Abstract—This paper introduces an extension to PyStruct, which is an open source Python library for structured machine learning, based on general Conditional Random Field (CRF) models. We have extended it by supporting multi-type CRF to jointly classify objects of different natures, and by supporting logical constraints at prediction time. Our motivation for extending the library consists in addressing Document Understanding (DU) tasks by collectively classifying the objects present on one or several pages, jointly considering textual objects and objects of other nature like table or images or whatever object can be recognized. The logical constraints are meant to reflect prior knowledge on the DU task. We focus here on the improvement made over PyStruct, giving practical information as well. We also present a reproducible experiment. This new library is publicly available on GitHub under the Simplified BSD License.

Keywords—structured machine learning, Conditional Random Field, open source, python.

I. INTRODUCTION

Structured prediction is about predicting a structured output rather than a scalar value, given some input. While there are several models and relevant methods to support structured machine learning, our focus here is on the popular graphical model named Conditional Random Field (CRF). A CRF model can be learned from a set of labelled graphs and is able to predict the labels of a new graph, i.e. it collectively predicts the label of each node of the new graph.

CRF was first introduced by Lafferty et al. in 2001 [1] and later applied on many tasks in computer vision, natural language processing, document understanding to cite a few. Indeed, the advantage of structured machine learning derives from its capability of jointly classifying all objects of interest, instead of classifying each independently of the others.

A Python open source library named PyStruct [2] was recently released. This library proposes several CRF model variants, depending on the topology of the graph, i.e. chain, grid, etc. It also supports several training methods as well as inference methods, inference being the technical term used for prediction in the context of a CRF model.

We have contributed an extension to PyStruct to support another family of graphs, where nodes are typed. In other words, each node has a label space that depends on its type, which is known information.

This extension generalizes the notion of Factorial CRF [5], which itself is a generalization of linear-chain CRF that repeats structure and parameters over a sequence of state vectors. In this approach, the labels of a chain belong each to a certain temporality, with connection between labels that are consecutive within-chain and between consecutive chains. One advantage of this model is to jointly solve multiple labelling tasks, on the same sequence, instead of applying multiple separate models. Recently, Factorial CRF was applied to jointly perform Chinese word recognition and segmentation [6]. We showed in [3] how to implement a Factorial CRF using the typed CRF model.

Another aspect of the extension considers logical constraints between labelled nodes at inference time, for both untyped- and typed-CRF models.

In this paper, we first explain our motivation for supporting multi-type CRFs, then we briefly describe the original PyStruct library before detailing the extended version. Finally, we illustrate its use on two examples.

II. MOTIVATION

Our motivation for supporting nodes of different natures in a CRF model originates from a document understanding task, where we wanted to assign a semantic label to each “object” located on a page of a document.

Assume for instance that a page is segmented in textual blocks and that one wants to assign to each block one of the labels: title, body, footnote, heading, caption, cell text. It makes sense that to jointly classify all blocks at once rather than classifying each block individually, because labels of blocks have some interdependencies in a document. This is where structured machine learning fits well: one reflects the text blocks of the page as the nodes of a graph, while edges indicate some label dependency between two nodes, for instance because they are close to each other, or exhibit some geometric alignment, etc.

Now assume you also want to label other types of page object, like images as figure, table, picture, graphical line. One could setup another classifier dedicated to images. But both task have some inter-dependencies:

- Between a text blocks labeled caption and an image object labelled figure or table or picture
- Between footnote or heading objects and graphical line objects
This simplistic example aims at giving the intuition that classifying jointly all the objects of the page, whatever their type is, might be valuable.

Therefore, we contributed a CRF model where each node of the graph is typed, and each type defines a certain set of possible labels. An edge in the graph indicates a dependency between the labels of a pair of nodes, whatever their types. In some sense, each edge is typed and its type is the pair of types of the pair of nodes it links. In this approach, we assume that the type of a node is known.

Back to our example, we would reflect each page as a single graph with two node types: text blocks and images. The graph is then populated based on the objects of the page, whatever their nature. In this way, it becomes possible to jointly classify the labels of all objects of the page, assuming a multi-type CRF model is supported.

