REEL:
A Relation Extraction Learning Framework
— INESC-ID Lisboa Tech. Rep. 15/2014 —

Pablo Barrio♯, Gonçalo Simões*, Helena Galhardas*, Luis Gravano♯
*INESC-ID and Instituto Superior Técnico, Universidade de Lisboa, Av. Prof. Dr. Cavaco Silva, 2744-016 Porto Salvo, Portugal
♯Columbia University, 1214 Amsterdam Avenue, New York, NY 10027, USA

This revision: June 13, 2014
REEL: A Relation Extraction Learning Framework

Pablo Barrio, Gonçalo Simões, Helena Galhardas, Luis Gravano

*INESC-ID and Instituto Superior Técnico, Universidade de Lisboa, Av. Prof. Dr. Cavaco Silva, 2744-016 Porto Salvo, Portugal
*Columbia University, 1214 Amsterdam Avenue, New York, NY 10027, USA

Abstract. Natural language text documents often contain information that is structured in nature. In its structured form, this information is more suitable for querying and analysis than as unstructured text. Information extraction systems extract such structured data from text documents through sophisticated text analysis. In particular, relation extraction systems, which are the focus of this paper, extract entity mentions and their relations from natural language text. For this, relation extraction systems employ a variety of text processing tools, such as sentence segmentation, tokenization, entity recognition, and part-of-speech tagging. Unfortunately, few extraction systems are publicly available and, even when they are, it is usually challenging to adapt them to learn to extract new relations and to operate over new text collections. To address these challenges, we present REEL (Relation Extraction Learning framework), an open source framework that facilitates the development and evaluation of relation extraction systems. To define a relation extraction system for a new relation and text collections, users only need to specify the parsers to load the collections, the relation and its constraints, and the learning and extraction techniques, which makes REEL a powerful framework to enable the deployment and evaluation of relation extraction systems for both application building and research.

Keywords: Information Extraction, Relation Extraction, Machine Learning, Open Source

1 Introduction

Relation extraction systems are sophisticated information extraction tools that automatically discover structured relations between entities in natural language text. For example, a properly trained relation extraction system would extract the tuple (earthquake, Chile’s northern Pacific shore) of the relation Occurs-In(Natural Disaster, Location) from the text excerpt “A strong 6.7-magnitude earthquake shook Chile’s northern Pacific shore.” To extract such structured information from text documents, state-of-the-art relation extraction systems usually employ a variety of text processing tools, such as sentence segmentation, tokenization, entity recognition, and part-of-speech tagging, and many times require specifying and enforcing constraints on the extracted information, such as requiring that extracted entities be of a certain type or that entities in an extracted relation be mentioned within N words of each other [26].

Many relation extraction systems have been proposed in the literature (e.g., [9,14,24,30,31]); however, few such systems are publicly available and, even when they are, it is usually problematic to adapt and evaluate them over new relations and text collections. To avoid implementing such complex systems from scratch, and to facilitate their adaptation and evaluation, developers often rely on relation extraction toolkits. One such toolkit is T-Rex [19], which is based on a text processing and entity extraction learning framework [20]. T-Rex splits the relation extraction task into relatively coarse modules, which limits the reuse of text processing and learning components across relation extraction tasks, and hence complicates the implementation for new extraction tasks. Also, T-Rex does not impose constraints on the output of its modules and of the overall extraction process, which complicates the experimental comparison of different extraction strategies and their output. As a result, to experimentally evaluate and compare relation extraction systems in T-Rex, we must rely on ad hoc solutions, which is far from ideal.

Other toolkits originally proposed for related text-centric tasks, such as text processing (e.g., UIMA [16]), machine learning (e.g., MALLET [23]), natural language processing (e.g., StanfordNLP [2], ClearTK [25]), and entity extraction (e.g., MinorThird [12]) provide low-level building blocks that are helpful for relation extraction, but lack the code and infrastructure to directly support relation extraction. To use these frameworks for relation extraction we could extend them by including the infrastructural elements missing in each framework. However, this would require in many cases a significant implementation
effort and a drastic redesign of the toolkit, since we would have to incorporate full support for the missing infrastructural elements. A more promising approach, which we advocate in this paper, is to integrate and complement valuable text processing toolkits—to exploit their powerful implementations of low-level text operations—and machine learning toolkits—to exploit their powerful implementations of relevant learning operations—for our relation extraction task. Our approach is then similar in spirit to that of ClearTK [25] but for a different problem. (ClearTK focuses on statistical natural language processing.)

Specifically, in this paper we introduce REEL (RElation Extraction Learning framework), an open-source framework to easily develop and evaluate relation extraction systems. REEL provides the code and infrastructure to: (i) handle various input text formats, which enables operations over different text collections; (ii) plug in appropriate text processing steps and tools, which enables diverse processing of the text with minimal effort; (iii) define and combine conceptual relation constraints that are automatically enforced; (iv) decouple learning and extraction from the text processing, which enables the straightforward integration and re-usability of different extraction algorithms; and (v) uniformly execute and evaluate relation extraction systems, which enables the testing and fair assessment of these systems. REEL, in contrast to existing toolkits, effectively modularizes the key components involved in relation extraction systems. To define a relation extraction system for a new relation and text collections, users only need to specify the parsers to load the collections, the relation and its constraints, and the learning and extraction techniques, which makes REEL a powerful framework to enable the deployment of relation extraction systems for both research and application building.

