ClearTK: A UIMA Toolkit for Statistical Natural Language Processing

Philip V. Ogren, Philipp G. Wetzler, Steven J. Bethard
Center for Computational Language and Education Research
University of Colorado at Boulder
Boulder, CO USA
E-mail: philip@ogren.info, Philipp.Wetzler@colorado.edu, Steven.Bethard@colorado.edu

Abstract

This paper describes a toolkit, ClearTK, which was developed at the Center for Computational Language and Education Research at the University of Colorado at Boulder. ClearTK is a framework that supports statistical natural language processing and implements a number of tasks such as part-of-speech tagging, named entity identification, and semantic role labelling. ClearTK is written in the Java programming language on top of Apache UIMA. The core of this framework contains a flexible and extensible feature extraction library and a set of interfaces to several popular machine learning libraries including OpenNLP's MaxEnt, Mallet's Conditional Random Fields, LibSVM, SVM^{light}, and Weka. We demonstrate that ClearTK can be used to achieve state-of-the-art performance on biomedical part-of-speech tagging.

1. Introduction

The Center for Computational Language and Education Research (CLEAR)[1] has had a strong presence in the Natural Language Processing (NLP) community for over a decade publishing dozens of high quality research papers. Despite this research success the center has not produced software that has been widely used outside of it other than the speech recognition software Sonic[2]. Software written for research purposes is not necessarily easy to install, compile, use, debug, and extend. As such, we undertook to create a framework for statistical NLP that would be a foundation for future software development efforts that would encourage wide distribution by being well documented, easily compiled, designed for reusability and extensibility, and extensively tested. The software we created, ClearTK[3], is a framework that supports statistical NLP by providing a rich feature extraction library, interfaces to popular machine learning libraries, and a set of components for tackling NLP tasks such as tokenization, part-of-speech tagging, syntactic parsing, named entity identification, and semantic role labeling. ClearTK is available with source code under a research-use only license[4]. ClearTK currently consists of 175 classes and over 20 unit test suites containing nearly 1300 assertions.

ClearTK is built on top of the increasingly popular Unstructured Information Management Architecture (UIMA) described in (Ferrucci and Lally 2004). Briefly, UIMA provides a set of interfaces for defining components for analyzing unstructured information and provides infrastructure for creating, configuring, running, debugging, and visualizing these components. In the context of ClearTK, we are focused on UIMA's ability to process textual data. All components are organized around a *type system* which defines the structure of the annotations that can be associated with each document. This information is instantiated in a data structure called the Common Analysis Structure (CAS). There is one CAS per document that all components that act on a document can access and update. Every annotation that is created is posted to the CAS which is then made available for other UIMA components to use and modify. Here is a short list of the most important kinds of components:

- Collection Reader – a component that reads in documents and initializes the CAS with any available annotation information.
- Analysis Engine – a component that performs analysis on the document and adds annotations to the CAS or modifies existing ones.
- CAS Consumer – a component that processes the resulting CAS data (e.g. write annotations to a database or a file)
- Collection Processing Engine (CPE) – an aggregate component that defines a pipeline that typically consists of one collection reader, a sequence of analysis engines, and one or more CAS consumers.

We chose UIMA for a wide variety of reasons including but not limited to its open source license, wide spread community adoption, strong developer community, elegant APIs that encourage reusability and interoperability, helpful development tools, and extensive documentation. While UIMA provides a solid foundation for processing text, it does not directly support statistical NLP. ClearTK provides a framework for creating UIMA components that use statistical learning as the foundation for decision making and annotation creation.

In the following sections we describe how statistical NLP is performed with ClearTK (section 2), give an overview of some of the NLP tasks ClearTK supports (section 3), and give results on a part-of-speech tagger written using ClearTK (section 4).

