Apache OpenNLP
Developer Documentation



Written and maintained by the Apache OpenNLP Development Community

Apache OpenNLP Developer Documentation

by Written and maintained by the Apache OpenNLP Development Community

Version 3.0.0

Copyright © 2011, 2026 The Apache Software Foundation

License and Disclaimer. The ASF licenses this documentation to you under the Apache License, Version 2.0 (the "License"); you may not use this documentation except in compliance with the License. You may obtain a copy of the License at

https://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, this documentation and its contents are distributed under the License on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Table of Contents

1. Introduction 1
    - Description 1
    - General Library Structure 1
    - Application Program Interface (API) 1
        - Generic Example 1
    - Command line interface (CLI) 2
        - Description 2
        - List of tools 2
        - Setting up 2
        - Generic Example 2
    - OpenNLP Models 5
        - OpenNLP Models 5
        - ONNX Models 5
    - Closing and resetting input streams 5
    - System Properties 6
        - String Interning 6
        - Whitespace Definition 6
    - Code-point case mapping 6
    - Service providers 6
2. Project Structure 9
    - Overview 9
    - API Module 9
    - Runtime Module 9
    - Machine Learning Modules 9
    - Models Module 10
    - Formats Module 10
    - Deep Learning Modules 11
    - CLI Module 11
    - Tools Module (Aggregated Jar) 11
    - Extension Modules 11
    - Migrating from 2.x to 3.x 12
3. Language Detector 13
    - Classifying 13
    - Language Detector Tool 13
    - Language Detector API 14
    - Training 14
        - Training Tool 15
        - Training with Leipzig 15
        - Training API 15
4. Sentence Detector 17
    - Sentence Detection 17
        - Sentence Detection Tool 17
        - Sentence Detection API 17
    - Sentence Detector Training 18
        - Training Tool 18
        - Training API 20
    - Evaluation 21
        - Evaluation Tool 21
5. Tokenizer 22
    - Tokenization 22
        - Tokenizer Tools 22
        - Tokenizer API 23
    - Explicit token character policies 24
    - Independent whitespace tokenizers 25
    - Tokenizer Training 25

Training Tool 25
Training API 26
Detokenizing 27
Detokenizing API 28
Detokenizer Dictionary 29
Unicode Word Segmentation (UAX #29) 29
Word Segmenter 30
Word Tokenizer 30
Lattice tokenization for CJK 31
Subword Tokenization 31
WordPiece 32
6. Text Normalization 33
Introduction 33
The normalizer family 33
Composing a pipeline 35
Offset-aware pipelines 35
Use in DL components 36
Diacritic folding and multilingual safety 37
The CharClass engine and code point sets 37
The layered term model 38
Emoji annotations 39
Confusable (homoglyph) folding 40
Per-language profiles 40
Reference data 41
7. Stopword Filtering 42
Overview 42
Loading a bundled list 42
Loading a custom list 42
Extending or overriding defaults 43
Multi-word stopwords 43
Decorating a Tokenizer 44
Streaming usage 44
Command-line usage 45
Bundled languages 45
Thread-safety 46
8. Name Finder 47
Named Entity Recognition 47
Name Finder Tool 47
Name Finder API 47
Name Finder Training 49
Training Tool 50
Training API 51
Custom Feature Generation 52
Evaluation 54
Evaluation Tool 54
Evaluation API 54
Named Entity Annotation Guidelines 55
9. Document Categorizer 56
Classifying 56
Document Categorizer Tool 56
Document Categorizer API 56
Training 59
Training Tool 59
Training API 59
Training an SVM Model 60
10. Sentiment Analysis 61
Classifying 61
Sentiment Analysis Tool 61

Sentiment Analysis API ... 61
Training ... 62
Training Tool ... 62
Training API ... 62
Evaluation ... 62
Evaluation Tool ... 62
Cross Validation ... 63
Evaluation API ... 63
11. Part-of-Speech Tagger ... 64
Tagging ... 64
POS Tagger Tool ... 64
POS Tagger API ... 64
Training ... 65
Training Tool ... 65
Training API ... 66
Tag Dictionary ... 67
Evaluation ... 69
Evaluation Tool ... 69
12. Lemmatizer ... 71
Lemmatizer Tool ... 71
Lemmatizer API ... 71
Lemmatizer Training ... 73
Training Tool ... 74
Training API ... 74
Lemmatizer Evaluation ... 75
13. Stemmer ... 76
Introduction ... 76
StemmerFactory ... 76
Sharing and caching ... 76
Hunspell dictionaries ... 76
14. Document Annotation Container ... 79
Introduction ... 79
What the layers contain ... 80
Building a pipeline ... 80
Writing a custom annotator ... 81
Term vectors ... 82
15. SpellChecker ... 84
Overview ... 84
SpellChecker Tools ... 84
Building a Model ... 84
Correcting Text ... 85
SpellChecker API ... 87
Choosing an Edit Distance ... 88
Dictionary and Model Format ... 88
Multi-language Models ... 89
Pipeline Integration ... 90
16. Chunker ... 92
Chunking ... 92
Chunker Tool ... 92
Chunking API ... 92
Chunker Training ... 93
Training Tool ... 94
Training API ... 95
Chunker Evaluation ... 95
Chunker Evaluation Tool ... 95
17. Parser ... 97
Parsing ... 97
Parser Tool ... 97

Parsing API 98
Parser Training 99
Training Tool 99
Training API 100
Parser Evaluation 102
Parser Evaluation Tool 102
Evaluation API 103
18. Dependency Parsing 104
Introduction 104
Training data 104
Training and loading a model 104
Command-line tools 104
Feedforward parser 105
Command-line tools 105
Parsing and evaluation 105
19. Entity Linker 107
Overview 107
Preparing Input Spans 107
Implementing an Entity Linker 107
Loading an Implementation 110
Command Line Tool 110
20. Coreference Resolution 111
21. Classpath Loading of OpenNLP Models 112
Background 112
Loading a pre-trained model 112
Bundling a self-trained model 113
Installing third-party resources 114
22. Extending OpenNLP 116
Extension Class Allowlist 116
23. Corpora 117
CONLL 117
CONLL 2000 117
CONLL 2002 119
CONLL 2003 120
CoNLL-U 123
CoNLL-U multiword token lines 123
Arvores Deitadas 123
Getting the data 123
Converting the data (optional) 123
End of file in AD input 124
AD leaf tags with embedded equals signs 124
Text boundaries in AD name samples 124
Training and Evaluation 124
OntoNotes Release 4.0 125
Name Finder Training 125
BioNLP/NLPBA 2004 126
Getting the data 126
Input format 126
Converting the data 127
Brat Format Support 127
Training 128
Evaluation 129
Cross Validation 130
MASC 131
24. Machine Learning 132
Maximum Entropy 132
Implementation 132
Text embedding providers 133

25. UIMA Integration 135
Adding the Dependency 135
Type System 135
Descriptor Structure 135
Sentence Detector 136
Tokenizer 136
Learnable Tokenizer 136
Simple Tokenizer 137
Whitespace Tokenizer 137
Name Finder 137
Learnable Name Finder 137
Dictionary Name Finder 138
POS Tagger 138
Chunker 138
Parser 139
Document Categorizer 139
Language Detector 139
Normalizer 140
Building an Aggregate Pipeline 140
Using Custom Type Systems 141
Running the PEAR Sample in CVD 142
26. Morfologik Addon 143
Morfologik Integration 143
Morfologik CLI Tools 144
27. The Command Line Interface 145
Doccat 145
Doccat 145
DoccatTrainer 145
DoccatEvaluator 145
DoccatCrossValidator 146
DoccatConverter 147
Langdetect 147
LanguageDetector 147
LanguageDetectorTrainer 147
LanguageDetectorConverter 148
LanguageDetectorCrossValidator 148
LanguageDetectorEvaluator 149
Dictionary 150
DictionaryBuilder 150
Tokenizer 150
SimpleTokenizer 150
TokenizerME 150
TokenizerTrainer 150
TokenizerMEEvaluator 152
TokenizerCrossValidator 154
TokenizerConverter 156
DictionaryDetokenizer 157
Sentdetect 157
SentenceDetector 157
SentenceDetectorTrainer 157
SentenceDetectorEvaluator 159
SentenceDetectorCrossValidator 161
SentenceDetectorConverter 164
Namefind 165
TokenNameFinder 165
TokenNameFinderTrainer 165
TokenNameFinderEvaluator 167
TokenNameFinderCrossValidator 169

TokenNameFinderConverter 172
CensusDictionaryCreator 173
Postag 173
POSTagger 173
POSTaggerTrainer 173
POSTaggerEvaluator 175
POSTaggerCrossValidator 176
POSTaggerConverter 177
Lemmatizer 178
LemmatizerME 178
LemmatizerTrainerME 179
LemmatizerEvaluator 179
Chunker 180
ChunkerME 180
ChunkerTrainerME 180
ChunkerEvaluator 181
ChunkerCrossValidator 181
ChunkerConverter 182
Parser 183
Parser 183
ParserTrainer 183
ParserEvaluator 184
ParserConverter 184
BuildModelUpdater 185
CheckModelUpdater 185
TaggerModelReplacer 186
Entitylinker 186
EntityLinker 186
Languagemodel 186
NGramLanguageModel 186
28. Evaluation Test Data 187
What is it? 187
Where is it? 187
How to use the evaluation test data to run test? 187
How to change evaluation data? 187

List of Tables

3.1. Normalizers 13
8.1. Feature Generators 53
14.1. Layers provided by the toolkit's adapters 80
24.1. Separators written for the regular expression of 2.x 132

List of Examples

14.1. A document with three annotation layers 79

Chapter 1. Introduction

Description

The Apache OpenNLP library is a machine learning based toolkit for the processing of natural language text. It supports the most common NLP tasks, such as tokenization, sentence segmentation, part-of-speech tagging, named entity extraction, chunking, parsing, coreference resolution, and sentiment analysis. These tasks are usually required to build more advanced text processing services. OpenNLP also includes maximum entropy and perceptron based machine learning.

The goal of the OpenNLP project will be to create a mature toolkit for the aforementioned tasks. An additional goal is to provide a large number of pre-built models for a variety of languages, as well as the annotated text resources that those models are derived from.

General Library Structure

The Apache OpenNLP library contains several components, enabling one to build a full natural language processing pipeline. These components include: sentence detector, tokenizer, name finder, document categorizer, sentiment analyzer, part-of-speech tagger, chunker, parser, coreference resolution. Components contain parts which enable one to execute the respective natural language processing task, to train a model and often also to evaluate a model. Each of these facilities is accessible via its application program interface (API). In addition, a command line interface (CLI) is provided for convenience of experiments and training.

Application Program Interface (API)

OpenNLP components have similar APIs. Normally, to execute a task, one should provide a model and an input.

Generic Example

A model is usually loaded by providing a FileInputStream with a model to a constructor of the model class:

```
try (InputStream modelIn = new FileInputStream("lang-model-name.bin")) {
  SomeModel model = new SomeModel(modelIn);
}
```

After the model is loaded the tool itself can be instantiated.

```
ToolName toolName = new ToolName(model);
```

After the tool is instantiated, the processing task can be executed. The input and the output formats are specific to the tool, but often the output is an array of String, and the input is a String or an array of String.

```
String[] output = toolName.executeTask("This is a sample text.");
```

Command line interface (CLI)

Description

OpenNLP provides a command line script, serving as a unique entry point to all included tools. The script is located in the bin directory of OpenNLP binary distribution. Included are versions for Windows: opennlp.bat and Linux or compatible systems: opennlp.

