The MITRE Corporation

jCarafe – v0.9.7

1 Overview

jCarafe provides a statistical machine learning framework for creating systems that can automatically extract information from natural language free text. A common example includes the task of identifying Named Entities such as persons, organizations and locations.

1.1 What this Guide Contains

This document is focused primarily on providing details on how to use jCarafe from the command line to train phrase extraction models and then how to apply them to "raw" (i.e. un-annotated) data.

jCarafe performs important steps in the annotate-train-test paradigm for developing systems using a machine learning component. In the course of developing a system, however, the processes of annotating training data and employing proper metrics and error reporting for system evaluation are also vital aspects of building a high-accuracy system. This document does not address these two areas in much detail. Evaluation frameworks can be found within MAT (The MITRE Annotation Toolkit) along with software for creating annotations. Other annotation tools include Callisto, and WordFreak.

It is worth noting that jCarafe's underlying algorithms apply much more broadly than the focus of this document. The focus here is on rapidly and easily developing *phrase extraction systems* with jCarafe. However, the underlying algorithms, based on Conditional Random Fields, have been employed for a variety of tasks, including: part-of-speech tagging, shallow parsing, event extraction, co-reference, syntactic dependency parsing, word alignment, semantic role labeling, dialog act tagging, summarization and discourse parsing.

1.2 System Requirements and Notes on Performance

jCarafe should run on any JRE version 1.5.x or greater. jCarafe is quite memory and CPU intensive due to the nature of the underlying statistical inference and parameter estimation algorithms and typical use with large amounts of data and large numbers of features. Some attention has been placed on optimizing the system's runtime behavior in terms of both memory and CPU usage, but for large data sets powerful hardware will be required.

Training usually involves holding the entire training set and all features extracted from it in main memory; numerous training iterations over the entire data set are then required to learn a model. Memory requirements and training times can be reduced using disk-caching and stochastic gradient descent learning methods (see the `--psa` flag below in Table 1); this is required for training on very large datasets (e.g. greater than 1 million words) and/or for complicated tasks involving many categories.

Decoding times are usually quick, but the time is quadratic in the number of categories (e.g. a doubling in the number of categories results in a four-fold increase in decoding times) and is also affected roughly linearly in the number of features. Decoding can also require significant memory since trained models can involve millions of parameters; the entire model must reside in memory during decoding. Also, for large models, which sometimes are 100's of megabytes on disk, the startup time for the decoder is significant since the entire model must be loaded into memory. Accordingly, the jCarafe decoder typically operates as a server or is used to process a large number of files in batch.

Recent enhancements to jCarafe allow for multi-threading in a variety of contexts. See

1.3 Memory Considerations

For training on large datasets, the maximum heap size of the JVM may need to be increased. This is achieved by adding an option to the java invocation. The flag has the form –Xmx*number***unit** where *number* is an integer and **unit** is typically either `m` for megabytes or `g` for gigabytes. So `–Xmx1500m` would set the maximum heap size to 1500 megabytes. Note that the default heap sizes vary by platforms and JVM version and install differences.

2 Input/Output Formats

The two input/output formats are discussed below.

2.1 InLine

This format assumes input files are plain text. Annotations, at training time, are provided as inline XML/SGML elements wrapped around annotated phrases, just as with the XML input/output mode. However, the document need not be well-formed XML. Further, the text processing mode carries out its own tokenization and sequence boundary identification. At training time, an appropriate example input is provided below:

```
The CEO, <PERSON>John Smith</PERSON>, left for <LOCATION>Boston</LOCATION>
```

While the input is generally assumed to be plain text, modulo XML elements denoting phrase annotations, this processing mode will accept and recognize SGML elements and ignore them for the purposes of determining tokens and/or sentence boundaries. In practical terms, this means that the less-than character, < , needs to be escaped otherwise identifying SGML elements is problematic.

Note that this format will happily accept well-formed XML, but documents are not required to be XML-compliant.

2.1.1 Pre-processing

jCarafe uses its own tokenizer to identify words as well as potential sentence boundaries. In some cases, tokens have already been identified by an earlier processing stage – e.g. a part of speech tagger. In other cases, the user may want to ensure that jCarafe treats a sequence of characters in a document as a single token. Such pre-identified tokens can be specified by using `<lex>` tags in the input data. Below is an example:

```
<lex>The</lex> <lex>CEO</lex><lex>,</lex> <PERSON><lex>John</lex>
<lex>Smith</lex></PERSON><lex>,</lex> <lex>left</lex> <lex>for</lex>
<LOCATION><lex>Boston</lex></LOCATION>
```

2.1.1.1 Token Attributes as Features

Tokens, specified with `lex` tags, can have associated with them attribute value pairs that are denoted as attribute value pairs within the `lex` tag element. The attribute value pairs can then be used as features by the trainer and decoder. This provides a hook that allows for arbitrary information about individual tokens to be provided by an external pre-processing component. In this way, rich task-specific features can be introduced without needing to modify jCarafe. This is especially preferable when existing scripts or tools are available for extracting certain types of features and/or the user does not want to write Java or Scala code in order to extend jCarafe's inventory of feature extraction functions.

2.1.1.2 Zoning

The general problem of zoning – i.e. identifying the parts of an input stream that should, in fact, be processed, is difficult. Further, in complicated applications different zones of a given input should be processed by *different* extraction tools (or models). jCarafe does not provide an architecture to handle this level of zoning but instead provides some basic mechanisms to prevent the processing of regions of an input document.

There are two special tags: <IGNORE> and <BREAK_IGNORE>

The <IGNORE> tag specifies that the region within the ignore block should be ignored as if it were simply not present in the file. The <BREAK_IGNORE> tag, on the other hand, will be ignored as if it were not present but forces a sequence boundary end just prior to the beginning tag and starts a new sequence beginning immediately after the corresponding closing tag.