[1] http://clear.colorado.edu, formerly named the Center for Spoken Language Research
[2] http://clear.colorado.edu/Sonic/
[3] http://clear.colorado.edu/ClearTK/
[4] https://www.cusys.edu/techtransfer/downloads/Bulletin-ResearchLicenses.pdf

2. Statistical NLP in ClearTK

ClearTK was designed and implemented with special attention given to creating reusable and flexible code for performing statistical NLP. As such, the library provides classes that facilitate extracting features, generating training data, building classifiers, and classifying annotations. ClearTK introduces *classifier annotators* which are analysis engines that perform feature extraction, classify the extracted features using a machine learning model, and interpret the results of the classification by e.g. labeling annotations or creating new annotations. A classifier annotator can also be run in training mode in which it performs feature extraction and then writes out training data which is then used for building a model.

2.1. Feature Extraction

The ClearTK feature extraction library is highly configurable and easily extensible. Each *feature extractor* produces a feature or set of features for a given annotation (or pair or collection of annotations as the feature extractor requires) for the purpose of characterizing the annotation in a machine learning context. A feature in ClearTK is a simple object that contains a value (i.e. a string, boolean, integer, or float value), a name, and a context that describes how the feature value was extracted. Most features are created by querying the CAS for information about existing annotations. Because features are typically many in number, short lived, and dynamic in nature (i.e. features often derive from previous classifications), they are not represented in the CAS but rather as simple Java objects.

The *spanned text extractor* is a very simple example of a feature extractor that takes an annotation and returns a feature corresponding to the covered text of that annotation. The *type path extractor* is a slightly more complicated feature extractor that extracts features based on a path that describes a location of a value defined by the type system with respect to the annotation type being examined. For example, Figure 1 shows a simple hypothetical type system. A type path extractor initialized with the path `headword/partOfSpeech` can extract features corresponding to the part-of-speech of the head word of examined constituents.

A much more sophisticated feature extractor is the *window feature extractor*. It operates in conjunction with

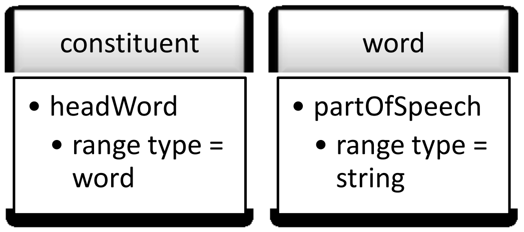


Figure 1: A hypothetical type system that contains the path `headWord/partOfSpeech`.

a simple feature extractor (such as the *spanned text extractor* or *type path extractor*) and extracts features over some numerically bounded and oriented range of annotations (e.g. five token to the left) relative to a focus annotation (e.g. a named entity annotation or syntactic constituent) that are within some window annotation (e.g. a sentence or paragraph annotation.) The "featured" annotations, the focus annotation and the window annotation are all configurable with respect to the type system. This allows the window feature extractor to be used in a wide array of contexts. The window feature extractor also handles boundary conditions such that e.g. words appearing outside the sentence that the focus annotation appears in would be considered as "out-of-bounds." This feature extractor allows one to extract features such as:

- The three part-of-speech tags to the left a word.
- The part-of-speech tag of the head word of constituents to the right of an annotation.
- The identifiers of recognized concepts to the left an annotation.
- The penultimate word of a named entity mention annotation.
- The last three letters of the first two words of a named entity mention annotation.
- The lengths of the previous 10 sentences.

A feature extractor is any class that generates feature objects. For example, the window extractor has a method that takes a focus annotation (e.g. a word) and a window annotation (e.g. a sentence) and produces features relative to these two annotations according to how the feature extractor was initialized. Many feature extractors implement an interface that designate them as *simple feature extractors* which allows them to be used by more complicated feature extractors such as the window extractor. It is the responsibility of the classifier annotator to know how to initialize feature extractors and how to call them. Table 1 lists some of the feature extractors provided by ClearTK.

Similar to feature extractors, *feature proliferators* create features suitable for characterizing an annotation in a machine learning context by taking as input features created by a feature extractor and creating new features. An example of a feature proliferator is *lower case proliferator* which takes features created by e.g. the spanned text extractor and creates a feature that contains the lower cased value of the input feature. Another example of a feature proliferator is the *numeric type proliferator* which examines a feature value and determines if it is a string that contains some digits, contains only digits, looks like a year, looks like a Roman numeral, etc. Table 2 lists some of the feature proliferators provided by ClearTK.