The rest of the paper is organized as follows. Section 2 provides the necessary background and discusses the limitations of existing work. Section 3 describes the architecture of REEL, focusing on its main components and their interaction for developing and evaluating relation extraction systems. Section 4 shows how to use REEL in practice, by providing an end-to-end implementation of a typical relation extraction system. Finally, Section 5 concludes the paper and suggests directions for future work. REEL is publicly available, as open source under the General Public License Version 3 (GPLv3) license, at http://reel.cs.columbia.edu/.

2 Background and Related Work

Relation extraction systems are sophisticated software tools that automatically discover structured relations between entities in natural language text. For example, a properly trained relation extraction system would extract the tuple ⟨Mark Chapman, second-degree murder, 1981⟩ of the relation Charged(Person, Charge, Date) from the text excerpt “John Lennon’s killer, Mark Chapman, was sentenced in 1981 to 20 years to life in prison after pleading guilty to second-degree murder.” Relation extraction is a complex task that relies on a wide variety of text processing and machine learning steps to extract tuples from text, which makes the definition, deployment, and evaluation of systems for new extraction tasks and text collections a challenging proposition.

To discover relations in a text document, a typical relation extraction system (e.g., [3,8]) performs a series of steps, which are illustrated in Figure 1. The extraction system starts by loading the contents of a given text document, tagging the entities of interest and splitting the tagged text into text segments (e.g., sentences). Then, it generates, for each segment, zero or more candidate text segments. In a nutshell, a candidate is a text segment that satisfies all the relation and entities constraints for the extraction task at hand (e.g., entities need to be of a certain type, say, Person, Charge, or Date, or entities need to be mentioned within $N$ words of each other). Hence, candidates contain a mention of a potential tuple. Then, from each candidate, the system extracts relevant features, such as tokens, part-of-speech tags, dependency graphs, or distance between entities. Finally, the extraction system uses these features as input to the tuple extraction algorithm (e.g., [9,10,13,14,17,22,24,30,31]), which often relies on a binary classification approach to determine whether the entities in the candidate are related as needed. Specifically, based on decisions learned during a training step, the binary classifier labels a candidate as positive, thus concluding that the entities are related, or negative.

Tuple extraction algorithms come in two flavors, namely, pattern- and statistics-based. Most of the early approaches to relation extraction were pattern-based [3,8,15,24] and aimed at identifying, for a given relation of interest, text and grammatical patterns that signal a relation between entities. For instance, the text pattern “⟨Person⟩ was arrested for ⟨Charge⟩ on ⟨Date⟩” may determine the existence of the above
Fig. 1. Key steps in relation extraction.

Charged(Person, Charge, Date) relation. Recent strategies for relation extraction have focused on statistical approaches. These approaches are usually divided into two categories, namely, feature- and kernel-based. Feature-based relation extraction systems (e.g., [14,17,22,31]) are usually based on linear classifiers, over a predefined feature space (e.g., part-of-speech tags, words, entity types). Kernel-based relation extraction systems (e.g., [9,10,13,30]), on the other hand, take advantage of kernels to analyze text segments according to structured representations (e.g., sequences, dependency graphs). These techniques usually explore a larger, and not explicitly predefined, feature space that feature-based approaches cannot handle. Kernel-based systems are able to learn that, for example, certain shapes of dependency graphs are more likely to include related entities than others. (We illustrate the dependency graph of our running example in Figure 6c.)

Because of all the steps involved in relation extraction, and because of the wide variety of tuple extraction algorithms, developing relation extraction systems is a rather challenging and time-consuming process. To avoid implementing such complex systems from scratch, and to facilitate their adaptation and evaluation, developers often rely on relation extraction toolkits. One such toolkit is T-Rex [19], which is based on the text processing and entity extraction learning framework RUNES [20]. T-Rex models the various relation extraction steps (e.g., word tokenization, entity recognition, learning of the relation extraction model) as additional text processing steps that can be coupled together—via a common interface—with RUNES’s native components.

Unfortunately, T-Rex splits the relation extraction task into relatively coarse modules, which limits the reuse of text processing and learning components across relation extraction tasks, and hence complicates the implementation of new extraction tasks. In particular, since all components share a generic interface that does not reveal their internal functionality properly, the developer needs to be aware of the low-level details of these components to reuse them successfully. Also, the coarse T-Rex modules do not constrain their output, which complicates the experimental comparison of different relation extraction strategies. As a result, to experimentally evaluate and compare relation extraction systems in T-Rex, we must rely on ad-hoc solutions, which is undesirable. Furthermore, changes in the evaluation process, such as using new evaluation measures, may lead to ubiquitous and fine-grained source code modifications across systems, which is problematic.