List of tools

The list of command line tools for Apache OpenNLP 3.0.0, as well as a description of its arguments, is available at section Chapter 27, *The Command Line Interface*.

Setting up

OpenNLP script uses JAVA_CMD and JAVA_HOME variables to determine which command to use to execute Java virtual machine.

OpenNLP script uses OPENNLP_HOME variable to determine the location of the binary distribution of OpenNLP. It is recommended to point this variable to the binary distribution of current OpenNLP version and update PATH variable to include $OPENNLP_HOME/bin or %OPENNLP_HOME%\bin.

Such configuration allows calling OpenNLP conveniently. Examples below suppose this configuration has been done.

Generic Example

Apache OpenNLP provides a common command line script to access all its tools:

```
$ opennlp
```

This script prints current version of the library and lists all available tools:

```
OpenNLP <VERSION>. Usage: opennlp TOOL
where TOOL is one of:
  Doccat                        learnable document categorizer
  DoccatTrainer                 trainer for the learnable document categorizer
  DoccatConverter               converts leipzig data format to native OpenNLP format
  DictionaryBuilder             builds a new dictionary
  SimpleTokenizer               character class tokenizer
  TokenizerME                   learnable tokenizer
  TokenizerTrainer              trainer for the learnable tokenizer
  TokenizerMEEvaluator          evaluator for the learnable tokenizer
  TokenizerCrossValidator       K-fold cross validator for the learnable tokenizer
  TokenizerConverter            converts foreign data formats (namefinder,conllx,pos) to native OpenNLP format
  DictionaryDetokenizer
  SentenceDetector              learnable sentence detector
  SentenceDetectorTrainer       trainer for the learnable sentence detector
  SentenceDetectorEvaluator     evaluator for the learnable sentence detector
  SentenceDetectorCrossValidator  K-fold cross validator for the learnable sentence detector
  SentenceDetectorConverter     converts foreign data formats (namefinder,conllx,pos) to native OpenNLP format
  TokenNameFinder               learnable name finder
  TokenNameFinderTrainer        trainer for the learnable name finder
  TokenNameFinderEvaluator      Measures the performance of the NameFinder model with the reference data
  TokenNameFinderCrossValidator   K-fold cross validator for the learnable Name Finder
  TokenNameFinderConverter      converts foreign data formats (bionlp2004,conll03,conll02,ad) to native OpenNLP
 format
  CensusDictionaryCreator       Converts 1990 US Census names into a dictionary
  POSTagger                     learnable part of speech tagger
  POSTaggerTrainer              trains a model for the part-of-speech tagger
  POSTaggerEvaluator            Measures the performance of the POS tagger model with the reference data
  POSTaggerCrossValidator       K-fold cross validator for the learnable POS tagger
  POSTaggerConverter            converts conllx data format to native OpenNLP format
  ChunkerME                     learnable chunker
  ChunkerTrainerME              trainer for the learnable chunker
  ChunkerEvaluator              Measures the performance of the Chunker model with the reference data
  ChunkerCrossValidator         K-fold cross validator for the chunker
  ChunkerConverter              converts ad data format to native OpenNLP format
  Parser                        performs full syntactic parsing
  ParserTrainer                 trains the learnable parser
  ParserEvaluator     Measures the performance of the Parser model with the reference data
  BuildModelUpdater             trains and updates the build model in a parser model
  CheckModelUpdater             trains and updates the check model in a parser model
  TaggerModelReplacer           replaces the tagger model in a parser model
All tools print help when invoked with help parameter
Example: opennlp SimpleTokenizer help
```

OpenNLP tools have similar command line structure and options. To discover tool options, run it with no parameters:

```
$ opennlp ToolName
```

The tool will output two blocks of help.

The first block describes the general structure of this tool command line:

```
Usage: opennlp TokenizerTrainer[.namefinder|.conllx|.pos] [-abbDict path] ...  -model modelFile ...
```

The general structure of this tool command line includes the obligatory tool name (TokenizerTrainer), the optional format parameters ([.namefinder|.conllx|.pos]), the optional parameters ([-abbDict path] ...), and the obligatory parameters (-model modelFile ...).

The format parameters enable direct processing of non-native data without conversion. Each format might have its own parameters, which are displayed if the tool is executed without or with help parameter:

```
$ opennlp TokenizerTrainer.conllx help
```

```
Usage: opennlp TokenizerTrainer.conllx [-abbDict path] [-alphaNumOpt isAlphaNumOpt] ...

Arguments description:
	-abbDict path
		abbreviation dictionary in XML format.
	...
```

To switch the tool to a specific format, add a dot and the format name after the tool name:

```
$ opennlp TokenizerTrainer.conllx -model en-pos.bin ...
```

The second block of the help message describes the individual arguments:

```
Arguments description:
	-type maxent|perceptron|perceptron_sequence
		The type of the token name finder model. One of maxent|perceptron|perceptron_sequence.
	-dict dictionaryPath
		The XML tag dictionary file
	...
```

Most tools for processing need to be provided at least a model:

```
$ opennlp ToolName lang-model-name.bin
```

When tool is executed this way, the model is loaded and the tool is waiting for the input from standard input. This input is processed and printed to standard output.

Alternative, or one should say, most commonly used way is to use console input and output redirection options to provide also an input and an output files:

```
$ opennlp ToolName lang-model-name.bin < input.txt > output.txt
```

Most tools for model training need to be provided first a model name, optionally some training options (such as model type, number of iterations), and then the data.

A model name is just a file name.

Training options often include number of iterations, cutoff, abbreviations dictionary or something else. Sometimes it is possible to provide these options via training options file. In this case these options are ignored and the ones from the file are used.

For the data one has to specify the location of the data (filename) and often language and encoding.

A generic example of a command line to launch a tool trainer might be:

```
$ opennlp ToolNameTrainer -model en-model-name.bin -lang en -data input.train -encoding UTF-8
```

or with a format:

```
$ opennlp ToolNameTrainer.conll03 -model en-model-name.bin -lang en -data input.train \
                        -types per -encoding UTF-8
```

Most tools for model evaluation are similar to those for task execution, and need to be provided fist a model name, optionally some evaluation options (such as whether to print misclassified samples), and then the test data. A generic example of a command line to launch an evaluation tool might be:

```
$ opennlp ToolNameEvaluator -model en-model-name.bin -lang en -data input.test -encoding UTF-8
```

OpenNLP Models

OpenNLP Models

OpenNLP supports training NLP models that can be used by OpenNLP. In this documentation we will refer to these models as "OpenNLP models." All NLP components of OpenNLP support this type of model. The sections below in this documentation describe how to train and use these models. Pre-trained models [https://opennlp.apache.org/models.html] are available for some languages and some OpenNLP components.

ONNX Models

OpenNLP supports ONNX models via the ONNX Runtime for the Chapter 8, *Name Finder* and Chapter 9, *Document Categorizer*. This allows models trained by other frameworks such as PyTorch and Tensorflow to be used by OpenNLP. The documentation for each of the OpenNLP components that supports ONNX models describes how to use ONNX models for inference. DL inference uses Unicode-aware text chunking and optional input normalization; see the section called “Use in DL components”.

Note

OpenNLP does not support training models that can be used by the ONNX Runtime - ONNX models must be created outside OpenNLP using other tools.

Closing and resetting input streams

Close corpus streams after use, preferably with try-with-resources. `PlainTextByLineStream.reset()` closes the reader from the previous pass before asking its `InputStreamFactory` for a new stream. The factory must therefore provide a new, readable stream for each pass, rather than return the stream that has just been closed.

The markable file stream that the command line tools and the format converters open for corpus files releases its file handle on close, including when closed through a wrapping reader. The Leipzig language-sample reader also closes the stream used to count source lines.

System Properties

String Interning

OpenNLP provides different implementations for String interning to reduce memory footprint. By default, OpenNLP uses a custom String interner implementation.

Users may override by setting the following system property:

```
-Dopennlp.interner.class=opennlp.tools.util.jvm.JvmStringInterner
```

In addition, users can provide custom String interner implementations by implementing the interface 'StringInterner' and specify this class via 'opennlp.interner.class'.

Whitespace Definition

The Unicode standard defines whitespace as the 25 code points with the binary `White_Space` property (`PropList.txt` [https://www.unicode.org/Public/18.0.0/ucd/PropList.txt], Unicode 18.0.0). The set includes space and tab, the next-line control (`U+0085`), no-break and typographic spaces such as `U+00A0` and `U+2007`, and the line and paragraph separators (`U+2028`/`U+2029`). Java's `Character.isWhitespace` omits the no-break spaces and `U+0085` but includes the information separators `U+001C`..`U+001F`; Java's default `\s` regex class matches ASCII whitespace only unless Unicode character-class mode is enabled. Text extracted from PDFs, HTML, and word processors often contains the wider set, on which the two definitions produce different tokens.

Tokenization, corpus format parsing, and feature generation therefore classify whitespace using the Unicode `White_Space` property by default, via `StringUtil.isWhitespace` and the related `StringUtil.splitOnUnicodeWhitespace` / `trimUnicodeWhitespace` helpers. Models trained with OpenNLP 1.x or 2.x, which include the models published by the project, used the legacy definition (the union of `Character.isWhitespace` and the Unicode `Zs` category). Set the property below when applying those models to reproduce their tokenization at the code points where the two definitions differ.

Users may switch to the legacy definition by setting the following system property:

```
-Dopennlp.whitespace.mode=LEGACY
```

Accepted values are `LEGACY` and `UNICODE` (the default), case-insensitive; any other value throws an `IllegalArgumentException`. The setting applies process-wide, so training and decoding of a given model must use the same value. Legacy support is scheduled for removal in 4.0.

Code-point case mapping

`StringUtil.toLowerCase` and `StringUtil.toUpperCase` map individual Unicode code points independently of the default locale. Uppercase conversion includes supplementary letters such as Deseret. These helpers do not expand one character into several characters or apply contextual final-sigma rules. Unpaired UTF-16 surrogates are preserved.

Service providers

OpenNLP has two ways to reach an implementation that core does not know. `ExtensionLoader` instantiates a class whose name comes from a model manifest, an XML descriptor or a training parameter, so the caller must

already know that name. The provider SPI in `opennlp.tools.util.ext` works the other way around: a jar on the class path announces what it can do, and OpenNLP picks an implementation for the request at hand. It is meant for optional implementations that ship separately, such as an annotator, a stemmer or an embedder.

