HHS Public Access

Author manuscript

LREC Int Conf Lang Resour Eval. Author manuscript; available in PMC 2017 November 02.

Published in final edited form as:

LREC Int Conf Lang Resour Eval. 2014 May ; 2014: 3289–3293.

ClearTK 2.0: Design Patterns for Machine Learning in UIMA

Steven Bethard[1], **Philip Ogren**[2], and **Lee Becker**[2]

[1]University of Alabama at Birmingham, Birmingham, AL, USA

[2]University of Colorado at Boulder, Boulder, CO, USA

Abstract

ClearTK adds machine learning functionality to the UIMA framework, providing wrappers to popular machine learning libraries, a rich feature extraction library that works across different classifiers, and utilities for applying and evaluating machine learning models. Since its inception in 2008, ClearTK has evolved in response to feedback from developers and the community. This evolution has followed a number of important design principles including: conceptually simple annotator interfaces, readable pipeline descriptions, minimal collection readers, type system agnostic code, modules organized for ease of import, and assisting user comprehension of the complex UIMA framework.

Keywords

NLP frameworks; machine learning; UIMA

1. Introduction

The Unstructured Information Management Architecture (UIMA) framework for developing natural language processing pipelines has grown in popularity since it was open-sourced by IBM in 2005. More recently, UIMA has gained recognition as the underlying architecture of the IBM Watson system that defeated human champions in the game show Jeopardy! (Ferrucci et al., 2010). However, the framework only establishes an architecture for connecting NLP components and does not directly support constructing machine learning classifiers based on sets of features.

The ClearTK framework[1] was introduced to address this gap (Ogren et al., 2008; Ogren et al., 2009) by providing:

- A common interface and wrappers for popular machine learning libraries such as SVMlight, LIBSVM, LIB-LINEAR, OpenNLP MaxEnt, and Mallet.
- A rich feature extraction library that can be used with any of the machine learning classifiers. Under the covers, ClearTK understands each of the native

bethard@cis.uab.edu, ogren@colorado.edu, lee.becker@colorado.edu

[1]http://www.cleartk.org/

machine learning libraries and translates features into a format appropriate to whatever model is being used.

- Infrastructure for using and evaluating machine learning classifiers within the UIMA framework.

Since its inception in 2008, ClearTK has been adopted by multiple developers worldwide in both academia and industry (including University of Colorado, Technische Universität Darmstadt, Apache cTAKES, Thomson Reuters, and 3M) and has been employed on diverse domains including clinical text, social media and student writing. ClearTK has been downloaded over 1700 times in just the past year, the project site receives over 100 new visits a month, and 78 developers have starred the project in Google Code. This growing user and developer base has provided a wealth of feedback that has led to a large number of changes. In this paper, we reflect on key lessons learned over the last 5 years, and how they generally inform the design of natural language processing frameworks.

2. Annotators should be conceptually simple

A core aspect of UIMA is the Annotator, which provides a process method that inspects the document (a JCas in UIMA), performs analyses, and stores resulting annotations. This is a familiar construct, even to novice UIMA users, as there are analogs in other frameworks, e.g. Stanford CoreNLP's CoreMap[2] and GATE's Document (Cunningham et al., 2011). Our experience suggests that annotators' process methods should orchestrate the core analysis code in ways that are as straightforward and intuitive as possible. ClearTK in previous iterations has suffered from over-abstraction of analysis code away from the process method – resulting in difficult to understand code.

In the first versions of ClearTK, developers were required to write two separate annotators: one for writing training data and another for classification. In practice, the bulk of the code in both annotators shared the same feature extraction steps. Consequently, this overlapping functionality was structured to remove this redundancy. The first version of this abstraction used a call-back approach where users only had to implement the feature extraction code. However, users found the call-back style unintuitive as it diverged heavily from the UIMA process conventions. Thus we redesigned this abstraction into ClassifierAnnotator, which allows users to write their feature extraction code in the standard process method, but requires them to handle two cases: training mode and classification mode. An example of such an annotator is shown in Figure 1. Though the change to ClassifierAnnotator requires additional coordination within a single annotator, it is conceptually simpler to learn.

We learned a similar lesson in designing an API for Begin-Inside-Outside (BIO) style chunking classification. The original approach consisted of a subtype of ClassifierAnnotator that abstracted away aspects of chunking, such as feature extraction and converting labels from token-based chunk-based and back, into separate classes that were dynamically loaded at instantiation time. In ClearTK 1.2 we replaced this architecture with a set of simple utility objects for converting between chunk labels and tokens labels, allowing for direct use of

[2] http://nlp.stanford.edu/software/corenlp.shtml

chunking within a standard process method (as shown in Figure 1) without need for specialized implementations of interfaces.