2.2. Classification

After a classifier annotator extracts features it then

Extractor	features extracted derived from…
spanned text	spanned text of an annotation
distance	the "distance" between two annotations
type path	value defined by a path through the type system
syntactic path	the syntactic path from one syntactic constituent to another
white space	existence of whitespace before or after an annotation
gazetteer	entries from a gazetteer found in the text
head word	headword of syntactic constituents
relative position	the relative position of two annotations (e.g. before, overlap left, etc.)
window	some window of annotations in or around the focus annotation
n-gram	generates n-gram style features relative to some window of annotations in or around the focus annotation
bag	generates bag-of-words style features

Table 1: Feature extractors provided by ClearTK

classifies the features and interprets the results. Classification is performed by a *classifier* which is a wrapper class that handles the details of providing a set of features to a machine learning model so that it can classify them and return a result. Currently there are classifier implementations for LibSVM[5] described in (Chang and Lin 2001), Mallet Conditional Random Fields (CRF)[6] described in (McCallum 2002), OpenNLP MaxEnt[7], SVM^{light}[8] described in (Joachims 1999) and Weka[9] described in (Witten and Frank 2005)[10]. Because the classifier wrappers handle the details of passing features to and results from a machine learning library, the developer of a classifier annotator does not have to worry about the low level details of working with each library and can focus on the NLP task itself. An additional benefit of the classifier abstraction is that it allows one to swap out one machine learning library for another and compare and contrast the fitness of a machine learning library for a particular task (see the results section below).

Each of these machine learning libraries are implemented in Java except for SVM^{light} which is implemented in the C programming language. As is common for machine learning libraries, the code for training is much more complicated than the code that performs classification. As such, we re-implemented the classifier in Java such that it can directly classify using the models generated by SVM^{light}. We do not support all of the many variations of SVM^{light} (e.g. SVM^{struct}, SVM^{cfg}, and SVM^{hmm}) or all of the various kernels that are available in SVM^{light}.

[5] http://www.csie.ntu.edu.tw/~cjlin/libsvm/

[6] http://mallet.cs.umass.edu/

[7] http://maxent.sourceforge.net/

[8] http://svmlight.joachims.org/

[9] http://www.cs.waikato.ac.nz/ml/weka/

[10] There are licensing incompatibilities between ClearTK and Weka and SVM^{light} depending on how ClearTK is to be used. Weka and SVM^{light} are not distributed with ClearTK.

Proliferator	Description
capital type	all uppercase, all lowercase, initial uppercase, mixed case
numeric type	all digits, 4 digit year, some digits, roman numeral
character n-grams	character prefixes and suffixes
hyphen	contains hyphen
lower case	Lower case version of string feature

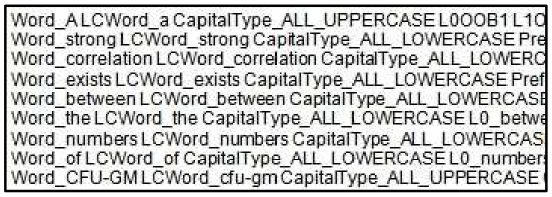
Table 2: Feature proliferators provided by ClearTK

The classifier interface has two classification methods. The first is a method that takes a set of features corresponding to an instance to be classified and returns a single result. The second is a method that takes a list of feature sets that correspond to a sequence of instances to be classified together (or in sequence) and returns a list of results. The latter method is needed for sequential learners such as Conditional Random Fields or Hidden Markov Models because these learners must be able to view the full sequence of instances at once.

After a result or list of results is returned from the classifier the classifier annotator is responsible for interpreting those results by updating existing annotations or creating new ones. For example, part-of-speech tagging is typically accomplished by classifying features extracted for each word annotation. The classification returned will correspond to a part-of-speech tag which can be used to set the part-of-speech of the word annotation in the CAS. This example assumes a type system similar to the one shown in Figure 1. For other tasks, such as named entity identification, the resulting classifications will result in new annotations. Named entity identification is often performed by classifying each word as beginning, inside, or outside a named entity (or some variant of this basic approach). In this example, the classifier annotator would be responsible for taking these word-level classifications and creating named entity annotations for the words that are classified as beginning or inside a named entity. Other NLP tasks such as semantic role labeling are significantly more complicated and require classification of pairs of annotations and as such, feature extraction is performed, in part, on pairs of annotations using feature extractors such as the distance extractor and the syntactic path extractor (see Table 1). In this context, the classification results are interpreted as establishing e.g. a predicate-argument relationship.