Other toolkits originally proposed for related text-centric tasks provide low-level building blocks that are helpful for relation extraction, but lack the code and infrastructure to support all steps involved in relation extraction. Specifically, text processing toolkits (e.g., UIMA [16]) tend to only provide support for the entity tagging and feature extraction steps described above and, as a result, lack infrastructure for the remaining steps. Machine learning libraries (e.g., MALLET [23], LibSVM [11], Weka [18]), in contrast, provide the foundation for learning the tuple extraction algorithm, although they lack support for the variety of steps that relation extraction systems routinely need, such as entity tagging and candidate enumeration. Finally, natural language processing suites (e.g., NLTK [6], OpenNLP [1], StanfordNLP [2], ClearTK [25], LingPipe [5]) and entity extraction frameworks (e.g., RUNES [20], MinorThird [12]) consolidate the fea-
The text processing and machine learning libraries, although they lack support for relation extraction altogether, in that they do not offer infrastructure for, say, candidate generation and relation constraints.

To support relation extraction, one alternative would be to extend the toolkits above by including the missing components. However, this would require in many cases a significant implementation effort and a drastic redesign of the toolkit, since we would have to incorporate full support for the missing steps. A more promising approach, which we adopt in this paper, is to integrate the existing toolkits into a properly designed relation extraction infrastructure, to leverage their capabilities as needed.

3 The REEL Framework

In Section 2, we introduced the fundamental concepts for relation extraction and discussed the limitations of existing toolkits. We now describe REEL, an open-source framework for the implementation and evaluation of relation extraction systems that overcomes the limitations of previous approaches. Specifically, in Section 3.1 we provide a high-level overview of the REEL architecture and its components. Then, in Sections 3.2 to 3.7 we describe each component in detail.

3.1 System Architecture

We now introduce REEL’s flexible, modular architecture (see Figure 2). REEL’s components can be divided into two types. The Text Processing components are responsible for transforming the input text documents into the input format for the relation extraction techniques. Then, the Learning and Extraction components perform the most important relation extraction tasks, namely, to learn, execute, and evaluate relation extraction systems. We now provide details about these two types of components.

The Text Processing components (see Figure 2), which include the Text Segment Loading, Candidate Generation, and Feature Extraction and Operable Structure Generation components, focus on processing the input documents for the relation extraction system. First, the Text Segment Loading component, which we describe in Section 3.2, loads the documents in a text collection and transforms them into text segments that the relation extraction techniques can then process. The key challenge here is to allow the integration of different text processing tools (e.g., file loaders, XML parsers) so that it is easy to use different types of collections in REEL. Second, the Candidate Generation component, which we describe in Section 3.3, produces the candidate text segments that we introduced in Section 2. As discussed, these candidate text segments satisfy the constraints associated with the relation at hand (e.g., the text segment must include the Person, Charge, and Date entities within $N$ words of each other). The key challenge for this component is to enable a flexible definition of constraints over the entities and the relation itself. Finally, the Feature Extraction and Operable Structure Generation component, which we describe in Section 3.4, extracts the...
features required by a specific relation extraction algorithm and produces the data structure for the extraction algorithm (e.g., sequences or trees of features). These data structures, which we refer to as *operable structures*, are a feature-enriched version of the candidate text segments on which the learning and extraction algorithms will operate. The key challenge is then to provide a unified interface for the extraction of features that supports the wide variety of features and structures that different learning algorithms may require.

The Learning and Extraction components (see Figure 2), which include the Relation Extraction Training, Tuple Extraction, and Relation Extraction Evaluation components, focus on the relation extraction algorithms. First, the Relation Extraction Training component, which we describe in Section 3.5, automatically produces a *relation extraction model* using, as training input, labeled operable structures, which indicate whether the relation of interest holds among their entities. Second, the Tuple Extraction component, which we describe in Section 3.6, uses the model produced in the Relation Extraction Training component to extract tuples corresponding to related entities. Notably, Tuple Extraction performs a classification task over unlabeled operable structures and produces tuples of entities that are likely related. Third, the Relation Extraction Evaluation component, which we describe in Section 3.7, evaluates the relation extraction systems according to a given set of evaluation metrics. When proposing an architecture for the learning and extraction components, the key challenge is to provide a unified interface for different relation extraction techniques to help train, execute, and evaluate the resulting models with minor changes in the code.

Next, we describe the different components discussed so far in detail. Later, in Section 4, we show the implementation of the *Charged* running example in REEL, which demonstrates how our framework helps in writing relation extraction systems in a simple and easy-to-understand manner.

### 3.2 Text Segment Loading

The first component of REEL, Text Segment Loading, is the interface between the original representation of documents (e.g., XML files, file directories, or document indexes) and the internal representation that REEL employs to represent the text segments. The main goal of this component is to detach the formatting subtleties of the text collections from the further operations to be run over them by producing text segments that include mentions to the entities of interest.

The Text Segment Loading component in REEL starts by reading the text documents from their original source (see *Document Loader* in Figure 3). Once the documents are loaded, REEL performs the tagging of entities required by the relation of interest (see *Entity Tagger* in Figure 3). For example, in our running example, the user needs to provide entity taggers to annotate “John Lennon” and “Mark Chapman” as *Person*, “second-degree murder” as *Charge*, and “1981” as *Date* entities. This tagging process can differ substantially according to the text collections (e.g., some datasets, such as ACE 2005 [29], already contain named entity annotations) and available tagging resources (e.g., some toolkits provide off-the-shelf, pre-trained named entity recognition models). REEL supports different types of entity taggers that range from simple loaders from the document files to models from open-source named entity tagging frameworks, such as Stanford NER\(^1\) and E-txt2db\(^2\).