III. PYSTRUCT

As said, PyStruct is an open source library for structured machine learning. As there are plenty of explanations and examples on its web site, http://pystruct.github.io, we will here limit ourselves to a summary of aspects that are salient with respect to the extension.

When using PyStruct, one must select one of the available CRF variants and one of the available inference methods.

The CRF variants defines the graph topology, e.g., chain-like, regular grid or general possibly loopy graphs. It may also determine if the edges are associated with a vector of features or not. Indeed, edges represent the pairwise interaction between the labels of two nodes. This interaction may depend solely on the labels of the node pair or in addition may be influenced by some intrinsic characteristics of the edge, captured by the edge feature vector.

PyStruct tries to stay as close as possible to the interface and conventions of scikit-learn: http://scikit-learn.org. It also benefits from many of its packages, especially from its preprocessing package that is very convenient for feature extraction. As in scikit-learn, training is done by calling the fit method of the model, while prediction is done by calling the predict method. However, their input and output values differ since PyStruct is about structured machine learning, so the input is not a single feature vector and the output is not a single label. In fact, for performance reasons scikit-learn takes a series of feature vectors, i.e., a matrix, and produces a series of labels, despite no label interaction is considered. And the same comment applies to PyStruct.

IV. PYSTRUCT EXTENSION

PyStruct CRF graphs assumes that the nodes of the graph share all the same nature. In consequence, all nodes share the same model parameters, called weights, and the same set of possible labels. Similarly, all edges have the same nature and share the same edge weights. This was a limitation with regards to our needs, as explained in section II.

So, we propose a new CRF model [3] called NodeTypeEdgeFeatureGraphCRF, which is available at https://github.com/jlmeunier/pystruct together with the pre-existing PyStruct code.

A. Multi-type CRF Model

NodeTypeEdgeFeatureGraphCRF supports nodes of multiple natures, which we call node types. Each type has its own weights and set of possible labels. Similarly, edges have different nature depending on the type of their sources and targets-nodes. In a graph with N node types, there are N^2 types of edges.

The new model generalizes the native PyStruct EdgeFeatureGraphCRF, where edges have features.

Its constructor takes the form shown in Fig. 1. Beside the number of type, one must indicate the number of feature per node type and per edge type.

```python
def __init__(self, 
    n_types, # how many node type? 
    1_n_states, # how many labels per node type? 
    1_n_features, # how many features per node type? 
    a_n_edge_features, # how many features per edge type? 
    l_edges, # how many edges per type? 
    inference_method="ad3", 
    1_class_weight=None # class_weight per node type or None 
): 
```

Fig. 1. Constructor of a typed CRF model

Similarly, the fit and the predict method require input values that are split by type.

In single type, an input X is a graph represented by a triplet (node_features, edges, edge_features), formed of a node feature matrix, a 2-column edge definition matrix, and an edge feature matrix. Each row of the edge definition matrix gives the indices of the two nodes connected by the edge.

In multi-type, an input X is a graph represented by a triplet (node_features, edges, edge_features), formed by a list of node feature matrix (one by node type), a list of edge definition matrix (one by edge type), etc. In a graph with N types, the edges from I^th type to J^th type are defined in the (I x N + J)^th matrix of the edges list. The features of these edges are in the (I x N + J)^th matrix of the edge_features list.

This is explained in detail in GitHub.

B. Logical Constraints

Another extension, not mentioned yet, consists in supporting inference under logical constraints. Once a graph is constructed, it is possible to define logical constraint between the labels of certain nodes of the graph. This is a way to constrain the inferred labelling of the graph to respect some prior knowledge you have.

This extension opportunistically builds on the built-in capability of the AD3 [4] inference library, which is one of the libraries referred to by PyStruct. The logic operators supported by AD3 are XOR, XOR OUT, AT MOST ONE, OR, OR OUT, AND OUT, IMPLY.
A constraint consists in one of the operator above applied on a set of triplets \((label, node, negated)\) where negated is a Boolean value indicating if the corresponding argument is negated in the logical relation. For instance, one can express that at most one of a set of \(N\) nodes must have a label \(y\), by specifying a AT\_MOST\_ONE constraint over the set of triplets \((y, node, False)\) for \(i\) in 1..\(N\).