Each service interface has a provider interface of its own that extends `Provider`, for example `DocumentAnnotatorProvider extends Provider<DocumentAnnotator>`. An implementation needs a public no-argument constructor and is listed in `META-INF/services/` under the fully qualified name of that provider interface, the usual `java.util.ServiceLoader` layout. A provider states its `name()`, a `priority()`, whether it `isAvailable()` in the running process, for example whether an optional dependency is present, and whether it `supports` a request. The request is a `ProviderSpec`: an optional location, such as a model file, and string options. Neither the constructor nor the static initializer, the availability check or `supports` may load a model or initialize a native library, because they run while the candidates are collected, before one of them is chosen.

```
DocumentAnnotator createAnnotator() throws IOException {
  ProviderSpec spec = ProviderSpec.of(Map.of("algorithm", "english"));
  Providers<DocumentAnnotatorProvider> providers = Providers.of(DocumentAnnotatorProvider.class);
  return providers.select(spec).create(spec);
}
```

`Providers.of` scans the class path once, so an application creates its instance at startup and keeps it, rather than resolving per request. The instance is immutable and thread-safe. Keep it in a field owned by the application, for example a servlet context attribute or an application scoped bean; a `static` field is wrong when OpenNLP is deployed in a class loader shared by several applications, because it would keep one application's providers and their class loader alive. The lookup uses the thread context class loader when that loader sees the provider interface, and the class loader of the provider interface otherwise. On a thread whose context class loader is not the application's, such as a thread of a pool, only the providers of the provider interface's own class loader are found, so a library or a framework integration captures the class loader at startup and passes it to `Providers.of(spi, loader)`. Resolving is not free either: keep what `select` returned instead of resolving per request, and keep in mind that a provider blocking in `isAvailable()` or `supports()` blocks every resolution of that provider interface.

`select` returns the available provider with the highest priority that supports the request. If several share the highest priority, an `AmbiguousProviderException` names them; if none supports the request, an `UnsatisfiedProviderException` names the installed providers. `supporting` returns every matching provider, best first, and `byName` resolves one by name. Providers may share a name, in which case the higher priority wins, which is how a module replaces the provider of another module. What `create` returns belongs to the caller, who also closes it if it is closeable.

Three keys configure resolution, all named after the provider interface. `opennlp.provider.<interface>` selects a provider by name, for example `opennlp.provider.opennlp.tools.document.DocumentAnnotatorProvider=stemmer`. It narrows the providers that support the request, so a selected provider that does not support the request is reported as unsatisfied rather than used. `opennlp.provider.<interface>.disabled` holds a comma separated list of provider names and provider class names that are not resolved; a name stays listed by `installed()`, while a class name is never instantiated, which keeps a provider out of the process entirely and therefore has to be configured before the providers are looked up. The third key, `opennlp.provider.<interface>.allowed`, is described below. The first two are read from `META-INF/opennlp/providers.properties`, encoded in UTF-8, of the class loader the providers were looked up with; where several files carry the same key, the one with the highest `configuration.ordinal` wins, and a system property of the same name wins over all files. In a native image the file is read only when it is registered as a resource.

Adding a jar to the class path is a decision to trust its code: every provider of the provider interface is instantiated when the providers are looked up, and any jar can carry a `providers.properties` that selects or disables a provider, which OpenNLP logs with the file it came from. Providers may also share a name, so a name is a convenient handle and not a guarantee of which implementation answers. An application that must not be reconfigured by a library on its class path passes its own source, which is also how two deployments in one application server configure themselves independently: `Providers.of(spi, loader, key -> myConfiguration.get(key))`.

An operator who wants to decide which implementations may run at all uses `opennlp.provider.<interface>.allowed`, a comma separated list of provider class names and of package prefixes, which have to end with a dot. Only those classes are instantiated, every other provider of that interface is left out whatever it is named, an unset or blank value allows all of them and a value that holds only separators allows none. It is read from a system property or from the source an application passes, never from a `providers.properties` that any jar could carry, and it is checked before a provider class is instantiated, so it has to be configured before the providers are looked up. A provider left out this way is counted in the message of an `UnsatisfiedProviderException` together with the key.

What the allowed key decides is which class names may be instantiated, not where their code comes from. Two jars can carry the same class name, and which one is loaded is the class loader's decision, so a package prefix widens that further. A provider name is weaker still: names may be shared between jars, so `opennlp.provider.<interface>` is a convenience for choosing between implementations, not a security control.

Where the origin of an implementation matters, decide by the class instead of by its name:

```
Providers<DocumentAnnotatorProvider> providers = Providers.ofTrusted(
    DocumentAnnotatorProvider.class, loader,
    type -> expected.equals(type.getProtectionDomain().getCodeSource().getLocation()));
```

The predicate is asked for every provider class the configuration keys leave, before an instance of it is created, and it can look at the code source, the module or the signers of the class. The class is loaded and linked before it is asked, including whatever a custom class loader does while loading it, but it is not initialized: a rejected provider runs neither its static initializer nor its constructor, and is counted among the skipped ones. The predicate must therefore not initialize the class itself, for instance by reading a static field, and must be deterministic, thread-safe and free of blocking calls. A predicate that throws ends the lookup, so a failure never widens the policy. The strongest boundary remains the class path itself: look the providers up with a class loader that sees only the jars you trust. The lookup also logs at debug level which location each provider class was loaded from, for auditing.

A provider that cannot be loaded, whose name is unusable, or that throws from `name()` or `priority()` is skipped while the providers are looked up, so one broken jar does not hide the others, and the number of skipped providers appears in the message of an `UnsatisfiedProviderException`. A provider that throws from `isAvailable()` or `supports()` is left out of that one resolution instead. Both are reported once per provider and method as a warning and at debug level afterwards. A configuration file that cannot be read, and a class loader that cannot list its files at all, fail the lookup with a `ProviderResolutionException`. Discovery works on the class path; OpenNLP declares no JPMS `provides` directives for the module path.

The runtime registers one provider, `stemmer`, which creates a `StemmerAnnotator` for a request without location whose only option is `algorithm`: `porter`, which is also the default when the option is absent, for the Porter stemmer, or the name of a Snowball algorithm.

Chapter 2. Project Structure

Overview

Starting with version 3.0, Apache OpenNLP has been reorganized from a single monolithic `opennlp-tools` artifact into a set of fine-grained modules. This modularization allows users to depend only on the components they actually need, resulting in a smaller dependency footprint. At the same time, the public API remains stable and fully compatible with previous 2.x releases.

The following sections describe each module, its purpose, and when to include it as a dependency.

API Module

The `opennlp-api` module defines the public interfaces and abstractions that form the contract between OpenNLP and its users. It contains the core interfaces such as `Tokenizer`, `SentenceDetector`, `POSTagger`, `TokenNameFinder`, `Chunker`, `Parser`, `LanguageDetector`, `Lemmatizer`, and `DocumentCategorizer`.

This module also provides shared base classes such as `BaseModel`, the `ObjectStream` abstraction for data processing, the command-line argument parsing framework, and common utility types. It is a transitive dependency of `opennlp-runtime` and typically does not need to be declared explicitly.

```
<dependency>
   <groupId>org.apache.opennlp</groupId>
   <artifactId>opennlp-api</artifactId>
   <version>CURRENT_OPENNLP_VERSION</version>
</dependency>
```

Runtime Module

The `opennlp-runtime` module is the primary dependency for most users. It contains the core NLP tool implementations including sentence detection, tokenization, part-of-speech tagging, named entity recognition, chunking, parsing, language detection, lemmatization, and document categorization.

By default, `opennlp-runtime` ships with the Maximum Entropy machine learning implementation. If you need other ML algorithms, add the corresponding ML module as described below.

```
<dependency>
   <groupId>org.apache.opennlp</groupId>
   <artifactId>opennlp-runtime</artifactId>
   <version>CURRENT_OPENNLP_VERSION</version>
</dependency>
```

Machine Learning Modules

The machine learning implementations have been separated into individual modules so that applications can include only the algorithms they use. Each module provides a specific ML algorithm and is loaded at runtime via the `ExtensionLoader` service discovery mechanism.

- `opennlp-ml-commons` — Shared ML utilities and base classes used by all ML algorithm modules. This is a transitive dependency of each ML module and does not need to be declared explicitly.
- `opennlp-ml-maxent` — Maximum Entropy classifier. This is the default algorithm and is included transitively via `opennlp-runtime`.
- `opennlp-ml-perceptron` — Perceptron-based learning algorithm. Add this dependency if your models use the Perceptron or Perceptron Sequence trainer.
- `opennlp-ml-bayes` — Naive Bayes classifier. Add this dependency if your models use the Naive Bayes trainer.
- `opennlp-ml-libsvm` — Support Vector Machine (SVM) based text classification via zlibsvm [https://github.com/rzo1/zlibsvm]. This module provides `DocumentCategorizerSVM`, an SVM-based implementation of the `DocumentCategorizer` interface with configurable term weighting (binary, TF, TF-IDF, log-normalized TF), feature selection (information gain, chi-square, term frequency, document frequency), and feature scaling.

For example, to use the Perceptron trainer alongside the default Maximum Entropy, add:

```
<dependency>
  <groupId>org.apache.opennlp</groupId>
  <artifactId>opennlp-ml-perceptron</artifactId>
  <version>CURRENT_OPENNLP_VERSION</version>
</dependency>
```

Models Module

The `opennlp-model-resolver` module provides classpath-based model discovery and loading. It enables applications to bundle pre-trained OpenNLP models as JAR files and load them at runtime without explicit file path references. See Chapter 21, *Classpath Loading of OpenNLP Models* for details on classpath model loading.

```
<dependency>
  <groupId>org.apache.opennlp</groupId>
  <artifactId>opennlp-model-resolver</artifactId>
  <version>CURRENT_OPENNLP_VERSION</version>
</dependency>
```

Formats Module

The `opennlp-formats` module supports reading and writing various NLP training and evaluation data formats, including CoNLL, BioNLP, BRAT, AD (Floresta), Leipzig, and others. Include this module if you need to train models from data in non-native OpenNLP formats.

```
<dependency>
  <groupId>org.apache.opennlp</groupId>
  <artifactId>opennlp-formats</artifactId>
  <version>CURRENT_OPENNLP_VERSION</version>
</dependency>
```

Deep Learning Modules

OpenNLP provides optional support for ONNX-based neural models via two modules:

- `opennlp-dl` — Integrates the ONNX Runtime for CPU-based inference. This module enables the use of models trained by external frameworks such as PyTorch or TensorFlow, exported in the ONNX format.
- `opennlp-dl-gpu` — Replaces the CPU ONNX Runtime with the GPU-accelerated variant for systems with supported GPU hardware. Use this module instead of `opennlp-dl` when GPU acceleration is available and desired.

```
<!-- CPU variant -->
<dependency>
   <groupId>org.apache.opennlp</groupId>
   <artifactId>opennlp-dl</artifactId>
   <version>CURRENT_OPENNLP_VERSION</version>
</dependency>

<!-- OR GPU variant (do not include both) -->
<dependency>
   <groupId>org.apache.opennlp</groupId>
   <artifactId>opennlp-dl-gpu</artifactId>
   <version>CURRENT_OPENNLP_VERSION</version>
</dependency>
```

CLI Module

The `opennlp-cli` module provides the command-line tools for training, evaluating, and running OpenNLP models from a terminal. It is included in the binary distribution and not typically needed as a library dependency. See Chapter 27, *The Command Line Interface* for details on available CLI commands.

Tools Module (Aggregated Jar)

The `opennlp-tools` module is an aggregated artifact that bundles all core modules (`opennlp-api`, `opennlp-runtime`, all ML modules, `opennlp-model-resolver`, `opennlp-formats`, and `opennlp-cli`) into a single JAR. It is provided for backwards compatibility with 2.x and for the binary distribution.

For new projects, we recommend depending on `opennlp-runtime` plus only the specific additional modules you need, rather than pulling in the full `opennlp-tools` artifact.