---

Finally, REEL splits the output of the text document loader into text segments according to the needs of the extraction (see Text Segment Splitter in Figure 3). These text segments can be different subsets of the documents, such as sentences or paragraphs, and can vary according to the extraction task. This task may also depend on the text, and REEL accommodates these different scenarios. For instance, some collections such as Aimed provide the input documents already split into sentences.

The text segments resulting from the Text Segment Loading component contain all the entities that can potentially appear in an extracted tuple. In the next section, we discuss how to generate candidate text segments (i.e., text segments that represent a potential relation between a specific set of entities).

3.3 Candidate Generation

Now that we have explained how the Text Segment Loading component produces text segments annotated with entities, we describe the generation of candidate text segments. This component receives as input the tagged text segments and the constraints of the relation. Then, for each input text segment, it produces all the candidate text segments that comply with the input constraints (Figure 4). REEL supports two types of constraints, namely, Entity Constraints and Relation Constraints, that help to conceptually define the relation of interest. REEL offers users the flexibility to define their own constraints and combine them with others of the same type via logical Boolean expressions. We define these constraints as follows:

- **Entity Constraint**: Entity Constraints are the conditions that entities need to satisfy to be part of the relation of interest. Examples of such constraints are the entity type constraints, which define to which types an entity can belong (e.g., all Charged relations must be between a Person, a Charge, and a Date); and non-mandatory constraints, which define whether the occurrence of an entity is optional (e.g., in the Charged relation, we may omit the Date but neither the Person nor the Charge can be omitted).

- **Relation Constraint**: Relation Constraints are the conditions that apply to the relation as a whole. Among these constraints we may find distance constraints (e.g., the distance between entities should not exceed 10 words) and number of occurring entities (e.g., at least two entities need to participate in the relation even if some are optional).

Candidate text segments represent potential tuples for the relation at hand in the text segment, and, in effect, a single text segment may derive multiple candidates. REEL automatically computes these candidate text segments (see Algorithm 1), thus effectively hiding the complexity of the process when multiple

---

Algorithm 1: \(\text{generateCandidates}(\text{Seg}, \text{Rels})\)

1: \(Cands \leftarrow \emptyset\)
2: \(\text{for each relation } \text{Rel} \in \text{Relations} \text{ do} \)
3: \(\quad \text{CandE} \leftarrow \emptyset\)
4: \(\text{for each role } r \in \text{Rel.roles} \text{ do} \)
5: \(\quad C \leftarrow \text{Rel.entityConstraint}(r)\)
6: \(\quad \text{Ents} \leftarrow C.\text{compatibleEntities}(\text{Seg.Entities})\)
7: \(\quad \text{CandE} \leftarrow \text{CandE} \cup \langle r, \text{Ents} \rangle\)
8: \(\text{end for}\)
9: \(\text{RelC} \leftarrow \text{Rel.relationConstraints()}\)
10: \(\text{for each mapping } m \in \text{Mappings}(\text{CandE}, \text{RelC}) \text{ do} \)
11: \(\quad \text{for each entity } E \in m \text{ do} \)
12: \(\quad \quad N \leftarrow \text{Rel}(E.\text{role}, E.\text{entity})\)
13: \(\quad \text{end for}\)
14: \(\quad \text{Cands} \leftarrow \text{Cands} \cup N\)
15: \(\text{end for}\)
16: \(\text{end for}\)
17: \(\text{return } Cands\)

entities and constraints are involved. Algorithm 1 receives as input a text segment with annotated entities and the relations of interest. For example, a valid input would be “John Lennon’s killer, Mark Chapman, was sentenced in 1981 to 20 years to life in prison after pleading guilty to second-degree murder” and the "Charged" relation. Then, for each of the relations of interest (line 2), the algorithm obtains all possible entities in the text segment that comply with their corresponding entity constraints (lines 5-7). In our example, we would obtain “John Lennon” and “Mark Chapman” for Person, “second-degree murder” for Charge, and “1981” for Date. Then, the algorithm produces all possible combinations (see Mappings function in line 10) according to the relation constraints (line 9). For example, if the relation constraints indicate that entities should occur within 10 words and that Date is optional, the possible mappings will be: \(\langle \text{John Lennon, second-degree murder, 1981} \rangle, \langle \text{Mark Chapman, second-degree murder, 1981} \rangle, \langle \text{John Lennon, second-degree murder} \rangle, \text{and } \langle \text{Mark Chapman, second-degree murder} \rangle\). Then, the algorithm creates the candidate sentence and incorporates the corresponding entities (lines 11-13). Finally, the candidate sentences are added to the result set (line 14). Notice that some candidate text segments are subsumed by others in that their tuples are a subset of the tuples in other candidate text segments. For example, the candidate text segment that includes the potential tuple \(\langle \text{Mark Chapman, second-degree murder} \rangle\) is subsumed by another candidate text segment that includes the potential tuple \(\langle \text{Mark Chapman, second-degree murder, 1981} \rangle\). Interestingly, the entities in the tuple may only be related in the absence of the Date entity, and thus reporting only the larger tuple may miss crucial information. Thus, the decision on whether to report these two tuples as different tuples or only return the tuple that includes all entities is left to the relation extraction system implementation per se, as we describe in Section 3.6.

