EDISON: Feature Extraction for NLP, Simplified

Mark Sammons[1], **Christos Christodoulopoulos**[1], **Parisa Kordjamshidi**[1], **Daniel Khashabi**[1], **Vivek Srikumar**[2], **Paul Vijayakumar**[1], **Mazin Bokhari**[1], **Xinbo Wu**[1], **Dan Roth**[1]
[1]Department of Computer Science,
University of Illinois, Urbana-Champaign.
{mssammon,christod,kordjam,khashab2,pvijaya2,mbokhar2,xinbowu2,danr}@illinois.edu

[2] School of Computing,
University of Utah.
svivek@cs.utah.edu

Abstract

When designing Natural Language Processing (NLP) applications that use Machine Learning (ML) techniques, feature extraction becomes a significant part of the development effort, whether developing a new application or attempting to reproduce results reported for existing NLP tasks. We present EDISON, a Java library of feature generation functions used in a suite of state-of-the-art NLP tools, based on a set of generic NLP data structures. These feature extractors populate simple data structures encoding the extracted features, which the package can also serialize to an intuitive JSON file format that can be easily mapped to formats used by ML packages. EDISON can also be used programmatically with JVM-based (Java/Scala) NLP software to provide the feature extractor input. The collection of feature extractors is organised hierarchically and a simple search interface is provided. In this paper we include examples that demonstrate the versatility and ease-of-use of the EDISON feature extraction suite to show that this can significantly reduce the time spent by developers on feature extraction design for NLP systems. The library is publicly hosted at https://github.com/IllinoisCogComp/illinois-cogcomp-nlp/, and we hope that other NLP researchers will contribute to the set of feature extractors. In this way, the community can help simplify reproduction of published results and the integration of ideas from diverse sources when developing new and improved NLP applications.

Keywords: Natural Language Processing Tools, Feature Extraction, Machine Learning, NLP, Reproducibility

1. Motivation

The process of designing and building Natural Language Processing (NLP) applications that use Machine Learning (ML) techniques has a general structure that developers must follow. Given a specific task – co-reference resolution, for example – the developers acquire a corpus that exemplifies this task. Next, they consider what existing tools and data resources might contribute to a solution to this problem, and decide on the set of data structures that will support the task. As part of this decision, they will decide on what machine learning models would be most appropriate, as this may affect other design choices (programming language, data structures, input resources). As they build the application, a significant part of their effort will go into designing features that will be extracted from the source data and used as inputs to the machine learning component, and then implementing the corresponding feature extraction code. Often, other researchers have worked on the same task and published their own approaches, in which case the developers may try to recreate the features described in these publications. Frequently, the published description is terse due to space constraints, and many small but important details are omitted. In the best case, the authors have published their code, but even then it may take significant effort to understand the code and isolate the feature extraction, and to port it to a different programming language and/or a different set of data structures.

A number of NLP and ML packages provide libraries that support various aspects of this process. NLP packages provide components to generate inputs for a new problem, together with data structures that can be used programmatically to represent them. ML packages generally specify an input representation based on feature vectors at a distant remove from NLP data structures. Few packages to date have tried to bridge the gap between these two resources, in which ML features can be specified and extracted from NLP data structures and either written to file in a generic format, or programmatically fed to a ML system.

EDISON is a feature extraction library based on generic NLP data structures from the University of Illinois Cognitive Computation Group (CogComp)'s core NLP libraries (illinois-core-utilities). It provides an intuitive and versatile feature extraction API that can be used programmatically or as a stand-alone application generating files in JSON format for ML input. It contains reference implementations for feature extractors used in several CogComp NLP applications: Part of Speech (Roth and Zelenko, 1998), Chunking (Punyakanok and Roth, 2001), Named Entity Recognition (Ratinov and Roth, 2009), and Semantic Role Labeling (Punyakanok et al., 2008). We plan to add reference implementations for Coreference (Peng et al., 2015) and for Wikifier (Cheng and Roth, 2013).