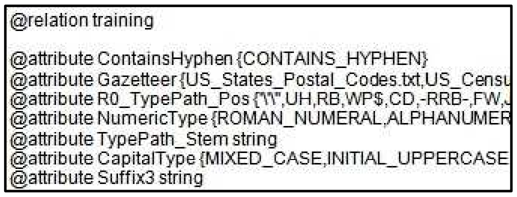
2.3. Training Data Consumers

When a classifier annotator is in training mode it creates training data from extracted features rather than classifying them. Each machine learning library specifies its own particular format that it expects as input when learning a model. Figures 2, 3, and 4 show snippets of training data for MaxEnt, Weka, and LIBSVM, respectively. In ClearTK, training data are created by *training data consumers* which are classes that know how to take a set of features and an outcome and write training



```
Word_A LCWord_a CapitalType_ALL_UPPERCASE L0OOB1 L1O
Word_strong LCWord_strong CapitalType_ALL_LOWERCASE Pre
Word_correlation LCWord_correlation CapitalType_ALL_LOWERC
Word_exists LCWord_exists CapitalType_ALL_LOWERCASE Pref
Word_between LCWord_between CapitalType_ALL_LOWERCASE
Word_the LCWord_the CapitalType_ALL_LOWERCASE L0_betwe
Word_numbers LCWord_numbers CapitalType_ALL_LOWERCAS
Word_of LCWord_of CapitalType_ALL_LOWERCASE L0_number
Word_CFU-GM LCWord_cfu-gm CapitalType_ALL_UPPERCASE
```

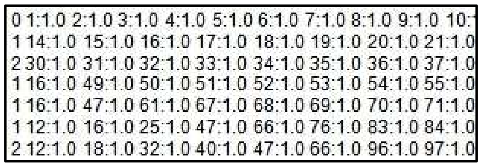
Figure 2: Example MaxEnt training data



```
@relation training

@attribute ContainsHyphen {CONTAINS_HYPHEN}
@attribute Gazetteer {US_States_Postal_Codes.txt,US_Censu
@attribute R0_TypePath_Pos {"\"",UH,RB,WP$,CD,-RRB-,FW,J
@attribute NumericType {ROMAN_NUMERAL,ALPHANUMER
@attribute TypePath_Stem string
@attribute CapitalType {MIXED_CASE,INITIAL_UPPERCASE
@attribute Suffix3 string
```

Figure 3: Example Weka training data



```
0 1:1.0 2:1.0 3:1.0 4:1.0 5:1.0 6:1.0 7:1.0 8:1.0 9:1.0 10:
1 14:1.0 15:1.0 16:1.0 17:1.0 18:1.0 19:1.0 20:1.0 21:1.0
2 30:1.0 31:1.0 32:1.0 33:1.0 34:1.0 35:1.0 36:1.0 37:1.0
1 16:1.0 49:1.0 50:1.0 51:1.0 52:1.0 53:1.0 54:1.0 55:1.0
1 16:1.0 47:1.0 61:1.0 67:1.0 68:1.0 69:1.0 70:1.0 71:1.0
1 12:1.0 16:1.0 25:1.0 47:1.0 66:1.0 76:1.0 83:1.0 84:1.0
2 12:1.0 18:1.0 32:1.0 40:1.0 47:1.0 66:1.0 96:1.0 97:1.0
```

Figure 4: Example LIBSVM training data

data suitable for a given machine learning library. There is one (or more) training data consumer for each machine learning library that ClearTK supports.