The output of the Candidate Generation component includes sets of entities that may constitute a relation. However, the relation extraction algorithms usually need additional hints, in the form of features, to decide if there is a relation between the entities of a candidate text segment. In the next section, we describe how we enrich candidate text segments with such features.

3.4 Feature Extraction and Operable Structure Generation

So far, we have described how REEL generates the candidate text segments for a relation of interest. These candidate text segments only include the entities that satisfy the entities and relation constraints and hence, are not specific to any relation extraction technique. To make the candidate segments usable for a relation extraction technique, REEL needs to “enrich” these candidates with features (e.g., lemmas, part-of-speech tags, dependency graphs) and produce their corresponding operable structures (Figure 5).
Fig. 5. Operable Structure Generation.

(a) Weka instance as example of vector-based features.
(b) Part-of-speech tags as example of sequence-based features.
(c) Dependency graph as example of graph-based features.

For this, REEL considers the requirements of the relation extraction technique of interest, including its features and how to store them in its operable structure. Such information is carried in what we refer to as the core of the relation extraction technique. In particular, cores are responsible for two crucial tasks. First, they guarantee that operable structures include the mandatory features. For example, if our relation extraction system requires tokens and their part-of-speech tags, the core must report these features as mandatory. Second, cores guarantee that the operable structures are represented in appropriate data structures for training. For example, if the training algorithm requires numeric vectors to represent each training instance, the core must store the operable structures in that form. To define cores along with their mandatory features, REEL provides a simple interface that is shared across relation extraction systems. This interface is general enough to enable the incorporation of additional features to existing cores, which in turn helps to effortlessly experiment with these features in other extraction systems.

Most of the features for relation extraction are typically shared across techniques, as discussed in Section 2, and as such, should be computed uniformly. For this, REEL provides Feature Extractors (Figure 5). Feature extractors respond to a unified interface and can be re-used for different cores with no additional effort. Each core is then responsible for storing the extracted features in their own format, as discussed. REEL defines, but is not limited to, three types of features, namely, vector-based, sequence-based, and graph-based, which we define as follows:

- Vector-based features refer to the most common feature representation in classification tasks. In this representation each characteristic of the candidate text segment is represented as a number (usually in a binary representation) and the entire set of *m* features forms an *m*-dimensional space. Several external...
tools use this representation, as is the case for the machine learning toolkit Weka⁴ and its Instance
object. For example, we can create in REEL an Instance-based operable structure as illustrated in
Figure 6a, where each @attribute line corresponds to a feature and the last line corresponds to pairs
that include the index of the feature and a Boolean value that indicates whether the feature occurs in
the text or not.

- Sequence-based features refer to the text segment features that are modeled as sequences. As an ex-
example, consider part-of-speech tags, which produce a sequence of features, one for each token in the
sentence. Figure 6b illustrates the part-of-speech tags of the text excerpt “Mark Chapman was sen-
tenced in 1981 to 20 years to life in prison after pleading guilty to second-degree murder,” where each
part-of-speech tag corresponds to one term or punctuation symbol in the text excerpt.

- Graph-based features refer to the text segment features that are modeled as a graph. As an example,
consider dependency graphs, which move away from the linearity of sequence-based features to a more
complex feature space. Figure 6c illustrates the dependency graph of the text excerpt “Mark Chapman
was sentenced in 1981 to 20 years to life in prison after pleading guilty to second-degree murder.” As
we see in this figure, there are (directed) connections between part-of-speech tags that together form a
graph of features.

Interestingly, dependencies between features are common in information extraction. For example, the
dependency graph needs the part-of-speech tags to be computed first (see Figure 6c). To avoid computing
the same features multiple times, which may incur a substantial overhead, REEL provides feature caching,
so that each feature set is computed only once.⁵ Specifically, REEL maintains two caches: one cache stores
the features extracted from the candidate text segment, which depend on the tagged entities (e.g., distance
between entities), whereas the other cache stores the features derived from the text segment, which do not
depend on the entities (e.g., tokens or part-of-speech tags). Such a distinction is necessary, since the cache
for text segment features is stored only once and shared across its (derived) candidates.

After Feature Extraction and Operable Structure Generation, the documents are fully processed and
can serve as input to relation extraction algorithms. In the next section, we explain how to train relation ex-
traction systems with REEL given the results of the Feature Extraction and Operable Structure Generation
component.

### 3.5 Relation Extraction Training

As we have argued throughout this paper, there are multiple techniques to train relation extraction systems
and our framework should be flexible enough to support different learning settings (e.g., techniques, train-
ing algorithms, existing libraries). REEL includes two concepts, namely, engine and model, that together
with the core (Section 3.4) provide these capabilities. In a nutshell, the engine runs the training algorithm:
Given a set of labeled operable structures—generated under the constraints of a core—the engine produces
a relation extraction model (Figure 7). The engine must then be aware of the internal representation of the
operable structure to guarantee the effective usage of the included features. For example, if the operable
structure is represented with graph-based features (Section 3.4), the learning algorithm in the engine should
be aware of such structure to manage it successfully.