3. Pipelines should look like pipelines

Once a user has developed a number of annotators, they typically string them together in a *pipeline*, indicating the sequence in which these annotators analyze a text. In ClearTK, users develop a variety of pipelines for different tasks such as training classifiers, making predictions with trained classifiers, testing classifier predictions against a gold standard, etc. Our experience suggests that pipeline-based code should be structured to make it easy to quickly understand what annotators are running in what order.

Consider the case of model training and evaluation. ClearTK's first abstraction separated evaluation into various classes and methods:

- The reader that loaded the training and testing data
- The preprocessing portion of a pipeline
- The classifier training portion of a pipeline
- The classifier prediction portion of a pipeline
- The evaluation portion of a pipeline

These items are easily separable and splitting them reduced code duplication. (For example, the preprocessing portion of the pipeline would be identical for training and testing). However, because each of these items was implemented in a different class or method, it was often difficult for a reader to understand the big picture of what exactly was running in each pipeline. In ClearTK 1.2, we simplified this abstraction, resulting in a single evaluation class with just three methods that must be defined:

1. Read a subset of data with a CollectionReader
2. Train a model given a CollectionReader
3. Test a model given a CollectionReader

A partial example of such an evaluation class is shown in Figure 2. In exchange for some duplication (e.g. if training and testing used the same preprocessing) developers are rewarded with more interpretable, self-contained pictures of the training and testing pipelines.

Feature transformation (e.g. normalizing feature values to z-scores or scaling term counts by inverse document frequency) is another example of structuring concerns around pipelines. In early versions of ClearTK, these kinds of transformations required running a specialized pipeline separately before the real pipeline to collect the sufficient statistics. This was confusing to users because (1) two pipelines were required for what was conceptually a single pipeline and (2) feature transformations conceptually happen after training data is written, not before. ClearTK 1.2 introduced TrainableFeatureExtractors where a user instead:

- Runs the original pipeline for writing training data. The TrainableFeatureExtractor will flag features that need additional post-processing.
- Ends the pipeline with an InstanceDataWriter that serializes the features for re-use.
- Invokes the TrainableFeaturesExtractor's train method on the serialized features to store sufficient statistics.
- Uses the original data writer on the transformed features to write out training data for a classifier.

We found that this approach aligned better with the conceptual expectations of our users.

4. Collection readers should be minimal

In UIMA, a CollectionReader is the connection between the source (file, URL, etc.) and the UIMA document (JCas) object. Early versions of ClearTK used the CollectionReader mechanism to both read in the text and import various annotation formats (TreeBank, PropBank, etc.).

However, as ClearTK developed support for importing more annotation formats, it became clear that this approach was problematic. UIMA allows only a single CollectionReader at the beginning of each pipeline, so you cannot, for example, have both a CollectionReader for TreeBank and one for TimeML in the same pipeline, even if both layers of annotation exist for your document. The solution to this problem is to view these TreeBank and TimeML readers not as CollectionReaders, but as regular UIMA annotators whose process(JCas) method utilizes external resources (e.g. a .mrg or a .tml file) to produce annotations to be added to the JCas.

ClearTK now recommends only one CollectionReader, URICollectionReader, which does nothing more than create a JCas containing the source's Uniform Resource Identifier (URI). Reading the text or annotations over the text is the responsibility of subsequent annotators. This approach to developing readers has several advantages, including more parallizable pipelines (which UIMA-AS can take advantage of) and added accessibility by leveraging users' existing familiarity with UIMA annotators. Figure 2 shows an example usage of URICollectionReader.

5. Code should be type system agnostic

All UIMA annotators must declare a type system, which defines the annotations and attributes that an annotator may add to documents. Due to varying requirements imposed by different domains and use cases, there is not yet a generally agreed upon NLP type system for UIMA, and thus many UIMA annotators cannot be combined easily. In ClearTK, we have always been careful to decouple the machine learning framework from the type system. All of the machinery for creating classifier-based annotators including feature extraction, feature normalization, chunking, training, classification, etc. is completely type system independent.

However, other parts of ClearTK do depend on a specific type system, e.g. for reading different annotations from corpora, for wrapping the output produced by non-UIMA annotators, and for constructing state-of-the-art systems like ClearTK-TimeML (Bethard, 2013). It is quite difficult to write a truly type-system agnostic UIMA annotator. For example, the OpenNLP UIMA annotators are intended to be type system agnostic, but in fact make type-system specific assumptions, like representing the part-of-speech as a string-valued attribute of a token annotation. To avoid this level of specific type system dependence, we have found it to be necessary to define interfaces for the various operations on tokens, sentences, parses, etc. Such an approach has been implemented in ClearTK 2.0's wrappers for ClearNLP, and we plan to extend this to other areas in the future.