The key contributions of this work are: 1. A Java library, EDISON, that provides a simple, intuitive programmatic framework to apply feature extraction functions to generate new feature representations, and which comes with a simple search interface to help users find existing features that meet their needs. 2. A simple application wrapper for EDISON that can be used as a standalone component to generate JSON-format feature files. 3. A large suite of feature extractors derived from existing state-of-the-art NLP tools

that serve as reference implementations for those tools' features, and which facilitate the reproduction of state-of-the-art-results for those tools. 4. A feature classification and naming scheme that can be used in textual descriptions of NLP/ML systems so that they become easier to replicate even when their source code is not available.
This paper describes EDISON and its use in developing NLP applications with machine learning elements.

2. Feature Extraction

Feature extraction occurs in the context of machine learning applications. Machine learning algorithms are used to automatically generate decision functions (*classifiers*) that would be very hard or impossible to implement programmatically. They do this by building statistical models that map some representation of the input data to some predefined set of meaningful outputs. For example, a spam classifier would take as input an email and output either a discrete label (such as **spam** or **not spam**), or a score indicating likelihood of being spam (perhaps normalized to the form of a probability). A document classifier would take as input a digital document and might output one or more labels indicating topics (such as **politics**, **sport**, **religion**). An image processor would take a digital image and might return names of objects and specify the regions of the image in which they occurred. A named entity recognizer (NER) will take digital text as input and return a set of annotations indicating which sequences of words represent names and what type of entity each represents. Other applications may produce complex, highly structured output labels (e.g. groups of objects representing body parts belonging to different people in an image, or representations of events described in text), or might not label inputs, but group them into similar sets (clusters). Data inputs are referred to here as *examples*, and the desired output values as *labels*.
However, learning algorithms require examples in specific formats. Typically, examples are represented as a list of *features*; when training or evaluating the learned algorithm, each example will also be associated with any relevant labels. These features represent an abstraction over the raw input at the level of whether or not (or to what degree) some characteristic is present for a given example. For the spam classifier, features might represent the presence of specific words or phrases (e.g. "fast cash"), or the number of unknown words in the email, or the domain of the sender. The application developer specifies the relevant abstractions, and must map the raw input into these features. Often, this involves processing the raw input with other tools. We will now sketch an NER implementation to flesh out some details that will help illustrate key concepts in feature extraction. This application takes a plain text document as input, iterating over the sequence of words in the text and producing a corresponding sequence of examples. Each example is an abstract representation of the corresponding word. Each example is classified by the machine learned classifier to assign it an entity type, and if that type is not "NONE", whether or not that word is at the beginning of an entity or not. Figure 1 shows the corresponding labels for a short word sequence. We will call the word that is currently being processed the *focus*.
One useful feature to distinguish between (proper) names and non-names includes capitalization, but it is not sufficient and does not help determine the entity type. One possibly useful feature is the type of word that appears before or after the focus; if it is a verb, it may be more likely that the focus represents a person or organization rather than a location. Typically this information would be acquired by processing the input text with a Part of Speech tagger. Other tools might also be used to provide different abstractions – the role of the focus in the sentence relative to the main verb (from a syntactic parser), or position within a phrase (from a shallow parser). Generally, these information sources are collected in a set of data structures which are then processed to generate examples by composing lists of features extracted from specified patterns in the data structures. These features can be very expressive and carefully specified, and in such cases tend to be time-consuming to write. The classifier can only make appropriate distinctions if the relevant information is exposed (if every example has the same representation, the classifier can make no useful prediction; and some features are more generally characteristic of specific focus items than others), and so the application developer has good reason to spend time carefully specifying potentially useful features.

3. EDISON

EDISON is a Java library to support feature extraction in Natural Language Processing. It uses the data structures from `illinois-core-utilities`[1], another Java library from the Cognitive Computation Group[2]. Together, these expand on an older version of EDISON described in (Clarke et al., 2012).