Engines bring flexibility into the development of relation extraction systems along several dimensions.
First, engines allow developers to use their machine learning libraries of choice (e.g., JlibSVM⁶, Weka)
with no restrictions, or to develop their own techniques to learn models (e.g., pattern-based techniques
such as PRDualRank [15]). Second, engines enable the modification of several learning decisions for a
given core with minimal effort. For example, for an SVM-based core, we can customize existing engines
with different learning strategies (e.g., batch vs. online) and different learning parameters (e.g., convergence
criterion).

The flexibility of the engines described above is carried over to the learned models, which include
the necessary information to perform the classification task. For example, if the relation extraction system

---

⁴ http://www.cs.waikato.ac.nz/ml/weka/

⁵ Such caching is possible because feature extractors are deterministic (i.e., they produce the same output given the
same text segment).

⁶ http://dev.davidsoergel.com/trac/jlibsvm/.
relies on an SVM-based classifier [21], the model will include the support vectors, whereas if the relation extraction system relies on patterns, the model will include the learned patterns and how they are matched with the text.

The next section discusses how to use the relation extraction model that results from the Relation Extraction Training to extract tuples from text.

### 3.6 Tuple Extraction

As we discussed in Section 3.1, REEL performs the tuple extraction as a classification task: REEL uses the model learned by the Relation Extraction Training component (Section 3.5) to classify unlabeled operable structures as positive (i.e., their entities are related) or negative (i.e., their entities are not related). Therefore, during tuple extraction, REEL observes a set of unlabeled operable structures and whenever one of these structures is classified as containing related entities, REEL produces a tuple with those entities (see Figure 8).

For example, consider a relation extraction model that receives operable structures based on dependency graphs and decides if their entities are an instance of the Charged relation in our running example. For our example sentence from Section 2, the operable structures would resemble that in Figure 6c. Since there are four candidate text segments for this sentence (see Section 3.3), the model would evaluate all the alternatives and output as relation instances (Mark Chapman, second-degree murder) and (Mark Chapman, second-degree murder, 1981). Furthermore, the model would attach information on tuple subsumption to the tuples, so that applications or end users can decide what tuples to consider.

Beyond the extraction of individual relations, we now discuss another important feature of REEL, namely, how it supports the comprehensive experimental evaluation of alternative relation extraction models.
3.7 Relation Extraction Evaluation

In addition to the definition of individual relation extraction systems, REEL supports the experimental evaluation and comparison of multiple relation extraction systems, a task of critical importance to facilitate research in information extraction. For this, REEL provides the notion of Evaluators. An evaluator helps to: (i) compare different configurations of the same relation extraction system to find the best performing setup; (ii) compare the performance of a relation extraction system over different text collections to demonstrate robustness; and (iii) compare different relation extraction systems over the same text collection to identify the best performers. To achieve this, the evaluator receives as input both the real and predicted labels of a set of operable structures together with their prediction properties (e.g., confidence of the output), and returns the measured performance (see Figure 9a).

The REEL Relation Extraction Evaluation component considers two important factors for the evaluation of relation extraction systems, namely, how to split sets of instances (e.g., documents, candidate sentences, or operable structures) and what performance measures to use. Regarding the first factor, REEL provides the generation of instance splits to facilitate the generation of principled training and test sets, as illustrated in Figure 9b. Specifically, REEL is able to create these splits over sets of documents, candidate text segments, and operable structures, thus offering different evaluation capabilities. For example, splitting over candidate text segments allows segments from the same text document to belong to both the training and test sets, which would not be possible if we could only split over entire documents. REEL currently provides two types of splitting, namely, percentage split, which splits elements at a given fraction (e.g., 70% for training and the rest for testing), and \( K \)-fold split, which splits elements into \( K \) equally-sized groups suitable for \( K \)-fold cross validation [28].

Regarding the choice of performance measures, REEL provides common measures for binary classifiers, namely, precision, recall, and F-measure, which can be used in the evaluator, as illustrated in Figure 9a. In addition to these metrics, REEL allows the computation of micro and macro averages over them, which is crucial during the evaluation of relation extraction systems that extract multiple relations simultaneously. Altogether, these performance measures enable the principled evaluation of relation extraction systems. REEL also provides support for additional performance measures. Specifically, users can implement measures that take as an input a set of operable structures along with their true labels, predicted labels, and other prediction properties (e.g., confidence of the prediction, probability of entities being related), and ultimately produce an output value. Moreover, users can define new prediction properties that the models can then explicitly report. These new prediction properties would incur source code changes; however, these changes would only affect the models, since that is where the actual prediction takes place.

Now that we have discussed the main components of REEL, we show how we use REEL to develop a relation extraction system for our running example.

4 Using REEL in Practice

In the previous section, we introduced the architecture of REEL and described its components in detail. We now illustrate how to use REEL in practice by providing an end-to-end implementation of a typical relation extraction system for our Charged relation example. Specifically, we show the Java source code to perform text segment loading and candidate generation, feature extraction and operable structure generation, relation extraction training, tuple extraction, and relation extraction evaluation.