2.4. Workflow

A typical workflow for a statistical NLP task in ClearTK involves creating training data, building a model, and then classifying annotations on unseen or test data. Creating training data involves taking an annotated corpus provided by some third party (e.g. Penn Treebank or GENIA) and transforming the data along with a host of features into a data format consumable by a machine learner. Creating training data in ClearTK involves the following steps for each document in the corpus:

a) A collection reader reads in a document from an annotated corpus from its distribution format and adds whatever useful annotation information is provided in the format (e.g. part-of-speech labels, named entities, syntax parse, etc.) to the CAS.
b) A series of analysis engines process the document by creating the annotations needed for the classifier annotator's feature extractors.
c) A classifier annotator in training mode iterates through annotations that are to be classified. For each focus annotation(s) the following two steps are performed:
    - extract features for the annotation(s)
    - pass the features to a training data consumer

The result of this process is a training data file that can be read in by a machine learning library. Model building is performed directly by the machine learning library and is typically invoked from the command line or via a simple script. ClearTK provides basic scripts and examples to invoke the various learners. After a model has been built it is packaged up into a jar[11] file along with some additional meta-data so that a classifier wrapper class can be instantiated from the jar file.

When a classifier annotator is run in classification mode the following steps are performed on unseen or separate test data for each document:

a) A collection reader reads in a document
b) A series of analysis engines process the document by creating the annotations needed for the classifier annotator's feature extractors.
c) The classifier annotator iterates through annotations that are to be classified. For each focus annotation(s) the following three steps are performed:
    - extract features for the annotation(s)
    - pass the extracted features to the classifier wrapper for classification
    - interpret results of the classification
d) A CAS consumer writes out results in a format appropriate for an evaluation script.

3. NLP Components in ClearTK

ClearTK provides a growing library of UIMA components that support a variety of NLP tasks. The library consists of three main types of components: collection readers, analysis engines, and classifier annotators which are summarized in Table 3. The collection readers of particular interest provided by ClearTK are those that read in widely used annotated corpora such as Penn Treebank[12] or PropBank[13]. The Penn Treebank reader reads in constituent parse trees into the CAS such that the full syntactic parse of each sentence is represented in the CAS such that constituents and their relations can be retrieved. The PropBank reader extends this reader by layering on the predicate/argument structure provided by the PropBank corpus. There are also collection readers for reading in the ACE 2005 corpus[14] and the CoNLL 2003[15] shared task data.

The analysis engines provided by ClearTK include a pattern-based tokenizer, a gazetteer annotator, and various wrappers around other NLP libraries. The tokenizer is based on Penn Treebank tokenization rules[16]. The gazetteer annotator finds entries from a gazetteer in text using simple string matching. Other analysis engines include wrappers around the OpenNLP part-of-speech tagger, sentence detector, and syntax parser and a wrapper around the Snowball stemmer[17,18].

[11] http://java.sun.com/javase/6/docs/technotes/guides/jar/index.html
[12] http://www.cis.upenn.edu/~treebank/
[13] http://verbs.colorado.edu/~mpalmer/projects/ace.html
[14] http://www.nist.gov/speech/tests/ace/2005/
[15] http://www.cnts.ua.ac.be/conll2003/ner/
[16] http://www.cis.upenn.edu/~treebank/tokenization.html
[17] http://snowball.tartarus.org/
[18] We use a modified version of the Snowball stemmer

component	type	description
Penn Treebank reader	CR	Reads the Penn Treebank corpus
PropBank	CR	Reads the PropBank corpus
ACE2005 reader	CR	Reads in named entity mentions from the ACE 2004 and 2005 tasks
CoNLL2003 reader	CR	Reads in named entity mentions from the CoNLL 2003 task
GENIA reader	CR	Reads in the GENIA corpus
tokenizer	AE	Penn Treebank style tokenizer
sentence detector	AE	Wrapper around OpenNLP sentence detector
syntax parser	AE	Wrapper around Open NLP syntax parser
stemmer	AE	Wrapper around the Snowball stemmer
gazetteer annotator	AE	Finds mentions of entries in a gazetteer using simple string matching
POS tagger	CA	performs part-of-speech tagging
BIO chunker	CA	performs BIO-style chunking
predicate annotator	CA	Identifies predicates
argument annotator	CA	Identifies and classifies semantic arguments of predicates

Table 3: Components provided by ClearTK. CR = collection reader, AE = analysis engine, and CA = classifier annotator