2.1.2 Labeling Individual Tokens

Not only can attributes be provided as input features, but the tagging task can be set up to label individual tokens (rather than spans or sequences of tokens). The most common instance of this sort of tagging task would be part-of-speech tagging, where each token receives a separate label and there isn't any notion of a "phrase".

```
<lex pos="DT">The</lex> <lex pos="NN">man</lex>
<lex pos="VBD">jumped</lex><lex pos=".">.</lex>
```

The above example shows how the part-of-speech tagging task could be represented in jCarafe. Attribute names (and values) that denote the labels of interest must be specified through the proper tagset.

2.1.3 Robust Decoding

In some cases, prohibiting the input data from containing the ‘<’ character is inconvenient at decoding time. It is possible to run the decoder using a different lexer that does not identify xml/sgml elements.

Identified entities will still be denoted using xml/sgml elements in the output, however. This form of decoding is possible by adding the `--no-tags` option when invoking the decoder.

2.2 MAT-JSON

The MAT-JSON input format uses the Java-Script Object Notation (JSON) format along with a simple schema to represent a text document together with associated metadata, including extracted or human-annotated phrase annotations. In contrast to Inline mode, annotations are represented using stand-off annotations. Each annotation specifies a beginning and ending character position within the text along with the type of the annotation. The full schema is discussed in more detail in the documentation for MAT. An instance of the MAT-JSON format for the simple, ongoing example is found below:

```
{"signal": "The CEO, John Smith, left for Boston.", "asets":
[{"type": "lex", "attrs":[], "annots": [[0,3], [4,7], [7,8],
[9,13], [14,19], [19,20], [21,25], [26,29], [30,36], [36,37]]},
{"type":"PERSON", "attrs":[], "annots": [[9,19]]},
{"type":"LOCATION", "attrs":[], "annots": [[30,36]]}]}
```

Without covering all the details, the reader will note that there is an annotation of type PERSON that extends from character position 9 to 19 (i.e. the text span covering "John Smith"). There is also a set of annotations provided here that denotes the boundaries for each word/token, which have the type "lex".

One key advantage of the MAT-JSON standoff representation is that the text, represented as the value of the "signal" attribute, is an arbitrary string. If XML/SGML elements appear in the signal they will be treated simply as text. This means that the signal does not need to be well-formed in anyway. It offers a more robust way of processing since data need not be pre-processed to escape < characters, for example. Also, as XML/SGML elements are treated as part of the text and are not ignored, they may be used as contextual cues which could be important with semi-structured data, for example.

2.2.1 Tokenization and Sequence Detection and Region Processing

By default, the MAT-JSON processing mode will carry out its own tokenization and sequence delineation (based on identifying sentence boundaries). jCarafe can accept pre-existing tokens that appear in the MAT-JSON document object as “lex” annotations. The `--no-pre-proc` flag can be used for prevent jCarafe from performing its own tokenization and to instead accept the provided “lex” annotations as the document tokens.

MAT-JSON processing utilizes the concept of region annotations to process only specific portions of the provided document signal. These annotations serve to represent document “zones” and/or to delineate sequences within a document. For example, if preprocessing was done that identified tokens and sentences separate region annotations could be provided that mark each sentence. In the case where pre-processing is handled within jCarafe, region annotations might mark entire document zones that should be processed.

When the `--no-pre-proc` flag is present, each region will be processed as a single sequence and any “lex” annotations present in the region will denote the elements of a sequence. If the –no-pre-proc flag is NOT present, then the standard build-in tokenization and sequence boundary detection routines will be applied, separately, to each identified region. So, multiple sequences may arise from a single region annotation depending on how many sentence boundaries are identified via the built-in pre-processing routines.

An example region annotation is shown below. This annotation covers the entire span of the single sentence in the ongoing example. If the document contained multiple sentences, there would be separate region annotations for each sentence.

```
{"type":"zone","attrs":["region_type"], "annots": [[0,37,"body"]]}
```

There can be multiple types of regions within a document. Regions must NOT OVERLAP (partial overlap or embedding). To specify which regions to process, the `--region` flag is used at the command-line. See below.

2.3 Basic

A final format that jCarafe accepts is a low-level, generic, representation for arbitrary sequence classification problems. This format is non-text specific. Its purpose is to provide a means to provide outside application-specific features directly to the CRF learning framework within jCarafe. This can be useful for problems in computer vision, or other language problems that don't operate at the "token-level" (e.g. sequences of sentences) or are otherwise not amenable to jCarafe's standard input representations (inline and MAT-JSON).

The format is line based, so that each element in a sequence is represented as a single line in a file with the format:

```
<sequence label>    <feature1>     <feature2> ……… <featureN>
```

The features and sequence label are tab-delineated. Each `<feature_i>` has the form:

```
<string>:<value>
```

The `:<value>` portion is optional. The `<string>` is an arbitrary alphanumeric string (containing no white-spaces and no ':' character) that serves to represent (or name) a feature. Features have a default value of 1.0 but the value can be changed by including a `<value>` for the feature. Below is a short example:

```
playSoccer	windy	sunny	temperature:85.0
notPlaySoccer	veryWindy	rainy	temperature:35.0
```

There are two elements in this sequence, the first labeled playSoccer, the second notPlaySoccer. Each has three features. The "temperature" feature is non-binary and the scalar values of 85.0 and 35.0 are provided.[1]

Sequences are delineated within a single file using blank lines (or lines that contain only the string "++"). Sequences can be denoted also by placing each sequence in its own, separate file in which case no blank line (or "++" line) is required.

---

[1] Note that in practice, real-valued features should be unit normalized to prevent numerical problems from arising.

3 Getting Started

This chapter describes how to use the jCarafe command-line interface to train a model, provided a corpus containing a well-defined set of annotations. Let's start by taking a look at the options provided to the jCarafe command-line application which can be invoked as follows:

```
> java -jar jcarafe-0.9.7.jar --help
```

This provides the full list of options for using the main jCarafe application, including both training and decoding/tagging modes. Each of these options is explained in further detail below.

