Arsenal

Daniel Elenius, Stéphane Graham-Lengrand,
Natarajan Shankar, and Eric Yeh

April 22, 2020

Overview The Arsenal system is designed to parse domain-specific natural language and convert it into a formal representation, offering a methodology to build *semantic parsers* for various domains. The use of natural language to express formal content, for instance descriptions of models, procedures, and requirements on them, has clear advantages when that content has to be understood by humans, but presents challenges for automation.

Controlled languages can be used to address those challenges, restricting the use of natural language by limiting the allowed vocabulary or pre-determining the syntactic forms that sentences can take [Kuh15]. Although abiding by those restrictions does simplifies parsing, it can have several drawbacks, including a decrease of legibility for humans, and requiring from users the same discipline as if they were reading and writing code.

Arsenal aims at retaining a more flexible use of natural language to express formal content, so that this content can be manipulated by users who may not be coders, while being amenable to automated processing for, e.g., formal modeling and analysis.

The Arsenal system makes use of machine learning to turn a flexible use of natural language into formal representations, in a context where the use of natural language is still specific to an application domain. It is designed to produce formal representations in the form of expression trees. Trees constitute a versatile data-structure for which several standard syntaxes exist, such as JSON or XML, allowing the formal representation to be communicated to consumers of the Arsenal output.

The Arsenal system expects its output to be clearly specified for the applicative domain considered, in the sense that a domain-specific *grammar* must be given specifying which expression trees are acceptable as output. The assumption is that the grammar captures those trees that have semantical meaning (as opposed to gibberish), or at least those trees that the consumer of Arsenal's output can make sense of. In other words, the nature of the concepts mentioned in the input natural language, and the kinds of relations that may hold between concepts, do not *emerge* out of the automated parsing process; otherwise dealing with them would become a burden on the consumer of the parsed output. Requiring a grammar describing what concepts and relations may be, instead of inferring them, is a reasonable assumption for domain-specific applications,

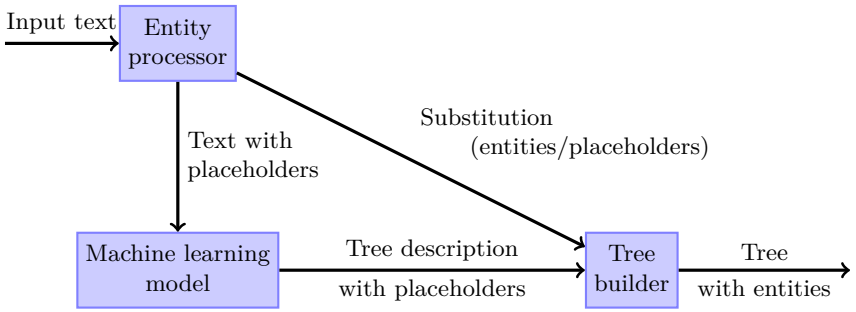

Figure 1: Arsenal's run-time component

involving for instance requirements, models, API documentation, etc; moreover, in an environment where the consumers of the parsed sentences are identified, their needs will prompt the specification of Arsenal's output grammar.

A selling point of the Arsenal system is that is guarantees that its output trees are *always* well-formed with respect to that grammar.

We illustrate Arsenal with the example of *regular expressions*. These are used pervasively in character string processing, identifying patterns, and a typical use for it is in a search or search-and-replace tool. A user may want to search all documents on his computer where the following pattern appears: "the word 'article' or the word 'paper' followed by any number of words, followed by the word '2020', the whole thing appearing on the same line, and that line containing nothing else than that". The search engine needs to understand this query, and the standard technology requires the user to express it with a regular expression such as

```
^(article|paper)([a-z]| )*2020$
```

This regular expression is shorter than the natural language for it, but requires technical skills. Arsenal would produce that regular expression from a natural language sentence.

Architecture

The run-time component A typical end-user of the Arsenal system will only interact with its run-time component, which performs the task described above, namely turning natural language sentences into formal representations as trees. This run-time component is described in Figure 1, and splits into three sub-components.

The main (sub-)component of the Arsenal's run-time is a machine learning model, whose role is to convert NL constructs into tree constructs from the grammar. The model has been trained from a set of labeled examples, which illustrate how basic NL constructs relate to tree nodes. Full sentences and trees

can be seen as the composition of those basic constructs and nodes, respectively, but they also usually involve a (typically much larger) set of *entities*. For instance, regular expressions only use a small number of constructs (such as repetition, alternatives, being anchored at the beginning and/or the end of a line), but also refer to characters drawn from a much larger pool (e.g. UTF8 characters). Rather than placing on the machine learning model the burden of learning those entities, which in some domains can be in infinite numbers, the Arsenal pipeline uses an *entity processor*, which detects those entities in the input text and replaces them with names called *placeholders*. It also produces a *substitution* recording how placeholders map to the substituted entities. Taking the example of a regular expression, described in natural language as

```
zero or more repetitions of any character between 'e' and 'h',
```

the entity processor would produce, on the one hand, the sentence

```
zero or more repetitions of any character between Character1
                                                  and Character2,
```

and, on the other hand, a substitution mapping `Character1` to `'e'` and `Character2` to `'h'`.