Extension Modules

OpenNLP provides optional extension modules for integration with external frameworks:

- `opennlp-morfologik` — Integrates the Morfologik [https://github.com/morfologik] library for dictionary-based stemming and lemmatization. See ??? for usage details.
- `opennlp-spellcheck` — Provides a native, language-agnostic spell corrector based on the SymSpell (Symmetric Delete) algorithm, including CLI tools, a pluggable edit distance, and pipeline integration. See Chapter 15, *SpellChecker* for usage details.

- `opennlp-uima` — Provides a set of Apache UIMA [https://uima.apache.org] annotators that wrap OpenNLP components for use in UIMA pipelines. See ??? for integration details.

Migrating from 2.x to 3.x

The 3.x release introduces no known breaking API changes. Existing code using the `opennlp-tools` artifact will continue to work without modification. However, we strongly recommend migrating to the modular dependency structure for a smaller footprint.

A minimal migration replaces:

```
<!-- 2.x: single monolithic dependency -->
<dependency>
   <groupId>org.apache.opennlp</groupId>
   <artifactId>opennlp-tools</artifactId>
   <version>2.x.y</version>
</dependency>
```

with:

```
<!-- 3.x: modular dependencies — add only what you need -->
<dependency>
   <groupId>org.apache.opennlp</groupId>
   <artifactId>opennlp-runtime</artifactId>
   <version>CURRENT_OPENNLP_VERSION</version>
</dependency>
<!-- Add opennlp-model-resolver, opennlp-ml-perceptron, opennlp-dl, etc. as needed -->
```

Note

The `opennlp-runtime` module includes the Maximum Entropy ML implementation by default. If your models were trained with the Perceptron or Naive Bayes algorithm, add the corresponding `opennlp-ml-perceptron` or `opennlp-ml-bayes` dependency.

Chapter 3. Language Detector

Classifying

The OpenNLP Language Detector classifies a document in ISO-639-3 languages according to the model capabilities. A model can be trained with Maxent, Perceptron or Naive Bayes algorithms. By default, normalizes a text and the context generator extracts n-grams of size 1, 2 and 3. The n-gram sizes, the normalization and the context generator can be customized by extending the LanguageDetectorFactory.

The default normalizers are:

Table 3.1. Normalizers

Normalizer	Description
EmojiCharSequenceNormalizer	Replaces each run of complete emoji (single emoji, emoji with U+FE0F, flags, keycaps, skin tone modifier sequences, ZWJ sequences and tag sequences) with one space, so `go####now` becomes `go now`. Text symbols, letters and ideographs from other planes, and an emoji with a stray joiner, modifier or tag next to it are kept as they are.
UrlCharSequenceNormalizer	Replaces each http or https URL and each email address with a blank space. A URL is removed as a whole, up to the next whitespace or delimiter, including a port, user information, percent escapes and non-ASCII hosts; URLs with other schemes are kept. Models trained with a release before 3.0.0 saw only the leading ASCII part of a URL removed and get that output again under `-Dopennlp.compat.mode=LEGACY` until they are retrained.
SocialMediaCharSequenceNormalizer	Replaces each hashtag, user handle, run of retweet markers and emoticon with one space, and shrinks repeated laughter to two repetitions.
NumberCharSequenceNormalizer	Replaces number sequences by blank spaces
ShrinkCharSequenceNormalizer	Shrink characters that repeats three or more times to only two repetitions.

EmojiCharSequenceNormalizer removes complete emoji only. Models trained with a release before 3.0.0 saw every hyphen, every character from U+E000 to U+FFFF and nearly every supplementary character removed, emoji or not. They get that output again under `-Dopennlp.compat.mode=LEGACY` until they are retrained.

Language Detector Tool

The easiest way to try out the language detector is the command line tool. The tool is only intended for demonstration and testing. The following command shows how to use the language detector tool.

```
$ bin/opennlp LanguageDetector model
```

The input is read from standard input and output is written to standard output, unless they are redirected or piped.

Language Detector API

To perform classification you will need a machine learning model - these are encapsulated in the LanguageDetectorModel class of OpenNLP tools.

First you need to grab the bytes from the serialized model on an InputStream - we'll leave it you to do that, since you were the one who serialized it to begin with. Now for the easy part:

```
InputStream is = ...
LanguageDetectorModel m = new LanguageDetectorModel(is);
```

With the LanguageDetectorModel in hand we are just about there:

```
String inputText = ...
LanguageDetector myCategorizer = new LanguageDetectorME(m);

// Get the most probable language
Language bestLanguage = myCategorizer.predictLanguage(inputText);
System.out.println("Best language: " + bestLanguage.getLang());
System.out.println("Best language confidence: " + bestLanguage.getConfidence());

// Get an array with the most probable languages
Language[] languages = myCategorizer.predictLanguages(null);
```

Note that the both the API or the CLI will consider the complete text to choose the most probable languages. To handle mixed language one can analyze smaller chunks of text to find language regions.

Training

The Language Detector can be trained on annotated training material. The data can be in OpenNLP Language Detector training format. This is one document per line, containing the ISO-639-3 language code and text separated by a tab. Other formats can also be available. The following sample shows the sample from above in the required format.

```
spa     A la fecha tres calles bonaerenses recuerdan su nombre (en Ituzaingó, Merlo y Campana). A la fecha, unas 50 \
  naves y 20 aviones se han perdido en esa área particular del océano Atlántico.
deu     Alle Jahre wieder: Millionen Spanier haben am Dienstag die Auslosung in der größten Lotterie der Welt verfolgt.\
   Alle Jahre wieder: So gelingt der stressfreie Geschenke-Umtausch Artikel per E-Mail empfehlen So gelingt der \
   stressfre ie Geschenke-Umtausch Nicht immer liegt am Ende das unter dem Weihnachtsbaum, was man sich gewünscht
 hat.
srp     ###### ########## ######## ## ###### ## ##### ### ########, #### ## ###### ## ###### ######## ###### #
 ###### \
  ####### # ######## ##### ##### ##### # ####. ###### ########## ###### ### ######.
lav     Egija Tri-Active proced#ru #paši iesaka izmantot silt#kajos gadalaikos, jo ziem# aukstums var š#ist ar# \
  nepat#kams. Vald#ba vienoj#s, ka izmai#as nodok#u politik# tiek konceptu#li atbalst#tas, tom#r deva \
  ned##u laika Ekonomikas ministrijai, Finanšu ministrijai un Labkl#j#bas ministrijai, lai ar vienotu \
  poz#ciju atgrieztos pie jaut#juma izskat#šanas.
```

Note

The line breaks marked with a backslash are just inserted for formatting purposes and must not be included in the training data.

Training Tool

The following command will train the language detector and write the model to langdetect-custom.bin:

```
$ bin/opennlp LanguageDetectorTrainer[.leipzig] -model modelFile [-params paramsFile] \
  [-factory factoryName] -data sampleData [-encoding charsetName]
```

Tip

To customize the language detector, extend the class opennlp.tools.langdetect.LanguageDetectorFactory add it to the classpath and pass it in the -factory argument.

Training with Leipzig

The Leipzig Corpora collection presents corpora in different languages. The corpora are a collection of individual sentences collected from the web and newspapers. The Corpora are available as plain text and as MySQL database tables. The OpenNLP integration can only use the plain text version. The individual plain text packages can be downloaded here: https://wortschatz.uni-leipzig.de/en/download

This corpora is specially good to train Language Detector and a converter is provided. First, you need to download the files that compose the Leipzig Corpora collection to a folder. The language of each file is taken from the first three characters of its name, the ISO 639-3 code in lower case, as in `eng_news_2005_10K-sentences.txt`. Apache OpenNLP Language Detector supports training, evaluation and cross validation using the Leipzig Corpora. For example, the following command shows how to train a model.

```
$ bin/opennlp LanguageDetectorTrainer.leipzig -model modelFile [-params paramsFile] [-factory factoryName] \
 -sentencesDir sentencesDir -sentencesPerSample sentencesPerSample -samplesPerLanguage samplesPerLanguage \
 [-encoding charsetName]
```

The following sequence of commands shows how to convert the Leipzig Corpora collection at folder leipzig-train/ to the default Language Detector format, by creating groups of 5 sentences as documents and limiting to 10000 documents per language. Then, it shuffles the result and select the first 100000 lines as train corpus and the last 20000 as evaluation corpus:

```
$ bin/opennlp LanguageDetectorConverter leipzig -sentencesDir leipzig-train/ -sentencesPerSample 5 -samplesPerLanguage
 10000 > leipzig.txt
$ perl -MList::Util=shuffle -e 'print shuffle(<STDIN>);' < leipzig.txt > leipzig_shuf.txt
$ head -100000 < leipzig_shuf.txt > leipzig.train
$ tail -20000 < leipzig_shuf.txt > leipzig.eval
```

Training API

The following example shows how to train a model from API.

```
InputStreamFactory inputStreamFactory = new MarkableFileInputStreamFactory(new File("corpus.txt"));

ObjectStream<String> lineStream = new PlainTextByLineStream(inputStreamFactory, StandardCharsets.UTF_8);
ObjectStream<LanguageSample> sampleStream = new LanguageDetectorSampleStream(lineStream);

TrainingParameters params = ModelUtil.createDefaultTrainingParameters();
params.put(TrainingParameters.ALGORITHM_PARAM, PerceptronTrainer.PERCEPTRON_VALUE);
params.put(TrainingParameters.CUTOFF_PARAM, 0);

LanguageDetectorFactory factory = new LanguageDetectorFactory();

LanguageDetectorModel model = LanguageDetectorME.train(sampleStream, params, factory);
model.serialize(new File("langdetect-custom.bin"));
```

Chapter 4. Sentence Detector

Sentence Detection

The OpenNLP Sentence Detector can detect that a punctuation character marks the end of a sentence or not. In this sense a sentence is defined as the longest white space trimmed character sequence between two punctuation marks. The first and last sentence make an exception to this rule. The first non whitespace character is assumed to be the start of a sentence, and the last non whitespace character is assumed to be a sentence end. The sample text below should be segmented into its sentences.

```
Pierre Vinken, 61 years old, will join the board as a nonexecutive director Nov. 29. Mr. Vinken is
chairman of Elsevier N.V., the Dutch publishing group. Rudolph Agnew, 55 years
old and former chairman of Consolidated Gold Fields PLC, was named a director of this
British industrial conglomerate.
```

After detecting the sentence boundaries each sentence is written in its own line.

```
Pierre Vinken, 61 years old, will join the board as a nonexecutive director Nov. 29.
Mr. Vinken is chairman of Elsevier N.V., the Dutch publishing group.
Rudolph Agnew, 55 years old and former chairman of Consolidated Gold Fields PLC,
   was named a director of this British industrial conglomerate.
```

Usually Sentence Detection is done before the text is tokenized and that's the way the pre-trained models on the website are trained, but it is also possible to perform tokenization first and let the Sentence Detector process the already tokenized text. The OpenNLP Sentence Detector cannot identify sentence boundaries based on the contents of the sentence. A prominent example is the first sentence in an article where the title is mistakenly identified to be the first part of the first sentence. Most components in OpenNLP expect input which is segmented into sentences.

Sentence Detection Tool

The easiest way to try out the Sentence Detector is the command line tool. The tool is only intended for demonstration and testing. Download the english sentence detector model and start the Sentence Detector Tool with this command:

```
$ opennlp SentenceDetector opennlp-en-ud-ewt-sentence-1.3-2.5.4.bin
```

Just copy the sample text from above to the console. The Sentence Detector will read it and echo one sentence per line to the console. Usually the input is read from a file and the output is redirected to another file. This can be achieved with the following command.

```
$ opennlp SentenceDetector opennlp-en-ud-ewt-sentence-1.3-2.5.4.bin < input.txt > output.txt
```

For the english sentence model from the website the input text should not be tokenized.