3.1 A Simple Example

Let's start with a very simple use-case of training a jCarafe model on a short, single file that contains three different types of annotations: `PERSON`, `LOCATION` and `ORGANIZATION`. This file is shown below.

```
Agreement on these points is a long way from a specific program, and
nobody expects <LOCATION>the U.S.</LOCATION> to rush toward radical
restructuring of the health-care system.

But there are signs that labor-management cooperation could change the
politics of health-care legislation and the economics of medicine.

"I can't remember a time when virtually everyone can agree on what the
problem is," says Mr. <PERSON>Seidman</PERSON>, who heads
<ORGANIZATION>the AFL-CIO</ORGANIZATION>'s department dealing with health
matters.

Because the <PERSON>Bush</PERSON> administration isn't taking the
initiative on health issues, business executives are dealing with
congressional Democrats who champion health-care revision.

"Business across the country is spending more time addressing this issue,"
says Sen. <PERSON>Edward Kennedy</PERSON> (D.,<LOCATION>Mass.</LOCATION>).
```

3.1.1 Training

We can train a model here by using the "inline" input mode. Either cut and paste the example file above into the relative file examples/examplefile1.txt or copy the examples/ directory and/or files here to the directory containing the jCarafe .jar file.

```
java -jar jcarafe-0.9.7.jar --mode inline --train --input-file \
"examples/examplefile1.txt" --model ./model1 --tag PERSON --tag \
ORGANIZATION --tag LOCATION --fspec default.fspec
```

The `--mode` option with `inline` indicates that inline i/o processing should be used, the `--train` option indicates we are estimating a model rather than applying it to new data, `--input-file` argument specifies the file path (relative or absolute) of the input, the `--model` argument specifies the path for the model file that will be created as a result of training. The `--tag` argument flags indicate which XML/SGML element names should be interpreted as annotations. Finally, the `--`

`fspec` flag indicates which features should be extracted from the input based on the feature specifications found in the specified file. Feature specifications are discussed in further detail in Customizing jCarafe's Features.

3.1.2 Decoding

With the resulting model, we can apply the decoder to a raw text file as follows:

```
java -jar jcarafe-0.9.7.jar –-mode inline --input-file \
"examples/example1.raw" --model ./model1 --output-file "out1"
```

This will take the raw input file, which in this case is the same file as was used during training, and run the decoder using the model file "model1" created by the training process. We could also invoke the decoder on the same file by processing the entire examples/ directory, but using a filter to only select the appropriate un-annotated file(s) for processing:

```
java -jar jcarafe-0.9.7.jar –-mode inline --input-dir ./examples/ \
--filter “.*raw” --model ./model1 --output-dir ./ --out-suffix .out
```

This will select all files that end with the three characters "raw" from the examples/ directory, process them, and place the results in the current directly with the input file names appended by the extension .out.

3.2 Overview of the Command Line Options

OPTION	DESCRIPTION	REQUIRE/WITHOUT
--batch-size	**Integer.** Applicable only when using SGD training with PSA. Specifies the number of sequences to be used for each sample/update of the gradient.	REQUIRES `--train`
--mode	**String.** Specifies the i/o mode for the application: inline, json and basic are the three options.	
--train	**Flag.** Perform training, rather than decoding/tagging which is the default.	
--no-pre-proc	**Flag**. Do NOT tokenize and sentence tag the input. Assumes “lex” tags are present when using inline mode and that “lex” annotations are provided in JSON mode.	
--input-dir	**A file path**. Input directory for training or decoding. All files within the directory will be assumed as input files unless the --filter option is used.	
--input-file	**A file path**. A single input file.	
--evaluate	**A file path.** Indicates that the data provided at decoding time should be used to evaluate the performance of the provided model. The tagset needs to be provided for the annotations present in the test data to be picked up.	REQUIRES `--tag` `--tagset`
--filter	**A regular expression**. Files within the specified input directory that match the expression are processed. E.g. `--filter .*sgm'` would select files that end with "sgm".	
--model	**A file path**. A file that will contain a serialized object that is the result of training. Used at runtime to specify the model to use for decoding.	
--lexicon-dir	**A directory path**. Specifies a directory containing files consisting of wordlists used to construct lexicon features.	

--seq-boundary	**List of colon-separated strings**. Specifies the set of tags used to delimit sequences (typically sentence tags) when using xml or json input-output modes.	
--tagset	**A file path**. A file containing a tagset specification	
--max-iters	**An integer**. Maximum number of training iterations.	REQUIRES `--train`
--neural	**Flag.** Use a "neural" CRF that contains one or more hidden nodes (or "gates") at each position in the sequence.	REQUIRES `--train` `--num-gates`
--no-begin	**Flag**. Do not introduce BEGIN states for each label category.	REQUIRES `--train`
--no-cache	**Flag.** By default expanded feature vectors are cached during training. Memory can be saved (at the cost of increased cpu time) by adding this flag to prevent caching. (REQUIRES `--train`)	
--num-gates	**Integer.** Specifies the number of hidden gates/nodes to use with a Neural CRF.	REQUIRES `--train` `--neural`
--gaussian-prior	**A positive float**. Specifies the variance of the Gaussian prior over parameters. Lower variances tend to more aggressively try to combat over-fitting. Default is 10.0.	
--l1	**Flag**. Use L1 regularization with SGD or PSA-based training	REQUIRES `--train` `--psa` or `--sgd`
--l1-C	**Double**. Specify the penalty factor for L1 regularization. (Default is 0.1)	
--psa	**Flag**. Use stochastic gradient descent training using Periodic Step-size Adapation(PSA). Much faster than standard training and usually provides a better model in fewer iterations than plain SGD training.	
--parallel	**Flag**. When using standard training, use multiple CPUs if available.	REQUIRES `--training`
--fspec	**A file path**. A file that specifies the feature functions to use during training.	
--nthreads	**An integer.** Number of threads to use for feature extraction during decoding. This can speed up decoding times on machines with multiple cpus/cores. Note that when pipelining decoders (using the `--pre-model` flag), *each* decoder will be provided the number of threads specified here.	REQUIRES `--parallel`
--num-states	**Integer.** Maximum number of states to use for a non-factored model. (ADVANCED)	
--output-file	**A file path**. Only available for decoding, this option specifies where the automatically annotated output file should be written to.	WITHOUT `--train`
--output-dir	**A directory path**. Only available for decoding, this option specifies a directory where output files should be written to.	WITHOUT `--train`
--out-suffix	**A string**. A string (usually an extension) that will be appended to the input file names to construct corresponding output file names.	WITHOUT `--train`
--pre-model	**A file path.** Specifies models to be applied as a pre-process to the current training or decoding run. Facilitates basic pipelining of decoders (see details below)	
--prior-adjust	**A float.** This parameter is used at decoding time to adjust the weight for the parameter associated with the class label bias that corresponds to "OTHER" to trade off precision and recall.	WITHOUT `--train`