3.1. Data Structures

The main data structure used by EDISON is called a `TextAnnotation`. It is used to represent a piece of text, such as a document, and collects all NLP annotations for that text. NLP annotations such as tokens, phrases, sentences, and other text-related constructs are represented in terms of spans of tokens/characters (`Constituents`) and edges (`Relations`) linking spans to each other. Each annotation source is represented as an independent `View` over the original text, that collects the `Constituents` and `Relations` generated by that source. `Constituents` and `Relations` can be labeled and assigned scores, and `Constituents` also allow an arbitrary number of attributes (key/value pairs) to be specified. The CogComp group has successfully used this representation to support NLP tasks ranging from part-of-speech tagging to semantic role labeling and co-reference resolution. Figure 2 illustrates these data structures as they might be used to represent token, part-of-speech, named entity, numerical quantity, and semantic role information for a short piece of text. In the context of a learning application, the developer populates these data structures using off-the-shelf NLP components. The CogComp NLP tools use these data structures

[1] `https://github.com/IllinoisCogComp/illinois-cogcomp-nlp/`
[2] `http://cogcomp.cs.illinois.edu`

John	Smith	said	Jane	Smith	bought	four	cakes	and	two	apples
B-PER	**I-PER**	**O**	**B-PER**	**I-PER**	**O**	**O**	**O**	**O**	**O**	**O**

Figure 1: Sequence of words and their labels in a named entity recognition (NER) task. **B** represents whether the focus word is "beginning" a particular label, **I** represents the word being "inside" the label, and **O** ("outside") represents that a word has no NER label.

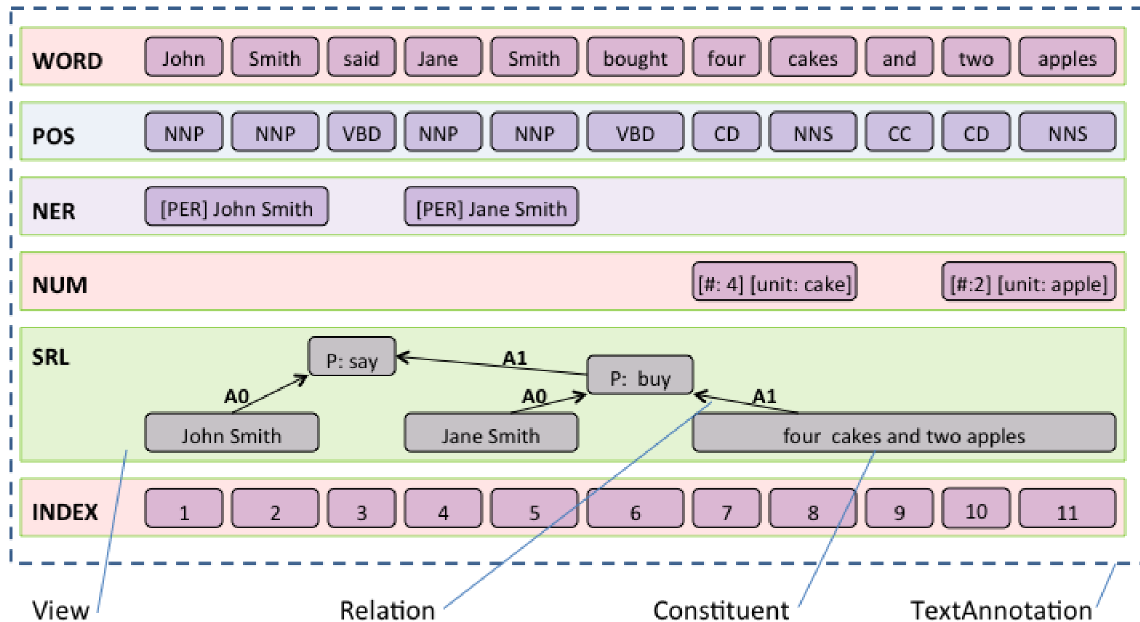

Figure 2: Data structures used by EDISON, showing the `TextAnnotation` of a sample text. The horizontal boxes are `Views` which represent different annotation sources: tokenization (**WORD**), part-of-speech tagging (**POS**), named entities (**NER**), numerical expressions (**NUM**) and semantic role labeling (**SRL**). Each `View` is comprised of `Constituents` and `Relations` (edges between `Constituents`). Notice the difference in the representation of the NER annotation compared to that of Figure 1: in `TextAnnotation` the adjoining labels **B-PER** and **I-PER** have been replaced by a two-token `Constituent`.

natively, but other tools could also be used to populate them with relatively little effort. `TextAnnotation` can also be populated from a JSON-format file.