Sentence Detection API

The Sentence Detector can be easily integrated into an application via its API. To instantiate the Sentence Detector the sentence model must be loaded first.

```
try (InputStream modelIn = new FileInputStream("opennlp-en-ud-ewt-sentence-1.3-2.5.4.bin")) {
  SentenceModel model = new SentenceModel(modelIn);
}
```

After the model is loaded the SentenceDetectorME can be instantiated.

```
SentenceDetectorME sentenceDetector = new SentenceDetectorME(model);
```

As of OpenNLP 3.0.0 SentenceDetectorME is thread safe and a single instance can be shared across threads; the legacy ThreadSafeSentenceDetectorME wrapper is retained for backward compatibility and is now deprecated. The Sentence Detector can output an array of Strings, where each String is one sentence.

```
String[] sentences = sentenceDetector.sentDetect("  First sentence. Second sentence.  ");
```

The result array now contains two entries. The first String is "First sentence." and the second String is "Second sentence." The whitespace before, between and after the input String is removed. The API also offers a method which simply returns the span of the sentence in the input string.

```
Span[] sentences = sentenceDetector.sentPosDetect("  First sentence. Second sentence.  ");
```

The result array again contains two entries. The first span beings at index 2 and ends at 17. The second span begins at 18 and ends at 34. The utility method Span.getCoveredText can be used to create a substring which only covers the chars in the span.

Sentence Detector Training

Training Tool

OpenNLP has a command line tool which is used to train the models available from the model download page on various corpora. The data must be converted to the OpenNLP Sentence Detector training format. Which is one sentence per line. An empty line indicates a document boundary. In case the document boundary is unknown, it's recommended to have an empty line every few ten sentences. Exactly like the output in the sample above. Usage of the tool:

```
$ opennlp SentenceDetectorTrainer
Usage: opennlp SentenceDetectorTrainer[.namefinder|.conllx|.pos] [-abbDict path] \
        [-params paramsFile] [-iterations num] [-cutoff num] -model modelFile \
        -lang language -data sampleData [-encoding charsetName]

Arguments description:
    -abbDict path
        abbreviation dictionary in XML format.
    -params paramsFile
        training parameters file.
    -iterations num
        number of training iterations, ignored if -params is used.
    -cutoff num
        minimal number of times a feature must be seen, ignored if -params is used.
    -model modelFile
        output model file.
    -lang language
        language which is being processed.
    -data sampleData
        data to be used, usually a file name.
    -encoding charsetName
        encoding for reading and writing text, if absent the system default is used.
    -useTokenEnd boolean flag
        set to false when the next sentence in the test dataset doesn't start with a blank space post completion of
        the previous sentence. If absent, it is defaulted to true.
```

To train an English sentence detector use the following command:

```
$ opennlp SentenceDetectorTrainer -model en-custom-sent.bin -lang en -data en-custom-sent.train -encoding UTF-8
```

It should produce the following output:

```
Indexing events using cutoff of 5

 Computing event counts...  done. 4883 events
 Indexing...  done.
Sorting and merging events... done. Reduced 4883 events to 2945.
Done indexing.
Incorporating indexed data for training...
done.
 Number of Event Tokens: 2945
     Number of Outcomes: 2
   Number of Predicates: 467
...done.
Computing model parameters...
Performing 100 iterations.
  1:  .. loglikelihood=-3384.6376826743144 0.38951464263772273
  2:  .. loglikelihood=-2191.9266688597672 0.9397911120212984
  3:  .. loglikelihood=-1645.8640771555981 0.9643661683391358
  4:  .. loglikelihood=-1340.386303774519 0.9739913987302887
  5:  .. loglikelihood=-1148.4141548519624 0.9748105672742167

 ...<skipping a bunch of iterations>...

 95:  .. loglikelihood=-288.25556805874436 0.9834118369854598
 96:  .. loglikelihood=-287.2283680343481 0.9834118369854598
 97:  .. loglikelihood=-286.2174830344526 0.9834118369854598
 98:  .. loglikelihood=-285.222486981048 0.9834118369854598
 99:  .. loglikelihood=-284.24296917223916 0.9834118369854598
100:  .. loglikelihood=-283.2785335773966 0.9834118369854598
Wrote sentence detector model.
Path: en-custom-sent.bin
```

Training API

The Sentence Detector also offers an API to train a new sentence detection model. Basically three steps are necessary to train it:

- The application must open a sample data stream
- Call the SentenceDetectorME.train method
- Save the SentenceModel to a file or directly use it

The following sample code illustrates these steps:

```
ObjectStream<String> lineStream =
  new PlainTextByLineStream(new MarkableFileInputStreamFactory(new File("en-custom-sent.train")),
 StandardCharsets.UTF_8);

SentenceModel model;
try (ObjectStream<SentenceSample> sampleStream = new SentenceSampleStream(lineStream)) {
  model = SentenceDetectorME.train("eng", sampleStream,
      new SentenceDetectorFactory("eng", true, null, null), TrainingParameters.defaultParams());
}

try (OutputStream modelOut = new BufferedOutputStream(new FileOutputStream(modelFile))) {
  model.serialize(modelOut);
}
```

Evaluation

Evaluation Tool

The command shows how the evaluator tool can be run:

```
$ opennlp SentenceDetectorEvaluator -model en-custom-sent.bin -data en-custom-sent.eval -encoding UTF-8

Loading model ... done
Evaluating ... done

Precision: 0.9465737514518002
Recall: 0.9095982142857143
F-Measure: 0.9277177006260672
```

The en-custom-sent.eval file has the same format as the training data.

Chapter 5. Tokenizer

Tokenization

The OpenNLP Tokenizers segment an input character sequence into tokens. Tokens are usually words, punctuation, numbers, etc.

The statistical tokenizers in this chapter assume conventional whitespace-separated training and test data. When input contains Unicode spacing or dash variants (no-break space, ideographic space, en dash, and similar characters from PDFs or the web), use the Unicode-aware preprocessing described in Chapter 6, *Text Normalization*. The DL components apply that machinery automatically for document chunking; see the section called “Use in DL components”.

Tokenizers classify whitespace using the Unicode `White_Space` property by default; see the section called “Whitespace Definition” for the system property that restores OpenNLP's legacy definition for models trained with 1.x or 2.x.

```
Pierre Vinken, 61 years old, will join the board as a nonexecutive director Nov. 29.
Mr. Vinken is chairman of Elsevier N.V., the Dutch publishing group.
Rudolph Agnew, 55 years old and former chairman of Consolidated Gold Fields
 PLC, was named a director of this British industrial conglomerate.
```

The following result shows the individual tokens in a whitespace separated representation.

```
Pierre Vinken , 61 years old , will join the board as a nonexecutive director Nov. 29 .
Mr. Vinken is chairman of Elsevier N.V. , the Dutch publishing group .
Rudolph Agnew , 55 years old and former chairman of Consolidated Gold Fields PLC ,
 was named a nonexecutive director of this British industrial conglomerate .
A form of asbestos once used to make Kent cigarette filters has caused a high
 percentage of cancer deaths among a group of workers exposed to it more than 30 years ago ,
 researchers reported .
```

OpenNLP offers multiple tokenizer implementations:

- Whitespace Tokenizer - A whitespace tokenizer, non whitespace sequences are identified as tokens
- Simple Tokenizer - A character class tokenizer, sequences of the same character class are tokens
- Learnable Tokenizer - A maximum entropy tokenizer, detects token boundaries based on probability model

Most part-of-speech taggers, parsers and so on, work with text tokenized in this manner. It is important to ensure that your tokenizer produces tokens of the type expected by your later text processing components.

With OpenNLP (as with many systems), tokenization is a two-stage process: first, sentence boundaries are identified, then tokens within each sentence are identified.

Tokenizer Tools

The easiest way to try out the tokenizers are the command line tools. The tools are only intended for demonstration and testing.

There are two tools, one for the Simple Tokenizer and one for the learnable tokenizer. A command line tool the for the Whitespace Tokenizer does not exist, because the whitespace separated output would be identical to the input.

The following command shows how to use the Simple Tokenizer Tool.

```
$ opennlp SimpleTokenizer
```

To use the learnable tokenizer download the english token model from our website.

```
$ opennlp TokenizerME opennlp-en-ud-ewt-tokens-1.3-2.5.4.bin
```

To test the tokenizer copy the sample from above to the console. The whitespace separated tokens will be written back to the console.

Usually the input is read from a file and written to a file.

```
$ opennlp TokenizerME opennlp-en-ud-ewt-tokens-1.3-2.5.4.bin < article.txt > article-tokenized.txt
```

It can be done in the same way for the Simple Tokenizer.

Since most text comes truly raw and doesn't have sentence boundaries and such, it's possible to create a pipe which first performs sentence boundary detection and tokenization. The following sample illustrates that.

```
$ opennlp SentenceDetector sentdetect.model < article.txt | opennlp TokenizerME tokenize.model | more
Loading model ... Loading model ... done
done
Showa Shell gained 20 to 1,570 and Mitsubishi Oil rose 50 to 1,500.
Sumitomo Metal Mining fell five yen to 692 and Nippon Mining added 15 to 960 .
Among other winners Wednesday was Nippon Shokubai , which was up 80 at 2,410 .
Marubeni advanced 11 to 890 .
London share prices were bolstered largely by continued gains on Wall Street and technical
    factors affecting demand for London 's blue-chip stocks .
...etc...
```

Of course this is all on the command line. Many people use the models directly in their Java code by creating SentenceDetector and Tokenizer objects and calling their methods as appropriate. The following section will explain how the Tokenizers can be used directly from java.

Tokenizer API

The Tokenizers can be integrated into an application by the defined API. The shared instance of the WhitespaceTokenizer can be retrieved from a static field WhitespaceTokenizer.INSTANCE. The shared instance of the SimpleTokenizer can be retrieved in the same way from SimpleTokenizer.INSTANCE. To instantiate the TokenizerME (the learnable tokenizer) a Token Model must be created first. The following code sample shows how a model can be loaded.

```
try (InputStream modelIn = new FileInputStream("opennlp-en-ud-ewt-tokens-1.3-2.5.4.bin")) {
  TokenizerModel model = new TokenizerModel(modelIn);
}
```

After the model is loaded the TokenizerME can be instantiated.

```
Tokenizer tokenizer = new TokenizerME(model);
```

As of OpenNLP 3.0.0 TokenizerME is thread safe and a single instance can be shared across threads; the legacy ThreadSafeTokenizerME wrapper is retained for backward compatibility and is now deprecated. The tokenizer

offers two tokenize methods, both expect an input String object which contains the untokenized text. If possible it should be a sentence, but depending on the training of the learnable tokenizer this is not required. The first returns an array of Strings, where each String is one token.

```
String[] tokens = tokenizer.tokenize("An input sample sentence.");
```

The output will be an array with these tokens.

```
"An", "input", "sample", "sentence", "."
```

The second method, tokenizePos returns an array of Spans, each Span contain the start and end character offsets of the token in the input String.

```
Span[] tokenSpans = tokenizer.tokenizePos("An input sample sentence.");
```

The tokenSpans array now contain 5 elements. To get the text for one span call Span.getCoveredText which takes a span and the input text. The TokenizerME is able to output the probabilities for the detected tokens. The getTokenProbabilities method must be called directly after one of the tokenize methods was called.

```
TokenizerME tokenizer = ...
String[] tokens = tokenizer.tokenize(...);
double[] tokenProbs = tokenizer.getTokenProbabilities();
```

The tokenProbs array now contains one double value per token, the value is between 0 and 1, where 1 is the highest possible probability and 0 the lowest possible probability.