--region	**String.** A specification of a single zone type following the syntax used for specifying tag/annotation types. Only applicable in JSON mode.	
--semi-crf	**Flag.** Use a semi-Markov CRF rather than a standard first-order CRF. Requires different types of features (see below for full details, this is an experimental feature)	REQUIRES `--train`
--streaming	**Flag.** This flag alters the processing during decoding so that very large files are processed incrementally which can save memory and improve processing times. Only applicable in inline processing mode.	WITHOUT `--train`
--tag	**A string**. A specification for a single tag type. Multiple --tag options can be provided, allowing for an entire tagset to be specified on the command line.	
--word-properties	**A file path.** Indicates a file that contains word properties that can be utilized as additional features.	

3.3 Detailed Description of Command Line Options

This section outlines some of the command-line arguments described above in more detail. In many cases, the default behavior of jCarafe obtained by ignoring these arguments is sufficient. However, in some cases, more advanced use is warranted and adjusting some of these settings can notably improve performance (accuracy and/or throughput).

3.3.1 Lexicon Directory (--lexicon-dir)

In certain cases, especially when training data is limited, a lexicon can introduce features that greatly improve performance. jCarafe provides a simple, standard method for easily introducing lexicon features. The argument to the `--lexicon-dir` option is a path to a directory that contains a set of files. Each file within the directory should be a word list that consists of one word on each line. The lexical features introduced, assuming the `lexFn` or `downLexFn` feature specifications have been added (see Table 2 below), will have names associated with the file names found in the specified lexicon directory. That is, if a word in the text is present in a word list, a feature instance associated with the name of that word list is introduced.

3.3.2 Tagset Specification (--tagset or --tag)

A tagset defines what tags, or annotations, the learner should pay attention to (i.e., try to learn) when trained on a specified corpus. An individual tag specification is simply a string that matches a tag element name or (e.g. `PERSON` to match an annotation such as `<PERSON>`John Smith`</PERSON>`), if the tag/annotation type is described with both a tag element and a single attribute value pair has the following form: `TAG:ATT=VAL`

So, for example, `ENAMEX:TYPE=PERSON` would identify the following annotation as a tag type to be learned: `<ENAMEX TYPE="PERSON">`John Smith`</ENAMEX>` Note that the tag specification is case sensitive and must match the strings found in the tags/annotations in the dataset exactly.

One further item to note is that the VAL or a tag specification using an attribute value pair may be a wild card so that all values of a particular attribute that have the correct tag element name will get picked up as annotations of a distinct type. So, for example, `ENAMEX:TYPE=*` would pick up as distinct annotation types `<ENAMEX TYPE="PERSON">` and `<ENAMEX TYPE="ORGANIZATION">`. Within a tagset file, each tag specification should be on a separate line.

> ***Warning: Ensure that the tagset specification file (and all files for that matter) use the appropriate line-ending encoding for the platform you are running on – i.e. '\r\n' on Windows and '\n' on all other platforms.***

3.3.3 Begin, inside, outside encoding (--begin)

The standard way to encode phrase identification tasks as sequence labeling tasks is to introduce three different types of states/labels. (B)egin states denote the beginning (i.e. first word/token) of a phrase of interest; (I)nside states denote a token within a phrase of interest that is not the first word of that phrase; and (O)utside tokens denote the tokens not part of any phrase of interest. An example of en encoding for the annotated text snippet shown earlier is below:

The	CEO	,	John	Smith	,	left	for	Boston
O	**O**	**O**	**B_P**	**P**	**O**	**O**	**O**	**B_L**

By default, a BIO encoding will be used. If the `--no-begin` flag is present, however, begin states will not be generated and the two separate states B_X and I_X (for beginning and inside of a phrase of type X, respectively) will be collapsed to a single state, X. This would be a preferred encoding, for example, if two phrases of the same type never occur immediately next to each other in a particular dataset.

3.3.4 Periodic Step-size Adaptation (--psa)

The default training method in jCarafe is based around trying to maximize the conditional log-likelihood (CLL) of the data. Computing the CLL and its gradient (required for maximizing the CLL) can be expensive, however. It requires computing the model expected value of each feature over the entire training set. This approach can be prohibitive with large datasets, however. Hundreds of iterations may be required for the model to converge and each such iteration may require hours of computation for very large datasets.

An alternative approach to parameter estimation is to use stochastic gradient descent (SGD) methods. Rather than computing the CLL and its exact gradient, SGD methods take a small sample of the training data at a time and compute the gradient for just that small sample. The parameters are then updated based on the "local gradient" according to a learning rate. The learning rate is given a momentum such that it decays over time and the updates performed on the parameters are of smaller and smaller magnitudes.

jCarafe includes a variation of SGD learning called Periodic Step-size Adaptation (PSA) that has proven to work extremely well in practice. PSA works by separately adjusts the learning rates for individual parameters. In practice, identifying the optimal momentum value with SGD and initial learning rate can be tricky and carried out through simple trial-and-error. Suboptimal momentum values or initial learning rates can mean that it takes much longer for SGD to converge to optimal parameters, in the learning sense, or that the optimal parameters are never reached (due to a learning rate that decays too rapidly).