In our example of an NER application, the application recognizes sequences of tokens representing individual named entities and assigns each entity a type. The end-to-end application would classify each word into either **Begin**, **Inside**, or **Outside**, representing its position inside or outside a named entity, and also whether a name represents a **Person**, **Organization**, or **Location**. We will assume that each word can belong to only one entity. The application would then combine the individual predictions to identify the entity boundaries, and also rationalize any inconsistent predictions (for example, a word labeled **Inside** but which does not follow a word labeled **Begin**, or a **Begin-Person** followed by an **Inside-Location**). The application would also track the correspondence between the individual words and the learned classifier decisions, allowing the user to provide the resulting annotation to other applications or to generate visual output for an end user.

3.2. Feature Extraction

EDISON supports a range of feature types, from the standard combinations provided by Fex (Cumby and Roth, 2003; Cumby and Roth, 2000) and FEXTOR (Broda et al., 2013) – such as collocations of constituents within a specified context window, features combining different levels of annotation, features based on dependency parse paths between constituents – to more specialized features proven useful in more complex NLP tasks like semantic role labeling (e.g. subcategorization frames, or *projected path* (Toutanova et al., 2008)). These features are extracted from a `TextAnnotation` data structure populated with the appropriate source annotations: a feature extractor for part-of-speech bigrams will extract features only from a `View` populated with that information.

In a supervised learning setting, the application uses labeled data to extract examples of the data items it wants the learning algorithm to classify. In the context of EDISON, we assume that this labeled data has been used to construct a `View` that contains a representation of the focus items as `Constituents`. The application code iterates over these focus items in the `TextAnnotation`, extracting features for each. In our NER example, these focus items are mapped to the word level, so we will make predictions at the word level. For each word, the feature extraction code runs a set of feature extractors on the `TextAnnotation` representation of that word (over various different `Views`) and generates a set of active features. During training or evaluation of the learned classifier, the code also extracts the true label for the current word. This list of features, together with any labels, is the *example* provided to the learn-

ing algorithm. EDISON provides some utilities to generate output compatible with two popular learning frameworks – see Section 3.3.

EDISON's feature extractors all implement the `FeatureExtractor` interface (see Figure 3) which has two methods: `getName()`, which returns the name of the feature (to be used as part of the feature's lexical representation, which will uniquely identify features extracted by the implementing class), and `getFeatures(Constituent c)`, which apply some logic to the `Constituent` it is passed to generate a set of features. These `Feature` objects store a feature name and an optional numerical term that may be used to represent a weight, count, or confidence.

Figure 3 shows a code snippet for some typical NLP features extracted using the EDISON library. Consider the NER constituent for "Jane Smith" in the example shown in Figure 2: when the feature extractor is called using the "Jane Smith" NER constituent as its argument, it will extract the features `[NNP-NNP,NNP-VBD,John-Smith,Smith-said]`. If this feature extractor were instead called with the word constituent for "Jane" as its argument, it will extract the same features because its left context is the same. However, a feature extractor that used the right context of the focus would produce different results, since the NER constituent covers an additional word, and the right context for that constituent will begin after the second word.

For a given focus constituent, the application will pool the sets of features produced by all its feature extractors plus the target label for the focus to create a single example for the learning algorithm. In our working example, if we have only the single feature extractor, the first word of the entity will be assigned the label **Begin-Person** and the application will create an example for this word of the form `B-PER, [NNP-NNP,NNP-VBD,John-Smith,Smith-said]`. This example, or some alternative representation of it (usually indexed to a set of integers), would be passed to the learning algorithm for processing.

3.3. Programmatic integration with learning frameworks

Programmatically, EDISON's feature extractors can be easily integrated into JVM-based learning frameworks such as LBJava (Rizzolo and Roth, 2010), Mallet (McCallum, 2002), Weka (Hall et al., 2009), and Saul (Kordjamshidi et al., 2015). Here we will show programmatic integrations for the CogComp tools (LBJava/Saul) and provide a file-based integration for the other ML tools. We plan to create programmatic interfaces for these tools in the near future.