ClearTK currently provides a small handful of classifier annotators: a part-of-speech tagger (described in section 4), a BIO-style chunker, and a pair of classifier annotators that support semantic role labelling. The BIO chunker performs text chunking using the popular **B**egin, **I**nside, **O**utside labelling scheme for classifying annotations as members of some kind of "chunk." For example, in named entity recognition labels such as "B-person" or "I-location" are used for words that begin a person mention or are inside a location mention, respectively. The BIO chunker is used for named entity recognition, shallow parsing, and tokenization. Semantic role labelling is achieved by the predicate and argument annotators. The predicate annotator decides whether constituents of a syntactic parse are predicates or not. The argument annotator runs subsequently and finds the arguments of a predicate.

4. Results

4.1. Biomedical part-of-speech tagging

To demonstrate the utility of a flexible feature extraction library coupled with interfaces to several popular machine learning libraries we report results for part-of-speech tagging on biomedical scientific literature. The part-of-speech tagger that was created using ClearTK consists of less than 60 lines of code. For training and testing we used the GENIA corpus which consists of 2000 MEDLINE abstracts and about 500,000 part-of-speech tagged words (Tateisi and Tsujii 2004). An inter-annotator agreement study was conducted on fifty of the abstracts and was found to be 98.62% (as simple agreement). The GENIA tagger performs at 98.49% accuracy (Tsuruoka, Tateishi et al. 2005) when trained on the first 90% of the data and tested on the remaining 10%. This represents state-of-the-art performance for this corpus.

distributed by Lucene: http://lucene.apache.org/

	description	Features
F1	current word	$word_i$
F2	word and word bigram features	F1, $word_{i+1}$, $word_{i+2}$, $word_{i-1}$, $word_{i-2}$, $word_i$+$word_{i-1}$, $word_i$+$word_{i+1}$
F3	previous tags	F2, pos_{i-1}, pos_{i-2}, pos_{i-1}+pos_{i-2}
F4	character prefixes and suffixes	F3, prefixes sizes 1, 2, and 3, suffixes sizes 1, 2, 3, 4, 5, and 6
F5	lexical characteristics	F4, capital type (e.g. all caps, initial caps, etc.), numeric type (e.g. all digits, contains digit, roman numeral, etc.), contains hyphen
F6	lower case	F5, lower_case($word_i$)
F7	Stem	F6, stem($word_i$)

Table 4: Feature sets used for training part-of-speech tagging models.

4.2. Feature sets

Table 4 provides a listing of the feature sets that were used for training a part-of-speech tagging model. These feature sets are loosely based on the features used in (Tsuruoka, Tateishi et al. 2005). Given a word in a sentence, $word_i$, each numbered feature set in Table 4 describes a set of features extracted for that word for part-of-speech classification. The feature sets are cumulative such that F2 contains all of the features in F1, F3 contains all of the features in F2, and so on. Feature set F3 is problematic for Mallet because it is a sequential learner tagging an entire sequence at once. This means that previous part-of-speech labels of words earlier in the sentence are not available as features for words later in the sentence. For this reason there are no performance results given for Mallet for feature set F3. For the Mallet experiments using features sets F4 through F7 the part-of-speech features introduced in F3 are excluded.

4.3. Part-of-speech tagging accuracy

Table 5 shows the results of the ClearTK part-of-speech tagger using the seven feature sets against four machine learning libraries. LibSVM out-performed the other learners for every feature set with a top performance of 98.63%. The general trend, as expected, is that adding

	LibSVM	MaxEnt	Mallet	SVM^{perf}
F1	**94.91**	93.40	91.80	90.21
F2	**96.85**	94.44	91.51	92.38
F3	**96.75**	93.76		91.94
F4	**98.49**	98.14	95.88	96.99
F5	**98.58**	98.22	96.35	97.31
F6	**98.58**	98.28	96.37	97.42
F7	**98.55**	98.16	96.34	97.43

Table 5: Part-of-speech tagging accuracy results

	LibSVM	MaxEnt	Mallet	SVM^{perf}
F1	343	0.5	755	102
F2	959	13	669	68
F3	469	16		81
F4	324	20	598	77
F5	258	21	646	83
F6	297	21	597	77
F7	280	21	435	83

Table 6: Training time for building models for part-of-speech tagger in minutes.