A key insight towards overcoming this problem with SGD is the fact that some of the parameters may converge more quickly than others. Accordingly, it would be preferable to update the learning rates for each parameter separately, rather than having a single learning rate for the entire set of parameters. The approach outlined in PSA takes exactly this approach and separately fine-tunes the learning rates for each parameter based on the recent history of adjustment for that parameter.

Stochastic Gradient Descent training using PSA is enabled with the `--psa` flag. Note that there is not automatic convergence test when using PSA. The user should set the number of iterations to a reasonable value. While the default value of 10 iterations is usually sufficient, often it is possible to train a quality model with 5-7 iterations. This value can be set with the `--max-iters` flag.

3.3.5 L1 Regularization (--l1 and --l1-C)

Regularization is a means to prevent models from overfitting by penalizing the learning objective function in a way that prevents parameter values from being adjusted in a way that fits the training data too closely. The standard regularizer for log-linear models like CRFs and maximum entropy models is an L2 regularizer such as the Gaussion prior already described. This assigns a penalty term based on the square of each parameter value. An L1 regularizer, on the other hand assigns a penalty term based on the absolute value of each parameter. L1 regularizers are somewhat harder to work with since the absolute value function is non-differentiable at zero. They have the benefit, however, of learning "sparse models" - that is, many parameters in the resulting learned model will have a value of zero. Such parameters can be removed entirely from the model. Such sparse models provide for more efficient decoders.

L1 regularization currently can be used with both standard stochastic gradient descent learning and PSA using the `--l1` flag. It is not available for batch training using the conditional log-likelihood objective. The penalty factor associated with the L1 regularizer, C, can be adjusted using the `--l1-C` option (e.g. `--l1-C 0.8` would set the value to 0.8). Its default value is 0.1. The optimal value must be manually tuned for each specific data set (perhaps using cross validation).

3.3.6 Pipelining Decoders (--pre-model)

Frequently, information extraction components are “pipelined”. A common pipeline might involve the following sequence of processing steps: 1) tokenization, 2) sentence identification, 3) part-of-speech tagging, 4) shallow parsing (i.e. chunking) and 5) Named Entity extraction. Developing pipelines of different components is a complex task, in general, and a number of architectures and frameworks have been proposed to handle this, including GATE and UIMA. This problem is beyond the scope of what jCarafe aims to offer, however there are certain “local pipelines” that occur frequently in practice such as part-of-speech tagging followed by Named Entity extraction, for example where the idea is to use the output of the part-of-speech tagging process to help improve the Named Entity extraction. jCarafe provides basic support to pipeline multiple processing stages, assuming each of those stages consists of a jCarafe decoder as specified by a jCarafe model file.

Note first that pipelining can be achieved in a general way with jCarafe by running up-stream components, perhaps not even jCarafe components, and encoding the output of those components as attributes on each token (i.e. “lex”) annotation/tag. For example, one could run a Brill-rule-based part-of-speech tagger and produce output that looks like:

```
<lex pos=”DT”>The</lex> <lex pos=”NN”>man</lex> <lex pos=”VBD”>ran</lex>
```

This could then be fed into a jCarafe-based Named Entity extraction component that uses the right types of feature functions (see below) to add features based on the identified parts-of-speech. The primary goal of jCarafe's pipelining mechanism is to simplify this process and obviate the need to create intermediate file representations containing the outputs from previous processing stages.

The `--pre-model` tag facilitates pipelining in a simple way. Each decoder (i.e. model file) that should be run as a pre-process is passed in via a `--pre-model` tag. Then, assuming the right feature functions are included in the feature specification(s) for the downstream model(s), the outputs from earlier stages are provided as features to downstream stages. It is possible to provide more than one pre-process to run using multiple `--pre-model` specifications. The *order in which the pre-processing steps are run is the same as the order they appear on the command line*.

3.3.6.1 Pipelining Example

Let's walk through an example of how the pipelining mechanism works. Let's assume we'd like to run a pipeline that consists of: part-of-speech tagging => shallow parsing => named entity extraction. The first step would involve training the part-of-speech tagger. We'd like the shallow parser to make use of the part-of-speech taggers output. We would train the shallow parsing using a command such as:

```
java –jar jcarafe-0.9.7-bin.jar --mode inline –-train –-input-dir
./shallow.input –-fspec shallowFspecWithPreProcessing –-pre-model ./part-
of-speech.MODEL --model shallow.MODEL
```

This would run the part-of-speech tagger specified via the `part-of-speech.MODEL` model file up front and then train the shallow parser using features produced from the output of the part-of-speech process. For this to work, the `shallowFspecWithPreProcessing` spec file should contain the "allTagFn" or the "attributeFn" feature specification in order to pick up the appropriate features from the part-of-speech tagger.

Now, in order to train the named entity extractor which should use both the outputs of the part-of-speech tagger and the shallow parser, we would invoke a training command such as:

```
java –jar jcarafe-0.9.7-bin.jar --mode inline –-train –-input-dir
./ne.input –-fspec namedEntityWithPreProcessing –-pre-model ./part-of-
speech.MODEL –-pre-model ./shallow.MODEL –-model namedEntity.MODEL
```

In order to apply our named entity model that requires part-of-speech tagging and shallow parsing as pre-processing steps, we need to provide this series of models at decoding time as well:

```
java –jar jcarafe-0.9.7-bin.jar --mode inline –-input-dir ./ne.toProcess
--pre-model ./part-of-speech.MODEL --pre-model ./shallow.MODEL –-model
namedEntity.MODEL –-output-dir ./ne.Processed
```

Note that while the preprocessing stages are carried out in series, there is no requirement that each be dependent on any of the previous processing stages. It would be possible, for example, to include a processing pipeline that consists of two separate part-of-speech taggers (that hopefully have somewhat uncorrelated errors) neither of which is dependent on the other but both of which are used by a downstream component to produce hopefully useful features.