For LBJava, features can be trivially wrapped as `Classifier` objects and used directly in the LBJava definition file. Figure 4 illustrates the way EDISON feature extractors can be used in the LBJava language, by creating a class that inherits from LBJava's `Classifier` interface to wrap the feature extractor so that it can be used directly in an LBJava definition file. The integration with Saul (Figure 5) is even easier since there is no need for a wrapper. Instead, EDISON's extractors can be directly called inside Saul's `property` definitions.

3.4. Writing to File for use in ML Frameworks

Many learning frameworks such as Mallet (McCallum, 2002) and Weka (Hall et al., 2009) support the use of formatted files to provide input to their learning algorithms. EDISON provides classes that write the extracted features to sparse vector formats used by each tool (`WriteToSvmLight` and `WriteToXrff` respectively).

3.5. Developer Support

EDISON's feature extractors have been carefully organized, named, documented, and assigned meaningful keywords (see section 4.). The goal is to make it as easy as possible for developers to find existing extractors that meet their needs. For convenience, we have created additional classes that group the feature extractors that correspond to some of CogComp's existing NLP tools[3].

To improve ease of use, EDISON comes with a simple search tool that opens a web browser and allows the user to search for feature extractors using keywords. This tool uses the in-code comments, keywords and EDISON's hierarchical package structure to identify the set of features most relevant to the user's query. Figure 6 shows a screenshot of the search interface. The search tool returns a set of snippets that link to the Javadoc descriptions for the relevant feature extractors, and for each snippet a link to a concrete example of use for the corresponding feature extractor.

4. Methodology

This section provides information about the way we organized the feature extractor classes and describes the process we used to port the feature extractors of some well-known CogComp NLP tools.

4.1. Organization and Navigation

We use descriptive names for feature extractors based on the types of information they use, and the way these pieces of information are combined. There are five key characteristics of the feature extractors we processed:

1. The source `View`(s) in the `TextAnnotation` (such as `Word`, `POS`, `NER`, or `SRL`).
2. Whether n-grams over features are taken (unigrams assumed if none specified). If only one n is specified (e.g. `Bigrams`) only that n-gram will be returned. Alternatively an arbitrary number of intermediate n-grams can returned by specifying a sequence of $0 \leq i \leq n$ terms (e.g. `TwoThreeFourGram`).
3. The operator for combining multiple extracted features. The default is disjunction (each feature is added separately to the list of returned features). Conjunction indicates that the set of features in the feature name are concatenated before being added to the list of returned features (so `POSWordConjThreeBefore` called on the word "library" in Figure 7 will return the single feature `NN-Construction`).
4. The relative position of the extracted feature, or the size of the context window in which features are extracted. If unspecified, assume just the focus is used.

[3] see the documentation for EDISON

```java
/**
 * An interface that specifies what a feature extractor should do.
 * In general, a feature extractor looks at a {@code Constituent} of a {@code
 * TextAnnotation} and generates a set of features for that
 * constituent.
 */
public interface FeatureExtractor {
    Set<Feature> getFeatures( Constituent c ) throws EdisonException;
    String getName();
}

/**
 * Extract part-of-speech and word bigrams from 3 words before the
 *    target constituent
 * @Keywords pos, word, bigrams, before
 */
public class POSAndWordBigramThreeBefore {
    public Set<Feature> getFeatures( Constituent c ){ //focus constituent
        NgramFeatureExtractor posBigrams =
            NgramFeatureExtractor.bigrams( WordFeatureExtractorFactory.pos );
        NgramFeatureExtractor wordBigrams =
            NgramFeatureExtractor.bigrams( WordFeatureExtractorFactory.word );
        // ContextFeatureExtractor specifies the window before and after
        //    the target within which to extract features
        ContextFeatureExtractor cfe = new ContextFeatureExtractor( 3, false, false );
        cfe.addFeatureExtractor( posBigrams );
        cfe.addFeatureExtractor( wordBigrams );
        return cfe.getFeatures( c );
    }
}
```

Figure 3: Sample code snippets showing the feature extractor interface and illustrating a feature extractor definition in EDISON. The extractor returns part-of-speech and word bigrams in a window of 3 tokens before the focus constituent.