The model outputs the decription of an output tree, involving the placeholders that were seen in the input. Technically, the description is the Polish notation of the tree, where the nodes of the tree are enumerated in a depth-first traversal. In the case of the above example, the Polish notation would simply be: `Plus CharacterRange Character1 Character2`.

The final tree is reconstructed from that description, and the substitution produced by the entity processor is used to replace the placeholders in the tree by the actual entities that they stood for. In the example above, that tree would have four nodes, and would be representable with the S-expression `(Plus (CharacterRange 'e' 'h'))`.

The design of the domain-specific grammar and the training of the model The other components of the Arsenal system, described in Fig. 2, are designed to produce a machine learning model trained to perform the run-time task. The exact nature of the model itself may vary, but it is always parameterised by *weights* and trained by a *supervised learning* process: we provide a set of pairs, each consisting of a natural language sentence and its formal representation as a tree, and tune the parameters so that the model, when run on these sentences, produce the correct trees. This is described in the lower part of Fig. 2.

This also raises the question of how such a labeled dataset is obtained, given that the training set is very specific to the tree grammar used in Arsenal's output. Arsenal's approach to this issue is to generate an entirely synthetic dataset, out of the chosen grammar.

The first task, performed by hand, is to actually describe the grammar (in code), in agreement with the consumers of Arsenal's output. The grammar is

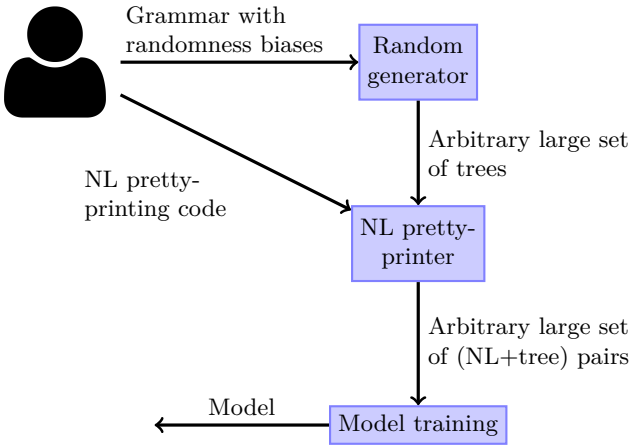

Figure 2: Producing Arsenal's models

fed as input to a component that will generate random trees belonging to the grammar, as described in the upper part of Fig. 2. The generator component is written once-and-for-all and does not need to be changed for different grammars or different application domains. However, in order to make sure that the generated data does cover the typical tree constructions that will be used in practice, it is convenient to bias the generation so that some constructs are more likely to be generated than others. These biases are provided by the grammar designer.

The second task is to pair each kind of nodes in the grammar (in finite numbers) with a couple of natural language renderings for it. These illustrate "typical ways" in which the grammar construct could be described in natural language (which often offers numerous ways of expressing the same content). Arsenal's *pretty-printer* then non-deterministically compounds the NL renderings defined by the grammar designer for each grammar construct, so as to produce a randomly generated NL sentence for each of the random trees previously generated. Those pairs constitute the labeled dataset with which the model can be trained. Since pretty-printing is specific to the grammar and relies on a finite set of NL renderings provided by the grammar designer, Arsenal provides a library for minimizing that manual effort. Once again, the designer can determine *biases* for choosing the most likely renderings of each grammar construct, which drives how the synthetic dataset is populated.

Finally, training the model from the labeled dataset is performed using the standard techniques of supervised machine learning. Programming the training can be done once-and-for-all together with the choice of model to be trained, although the more complicated the grammar is, the more nodes may be needed in the neural net.

In the work done so far, we have used a variant of *sequence-to-sequence* models equipped with an *attention* mechanism, which have been used very successfully for machine translation from natural language to natural language.

Such models are made of an *encoder and a decoder*. The encoder is a *Recurrent Neural Net* (RNN) that keeps an internal state (a vector of reals); as the input sentence is recursively traversed from one end to the other, the internal state is updated with every word that is read (and encoded as a vector of reals). The history of the encoder's internal state is recorded. The decoder produces a stream of tokens (in our case, the tokens that make up the Polish notation for the output tree). It also has an internal state, and an attention mechanism that focuses on certain parts of the encoder's internal state history. The decoder produces one token at a time, choosing it according to the last token it produced, its internal state and the attention, both of which are updated before producing the next token.

For Arsenal we adapted the *sequence-to-sequence* approach to produce tree descriptions in Polish notations, dynamically forcing the output tokens to describe a tree that is well-formed with respect to Arsenal's output grammar, and changing the halting condition of the output stream to make is stop as soon as the produced stream describes a complete, well-formed tree. This adaptation has provided an increase of accuracy of our models over standard ones, on semantic parsing tasks.

References

[Kuh15] T. Kuhn. A survey and classification of controlled natural languages. *CoRR*, abs/1507.01701, 2015 1