Finally, note that this pipelining framework does not scale particularly well in that all the processing components are part of the same memory image and the processing is in no way distributed.

3.3.7 Semi-Markov CRFs (--semi-crf)

Semi-Markov CRFs are a strictly more powerful model than a first-order sequential CRF in which sequences are jointly segmented and labeled. These models consider all possible segmentations for a sequence for segments up to some specified length, k, and the label/classify each of these segments. Crucially, label dependencies between adjacent *segments* are captured. This contrasts with a first-order CRF that captures label dependencies just between adjacent sequence elements. Semi-Markov CRFs (or simply Semi-CRFs) require different types of features than standard first-order sequential CRFs. Section 4.4 below describes some of the feature specifications available for Semi-CRFs. Semi-CRFs are enabled by using the Semi-CRF flag, `--semi-crf`. Note that Semi-CRF training is not fully supported in that the planned set of basic feature specifications is not yet complete. Further, only standard batch conditional log-likelihood training is supported.

3.3.8 Multi-threaded Decoding (--nthreads)

jCarafe offers support to speed up the process of applying a trained model to new data (i.e. decoding) by carrying out the cpu-intensive portions of this task, feature extraction and Viterbi decoding, with multiple threads. In the current implementation, there is a fair amount of overhead associated with this and speedups are typically only achieved if 3 or more cpus/cores are available. Despite the extra overhead, 5-fold speed-ups or more are possible on machines with 10 or more cpus. The `--nthreads` flag takes an integer that indicates the number of threads that will be spawned to carry out decoding, e.g. `--nthreads 10` specifies that 10 threads should be used. Note that the number of threads may exceed the number of available cpus/cores.

By default, the decoder performs multi-threaded decoding when it detects that more than 3 cpus are available on the current machine. It will then spawn 4n/5 threads where n is the number of detected cpus. To force single threaded behavior on a multi-cpu machine, the flag `--nthreads 1` must be added to the command line.

3.3.9 Parallel Training Options (--parallel and --nthreads)

It is possible to use multiple threads during both standard batch training as well as stochastic gradient descent training using PSA (see the `--psa` option). The `--parallel` flag indicates that multiple threads should be used. As with decoding, by default 4*n*/5 threads are created where *n* is the number of detected cpus. The `--nthreads` option can be used to explicitly set the number of threads.

3.3.10 Word Properties (--word-properties)

The `--word-properties` flag can be used to denote a file that contains a set of properties for words/observations. This is similar to using a lexicon. However, instead of a directory of files, a single file is provided where each line in the file denotes an entry, and each entry has the following format:

```
<word> <prop1> <prop2> …. <propN>
```

These properties can then be picked up as features by including the `wdPropFn` feature extractor (see below). This way of encoding word-specific features can be more appropriate than using a set of lexicons (with the `--lexicon-dir` option) when there are potentially many properties associated with each word.

3.3.11 Neural CRF (--neural and --num-gates)

CRFs provide a flexible framework for working with large numbers of highly non-independent features. Achieving high accuracies, however, often requires careful feature engineering. This is made simple with jCarafe, but still requires a fair bit of time and experimentation. A recent extension for CRFs has been proposed that introduces a “hidden layer” of gates between the observations and the states/label variables. This hidden layer can discover non-linear properties of the input that help with the target sequence labeling problem. The `--num-gates` option indicates how many such gates should be introduced at each position in the sequence. Only features that have been designated as “NEURAL” features will be connected to these hidden nodes. Non-neural features will be connected to the “output nodes” directly. See the discussion of neural features in the Feature Specifications section below.

3.4 Command-Line Argument Dependencies

4 Feature Specifications

This section covers jCarafe's framework for declaratively specifying the types of features to include in the model for a specific phrase extraction task. The features are the heart and soul of any extraction application. Careful attention to the features used in a model is frequently the determining factor in arriving at high-accuracy extraction systems. Furthermore, while jCarafe can comfortably accommodate models with millions of features, the overall throughput of the system is adversely affected by more features than necessary. Additionally, the model may well be more likely to over-fit with large numbers of features, providing degraded performance especially when it's applied to data that "looks different" (e.g. is of a different domain or genre).

In the future, jCarafe will have functionality aimed at automatically discovering or inducing feature specifications automatically from training data. Until then, however, specifying features must be done manually and is often driven by linguistic intuition and/or knowledge of the task and domain. A solid experimental framework (such as provided in the MITRE Annotation Toolkit) is recommended for iteratively developing feature specifications by improving on results using a held out development set and/or cross validation.

A feature specification can be thought of as a *template* for creating features of a certain type. To understand feature specifications, it helps to understand a little about the feature extraction process within jCarafe. For each sequence in the data set, at each position within the sequence (i.e. for each word) each feature specification is applied (or executed) at that position and results in a set of specific feature instances. This is best explained by a simple example. Consider the feature specification:

```
simple_word_function as wdFn;
```

The part that says `simple_word_function` is simply the name of this specification and can be an arbitrary string. The keyword 'as' separates the name from the specification body. The specification body in this case is the string `wdFn` which is a built-in, atomic extraction function. The `wdFn` function simply identifies the word at the current position and returns the word itself (unmodified) as a feature instance. In this way this simple specification will introduce many different feature instances, on the order of the size of the vocabulary of the training set.