If a single focus at a fixed relative position, indicate the position (e.g. `TwoBefore`). A window implies features are extracted at every position within that window: `WindowK` indicates a symmetric window of size K; if asymmetric, specify `KBefore` and/or `NAfter`. In all cases, K specifies relative positions in the `View` from which the focus is selected.

5. Whether the feature records the position of an extracted feature relative to the focus `Constituent`. If recorded, this results in much more specific features by distinguishing between e.g. the word "the" appearing before the focus and the same word appearing after the focus, which are then marked as distinct features.

To facilitate immediate understanding of their purpose, extractor names are based on these characteristics and follow a consistent order: source `Views` (in decreasing order of "expense" of processing), then level of n-grams used, then combination operator, then context window size, then position. The feature extractor in Figure 3 generates a set of part-of-speech bigrams and a set of word bigrams in the context preceding the focus `Constituent`, within a window of 3 `Constituents` in the same `View` as the focus `Constituent`. Following these naming guidelines, the feature extractor is named `POSAndWordBigramsThreeBefore`. Since the position of the bigrams relative to the focus is not recorded, the term "Position" is not used in the name; if the position *were* recorded, the name would be `POSAndWordBigramsPositionThreeBefore`.
Very complex feature extractors (e.g. verb subcategorization) cannot generally be completely described this way, so feature extractor code is documented with a description intended to clarify the extractor's behavior. This description also includes keywords intended to assist users in identifying relevant feature extraction behavior. Each feature extractor also has a corresponding unit test that exemplifies the behavior with a literal string indicating the features extracted. This is used by the search tool (see Section 3.5.), which indexes the description and keywords and which will return not just the link to the relevant Javadoc, but also the usage example from the corresponding unit test.
These additional documentation requirements are deemed necessary to make the library sufficiently accessible, as Javadoc alone is seldom sufficient to give the reader a concrete understanding of how a class can be used.

4.2. Creating reference feature implementations

To test the validity and ease-of-use of EDISON's feature extractor definitions, as well as to provide reference implementation for the features used in some of CogComp's most successful NLP tools, we recreate their features in EDI-

```
// Java wrapper for LBJava:
public class dependencyModifierFeatures implements Classifier {
        POSAndWordBigramThreeBefore fex = new POSAndWordBigramThreeBefore();
        ... // More feature extractors here
        @override
        public FeatureVector classify ( Object o ) {
                Constituent focus = (Constituent) o;
                Set<Feature> features = fex.extractFeatures( focus );
                // Convert the Edison Features to LBJava's FeatureVector
                FeatureVector featureVector = FeatureUtilities.getLBJFeatures( features );
                // Concatenate more features to the vector
                featureVector.addFeatures(otherFeatVector);
                return featureVector;
        }
}

// LBJava classifier definition:
// -- uses class simply by referring to its name
discrete namedEntityClassifier(NamedEntity eg) <-
           learn nerLabel
           using dependencyModifierFeatures
           ...
end
```

Figure 4: Sample code snippets showing part of an EDISON feature extractor in LBJava (Rizzolo and Roth, 2010). The extractor needs to be written as an LBJava `Classifier` class (upper half), and then it can be used in an LBJava `Learner` definition.

```
val posWordFeatures = property(predicates) {
     predicate: Constituent => {
        val fex = new POSAndWordBigramThreeBefore()
        // Convert the Set of Features into a comma-separated list
        fex.getFeatures(predicate).mkString(``,'')
     }
}
```

Figure 5: Sample Scala code snippet showing the use of an EDISON feature extractor in Saul (Kordjamshidi et al., 2015). Saul's `properties` (features or labels of `nodes`) are represented as strings.

Edison Home About

Edison Search Interface

bigran

Class Name	Description	Example
POSAndWordBigramThreeBefore	Extract part-of-speech and word bigrams from 3 words before the target constituent.	Example

```
// define a context feature extractor for a context of size two
FeatureExtractor fex = new POSAndWordBigramThreeBefore();
TextAnnotation ta = TextAnnotationUtilities
           .createFromTokenizedString("This is an extractor .");
// extract features for constituent at index 2
Constituent c = ta.getView(ViewNames.TOKENS)
          .getConstituentsCoveringToken(3);
Set features = fex.getFeatures(c);
String target = "[this_is, DT_VBZ, is_an, "
                              + "VBZ_DT, an_extractor, DT_NN]";
assertEquals(features.toString(), target);
```

© Cognitive Computation Group 2016

Figure 6: Screenshot of EDISON feature browser search interface. The user can search through the feature extractors' `Keywords` or their Javadoc header; the search is tolerant to spelling errors as shown in the example query for "bigran". The resulting entries (left) will contain a link to the corresponding Java API page as well as a code snippet (right) demonstrating their use and returned features.