The extension we made applies on both the original single-type CRF model as well as on the multi-type one.

To support those constraints at inference time, the initial graph is binarized, as explained in [3], to reflect each original graph node by \(L\) binary nodes, with \(L\) being the number of possible labels. A XOR constraint over those \(L\) binary nodes ensures that exactly one label is assigned to the initial node. Original graph edges are reified as binary nodes, with a more complex set of constraints and edges among binary nodes to preserve the meaning of the initial edge. Once the graph is binarized, any constraint defined by the user applies directly on the binarized graph.

Please refer to the online documentation of extended PyStruct for its precise usage.

V. EXAMPLE: THE HIDDEN SNAKE

To illustrate the use of a multi-type CRF model, we have modified a pre-existing example illustrating the merits of the PyStruct EdgeFeatureGraphCRF model, namely the Snake example.

A. Hidden Snake Task

In this example, the input is a series of images picturing a “snake”. Each snake consists in 10 parts, which occupies 1 pixel each. Those pixels are labelled from tail to head, i.e. 1 to 10. The color of each snake pixel denotes the direction of the corresponding snake body part, so that each part connects to the next part. There are 4 possible directions (up, right, down, left). The other pixels of the image are background pixels, labelled 0, and colored differently. Fig. 2 below illustrates a snake image, on right, and its corresponding labelling on left, where white denotes background label while tail to head is reflected by light grey to black colors.

Given an image, predicting the correct label of each pixel independently is not possible, which is why a collective prediction is required, and hence CRF comes into play. (one could properly predict only the head and the tail.)

To illustrate the use of multi-type CRF, we extend the task, considering that snakes hide themselves. So, some images do contain a snake, while the other images do not, despite looking like each other.

To this end, we generate additional images by damaging the pre-existing ones. Fig. 3 shows a generated image, where one pixel has been changed so that the image does not contain a complete 10-parts snake. The label of the pixels of such an image are all set to background.

The Hidden Snake task consists in labelling the pixels as well as labelling the image as Snake or NoSnake.

B. Single- and Multi-Type CRF Models

The chosen way of creating a (single-type) PyStruct CRF model for the Snake task consists in reflecting each pixel as a node that is linked to its 4 neighbors (horizontally and vertically), forming a regular grid graph, as shown in Fig. 4. Please refer to [2] for a detailed description of the node and edge features definition.

For the Hidden Snake task, we compare the original single-type CRF model with a multi-type CRF model where we also reflect the image itself as a node in the graph defined above. We chose to link that page-node to each pixel-node.

The vector representation of the image itself is constituted of 7 values: the height and width in pixels of the non-background area in the image, and the histogram of number of pixels for each of the 5 colors. This representation is arbitrary and for sure is not appropriate to decide if an image contains or not a snake. The vector representation of the pixel-to-image edges are made by concatenating the vector representation of the source and target node.
C. Hidden Snake Experiments

Exploiting the 200 training and 100 test images available in PyStruct, we generated 176 and 87 damaged images respectively. This lower number is because the damage done at random on a given image sometime still produces a snake image. In result, we have 387 training images and 176 test images.

On this dataset, we run 4 methods:

1. An oracle that predicts ‘background’ for all pixels.
2. PyStruct: The single-type CRF model. This is the baseline method.
4. PyStruct+Logic: here we do as in PyStruct+ (3 above) but at test time, we inject some logic constraints in the inference. We arbitrarily chose to express the fact that a snake has at most one cell of each possible cell label. So, for each test image, we create 10 AT MOST ONE constraints, one per label. Given a label, its AT MOST_ONE constraint covers all “pixel nodes” of the graph.