4.1 Built-in Feature Functions

jCarafe comes with a set of built-in feature extraction functions that serve as building blocks to create more complicated feature specifications. These functions are detailed below:

Feature Function	Description
wdFn	Extracts the word at the current position and returns it as a feature
caselessWdFn	Extracts the word at the current position and returns the result of downcasing it as a feature
regexpFn(name,regexp)	Takes two arguments. The first is a string that will serve as a name for this feature. The second is a string interpreted as a regular expression. If the regular expression accepts the token at the current position, the first argument name is returned as a feature
prefixFn(integer)	Takes an integer argument. Returns all prefixes from length 1 to the specified integer as features.
suffixFn(integer)	Takes an integer argument. Returns all suffixes from length 1 to the specified integer as features.
lexFn	Checks whether the current word appears in a specified lexicon. If so, the names of the lexical categories are returned as features.
wdPropFn	Checks whether the current word has any associated properties. If so, the names of the properties associated with the word are returned as features.
downLexFn	Downcases the current word and checks whether it appears in a specified lexicon. If so, the name of the lexical category is returned as a feature. Assumes, of course, that the lexicon provided has been downcased.
attributeFn(att)	Adds the feature “att=<val>” where ‘att’ is an attribute of the current lex tag and <val> is the value of that attribute. E.g. attributeFn(pos) would add the feature “pos=DT” for the lex tag (i.e. annotation): `<lex pos=’DT’>..</lex>` Note that in the feature spec file the attribute should NOT be surrounded with quotes. Whatever appears within the parentheses is considered part of the attribute name itself.
allTagFn	Extracts all attribute-value pairs present within *lex* tags/annotations as features.

4.2 Transition vs State Features

Features may correlate properties of the observed data with either individual states or state-pair transitions. By default all feature specifications (except the edgeFn feature spec) will create individual state features. To indicate that a feature should correlate with transitions the feature spec should end with the word TRANSITION in all caps and preceded by a space. Note that including transition features will often increase the number of features significantly since there are many more possible transition features than state features. Below is an example of two feature specs involving wdFn features, introducing both individual state and state pair (i.e. transition) features.

```
wdsWithStates       as wdFn;
wdsWithTransitions as wdFn TRANSITION;
```

4.3 Neural Features

Features may also be designated as “neural” features when one or more hidden nodes/gates are used along with the `--neural` option flag. This is done by adding the word NEURAL to the end of a feature specification:

```
wdsAsNeural       as wdFn NEURAL;
```

This specification will add all the features produced by the wdFn feature extractor as inputs to all the hidden nodes (the number of which is specified with the –num-gates option) as well as standard input features to the CRF. A full explanation of how these features work is outside the scope of this document at this time. A key point to mention is that adding many neural features will vastly increase the number of model parameters and will tend to result in over-fitting (the model can also be more difficult to train/estimate). As such, neural features are more appropriate to add for smaller numbers of input features. For example, instead of adding neural features for all the word features (produced by wdFn) it might make sense to add neural features for input features based on lexicon membership or based on part-of-speech attributes. As there are typically a smaller number of lexicon membership features (one feature type for each word-list) and part-of-speech features (number of part-of-speech tags), the number of parameters will not increase so much.

Note that the number of parameters will increase geometrically with the number of input features and number of gates.

4.4 Higher-order Feature Specifications

The atomic feature functions just detailed provide low-level building blocks for more complicated features that place these functions in context. These higher-order functions (or operators) take one or more feature specification bodies (either atomic feature functions or more complicated expressions) as arguments.

Let's start with a simple example that introduces how we can add a feature specification that constructs features regarding the word that appears immediately before and immediately after the word at the current position. This would be done as follows:

```
context_unigrams as wdFn over (-1,1);
```

Again, the first piece is just a name for the specification. The specification body itself says that the wdFn function should be applied to the previous position, -1 and the subsequent position, 1. The right argument to the higher-order function *over* can be either a list of integer offsets (relative to the current position) or a range of positions specified with the syntax (X to Y). So, for example, the following specification would capture features for the three positions to the right of the current word:

```
unigrams_to_the_right as wdFn over (1 to 3);
```

The other important higher-order function is the ngram operator. This function takes the same arguments as "over", but instead of introducing a set of features, one for each element in the specified range, the n-gram function concatenates words at the relative offsets into a single feature instance. So, the following specification would construct trigrams from the three words immediately to the right of the current word:

```
trigram_to_the_right as wdFn ngram (1 to 3);
```

A full description of the different higher-order feature functions can be found below:

Function Pattern	Description
Atomic_function **over** offsets	An atomic feature function is applied to all relative positions within the specified range or set of offsets. The features returned, for each offset value, are appended with the offset value and added as feature instances.
Atomic_function **ngram** offsets	An atomic function is applied to all relative positions in offsets and all features extracted from all relative positions are conjoined into a single feature together with the conjunction of offsets. E.g. wdFn ngram (1,2) would return the single feature instance "said_hello(1,2)" if applied at the position of the word "Smith" in the passage "John Smith said hello".
Spec_body **cross** Spec_body	This higher-order function takes the feature instances computed from the first specification body, F1,...,Fn and the feature instances computed from the second specification body G1,...Gn and computes the cross product of feature instances, G1F1, G1F2, ... , G1Fn, G2F1,..., G2Fn,..GnF1,..,GnFn This allows for conjunctions of features to easily be created over existing feature specifications.

4.5 Semi-CRF Feature Specifications

5 jCarafe API

The majority of jCarafe is written in Scala (www.scala-lang.org) , with some lower-level routines and libraries written in Java. It may be desirable, however, to treat jCarafe simply as a Java library use in larger applications. Accordingly, a Java API is provided. Currently, the API does not encompass *training*, just decoding. The API is in its early stages with additional functionality planned to more easily facilitate different input/output formats as well as to specify certain parameters to the decoder.