The	construction	of	the	library	finished	on	time	.
DT	**NN**	**IN**	**DT**	**NN**	**VBD**	**IN**	**NN**	.

CogComp Chunker	EDISON
POSWindowpp	**POSOneTwoThreeGramWindowThree**
IN	IN
DT	DT
NN	NN
VBD	VBD
.	.
IN_DT	IN_DT
DT_NN	DT_NN
NN_VBD	NN_VBD
VBD_IN	VBD_IN
IN_NN	IN_NN
NN_.	NN_.
IN_DT_NN	IN_DT_NN
DT_NN_VBD	DT_NN_VBD
NN_VBD_IN	NN_VBD_IN
VBD_IN_NN	VBD_IN_NN
IN_NN_.	IN_NN_.

Figure 7: Comparison of features extracted by a single feature extractor (`POSWindowpp`) from the CogcComp Chunker (shallow parser) pacakge with its new EDISON implementation, `POSOneTwoThreeGramWindowThree`, for the word "finished" in the sentence shown. Note that feature extractors return *sets* of features, so duplicate features are not repeated. If the feature extractor used relative positions of features, there would be additional 1-gram outputs because the same part-of-speech occurs before and after the focus.

SON. In creating the reference implementations for CogComp NLP tools, we defined and followed a process. For each tool, we first studied the existing feature extraction code to understand where the requisite data came from and how it would correspond to an implementation using the `illinois-core-utilities` data structures. Next, we wrote the EDISON implementation for the feature extractor, testing it on a `TextAnnotation` populated with the requisite `Views`, applied the naming convention described above, and added the description of behavior to the documentation. Finally, we verified that the new feature extractor followed the behavior of the original by running both extractors on the same focus item of equivalent inputs, and comparing the feature outputs.

For original applications that used their own data structures, it was necessary to populate comparable source package data structures as well as a `TextAnnotation` for the new EDISON implementation. We developed a simple application to generate the feature set for each member of the extractor pair, allowing side-by-side comparisons and direct human evaluation.

4.3. Validation and Experiments

During the reimplementation process, we wrote a simple application to compare two feature extractor outputs on sample text. This application ran each member of the pair on a deterministically created input text with other supporting annotations as needed. This generates a set of features for each focus item in the text (generally, each word).

The application generates a pair of extracted feature lists and compares the number and distribution of features (the original application may use a different feature output representation than EDISON). If these were not in agreement, we knew the behavior was different. For those in agreement, a human evaluator visually checked the outputs to verify that they matched. Figure 7 shows the output of the original and reference implementation of the feature extractor `POSOneTwoThreeGramWindowThree`.

For some reference implementations there were some minor differences with the original implementation. Specifically, some feature extractors that specified a window had unintuitive behavior at sentence boundaries which we decided to change, as we could see no principled reason for the exact original behavior.

5. Related Work

The most closely related feature extraction systems are FEXTOR (Broda et al., 2013) and Fex (Cumby and Roth, 2000). The latter is a C++ tool that can be run either programmatically or stand-alone to extract features for words, phrases, or simple entity-relation structures. It can be programmatically extended to introduce new features, but ships with a set of word-, part-of-speech-, and chunk-level features that can be easily combined for English text. It uses a very basic data structure to represent the input annotations it requires to generate the feature representation.

FEXTOR is implemented in Python and C++ and offers support for multiple languages. It supports feature extraction across sentence boundaries. The workflow is oriented towards file-based interaction – like Fex, FEXTOR uses a purpose-built scripting language to determine feature extraction behavior.