	LibSVM	MaxEnt	Mallet	SVM^{perf}
F1	40	0.3	0.5	0.6
F2	186	0.3	1.0	2.5
F3	200	0.4		2.6
F4	137	0.7	1.3	4.7
F5	123	0.7	1.5	4.4
F6	133	0.7	1.5	2.4
F7	127	0.7	1.5	3.1

Table 7: Tagging time for test data in minutes.

more features improves performance. Interestingly, feature set F3, which introduces part-of-speech tag features, performs worse than using feature set F2. Similarly, for feature set F7, which introduces stemmed word features, the performance generally degrades slightly.

Each data point in Table 5 represents five-fold cross validation. For the columns labelled LibSVM, MaxEnt, and SVM^{perf} (a variant of SVM^{light}) each fold is trained on 80% of the data and tested on the remaining 20%. The Mallet learner was prohibitively slow when training on 80% of the data and so only 20% of the data was used for training in each fold. We trained Mallet on 80% of the data using feature set F6 and it took five days to train, 2.5 minutes to tag, and performed at 97.84% accuracy. This was a frustrating result because Mallet has consistently outperformed the other learners on other sequential tagging tasks such as tokenization and named entity recognition.

4.4. Part-of-speech model learning time

Table 6 provides a general approximation of how long each learner requires to train a model for a given feature set. Because the model training took place on a wide variety of CPUs ranging from a single processor running at 1.7 GHz with 1GB of RAM to a doubly hyper-threaded CPU running at 3.4GHz with 4GB of RAM, these results are not strictly comparable. In general, however, the models that took longer to build were run on more powerful machines and so it is possible to make rough conclusions about the relative performance. The minimum training time across the five folds is given rather than the average. The clear trend is that the MaxEnt learner is much faster than the other three.

Despite the smaller amount of training data, Mallet is still consistently the slowest learner (as shown in Table 6.) We have trained Mallet models on similarly sized data sets with a much smaller training cost. However, the number of possible classification outcomes in these models was much less. This finding is consistent with the time complexity of training CRFs which includes a term that is the square of the number of outcomes.

4.5. Part-of-speech tagging time

Table 7 provides a general approximation of the time it took to assign part-of-speech tags to the test set. Again, for the same reasons described above, this data can only be used for rough comparison. Still, there are two trends which seem quite clear. LibSVM is by far the slowest classifier of the three and MaxEnt is the fastest. Another interesting trend is that there is little or no connection between how long it takes to train a model and how long it takes to classify words. For example, the slowest learner, Mallet, provides the second fastest tagger while the second slowest learner, LibSVM, provides the slowest tagger.

4.6. Discussion

Clearly, this experiment is nowhere near an exhaustive search through the space of possible feature sets and machine learning libraries. Furthermore, each learner has its own set of configuration parameters that can and should be tuned for a particular task. SVM^{light}, for example, comes in many flavors including SVM^{light}, SVM^{struct}, SVM^{hmm}[19], and SVM^{perf}. The latter being the one used in this experiment with essentially default configuration parameters, i.e. a linear kernel was used with the regularization parameter C set to 20.0. SVM^{perf} generates binary classifiers which were normalized using Platt's probabilistic outputs for SVMs as described by (Lin, Lin et al. 2007). LibSVM was trained using the linear kernel and default values for all other parameters. Similarly, Mallet was trained without changing any of the default parameter settings. MaxEnt was run with 150 iterations with a feature frequency cut-off of four. We experimented with using beam search with MaxEnt but found that it made very little difference in the outcomes. Weka was excluded from this experiment because it

[19] SVM^{hmm} is not currently supported by ClearTK but seems the most appropriate choice for part-of-speech tagging.

contains a wide variety of machine learning algorithms, many of which do not handle string features, and because we are not familiar enough with this library to make a confident selection among the many choices to represent this library well.