6. Modules should match natural subsets

ClearTK provides many different types of utilities (machine learning wrappers, readers for various corpora, UIMA wrappers for non-UIMA components like MaltParser or Stanford CoreNLP, etc.) and so it has been necessary to split ClearTK up into a small number of modules to allow users to depend on only those parts of ClearTK that they need. In early versions of ClearTK, we structured these based on the types of annotations being processed, e.g. code for reading PennTree-bank trees was put into the same module as our wrapper for OpenNLP's parser. The idea was that if you were working on, say, parsing, you would want access to all the different parsing algorithms. However, we found that this approach did not scale. For example, very few users would want to include all of OpenNLP, MaltParser, BerkeleyParser, Stanford CoreNLP, etc. just to read trees from a PennTreebank file. ClearTK 1.2 restructured the modules to match the natural subsets of ClearTK functionality:

- Type system agnostic machine learning libraries and feature extractors
- ClearTK's version of a UIMA type system for NLP
- Feature extractors based on the ClearTK type system (e.g. paths through constituency trees)
- Readers for various corpora, based on the ClearTK type system
- Wrappers for non-UIMA components, based on the ClearTK type system

We have found that this structure better matches the conceptual dependencies of ClearTK, and better enables ClearTK users to use only the parts they want.

7. Users need help past the UIMA overhead

After the many improvements to ClearTK interfaces and usability over the years, we have now reached a point where much of the overhead of learning ClearTK is actually the overhead of learning UIMA. To understand the UIMA framework, you need to understand not just how to write an annotator with a process(JCas) method – which is what is really at the heart of the framework – but also how to:

- Declare a type system that describes the annotations you want your annotator to create

- Configure your build system to generate Java classes from the type system
- Create code to read your training data into JCas objects
- Declare (using XML files or Java annotations) any parameters needed to initialize your annotator
- Create an AnalysisEngine object from your annotator and the initialization parameters
- etc.

These tasks are fairly easy for a UIMA expert, but are often challenging and overwhelming for a new UIMA user. Thus, to get new potential users of ClearTK up to speed, we have found it helpful to have a UIMA expert put the above items together. Then, the new users can focus on the core problems that ClearTK is designed for: extracting features and using the classifier in the process(JCas) method of the annotator. We applied exactly this approach with new users of ClearTK, and successfully developed both a student response analysis system for SemEval-2013 (Okoye et al., 2013), and a relation extraction system for Apache cTAKES (Dligach et al., 2013).

8. Discussion

The development of the ClearTK framework has revealed a number of key design patterns for NLP frameworks that can help new users to more quickly understand and adopt a framework. At their core, these patterns suggest aiming for intuitive interfaces that leverage existing user knowledge, and trying to minimize the number of conceptual dependencies between the various parts of the framework.

While the design patterns discussed here are driven by the specific needs of the ClearTK framework as it integrates machine learning into UIMA, we believe that these patterns could be generally useful across NLP frameworks such as Stanford CoreNLP, GATE and NLTK (Bird et al., 2009). For example, while Stanford CoreNLP does well from the perspective of having a simple annotator interface and encouraging readable pipelines, it does not support type system agnostic code – all code using Stanford CoreNLP must translate to and from a fixed set of annotation types. Or, for example, while NLTK does well at arranging its modules to allow users to import only the parts of NLTK that they need, a lot of functionality is packaged into the corpus readers rather than providing generic corpus parsing annotators that can be easily combined. These are not serious flaws that would prevent the use of any of these frameworks, but are potential avenues for improvement as the frameworks themselves evolve.

Acknowledgements

This research was supported in part by the Strategic Health IT Advanced Research Projects (SHARP) Program (90TR002) from the Office of the National Coordinator for Health Information Technology, and by Grant Number R01LM010090 from the National Library Of Medicine. The content is solely the responsibility of the authors and does not necessarily represent the official views of the Office of the National Coordinator for Health Information Technology, the National Library Of Medicine or the National Institutes of Health.

References

Bethard, S. Second Joint Conference on Lexical and Computational Semantics (*SEM), Volume 2: Proceedings of the Seventh International Workshop on Semantic Evaluation (SemEval 2013). Association for Computational Linguistics; Atlanta, Georgia, USA: Jun. 2013 ClearTK-TimeML: A minimalist approach to TempEval 2013.; p. 10-14.

Bird, S., Klein, E., Loper, E. Natural Language Processing with Python. O'Reilly Media; 2009.

Cunningham H, Maynard D, Bontcheva K, Tablan V, Aswani N, Roberts I, Gorrell G, Funk A, Roberts A, Damljanovic D, Heitz T, Greenwood MA, Saggion H, Petrak J, Li Y, Peters W. Text Processing with GATE (Version 6). 2011

Dligach D, Bethard S, Becker L, Miller T, Savova GK. Discovering body site and severity modifiers in clinical texts. Journal of the American Medical Informatics Association. 2013 pages amiajnl–2013.