TABLE I. shows the numerical results of the experiment.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>All-background Oracle</td>
<td>7015</td>
<td>0.86</td>
<td>0.00</td>
<td>n/a</td>
</tr>
<tr>
<td>PyStruct</td>
<td>7015</td>
<td>0.92</td>
<td>0.80</td>
<td>n/a</td>
</tr>
<tr>
<td>PyStruct+</td>
<td>7015</td>
<td>0.94</td>
<td>0.87</td>
<td>0.84</td>
</tr>
<tr>
<td>PyStruct+Logic</td>
<td>7015</td>
<td>0.94</td>
<td>0.89</td>
<td>0.82</td>
</tr>
</tbody>
</table>

This experiment shows the interest, on this task, of jointly classifying objects of different nature, pixels and the image here.

The use of logical constraints increases pixel accuracy at the price of decreased image accuracy. The value is less clear and deserves some more experiments.

To reproduce these results, please refer to GitHub file examples/plot_hidden_short_snakes_TYPED.py.

VI. EXAMPLE: TABLE SEGMENTATION

We report here a preliminary experiment with a Document Understating (DU) task, which consists in processing historical death records documents from the Passau Diocesan Archives\(^1\). We choose to apply supervised machine learning at one step of the task, where the table needs to be segmented in rows. In this section, we will focus on this precise step of the global task, which deserves a complete publication by itself.

A. Table Segmentation Task

After preprocessing of the document images and human annotation, we obtained 85 annotated pages. (We do not describe the preprocessing here, because this goes beyond this paper.) With respect to this experiment, we are interested in two specific output elements of the annotation:

- The text lines, i.e. the bounding boxes of each line of text in each table cell or elsewhere on the page
- The table row separator, i.e. lines drawn by the human annotator to separate the table rows. Some of these lines map onto graphical lines appearing on the page, some do not. (Absence of graphical line between two rows.)

Out of those two types of information, we generated our specific training data as follow:

- We labeled the text lines of each cell using a BIES scheme, column-wise. The top-most text line of a cell gets a label B (for Begin), the next ones get a I (Inside), and the last one gets a E (End). A text line alone in a cell gets a S (Singleton) label. Any text line outside of the table gets a label O (Other). This is done columns by column.

- We then ran a graphical line detector provided by a READ partner: The Computer Vision Lab2, from the Technische Universität Wien. By matching detected lines against human annotation, we labeled the detected lines as I (Inside separator) or O (Other). We call “inside separator” those lines in between two rows, which act as a separator of two table rows.

The Fig. 7 at the end of the article shows a page of the training set. But we noticed that our labeling, which was automatically generated using a basic heuristic, of the graphical lines is of poor quality and quite noisy. So, we won’t draw any strong conclusion from this experiment for the time being.

We hope that by solving this text line categorization problem, we will in turn be able to properly segment the page objects into a table properly segmented in rows. So it is the BIES categorization task which plays a key role in the overall task.

B. Single-Type CRF Model

We model each page as a graph, where each node reflects a text line. An edge in the graph reflects a neighboring relationship between two text lines, possibly long distance one. More precisely, whenever there is horizontal, respectively vertical, significant and direct overlap between two bounding box of the two text lines, we create a vertical, respectively horizontal, edge. ‘Significant’ means that the overlap must of at least a certain threshold. ‘Direct’ means that the two bounding boxes must be in line of sight of each other, i.e. without any obstructing block in between.

Fig. 5 shows in blue with white filling the neighbors of the red central block.

\(^1\) http://www.archiv.bistum-passau.de

\(^2\) http://www.caa.tuwien.ac.at/cvl
C. Multi-Type CRF Model

Considering the graphical lines as another type of elements, we extend the single-type graph with nodes reflecting the graphical lines. We also create new types of edges: text-to-line edges, as well as line-to-line edges. Fig. 6 shows such a multi-type graph.

D. Categorization Experiment

Once we have reflected our 85 pages as either 85 single type graphs, or 85 multitype graphs, we trained a CRF model on both kinds of graphs. The features used to represent the text line or lines are geometric ones (no text available). We also trained a linear regression model on the text lines using the same BIES labeling, and same features. This model lacks any contextual information and not surprisingly shows bad performance. Indeed, this categorization task deserves the use of a structured machine learning approach. Adding some context as additional features on line elements is doable but not so easy as the number of neighbors is variable and aggregating the neighbor’s features is not straightforward.

