Pyconstruct: Constraint Programming Meets Structured Prediction

Paolo Dragone1,2, Stefano Teso3 and Andrea Passerini1
1 University of Trento
2 TIM-SKIL, Trento, Italy
3 KU Leuven, Leuven, Belgium
paolo.dragone@unitn.it, stefano.teso@cs.kuleuven.be, andrea.passerini@unitn.it

Abstract
Constructive learning is the task of learning to synthesize structured objects from data. Examples range from classical sequence labeling to layout synthesis and drug design. Learning in these scenarios involves repeatedly synthesizing candidates subject to feasibility constraints and adapting the model based on the observed loss. Many synthesis problems of interest are non-standard: they involve discrete and continuous variables as well as arbitrary constraints among them. In these cases, widespread formalisms (like linear programming) cannot be applied, and the developer is left with writing her own ad-hoc solver. This can be very time consuming and error prone. We introduce Pyconstruct, a Python library tailored for solving real-world constructive problems with minimal effort. The library leverages max-margin approaches to decouple learning from synthesis and constraint programming as a generic framework for synthesis. Pyconstruct enables easy prototyping of working solutions, allowing developers to write complex synthesis problems in a declarative fashion in few lines of code. The library is available at: http://bit.ly/2st8nt3

1 Introduction
Many real-world problems involve learning to synthesize (potentially novel) structures or solutions from examples. Applications range from classical problems like syntactic parsing and image segmentation to design tasks like layout synthesis [Yu et al., 2011; Dragone et al., 2016], interface optimization [Gajos and Weld, 2005], and drug design [Lavecchia, 2015]. Learning in this setting involves iteratively synthesizing structures according to the current model, usually via mathematical optimization, and updating the latter based on some estimate of the corresponding loss. Well-known models include structured-output SVMs [Tsochantaridis et al., 2004] and conditional random fields [Sutton and McCallum, 2012].

A major issue with existing implementations is that they assume synthesis to be easily encodable in standard formalisms, like linear programming or dynamic programming. However, many problems of interest cannot be reformulated in such a way. Indeed, they may involve discrete and continuous variables (e.g., type and position of furniture pieces in layout synthesis) and arbitrary constraints between them (non-overlap, design guidelines). In this case, the developer is left with the task of writing her own ad-hoc solver.

We introduce Pyconstruct, a Python library specifically tailored for non-standard constructive tasks. Our goal is to cut the effort needed for writing a working learner. Pyconstruct combines two ingredients. First, synthesis is expressed in MiniZinc [Nethercote et al., 2007], a compact and general declarative language for constraint programming. MiniZinc enables writing non-standard synthesis problems in few lines of code. Other inference problems (such as separation [Joachims et al., 2009]) are expressed in the same way, reusing code snippets whenever possible. MiniZinc comes with several state-of-the-art backends for combinatorial, numerical and mixed problems, e.g., OptiMathSAT [Sebastiani and Trentin, 2015], Gecode [Schulte et al., 2010] and Gurobi [Gurobi Optimization, 2016]. Second, learning employs max-margin techniques, which decouple synthesis from learning. Therefore, the developer can change the synthesis problem without worrying about the learning algorithm at all. This speeds up prototyping solutions to non-standard constructive problems and adapting existing solutions to different settings. The library stems from several papers on constructive learning [Teso et al., 2016; 2017a; 2017b; Dragone et al., 2016; 2017; 2018a; 2018b].