Ferrucci D, Brown E, Chu-Carroll J, Fan J, Gondek D, Kalyanpur AA, Lally A, Murdock JW, Nyberg E, Prager J, Schlaefer N, Welty C. Building watson: An overview of the DeepQA project. AI Magazine. Jul; 2010 31(3):59–79.

Ogren PV, Wetzler PG, Bethard S. Clear TK: A UIMA toolkit for statistical natural language processing. Towards Enhanced Interoperability for Large HLT Systems: UIMA for NLP workshop at Language Resources and Evaluation Conference (LREC). 2008; 5

Ogren PV, Wetzler PG, Bethard SJ. Clear TK: a framework for statistical natural language processing. Unstructured Information Management Architecture Workshop at the Conference of the German Society for Computational Linguistics and Language Technology. 2009; 9

Okoye, I., Bethard, S., Sumner, T. Second Joint Conference on Lexical and Computational Semantics (*SEM). Volume 2: Proceedings of the Seventh International Workshop on Semantic Evaluation (SemEval 2013). Association for Computational Linguistics; Atlanta, Georgia, USA: Jun. 2013 CU: Computational assessment of short free text answers - a tool for evaluating students' understanding.; p. 603-607.

```
public class NamedEntityChunker extends CleartkSequenceAnnotator<String> {
  ...
  private BioChunking<Token, NamedEntityMention> chunking = new BioChunking<>(
      Token.class, NamedEntityMention.class, "mentionType");
  ...
  public void process(JCas jCas) throws AnalysisEngineProcessException {
    for (Sentence sentence : JCasUtil.select(jCas, Sentence.class)) {
      // extract features for each token in the sentence
      List<Token> tokens = JCasUtil.selectCovered(jCas, Token.class, sentence);
      List<List<Feature>> featureLists = new ArrayList<>();
      for (Token token : tokens) {
        List<Feature> features = new ArrayList<>();
        features.addAll(this.extractor.extract(jCas, token));
        features.addAll(this.contextExtractor.extract(jCas, token));
        featureLists.add(features);
      }
      // during training, convert NamedEntityMentions in the CAS into expected classifier outcomes
      if (this.isTraining()) {
        // extract the gold (human annotated) NamedEntityMention annotations
        List<NamedEntityMention> namedEntityMentions = JCasUtil.selectCovered(
            jCas, NamedEntityMention.class, sentence);
        // convert the NamedEntityMention annotations into token-level BIO outcome labels
        List<String> outcomes = this.chunking.createOutcomes(jCas, tokens, namedEntityMentions);
        // write the features and outcomes as training instances
        this.dataWriter.write(Instances.toInstances(outcomes, featureLists));
      }
      // during classification, convert classifier outcomes into NamedEntityMentions in the CAS
      else {
        // get the predicted BIO outcome labels from the classifier
        List<String> outcomes = this.classifier.classify(featureLists);
        // create the NamedEntityMention annotations in the CAS
        this.chunking.createChunks(jCas, tokens, outcomes);
      }
    }
  }
}
```

Figure 1.
The process method of a CleartkAnnotator for BIO-chunking

```
public class EvaluateNamedEntityChunker extends
    Evaluation_ImplBase<File, AnnotationStatistics<String>> {

  ...
  protected CollectionReader getCollectionReader(List<File> files) throws Exception {
    return CollectionReaderFactory.createReader(UriCollectionReader.getDescriptionFromFiles(files));
  }
  ...
  public void train(CollectionReader collectionReader, File outputDirectory) throws Exception {
    // assemble the training pipeline
    AggregateBuilder aggregate = new AggregateBuilder();
    // an annotator that loads the text from the training file URIs
    aggregate.add(UriToDocumentTextAnnotator.getDescription());
    // an annotator that parses and loads MASC named entity annotations (and tokens)
    aggregate.add(MascGoldAnnotator.getDescription());
    // an annotator that adds part-of-speech tags
    aggregate.add(PosTaggerAnnotator.getDescription());
    // our NamedEntityChunker annotator, configured to write Mallet CRF training data
    aggregate.add(AnalysisEngineFactory.createEngineDescription(
        NamedEntityChunker.class,
        CleartkSequenceAnnotator.PARAM_IS_TRAINING,
        true,
        DirectoryDataWriterFactory.PARAM_OUTPUT_DIRECTORY,
        outputDirectory,
        DefaultSequenceDataWriterFactory.PARAM_DATA_WRITER_CLASS_NAME,
        MalletCrfStringOutcomeDataWriter.class));
    // run the pipeline over the training corpus
    SimplePipeline.runPipeline(collectionReader, aggregate.createAggregateDescription());
    // train a Mallet CRF model on the training data
    Train.main(outputDirectory);
  }
  ...
}
```

Figure 2.
The getCollectionReader and train methods of a ClearTK evaluation class