Explicit token character policies

`TokenizerCharacterPolicy` tests token eligibility from caller-supplied `CodePointSet` values for letters, digits, and marks. A mark may follow a letter, a digit, or another mark, but cannot start a token. The policy does not infer Unicode character categories, so callers choose the exact script coverage. The predicate checks a complete candidate without finding token boundaries or changing its text. `TokenizerME` applies the policy of its model for the alphanumeric optimization; the training section explains how a model selects one.

```
TokenizerCharacterPolicy policy = TokenizerCharacterPolicy.of(
    CodePointSet.ofRange('A', 'Z').union(CodePointSet.ofRange('a', 'z')),
    CodePointSet.ofRange('0', '9'),
    CodePointSet.of(0x0300, 0x0301));

boolean word = policy.test("Cafe\u0301");

// Migrating from a custom 2.x alphaNumericPattern: pass the policy to the factory
TokenizerFactory factory = new TokenizerFactory("en", null, true, policy);
```

Use `TokenizerCharacterPolicy.ascii()` for the fixed ASCII letter and digit preset. Policy sets must be pairwise disjoint and cannot contain surrogate code points or whitespace. The getters return the immutable sets. Eligibility is still a model-specific decision: accepting every Unicode letter would incorrectly bypass segmentation for some writing systems. See Unicode word boundaries [https://www.unicode.org/reports/tr29/#Word_Boundaries] for the distinction between character classes and language-specific segmentation.

Independent whitespace tokenizers

Use `WhitespaceTokenizer.newInstance(boolean keepNewLines)` when a caller needs its own whitespace tokenizer configuration. Each call returns a distinct instance. Changing it does not change `WhitespaceTokenizer.INSTANCE`. `TokenizerME` uses its own whitespace tokenizers and does not change the shared instance. The shared instance is mutable; callers that need different settings should use separate instances instead of reconfiguring it.

Tokenizer Training

Training Tool

OpenNLP has a command line tool which is used to train the models available from the model download page on various corpora. The data can be converted to the OpenNLP Tokenizer training format or used directly. The OpenNLP format contains one sentence per line. Tokens are either separated by a whitespace or by a special <SPLIT> tag. Tokens are split automatically on whitespace and at least one <SPLIT> tag must be present in the training text. The following sample shows the sample from above in the correct format.

```
Pierre Vinken<SPLIT>, 61 years old<SPLIT>, will join the board as a nonexecutive director Nov. 29<SPLIT>.
Mr. Vinken is chairman of Elsevier N.V.<SPLIT>, the Dutch publishing group<SPLIT>.
Rudolph Agnew<SPLIT>, 55 years old and former chairman of Consolidated Gold Fields PLC<SPLIT>,
   was named a nonexecutive director of this British industrial conglomerate<SPLIT>.
```

Usage of the tool:

```
$ opennlp TokenizerTrainer
Usage: opennlp TokenizerTrainer[.namefinder|.conllx|.pos] [-abbDict path] \
            [-alphaNumOpt isAlphaNumOpt] [-params paramsFile] [-iterations num] \
            [-cutoff num] -model modelFile -lang language -data sampleData \
            [-encoding charsetName]

Arguments description:
      -abbDict path
            abbreviation dictionary in XML format.
      -alphaNumOpt isAlphaNumOpt
            Optimization flag to skip alpha numeric tokens for further tokenization
      -params paramsFile
            training parameters file.
      -iterations num
            number of training iterations, ignored if -params is used.
      -cutoff num
            minimal number of times a feature must be seen, ignored if -params is used.
      -model modelFile
            output model file.
      -lang language
            language which is being processed.
      -data sampleData
            data to be used, usually a file name.
      -encoding charsetName
            encoding for reading and writing text, if absent the system default is used.
```

To train the english tokenizer use the following command:

```
$ opennlp TokenizerTrainer -model en-custom-token.bin -alphaNumOpt true -lang en -data en-custom-token.train -encoding
 UTF-8

Indexing events with TwoPass using cutoff of 5

 Computing event counts...  done. 45 events
 Indexing...  done.
Sorting and merging events... done. Reduced 45 events to 25.
Done indexing in 0,09 s.
Incorporating indexed data for training...
done.
 Number of Event Tokens: 25
     Number of Outcomes: 2
   Number of Predicates: 18
...done.
Computing model parameters ...
Performing 100 iterations.
  1:  ... loglikelihood=-31.191623125197527 0.8222222222222222
  2:  ... loglikelihood=-21.036561339080343 0.8666666666666667
  3:  ... loglikelihood=-16.397882721809086 0.9333333333333333
  4:  ... loglikelihood=-13.624159882595462 0.9333333333333333
  5:  ... loglikelihood=-11.762067054883842 0.9777777777777777

 ...<skipping a bunch of iterations>...

 95:  ... loglikelihood=-2.0234942537226366 1.0
 96:  ... loglikelihood=-2.0107265117555935 1.0
 97:  ... loglikelihood=-1.998139365828305 1.0
 98:  ... loglikelihood=-1.9857283791639697 1.0
 99:  ... loglikelihood=-1.9734892753591327 1.0
100:  ... loglikelihood=-1.9614179307958106 1.0
Writing tokenizer model ... done (0,044s)

Wrote tokenizer model to
Path: en-custom-token.bin
```

Training API

The Tokenizer offers an API to train a new tokenization model. Basically three steps are necessary to train it:

- The application must open a sample data stream
- Call the TokenizerME.train method
- Save the TokenizerModel to a file or directly use it

The following sample code illustrates these steps:

```
ObjectStream<String> lineStream = new PlainTextByLineStream(new MarkableFileInputStreamFactory(new File("en-
custom-sent.train")),
   StandardCharsets.UTF_8);

ObjectStream<TokenSample> sampleStream = new TokenSampleStream(lineStream);
TokenizerModel model;
try {
  model = TokenizerME.train(sampleStream,
   TokenizerFactory.create(null, "eng", null, true), TrainingParameters.defaultParams());
}
finally {
  sampleStream.close();
}

OutputStream modelOut = null;
try {
  modelOut = new BufferedOutputStream(new FileOutputStream(modelFile));
  model.serialize(modelOut);
} finally {
  if (modelOut != null)
    modelOut.close();
}
```

The last argument of `TokenizerFactory.create` is the `TokenizerCharacterPolicy` that the alphanumeric optimization uses to accept a candidate token without consulting the model. Leaving it out selects the policy for the language code: ASCII letters and digits for most languages, and ASCII plus the letters of the language for Catalan (ca), Dutch (nl), French (fr), German (de), Italian (it), Polish (pl), Portuguese (pt) and Spanish (es); the ISO 639-2 codes are accepted too. None of these presets has marks, so a decomposed character such as `e` followed by U+0301 is not accepted by them; supply a policy with the combining marks of your text to accept it. The policy is written to the model manifest under the property `characterPolicy` as comma separated hex code points and ranges, for example `letters=0041-005A,0061-007A;digits=0030-0039;marks=` for the ASCII preset, and read back from there when the model is loaded. `TokenizerCharacterPolicy.toString()` produces this form and `TokenizerCharacterPolicy.parse(String)` reads it.

Models written by earlier releases carry the property `alphaNumericPattern` with a regular expression instead. A model with one of the expressions OpenNLP itself wrote for the languages above loads unchanged and uses the policy with the same characters, so the released models need no conversion. A model trained with a custom pattern loads too and tests candidate tokens with that regular expression. This regular expression support, with the `Pattern` arguments of the `TokenizerFactory` constructor, `TokenizerFactory.create` and the `TokSpanEventStream` constructor and `TokenizerFactory.getAlphaNumericPattern()`, is **deprecated** and will be removed in 4.0. To migrate such a model, express the letters, digits and marks of its pattern as a `TokenizerCharacterPolicy`, then either train the model again with that policy, or replace `alphaNumericPattern` in the `manifest.properties` entry of the model file with `characterPolicy` set to the text form of the policy and zip the model again.

A model with no policy and no pattern in its manifest uses the policy for its language code. OpenNLP 2.x reads only `alphaNumericPattern`, so it applies its default pattern to a model written with a custom policy.

Detokenizing

Detokenizing is simple the opposite of tokenization, the original non-tokenized string should be constructed out of a token sequence. The OpenNLP implementation was created to undo the tokenization of training data for the tokenizer. It can also be used to undo the tokenization of such a trained tokenizer. The implementation is strictly rule based and defines how tokens should be attached to a sentence wise character sequence.

The rule dictionary assign to every token an operation which describes how it should be attached to one continuous character sequence.

The following rules can be assigned to a token:

- MERGE_TO_LEFT - Merges the token to the left side.
- MERGE_TO_RIGHT - Merges the token to the right side.
- RIGHT_LEFT_MATCHING - Merges the token to the right side on first occurrence and to the left side on second occurrence.

The following sample will illustrate how the detokenizer with a small rule dictionary (illustration format, not the xml data format):

```
. MERGE_TO_LEFT
" RIGHT_LEFT_MATCHING
```

The dictionary should be used to de-tokenize the following whitespace tokenized sentence:

```
He said " This is a test " .
```

The tokens would get these tags based on the dictionary:

```
He -> NO_OPERATION
said -> NO_OPERATION
" -> MERGE_TO_RIGHT
This -> NO_OPERATION
is -> NO_OPERATION
a -> NO_OPERATION
test -> NO_OPERATION
" -> MERGE_TO_LEFT
. -> MERGE_TO_LEFT
```

That will result in the following character sequence:

```
He said "This is a test".
```

Detokenizing API

The Detokenizer can be used to detokenize the tokens to String. To instantiate the Detokenizer (a rule based detokenizer) a DetokenizationDictionary (the rule of dictionary) must be created first. The following code sample shows how a rule dictionary can be loaded.

```
InputStream dictIn = new FileInputStream("latin-detokenizer.xml");
DetokenizationDictionary dict = new DetokenizationDictionary(dictIn);
```

After the rule dictionary is loaded the DictionaryDetokenizer can be instantiated.

```
Detokenizer detokenizer = new DictionaryDetokenizer(dict);
```

The detokenizer offers two detokenize methods, the first detokenize the input tokens into a String.

```
String[] tokens = new String[]{"A", "co", "-", "worker", "helped", "."};
String sentence = detokenizer.detokenize(tokens, null);
Assert.assertEquals("A co-worker helped.", sentence);
```

Tokens which are connected without a space in-between can be separated by a split marker.

```
String sentence = detokenizer.detokenize(tokens, "<SPLIT>");
Assert.assertEquals("A co<SPLIT>-<SPLIT>worker helped<SPLIT>.", sentence);
```

The API also offers a method which simply returns operations array in the input tokens array.

```
DetokenizationOperation[] operations = detokenizer.detokenize(tokens);
for (DetokenizationOperation operation : operations) {
  System.out.println(operation);
}
```

Output:

```
NO_OPERATION
NO_OPERATION
MERGE_BOTH
NO_OPERATION
NO_OPERATION
MERGE_TO_LEFT
```

Detokenizer Dictionary

Detokenization Dictionary is the rule dictionary about detokenizer. tokens - an array of tokens that should be detokenized according to an operation. operations - an array of operations which specifies which operation should be used for the provided tokens. The following code sample shows how a rule dictionary can be created.

```
String[] tokens = new String[]{".", "!", "(", ")", "\"", "-"};
Operation[] operations = new Operation[]{
    Operation.MOVE_LEFT,
    Operation.MOVE_LEFT,
    Operation.MOVE_RIGHT,
    Operation.MOVE_LEFT,
    Operation.RIGHT_LEFT_MATCHING,
    Operation.MOVE_BOTH};
DetokenizationDictionary dict = new DetokenizationDictionary(tokens, operations);
```

Unicode Word Segmentation (UAX #29)

The package `opennlp.tools.tokenize.uax29` provides a tokenizer that follows the Unicode Text Segmentation algorithm (UAX #29 [https://www.unicode.org/reports/tr29/]), word boundary rules WB1 through WB999. It is rule-based and needs no trained model. It works directly over a `CharSequence` and reports character offsets so the original text is preserved for downstream processing such as the normalization described in

Chapter 6, *Text Normalization*. The boundary data comes from the bundled Unicode Character Database (currently Unicode 18.0) and the implementation passes the official `WordBreakTest.txt` conformance suite for that release.