To address these tasks a REEL user should use code templates provided in the toolkit. Then, the users could adapt these templates to their own needs by using different implementations of document loaders, constraints, feature generators, engines, and cores. Users can also implement their own version of these elements to exploit new techniques. First, as explained in Sections 3.2 and 3.3, we load the text segments of an input collection and derive their candidates (Sample Code 1.1). We define the Charged relation along with its constraints, which we save via serialization for future use (lines 2-9). We then load the documents from our collection (lines 11-13), each with their corresponding text segments. Users can write their own collection loaders, which are only required to produce documents in the REEL format as output. Next, we create a text splitter (lines 15-16), which defines the scope of the candidate text segments (e.g., sentences) and, in turn, where the (potential) tuples can occur. Finally, we use the REEL candidate generator to produce
the candidates from all documents in the collection (lines 17-21) and save them for the following steps (line 22).

**Sample Code 1.1. Candidate Generation.**

```java
// Define relationships and their constraints
String r = "CHARGED", t1 = "PER", t2 = "CHAR", t3 = "DAT";
RelationshipType rT = new RelationshipType(r, t1, t2, t3);
rT.setConstraints(new EntityTypeConstraint("PER"), t1);
rT.setConstraints(new EntityTypeConstraint("CHAR"), t2);
rT.setConstraints(new EntityTypeConstraint("DAT"), t3);
Set<RelationshipType> rTs = new HashSet<RelationshipType>();
rTs.add(rT);
SerializationHelper.write("rTypes.ser", rTs);
// Use a predefined Document Loader
Loader l = new MyLoader(rTs);
File AD = new File("/train/");
Dataset<Document> col = new Dataset<Document>(l, AD, false);
// Define sentence splitter for the candidate generator
String sp = "model.bin";
OpenNLPMESplitter spl = new OpenNLPMESplitter(sp);
CandidatesGenerator g = new CandidatesGenerator(spl);
Set<CandidateSentence> cand = new HashSet<CandidateSentence>();
for (Document d : col) {
candidates.addAll(g.generateCandidates(d, rTs));
}
SerializationHelper.write("train.ser", cand);
```
Once we generate the candidates, we must enrich them with features to produce the operable structures, as described in Section 3.4 (Sample Code 1.2). We start by retrieving the recently generated candidates (line 1). Then, we define the core (Section 3.5), which determines the tuple extraction algorithm (ShortestPathKernel) which implicitly defines its mandatory features. In this example, we add part-of-speech tags to the mandatory set of features (lines 6 and 7); however, we also included tokenization (line 4) and chunking (line 5) features, which are required to compute the part-of-speech tags. We then save this configuration (line 8), which we will use later during training and that we can use to produce operable structures from other candidate sentences for the Charged relation. Finally, we use the REEL StructureGenerator to produce the operable structures from the candidate sentences (line 9), and save them for later use (line 10).

Sample Code 1.2. Operable Structure Generation.

```java
Set<CandidateSentence> cand = (Set<CandidateSentence>) SerializationHelper.read("train.ser");
StructureConfiguration conf = new StructureConfiguration(new ShortestPathKernel());
FeatureGenerator<SequenceFS<Span>> tok = new OpenNLPTokenizationFG("modelT.bin");
FeatureGenerator<SequenceFS<Span>> fgCh = new EntityBasedChunkingFG(tok);
FeatureGenerator<SequenceFS<String>> fgChSt = new SpansToStringsConversionFG(fgCh);
FeatureGenerator<SequenceFS<String>> fgPOS = new OpenNLPPartOfSpeechFG("modelPOS.bin", fgChSt);
conf.addFeatureGenerator(fgPOS);
SerializationHelper.write("conf.ser", conf);
Set<OperableStructure> trD = StructureGenerator.generateStructures(cand, conf);
SerializationHelper.write("optr.ser", trD);
```

As described in Sections 3.5 through 3.7, we can use the operable structures to learn the relation extraction model, to extract tuples from text documents, and to perform a thorough evaluation of the relation extraction system. We now illustrate how REEL handles these operations.

For training, we load the operable structures, which are required to be labeled\(^7\), and learn a relation extraction model (Sample Code 1.3). Specifically, we load the definition of the relation that we created during candidate generation (line 1), the configuration of the features (line 2), and the operable structures (line 3) both defined during operable structure generation. Next, we create an engine (line 5), which we described in Section 3.5. The engine includes the learning algorithm to train the relation extraction model from the operable structure and thus, must support its internal structure (e.g., kernel), as discussed. REEL allows users to define their own engines using different machine learning toolkits. Finally, we train (line 6) and save the learned model (lines 8-9), which we can use for tuple extraction and evaluation, as we see next.

