(b) Recall on Nurse Rostering



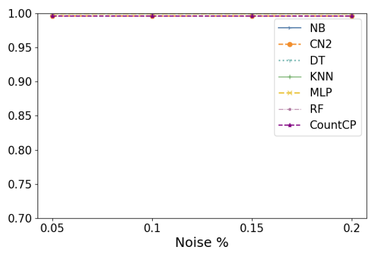
(c) Precision on Exam Timetabling



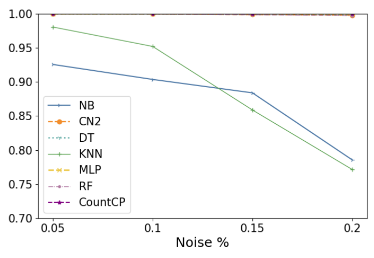
(d) Recall on Exam Timetabling



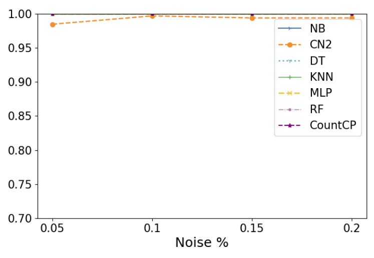
(e) Precision on Golomb



(f) Recall on Golomb



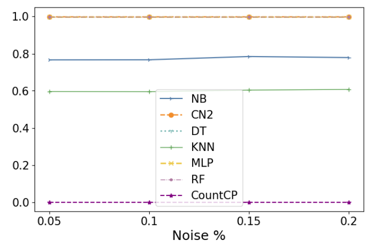
(g) Precision on Sudoku



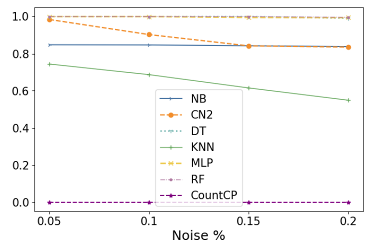
(h) Recall on Sudoku

Figure 4 Detailed results with the presence of FP noise

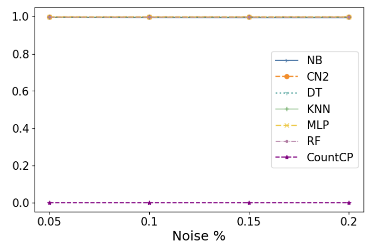
C Q3: Impact of FN noise per benchmark



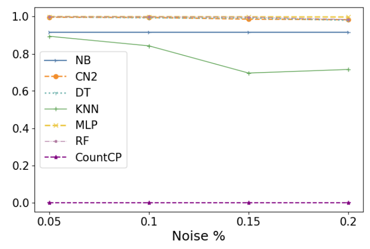
(a) Precision on Nurse Rostering



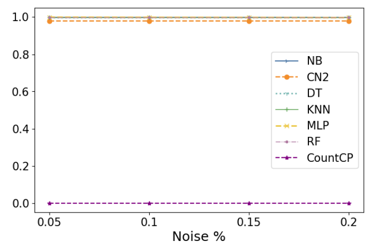
(b) Recall on Nurse Rostering



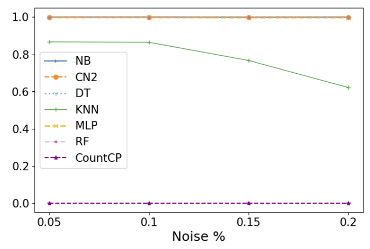
(c) Precision on Exam Timetabling



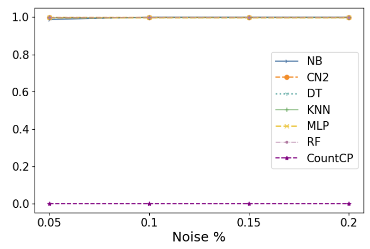
(d) Recall on Exam Timetabling



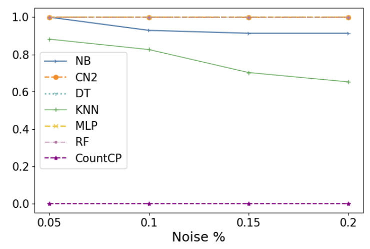
(e) Precision on Golomb



(f) Recall on Golomb



(g) Precision on Sudoku



(h) Recall on Sudoku

Figure 5 Detailed results with the presence of FN noise