Word Segmenter

`WordSegmenter` finds the word boundaries. It is a single forward cursor pass with constant-time property lookups and no regular expression. Every segment is reported, including whitespace and punctuation runs, so the segments are contiguous and together cover the whole text.

```
int[] boundaries = WordSegmenter.boundaries("The quick brown fox.");
List<Span> segments = WordSegmenter.segments("The quick brown fox.");
```

To process the segments without building a result list, stream them to a callback.

```
WordSegmenter.forEachSegment("The quick brown fox.", (start, end) -> {
  // handle the segment [start, end)
});
```

Word Tokenizer

`WordTokenizer` builds on the segmenter. It keeps the segments that are words (letters, digits, ideographs, kana, Hangul, a Southeast Asian script, or emoji), drops whitespace and punctuation, and classifies each token. Here emoji means any Extended_Pictographic code point, so symbol-like characters such as the copyright, trademark, and double-exclamation signs are kept rather than dropped. It implements the standard `Tokenizer` interface, so it can be used wherever a tokenizer is expected.

```
Tokenizer tokenizer = new WordTokenizer();
String[] tokens = tokenizer.tokenize("The quick brown fox.");
Span[] spans = tokenizer.tokenizePos("The quick brown fox.");
```

The tokens array contains "The", "quick", "brown", and "fox"; the trailing period and the spaces are dropped. The `tokenizeTyped` method additionally returns the category of each token as a `WordType`.

```
WordTokenizer wordTokenizer = new WordTokenizer();
for (WordToken token : wordTokenizer.tokenizeTyped("OpenNLP 3.0")) {
  System.out.println(token.text("OpenNLP 3.0") + " : " + token.type());
}
// OpenNLP : ALPHANUMERIC
// 3.0     : NUMERIC
```

The categories are `ALPHANUMERIC`, `NUMERIC`, `IDEOGRAPHIC`, `HIRAGANA`, `KATAKANA`, `HANGUL`, `SOUTHEAST_ASIAN`, and `EMOJI`.

A streaming overload reports each token to a handler without building a result list.

```
WordTokenizer wordTokenizer = new WordTokenizer();
wordTokenizer.tokenize("The quick brown fox.", (start, end, type) -> {
  // handle the token [start, end) of the given WordType
});
```

A token longer than the maximum token length is emitted as consecutive pieces without splitting a surrogate pair. The maximum defaults to `WordTokenizer.DEFAULT_MAX_TOKEN_LENGTH` and can be set through the constructor.

```
WordTokenizer wordTokenizer = new WordTokenizer(64);
```

Lattice tokenization for CJK

Languages written without spaces need a dictionary-backed segmenter. `LatticeTokenizer` scores paths over a MeCab-format dictionary and emits the cheapest segmentation with spans in original text coordinates. `UnigramSegmenter` does the same from a plain frequency lexicon. Install a dictionary archive with `MecabDictionaryInstaller`, load it as a `MecabDictionary`, and tokenize. Remote archives require an expected SHA-256 or SHA-512 digest; an application-supplied opt-in catalog (`installFromCatalog`) also needs `-Dopennlp.download.remote=true`. Fetching, verification, and unpacking use `ResourceInstaller`, described in the model loading chapter. Default limits are a 1 GiB download, 4 GiB unpacked, and 100000 archive entries. Set higher limits at JVM startup via `opennlp.download.max.bytes`, `opennlp.install.max.total.bytes`, `opennlp.install.max.entries`, and `opennlp.install.max.expansion.ratio` for larger dictionaries such as UniDic. Installed files are flattened to their base names. Existing target files are not replaced and must be removed before a refresh.

```
MecabDictionaryInstaller.install(archiveUri, dictionaryDirectory, sha512);
LatticeTokenizer tokenizer =
    new LatticeTokenizer(MecabDictionary.load(dictionaryDirectory));

String text = "#####";
String[] tokens = tokenizer.tokenize(text);
Span[] spans = tokenizer.tokenizePos(text);
List<Morpheme> morphemes = tokenizer.analyze(text);

UnigramSegmenter segmenter = UnigramSegmenter.load(Path.of("chinese-words.txt"));
String[] words = segmenter.tokenize("#########");
```

`matrix.def` must list a cost for every declared cell; matrix dimensions plus lexicon size are bounded by the shared `ResourceLimits.MAX_ENTRIES` limit and the matrix cell count by `ResourceLimits.MAX_MATRIX_CELLS`. Dictionary loading rejects empty lexicon surfaces, duplicate `matrix.def` entries, duplicate `char.def` categories, costs outside the signed 16-bit range, malformed text in the selected encoding, and `unk.def` templates for undefined categories. MeCab-quoted CSV fields are accepted, with three deviations from MeCab's reader: quotes open and close a quoted stretch wherever they appear in a field, text after a closing quote joins the field instead of being dropped, and an unterminated quoted field is rejected with its file and line instead of being read silently to the end of the line.

A `char.def` may define up to 17 categories, and category lengths range from 0 through 15, matching the MeCab character-property compiler. If a `char.def` mapping lists multiple categories, the first provides unknown-word settings. Grouping continues while successive assignments overlap. Unknown words use their category's `unk.def` template first, with `DEFAULT` used when no category template exists.

Subword Tokenization

Neural models commonly use subword units from a fixed vocabulary. The `SubwordTokenizer` interface encodes text as `SubwordPiece` values. Each value contains the normalized piece string, vocabulary id, and UTF-16 span in the original text. The span remains valid when model normalization changes the piece text. `encodeToIds` and `encodeToPieces` return only the ids or strings. An implementation can include model control pieces with

empty spans, such as the classification and separator pieces used by BERT. Thread safety is implementation specific.

The interface, `SubwordPiece`, and vocabulary-based `WordpieceEncoder` are in `opennlp-api`. Implementations that read trained model formats can implement the same contract.

A piece without source text has an empty span. When normalization reorders characters, the range can include adjacent source characters.

WordPiece

`WordpieceEncoder` in `opennlp.tools.tokenize` runs the full BERT tokenization pipeline: control character removal, whitespace normalization, CJK isolation, optional lower casing with accent stripping, punctuation isolation, and greedy longest-match wordpiece segmentation. It is constructed from the model's vocabulary, conventionally the lines of a `vocab.txt` file, where the line number is the id.

```
List<String> vocabulary = Files.readAllLines(Path.of("vocab.txt"));

WordpieceEncoder encoder = new WordpieceEncoder(vocabulary);        // uncased model
WordpieceEncoder cased = new WordpieceEncoder(vocabulary, false);   // cased model

String text = "Alice was beginning to get very tired.";
List<SubwordPiece> pieces = encoder.encode(text);
for (SubwordPiece piece : pieces) {
  CharSequence source = text.subSequence(piece.start(), piece.end());
  System.out.println(piece.id() + "\t" + piece.piece() + "\t" + source);
}
int[] ids = encoder.encodeToIds(text);
```

The classification and separator pieces appear at the beginning and end with empty spans. Words the vocabulary cannot cover, and words exceeding 100 normalized Unicode code points, become the unknown piece. A constructor parameter can set a different maximum. Models with other special tokens or non-contiguous ids are supported through the constructors taking explicit special tokens or a piece-to-id map.

Chapter 6. Text Normalization

Introduction

The package `opennlp.tools.util.normalizer` provides Unicode-aware text normalization for matching, search, and tokenization preprocessing. It cleans up the kinds of inconsistency that real text carries when it is copied from the web, PDFs, office documents, or multilingual sources: spacing that is not an ordinary space, the many dash and quotation variants, decomposed versus precomposed accents, non-ASCII digits, and invisible control characters.

The implementation follows three principles:

- **Standards-sourced.** Membership sets come from the Unicode Character Database (for example the `White_Space` and `Dash` properties in `PropList.txt` [https://www.unicode.org/Public/18.0.0/ucd/PropList.txt], Unicode 18.0.0), not from the JVM's locale-dependent or quirky character predicates. The normalization engine never relies on `Character.isWhitespace`, which disagrees with the Unicode standard.

- **Cursor-based, no regular expressions.** Every operation is a single forward pass over the input that tests membership in O(1) and advances by code point. This avoids the allocation and the catastrophic backtracking (ReDoS) risk of regular expressions, and it correctly recognizes Unicode characters that Java's `\s` does not.

- **Offset-preserving.** The original text is always the source of truth. Normalization produces a derived form for matching while the original character offsets are kept, so a search hit can be reported and highlighted against the source even when the normalized form has a different length.

Two engines underpin everything: the `CharSequenceNormalizer` family offers ready-made, composable normalizers, and the `CharClass` engine is the low-level, configurable building block they are made of. Built on these are four higher-level features documented below: a layered term model that projects a token through a configurable stack of transforms while keeping every intermediate form (see the section called “The layered term model”), an opt-in emoji annotation layer that turns pictographs into typed signal (see the section called “Emoji annotations”), per-language profiles that select the transforms appropriate to a language (see the section called “Per-language profiles”), and confusable folding that reduces lookalike characters for matching (see the section called “Confusable (homoglyph) folding”).

The normalizer family

Each normalizer implements the existing `opennlp.tools.util.normalizer.CharSequenceNormalizer` interface (`CharSequence normalize(CharSequence)`) and is, by default, a shared stateless instance obtained through `getInstance()` (case and accent folding also offer configured forms). They can therefore be combined with the existing `AggregateCharSequenceNormalizer`, or with the `TextNormalizer` builder described below.