Sample Code 1.3. Relation Extraction Training.

```java
Set<RelationshipType> rTs = (Set<RelationshipType>) SerializationHelper.read("rTypes.ser");
StructureConfiguration conf = (StructureConfiguration) SerializationHelper.read("conf.ser");
Set<OperableStructure> trD = (Set<OperableStructure>) SerializationHelper.read("optr.ser");
Engine eng = new JLibSVMBinaryEngine(conf, rTs);
```

\(^7\)The labels of the operable structures can be loaded from the input text collection (e.g., from the ACE 2005 collection [29]) or manually annotated.
Model svmM = eng.train(trD);

//Finally, we can store the model in order to use it later
String modF = "CHARGEDModel.svm";
SerializationHelper.write(modF, svmM);

For tuple extraction (Sample Code 1.4) we employ all the capabilities described thus far. We first load the recently learned model (lines 1-2) that we then use to define our relation extraction system (line 3). This relation extraction system, which is provided in REEL, enables users to directly plug in their learned models and have a fully functional relation extraction system that can be used in application building settings. In addition to the model, the relation extraction system receives a text splitter that defines the scope of the extraction (e.g., sentences), just as we did for candidate generation. To put the system to work, we load the documents from which we want to extract tuples (lines 4-7). (In this example we use the same type of dataset for training than we do for testing but we could easily plug in any other dataset and loader.) Finally, we iterate over the documents and print the extracted tuples (lines 8-10). The extractTuples method, provided in the relation extraction system, hides the complexity of processing the given document to obtain its operable structures and perform the extraction task.

**Sample Code 1.4. Tuple Extraction.**

```java
String modF = "CHARGEDModel.svm";
Model svmM = (Model) SerializationHelper.read(modF);
ClassifierBasedRelationshipExtractor ext = new ClassifierBasedRelationshipExtractor(svmM, new OpenNLPMESplitter("en-sent.bin"));
Set<RelationshipType> rTs = (Set<RelationshipType>) SerializationHelper.read("rTypes.ser");
Loader l = new MyLoader(rTs);
File AD = new File("/test/");
Dataset<Document> col = new Dataset<Document>(l, AD, false);
for (Document d : col) {
    System.out.println(ext.extractTuples(d));
}
```

The output of the code above for a document that includes the text excerpt “John Lennon’s killer, Mark Chapman, was sentenced in 1981 to 20 years to life in prison after pleading guilty to second-degree murder” in our running example will look like:

```
Charged[Person(Mark Chapman);
    Charge(second-degree murder);
    Date(1981)]
```

Finally, for evaluation (Sample Code 1.5), we require the learned model that we want to evaluate, as well as the labeled operable structures that will represent the ground truth. We start by loading the learned model (lines 1-2) as well as the labeled operable structures (lines 3-9). We then define the evaluator (line 10), which we described in Section 3.7, and the performance measures that we will consider in the evaluation. Here, we measure our model using recall, precision, and F-measure (lines 11-16), which are already implemented in REEL, although the user can incorporate other measures that can be used directly, as we explained. Finally, we invoke the printEvaluatorReport method (line 17), which outputs the recall, precision, and F-measure values, as requested.

**Sample Code 1.5. Relation Extraction Evaluation.**

```java
String modF = "CHARGEDModel.svm";
Model svmM = (Model) SerializationHelper.read(modF);
List<String> tF = FileUtils.readLines("testfiles.ser");
```
List<OperableStructure> l = new ArrayList<OperableStructure>();
List<OperableStructure> oS;
for (String s : tF) {
    oS = (List<OperableStructure>)SerializationHelper.read(s);
    l.addAll(oS);
}
Evaluator eval = new Evaluator();
Measure rec = new Recall();
eval.addMeasure(rec);
Measure pre = new Precision();
eval.addMeasure(pre);
Measure f = new FMeasure(1.0);
eval.addMeasure(f);
eval.printEvaluationReport(l, svmM);

In this section, we walked through the steps required to develop and evaluate a typical relation extraction system in REEL. As shown, the code needed to produce such an extraction system within REEL is simple and easy to understand. This makes REEL a powerful framework to enable the deployment and evaluation of relation extraction systems for both application building and research.

5 Conclusions and Future Work

In this paper, we introduced REEL, an open-source framework to easily develop and evaluate relation extraction systems. REEL provides an end-to-end infrastructure that handles relation extraction as a classification task, and leverages powerful existing toolkits for both text processing and machine learning subtasks. Moreover, REEL effectively addresses the complex requirements of relation extraction and helps developers and researchers produce simple and easy-to-understand source code for their relation extraction systems. As part of the REEL distribution—as open source under the General Public License Version 3 (GPLv3) license, at http://reel.cs.columbia.edu—we have included ready-to-use relation extraction systems (e.g., [9, 10]); we have also integrated several text processing and machine learning toolkits, to illustrate how to incorporate and leverage external algorithms and toolkits. As future work, we are currently expanding the existing selection of learning algorithms and execution strategies for extracting tuples from large text collections, which are critically important for relation extraction. Specifically, we are: (i) incorporating a family of online learning algorithms (e.g., [27]) that enable an efficient and incremental learning of the relation extraction systems; and (ii) including modules for the efficient, query-based execution of relation extraction systems over large text collections (e.g., QXtract [4] and FactCrawl [7]).

Acknowledgements

This work was supported by Fundação para a Ciência e a Tecnologia, under Project PEst-OE/EEI/LA0021/2013 and Ph.D. Grant SFRH/BD/61393/2009. This material is also based upon work supported by the National Science Foundation under Grant IIS-08-11038.

References