There is little explicit support for feature extraction for NLP in learning packages such as NLTK (Loper and Bird, 2002),

in NLP development frameworks like GATE (Cunningham et al., 2002), or in NLP software bundles like Stanford's CoreNLP (Manning et al., 2014). All these software frameworks provide integrated natural language processing tools, but they do not directly support extraction of arbitrary features by composing the outputs of these processes. Such features are essential to achieve good performance using machine learning models in many NLP tasks.

EDISON explicitly supports feature extraction but, in contrast to FEXTOR and Fex, is oriented more towards programmatic interaction (although the package can be run as an application that reads from and writes to files). It packages a suite of reference implementations of feature extractors used in a number of state-of-the-art NLP tools, with the goal of making it easier for developers and researchers to implement their own baseline NLP components with a minimum of effort. Its API also more directly supports integration of a machine-learned NLP system into an end-to-end application. EDISON is written in Java, due to Java's widespread use in the academic and commercial NLP developer communities and its compatibility with Scala.

6. Conclusions

This paper presents EDISON, a feature extraction library to support machine-learning NLP applications. EDISON provides a large suite of feature extractors, including reference implementations of features from a set of state-of-the-art NLP tools. We show how EDISON can be used to specify and implement feature extractors for use in a wide range of NLP applications using several popular Machine Learning frameworks. By providing reference implementations for many of our state-of-the-art NLP components, we hope to make it easier for other researchers to leverage our NLP experience. By making it accessible via github, we hope to launch a community-wide effort to collect and share feature extraction functionality. By pooling our collective knowledge we can develop NLP applications more efficiently and improve the reproducability of our NLP research.

7. Acknowledgements

This material is based on research sponsored by DARPA under agreement number FA8750-13-2-0008. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

Bibliographical References

Broda, B., Kedzia, P., Marcińczuk, M., Radziszewski, A., Ramocki, R., and Wardyński, A., (2013). *Computational Linguistics: Applications*, chapter Fextor: A Feature Extraction Framework for Natural Language Processing: A Case Study in Word Sense Disambiguation, Relation Recognition and Anaphora Resolution, pages 41–62. Springer Berlin Heidelberg, Berlin, Heidelberg.

Cheng, X. and Roth, D. (2013). Relational inference for wikification.

Clarke, J., Srikumar, V., Sammons, M., and Roth, D. (2012). An NLP Curator (or: How I Learned to Stop Worrying and Love NLP Pipelines). In *LREC*.

Cumby, C. and Roth, D. (2000). Relational representations that facilitate learning. In *KR*, pages 425–434.

Cumby, C. and Roth, D. (2003). On kernel methods for relational learning. In *ICML*, pages 107–114.

Cunningham, H., Maynard, D., Bontcheva, K., and Tablan, V. (2002). GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications. In *ACL*.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18.

Kordjamshidi, P., Roth, D., and Wu, H. (2015). Saul: Towards declarative learning based programming. In *IJCAI*.

Loper, E. and Bird, S. (2002). NLTK: the Natural Language Toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*.

Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. J., and McClosky, D. (2014). The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60.

McCallum, A. K. (2002). MALLET: A machine learning for language toolkit. http://www.cs.umass.edu/ mccallum/mallet.

Peng, H., Chang, K., and Roth, D. (2015). A joint framework for coreference resolution and mention head detection. In *CoNLL*, page 10, University of Illinois, Urbana-Champaign, Urbana, IL, 61801, 7. ACL.

Punyakanok, V. and Roth, D. (2001). The use of classifiers in sequential inference. pages 995–1001. MIT Press.

Punyakanok, V., Roth, D., and Yih, W. (2008). The importance of syntactic parsing and inference in semantic role labeling. *Computational Linguistics*, 34(2):257–287.

Ratinov, L. and Roth, D. (2009). Design challenges and misconceptions in named entity recognition. In *CoNLL*.

Rizzolo, N. and Roth, D. (2010). Learning Based Java for Rapid Development of NLP Systems. In *LREC*, Valletta, Malta.

Roth, D. and Zelenko, D. (1998). Part of speech tagging using a network of linear separators. In *COLING-ACL, The 17th International Conference on Computational Linguistics*, pages 1136–1142.

Toutanova, K., Haghighi, A., and Manning, C. D. (2008). A global joint model for semantic role labeling. *Computational Linguistics*.