Despite the limitations of the experiment (e.g. using 20% of the data for training Mallet, no parameter tuning, and limited feature space exploration) it is interesting to note that several of the configurations performed at state-of-the-art for this task. The results also point to possibilities for future experimentation such as combining the fastest taggers (MaxEnt and Mallet), removing the features introduced in feature set F3, and replacing SVM^{perf} with SVM^{struct} or SVM^{hmm}.

More important than the scientific contributions of this experiment are the observations on how ClearTK makes this kind of experimentation possible and easy. By having a feature extraction library that is highly configurable with respect to a user-defined type system it is possible for developers to reuse feature extractors for a wide variety of NLP tasks. In fact, the feature extractors used for the part-of-speech taggers were written in the context of supporting named entity recognition and there were no new feature extractors created for the part-of-speech tagger (though, admittedly, these are very similar tasks.) Additionally, because feature extraction is performed independently of any particular machine learning library, it is possible to swap out one learner for another and directly compare them with respect to performance (both accuracy and throughput) with all other factors being equal. This allows a best-of-breed approach to classification in which e.g. Mallet CRF could be used for certain sequential labelling tasks while LibSVM or SVM^{light} can be used for binary classifications required for semantic role labelling while implementations for all of these tasks share common code infrastructure.

This experiment was carried out using an experimental configuration file for defining a set of feature extractors to be used for training and classification. This allows decisions about which feature extractors to use at runtime without having to change the code that calls the feature extractors for each feature set of interest. Additionally, UIMA is highly configurable at runtime via the use of configuration files called *descriptors* which allows, for example, one to define a CPE using XML to specify a particular execution order of components. These two configuration mechanisms allowed each run of the experiment to be executed without any code changes or recompilation. Each run was executed by calling a single script that took in a set of properties associated with the learner being used, a feature extraction configuration file, a set of descriptor files, and parameters that determined the testing and training sets.

5. Conclusion

We have described a new framework for statistical NLP called ClearTK. ClearTK provides a rich feature extraction library, interfaces to several popular machine learning libraries, and a library of components that perform a variety of NLP tasks. We have demonstrated that ClearTK can perform at state-of-the-art level for the task of biomedical part-of-speech tagging and discussed and compared the performance of different learners on this task.

6. Acknowledgements

The authors gratefully acknowledge the following members of CLEAR for their input during the creation of ClearTK: Ying Chen, Ken Griest, James Martin, Martha Palmer, and Wayne Ward. We also thank the respective authors of the previously mentioned machine learning libraries and the members of the UIMA community that have answered our many questions via the UIMA users list.[20] We would also like to thank the University of Colorado Technology Transfer Office[21] who helped fund this project.

7. References

Chang, C. C. and C. J. Lin (2001). "LIBSVM: a library for support vector machines." Software available at http://www. csie. ntu. edu. tw/cjlin/libsvm **80**: 604–611.

Ferrucci, D. and A. Lally (2004). "UIMA: an architectural approach to unstructured information processing in the corporate research environment." Natural Language Engineering **10**(3-4): 327-348.

Joachims, T. (1999). "Making large-Scale SVM Learning Practical. Advances in Kernel Methods-Support Vector Learning." B. Scoelkopf, C. Burges, A. Smola.

Lin, H. T., C. J. Lin, et al. (2007). "A note on Platt's probabilistic outputs for support vector machines." Machine Learning **68**(3): 267-276.

McCallum, A. K. (2002). "Mallet: A machine learning for language toolkit." Unpublished. http://mallet. cs. umass. edu.

Tateisi, Y. and J. Tsujii (2004). "Part-of-speech annotation of biology research abstracts." Proceedings of LREC04.

Tsuruoka, Y., Y. Tateishi, et al. (2005). "Developing a Robust Part-of-Speech Tagger for Biomedical Text." Advances in Informatics: 10th Panhellenic Conference on Informatics, PCI 2005, Volos, Greece, November 11-13, 2005: Proceedings.

Witten, I. H. and E. Frank (2005). Data Mining: Practical Machine Learning Tools and Techniques, Morgan Kaufmann.

[20] http://incubator.apache.org/uima/mail-lists.html

[21] https://www.cusys.edu/techtransfer/