2 Structured Prediction with Pyconstruct
We follow the usual structured output setting [Bakir et al., 2007], where the goal is to induce a mapping \( f: \mathcal{X} \rightarrow \mathcal{Y} \) from inputs \( x \in \mathcal{X} \) (e.g., sentences, images, empty room layouts) to output structures \( y \in \mathcal{Y} \) (tags, segments, furnished rooms). Synthesis, or inference, amounts to solving the optimization problem \( f(x) = \arg\max_{y \in \mathcal{Y}} \langle w, \phi(x, y) \rangle \). Here \( \phi: \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}^d \) is a joint input-output feature map and \( w \in \mathbb{R}^d \) is a vector of parameters to be estimated.
from data. Learning can be performed in a number of ways. We focus on max-margin approaches [Tsochantaridis et al., 2005], since they only require an oracle able to invoke inference (or related problems [Joachims et al., 2009]) for the target structures. Existing implementations are usually limited to structures for which an efficient oracle is known, such as sequences, trees or graphs. In stark contrast, Pyconstruct makes the inference oracle programmable. In Pyconstruct the oracle is a solver-agnostic MiniZinc program, allowing the domain of objects to be defined independently from the inference algorithm, which can be chosen and plugged in at runtime. This layer of abstraction enables performing inference on non-standard objects and enforcing arbitrary constraints on them. Our library has two main components: (i) a Python learning framework implementing several state-of-the-art algorithms (e.g., SSG [Shalev-Shwartz et al., 2011] and Block-Coordinate Frank-Wolfe [Lacoste-Julien et al., 2013]); (ii) a constraint programming inference engine powered by MiniZinc [Nethercote et al., 2007]. A MiniZinc file encodes the domain of the structured objects, which defines the input and output variables, the feasibility constraints, the feature vector, and methods for solving the different inference problems. Upon inference, Pyconstruct runs a MiniZinc-compatible solver on the domain file, specifying which inference problem to solve and which model to use. Models are objects holding the learned parameters, e.g., a linear model contains a vector w of weights. A model is usually the output of a learner, which estimates the model parameters from data using some learning algorithm. Learners in Pyconstruct are compatible with Scikit-learn [Pedregosa et al., 2011], and can be used in conjunction with most of its utilities.

3 Example: OCR Equation Recognition

In this demo, we demonstrate how our library can be used to solve quite complex structured-output prediction problems in just a few lines of MiniZinc code. In particular, here we showcase a simple problem of handwritten equation recognition. In this task we have to recognize equations of the form \( a + b = c \), where \( a, b \geq 0 \). The input is a sequence of \( N \) images, one per symbol. Notice that \( N \) is not fixed and depending on the number of digits of each number it may grow arbitrarily. We assume there are also two images for the “+” and “=” symbols, and that the equations are all valid. Encoding this prior knowledge inside a standard, Viterbi-like, inference algorithm over sequences would be quite verbose and difficult to optimize. Using Pyconstruct we can code this problem very easily\(^1\), as shown in the \texttt{domain.pmzn} file in Figure 1. The length of the sequence \( N \) and the sequence of images (assumed to be 9 by 9 black or white pixels) are given as inputs. The output sequence \( \text{seq} \) is the only output variable and is encoded as a vector of length \( N \) of integers in \([0, 11]\), where \( 0 \) to \( 9 \) represent the digits themselves, while \( 10 \) and \( 11 \) represent + and = respectively. In line 10 we define an array containing the indices of the two operators. The following three lines encode what we know about those indices, i.e., their order and their correspondence with their respective symbols in the sequence. Also, the sequence must contain exactly one + and one = (line 13). Next, we define the three actual numbers as \( n_i = \sum_{d_j \in \text{dom}_{ij}} 10^j \cdot d_j, \) where \( n_i \) is the number of digits in number \( i \) and \( d_j \) is the \( j \)-th digit of the \( i \)-th number, from the least to the most significant digit (omitted in Figure 1 for brevity). Finally, we constrain the numbers to be positive and the equation to be valid (lines 20 and 21). For space limitations, we also omitted the code describing the feature array. This should be an array of features that correlate the images to the symbols, e.g. [Taskar et al., 2004]. The last two lines contain templating code that Pyconstruct compiles to a proper MiniZinc solve statement, depending on the inference problem to be solved and the current model. After defining the MiniZinc file encoding the objects domain, one can estimate a structured-output model over it. All is needed is to instantiate a Domain providing the MiniZinc file in Figure 1, instantiate a Learner and fit it with the data. This takes exactly two lines of Python code, as shown in the \texttt{learn.py} file in Figure 1, where we use an SSG learner and fit it with some data \((X, Y)\). As an example, we used Pyconstruct to solve the above formula recognition task on a small dataset extracted from the ICFHR’14 CROHME competition data\(^2\). We used an SSG learner and compared the performance of the domain in Figure 1 against a similar domain lacking the prior knowledge encoded with the MiniZinc constraints. After training the algorithm with 800 samples, the constrained model achieves an average Hamming loss on the predicted sequences over the test set (200 samples) of 0.102, while the unconstrained one stops at 0.373 (\( p < 10^{-30} \)). The learning curves are also widely separated: the training losses are on average 22% lower for the constrained domain (\( p < 10^{-6} \), confirming that the constraints help learning with less data.

\(^1\)Code and data available at http://bit.ly/2LX0sMO

\(^2\)Data available at http://bit.ly/2J7Dh4v
References