Table II. and Table III. show the numerical results of the experiment, on a 20 folds cross-validation. A Wilcoxon test on the 20-fold results shows a high statistical significance. (p-values in a range 0.001 – 0.004.)

Table III. RESULTS OF THE MULTI-TYPE EXPERIMENT.

<table>
<thead>
<tr>
<th></th>
<th>--- MICRO-AVERAGE ---</th>
<th>precision</th>
<th>recall</th>
<th>f1-score</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>text_B</td>
<td>0.922</td>
<td>0.936</td>
<td>0.929</td>
<td>49996</td>
<td></td>
</tr>
<tr>
<td>text_I</td>
<td>0.862</td>
<td>0.926</td>
<td>0.892</td>
<td>43476</td>
<td></td>
</tr>
<tr>
<td>text_E</td>
<td>0.916</td>
<td>0.929</td>
<td>0.922</td>
<td>49960</td>
<td></td>
</tr>
<tr>
<td>text_S</td>
<td>0.904</td>
<td>0.797</td>
<td>0.847</td>
<td>41962</td>
<td></td>
</tr>
<tr>
<td>text_O</td>
<td>0.032</td>
<td>0.041</td>
<td>0.036</td>
<td>844</td>
<td></td>
</tr>
<tr>
<td>sprtr_I</td>
<td>0.773</td>
<td>0.840</td>
<td>0.805</td>
<td>28430</td>
<td></td>
</tr>
<tr>
<td>sprtr_O</td>
<td>0.913</td>
<td>0.872</td>
<td>0.892</td>
<td>54457</td>
<td></td>
</tr>
<tr>
<td>avg / total</td>
<td>0.888</td>
<td>0.885</td>
<td>0.886</td>
<td>269125</td>
<td></td>
</tr>
</tbody>
</table>

However, we cannot draw any conclusion with respect to the appropriateness of the approach for table segmentation for several reasons: 1) we need to reinforce the baseline method; ii) the labeling of the graphical lines is noisy. However, we conclude here that the proposed extension to PyStruct is functioning properly.

VII. CONCLUSION

We have proposed an open source extension to PyStruct, which is an open source Python library for structured machine learning based on CRF models. We kept the same permissive license, namely the Simplified BSD license, for the extension.

The source code and experiments are available at https://github.com/jlmeunier/pystruct.

To our knowledge, PyStruct is the only open source Python library that supports CRF models without topological constraints, or in other words that works with general CRF graphs, possibly loopy ones.

The proposed extension adds the possibility to classify objects of different natures, in the same graph, and to account for dependencies between their labels, which are in different label spaces.

We believe this extension may be useful in Document Understanding tasks because the objects laid out on a document page have a dependency on each other despite their possible different natures. Since there exist physical and logical layout analysis methods, to identify these objects and their nature, we expect to be able to fruitfully use multi-type CRF in DU tasks.

We have experimented this extended approach on an artificial example derived from the machine learning literature and observed some significant advantage. We also did some preliminary experiment on a DU task showing the feasibility of the approach.

We are now going further with Document Understanding task in the context of the EU READ project. We would encourage, and will support, other practitioners to test this extended library and report in some way to us, or all of us, about the findings.
ACKNOWLEDGMENT

The author would like to thank his colleagues Jean-Marc Andréoli, Hervé Déjean, Matthias Gallé and Chunyang Xiao for their support and fruitful discussions, as well as Andreas Müller for his support on the PyStruct forum. We are also grateful to the ICDAR-OST reviewers for their comments and to our READ partners who contributed data and software for the table segmentation experiment.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 674943.

No animals were harmed

REFERENCES


Fig. 7. A table image. The BIES labels on text lines are shown as orange, green, light blue, yellow. The Inside, Other labels on separators are violet and grey, respectively. Note that some lines on top of page are wrongly labeled Inside.