The jCarafe Java API works in a very simple fashion via a single class: `org.mitre.itc.jcarafe.jarafe.JarafeTagger`. Currently, the API does not support training, so models must be derived by running jCarafe via the command-line interface. A Jarafe instance needs to specify an input/output mechanism along the lines of the inline, json options described earlier for batch processing. Once properly initialized the Jarafe object provides a set of methods for processing a single string as input (either inline or MAT-JSON). Full details on the Java API can be found in the javadocs. Here, we provide a couple of Java examples using the API.

```
//import API
import org.mitre.itc.jcarafe.jarafe.JarafeTagger;

//set up the JarafeTagger object
JarafeTagger tagger = new org.mitre.itc.jcarafe.jarafe.JarafeTagger();
String modelFilePath = "C:/Documents ….”
tagger.initializeAsText(modelFilePath);
String result = tagger.processString("Some string to process …”);
```

An alternative way to use the API is to pass in, as an array of strings, the command-line arguments that one would use from the command line to use the decoder/model:

```
//import API
import org.mitre.itc.jcarafe.jarafe.JarafeTagger;

//set up the JarafeTagger object
JarafeTagger tagger = new org.mitre.itc.jcarafe.jarafe.JarafeTagger();
String modelFilePath = "C:/Documents ….”
String[] args = {"--model", modelFilePath, "--mode","inline","--seq-
boundary", "s", "--nthreads","5"}
tagger.intialize(args);
String result = tagger.processString("Some long string to process..");
```

Note that the key difference with the second method is the ability to pass in arbitrary options to the decoder in a concise (but not type-safe) manner. The API will be extended in the future to provide methods to directly add decoder options.

6 jCarafe as a Service using XML-RPC

The jCarafe Java API described above allows users to integrate jCarafe-based extraction capabilities into their existing application(s). In this setting, a client sends a unit of text to be processed, a jCarafe server applies one or more extraction routines to the data and returns a set of annotations over the data in one of the primary representations discussed earlier: standoff annotations via JSON-MAT or inline annotations.

Using jCarafe-based services requires the jcarafe_services jar file that includes additional functionality beyond the core jCarafe distribution.

Given a jCarafe model, an XML-RPC service can be started with the following command:

```
java -cp jcarafe_xmlrpc-0.9.0-bin.jar \
org.mitre.itc.jcarafe_server.tagging.JarafeTaggerServer <port> <options>
```

The standard command line options appropriate for a jCarafe decoder can be passed in as the set of options, `<options>`.

Client applications need to call the method "jarafe.processBase64String" with a single argument that is a base 64 encoded string in the corresponding input/output format. So, for example, if the server is expecting "json" format, a base64 encoded string containing "signal":"Text to process" should be provided. If the server is in “inline” mode, the string "Text to process" would be provided, base64 encoded. Recall that JSON-encoded strings must conform to the JSON standard and cannot, for example, include newlines or other control characters. These must be properly escaped.

7 Additional Functionality

Besides the core functionality described in this guide, jCarafe provides other applications useful for processing text data as well as additional stand-alone learning algorithms.

7.1 Tokenizer

jCarafe's built-in tokenizer for English can be called as a separate application. The tokenizer will produce reasonable output for any latin-based character sets. The tokenizer can be invoked on a single file as follows:

```
java -cp jcarafe-0.9.7.jar org.mitre.itc.jcarafe.tokenizer.FastTokenizer -
-input-file <input file> --output-file <output file> [--json]
```

The resulting file will contain the original input file contents with `lex` tags wrapped around each identified token.

If the `--json` flag is added, the input is assumed to be in MAT-JSON format. The elements of the JSON document **signal** will be tokenized with standoff token annotations added with type `lex`; the resulting serialized JSON document object will be written to the specified output file.

7.2 Maximum Entropy Classification

The maximum entropy classifier can be invoked in a manner similar to the standard invocations for using jCarafe for phrase tagging described above. For training the invocation is:

```
java -cp jcarafe-0.9.7.jar org.mitre.itc.jcarafe.maxent.ME train --input-
file <input file> --model <model file> [--gaussian-prior <positive float>]
```

The result of this process is the learned model placed in *model file*. The model can be used on new data by applying the decoder as follows:

```
java -cp jcarafe-0.9.7.jar org.mitre.itc.jcarafe.maxent.ME decoder --
input-file <input file> --model <model file> --output-file <output file>
```

The output of this process sent to *output file* consists of a file where each line corresponds to the same line number in the input file. Each line in the output file has a tab-separated list of label and probability pairs of the form *label:probability* -- i.e., where the label and probability values are separated by a colon character.

7.2.1 Data Formats

For both training and decoding, the following format is used. Each line in the file represents a single training/test instance. The first element is a string denoting the label/category for that instance. The remaining elements are features, which can be arbitrary strings, followed by an optional real-valued feature value (the default is 1.0). Elements are separate with one or more single spaces. A comment can be provided for each instance (useful for keeping track of where it came from) at the end of the line initiated with a '#' character. Below is a simple example:

```
playUltimate sunny windy windspeed:20.0 # this was on Sunday April 9
notPlayUltimate rainy windy windspeed:30.0 # this was the next day
playUltimate johnPlays numberOfPlayers:10.0 numberOfPlayersGreaterThan7
...
```

At test time, if labels are included, the performance of the classifier will be provided. If labels are not available (because the data hasn't been labeled) some element must be still be included as a placeholder:

```
UNK sunny windy windspeed:20.0 # this was on Sunday April 9
UNK rainy windy windspeed:30.0 # this was the next day
UNK johnPlays numberOfPlayers:10.0 numberOfPlayersGreaterThan7
```

7.3 Log-linear Ranking Model

For ranking problems, the format is slightly different. Each "event" within a "ranking instance" is on a separate line having the form: *handle prob. mass feature vector*. Each group of instances making up the event is separated by a line with 5 or more '-' characters in a sequence.

The *handle* is an arbitrary label that is ignored. The *prob. mass* is a float between 0 and 1 indicating the "score" associated with that instance of the event. For example:

```
lab1 0.9 feature1 feature2
lab2 0.06 feature3 feature4
lab3 0.04 feature4 feature5
---------------------
lab1 0.7 feature1 feature10
lab2 0.2 feature3 feature5
lab3 0.05 feature4 feature5
lab3 0.05 feature3 feature6
```

The above includes 2 "events" where the first has 3 instances and the second has 4 instances. Any number of instances may be included for each event. The total probability mass should sum to 1 for each event (though these masses will be re-normalized to sum to 1 in any case).