Normalizer	Effect
`WhitespaceCharSequenceNormalizer`	Collapses each run of Unicode whitespace to a single ASCII space and trims the edges.
`LineBreakPreservingWhitespaceCharSequenceNormalizer`	Collapses horizontal whitespace runs to a single ASCII space but keeps a run containing a line break as a single newline, so paragraph structure survives; trims the edges.
`ParagraphPreservingWhitespaceCharSequenceNormalizer`	Unwraps hard-wrapped prose (Gutenberg-style fixed-width lines): a whitespace run with at most one line

Normalizer	Effect
	break collapses to a space, two or more collapse to a single newline, and `CRLF` counts as one break; trims the edges. Use before sentence detection on plain text, not on raw markdown with lists or code blocks.
`DashCharSequenceNormalizer`	Maps every Unicode dash to the ASCII hyphen-minus. The mathematical minus signs and the soft hyphen are not affected.
`QuoteCharSequenceNormalizer`	Folds typographic single quotes and apostrophes to `'` and double quotes (including guillemets) to `"`.
`DigitCharSequenceNormalizer`	Maps Unicode decimal digits (Arabic-Indic, Devanagari, fullwidth, ...) to ASCII `0-9` by their numeric value.
`EllipsisCharSequenceNormalizer`	Expands the horizontal ellipsis to `...` and the two-dot leader to `..`
`BulletCharSequenceNormalizer`	Replaces unambiguous list bullets with a space; the Catalan middle dot is left alone.
`InvisibleCharSequenceNormalizer`	Removes invisible format and bidirectional control characters (BOM, zero width space, bidi marks/overrides/isolates, ...). The zero width joiner and non-joiner and variation selectors are kept.
`NfcCharSequenceNormalizer`	Applies Unicode Normalization Form C (canonical composition); a safe, lossless baseline for matching.
`NfkcCharSequenceNormalizer`	Applies Unicode Normalization Form KC (compatibility composition); folds fullwidth forms, ligatures, and super/subscripts.
`CaseFoldCharSequenceNormalizer`	Lower cases for case-insensitive matching, using `Locale.ROOT` or a supplied locale. Does not apply full case folding; U+00DF is unchanged.
`FullCaseFoldCharSequenceNormalizer`	Applies Unicode full case folding (`CaseFolding.txt`, status `C` and `F`), including the expanding folds such as U+00DF to `ss` and U+FB00 to `ff`. The Turkic mappings are not applied. Expects NFC input.
`AccentFoldCharSequenceNormalizer`	Folds diacritics in a script-aware way (see below).
`GermanUmlautCharSequenceNormalizer`	Transliterates German umlauts and the eszett (a-umlaut to `ae`, eszett to `ss`; DIN 5007-2).
`ConfusableSkeletonCharSequenceNormalizer`	Reduces lookalike characters to a confusable skeleton for matching (UTS #39); see below.
`EmojiToEmoticonCharSequenceNormalizer`	Folds emoji to ASCII emoticons from a bundled project-authored mapping (U+1F642 to `:)`); an emoji presentation sequence (a pictograph followed by U+FE0F) folds as one unit. Unmapped content passes through.
`EmoticonToEmojiCharSequenceNormalizer`	Folds an ASCII emoticon that stands alone between whitespace to its pictograph (`:-)` to U+1F642). An embedded sequence, such as the `:/` in `https://`, is left alone. Applied before tokenization, it lets emoticons survive as single emoji tokens.

A single normalizer is applied directly:

```
CharSequenceNormalizer ws = WhitespaceCharSequenceNormalizer.getInstance();
String clean = ws.normalize("a   b").toString();   // "a b"

String hyphen = DashCharSequenceNormalizer.getInstance()
   .normalize("state—of–the#art").toString(); // "state-of-the-art"
```

Composing a pipeline

`TextNormalizer` is a fluent builder that composes the normalizers, in the order they are added, into a single `CharSequenceNormalizer`:

```
CharSequenceNormalizer pipeline = TextNormalizer.builder()
   .nfc()
   .caseFold()
   .accentFold()
   .build();

String term = pipeline.normalize("CAFÉ").toString();   // "cafe"
```

A conservative search-oriented chain (strip invisibles, NFC, collapse whitespace, fold quotes and dashes, case fold, then script-gated accent fold) is available directly:

```
CharSequenceNormalizer chain = TextNormalizer.defaultChain();

// byte order mark stripped, curly quotes folded, case and accent folded
String t = chain.normalize("“CafÉ”").toString();   // "\"cafe\""
```

Any custom `CharSequenceNormalizer` can be inserted with `with(...)`. The `TextNormalizer` pipeline and the individual `CharSequenceNormalizer` implementations are not applied automatically by statistical OpenNLP components; callers compose them explicitly when preprocessing text for search or matching; to recover original character offsets from the composed result, see the section called “Offset-aware pipelines”. The DL components described in the section called “Use in DL components” use a narrower, built-in subset of this machinery.

Offset-aware pipelines

`build()` returns a normalizer that yields only the cleaned text. When you also need to map a match found in the normalized text back to the original (to highlight a search hit in the source, say), build the pipeline with `buildAligned()` instead. It returns an `OffsetAwareNormalizer` whose `normalizeAligned(text)` returns an `AlignedText`: the normalized string together with an `Alignment` that maps spans between the two forms, composed across every stage, so a match maps straight back to the source even when a stage collapsed a run or folded a multi-unit character.

```
OffsetAwareNormalizer pipeline = TextNormalizer.builder()
   .whitespace()      // collapses runs, so it changes length and shifts offsets
   .dashes()
   .buildAligned();

String original = "find   the—match";
AlignedText aligned = pipeline.normalizeAligned(original);
// aligned.normalized()              ->  "find the-match"
Span hit = aligned.toOriginalSpan(5, 14);   // "the-match" in the normalized text
// hit.getCoveredText(original)        ->  "the—match"   (back to the source, em dash and all)
```

`OffsetAwareNormalizer` is a capability interface: it extends `CharSequenceNormalizer` and adds `normalizeAligned`, so a caller tests for it with a plain `instanceof`, the same pattern the name finder uses for `OffsetMappingNameFinder`. Every per-code-point fold implements it: whitespace, the line-break-preserving whitespace normalizer, dashes, invisible-control stripping, quotes, digits, ellipsis, bullets, the German umlaut transliteration, Unicode full case folding (`fullCaseFold()`, whose expansions come from a bundled table with known lengths, so it reports its edits), and the emoji/emoticon folds (`emojiToEmoticon()` and `emoticonToEmoji()`, backed by the same kind of known-length bundled table). The folds that route through `java.text.Normalizer` (NFC, NFKC, accent folding, and confusable folding) or through JDK case mapping (the locale-based `caseFold()`) cannot report their per-character edits, so they do not implement the interface, and a chain that contains one is rejected by `buildAligned()` with an `IllegalStateException` that names the step, rather than returning a pipeline that would hand back an offset that does not hold. The per-stage maps and their composition use the `Alignment` machinery described in the section called “The CharClass engine and code point sets”.

Use in DL components

`NameFinderDL` and `DocumentCategorizerDL` share Unicode-aware text handling through `AbstractDL`. Long inputs are split into overlapping chunks on the full Unicode `White_Space` set (no-break space, ideographic space, line and paragraph separators, and the other members listed under the section called “Reference data”), not on Java's six-character `\s` subset. Empty tokens from leading, trailing, or repeated whitespace are not produced.

`NameFinderDL` additionally locates reconstructed entity text in the original input with a cursor-based matcher: a space in the reconstructed span matches zero or more Unicode whitespace code points in the source, and every other code point is compared case-insensitively. This handles spacing copied from PDFs, the web, or non-Latin sources when resolving `Span#getCoveredText(...)`.

Optional input folding is controlled through `InferenceOptions` and is *off by default* so existing models keep their prior inputs unless you opt in:

- `setNormalizeWhitespace(true)` maps each Unicode whitespace code point to a single ASCII space before inference. The transform is one code point to one space, so character offsets stay aligned with the input.
- `setNormalizeDashes(true)` maps each dash in the default `CharClass.dashes()` set to the ASCII hyphen-minus. Mathematical minus signs and the soft hyphen are not affected unless you extend the set explicitly. This replacement is also one code point to one character for Basic Multilingual Plane dashes. A dash in the supplementary planes shrinks from two UTF-16 units to one, which shifts later offsets, so with this fold enabled `find` reports offsets into the normalized text in that case while `findInOriginal` maps them back to the original input.

Run-collapsing normalization (for example `WhitespaceCharSequenceNormalizer`, which collapses whitespace *runs* to a single space) is not enabled through these flags because it would shift character offsets. Use the `CharSequenceNormalizer` pipeline directly when you need that behavior on text that does not require offset-preserving span lookup. See also the section called “Using an ONNX Model” and the section called “Using an ONNX Model”.

```
InferenceOptions options = new InferenceOptions();
options.setNormalizeWhitespace(true);   // opt-in: NBSP, ideographic space, ... -> ASCII space
options.setNormalizeDashes(true);       // opt-in: en dash, em dash, ... -> hyphen-minus

NameFinderDL finder = new NameFinderDL(model, vocab, ids2Labels, options, sentenceDetector);
```

Diacritic folding and multilingual safety

`AccentFoldCharSequenceNormalizer` folds accents for search, but does so in a script-aware way that a Latin-only folding filter cannot. It decomposes the text, then drops nonspacing combining marks only for base characters whose script is configured for folding (Latin, Greek, and Cyrillic by default). Combining marks on other scripts are left untouched, because there they are essential orthography rather than decoration: dropping an Indic vowel sign or virama, an Arabic harakat, a Hebrew point, or a Thai vowel would change the word.

```
CharSequenceNormalizer fold = AccentFoldCharSequenceNormalizer.getInstance();

fold.normalize("café");              // "cafe"   (Latin accent folded)
fold.normalize("#");                 // "#" (Greek alpha-with-tonos -> alpha)
fold.normalize("##");            // unchanged (Devanagari is left intact)
```

Atomic Latin letters that do not decompose are mapped to an ASCII approximation by default: for example the stroke letters and ligatures, eszett, and thorn (`ø -> o`, `æ -> ae`, `ß -> ss`, `þ -> th`). Both behaviors are configurable through the constructor:

```
// fold only Latin, and do not map the stroke letters
CharSequenceNormalizer latinOnly = new AccentFoldCharSequenceNormalizer(
    java.util.Set.of(Character.UnicodeScript.LATIN), false);
```

Diacritic folding is a recall optimization, not a linguistically correct transform, so it is intended for a search or matching form rather than for display. Language-specific case and letter rules (for example German `DIN` umlaut expansion, or the Turkish dotless-i) are out of scope for the default folder and should be applied with an explicit locale upstream.

The CharClass engine and code point sets

The set-based normalizers are built on `CharClass`, a configurable class of Unicode code points paired with a single canonical replacement, backed by a `CodePointSet` with O(1) membership. You choose both the membership and the replacement code point with `CharClass.of(members, replacement)`; whitespace and dashes are the two built-in presets, and any other class is one more configured instance:

```
CharClass ws = CharClass.whitespace();   // Unicode White_Space -> U+0020
CharClass dash = CharClass.dashes();     // Unicode Dash (curated) -> U+002D

ws.collapse("a   b");                  // "a b"         (runs -> one space)
ws.trim("  hi  ");                   // "hi"
String[] tokens = ws.split("one two"); // ["one", "two"]  (offset-aware via splitSpans)
dash.normalize("a—b");               // "a-b"
```

A class applies its replacement in four ways, which differ in whether they collapse runs and whether they preserve character offsets:

- `normalize(text)` replaces each member one-for-one with the replacement, so it is length- and offset-preserving; use it when you still need spans back into the original text.
- `collapse(text)` reduces each maximal run of members to a single replacement; it changes length, so it is a search and match transform.
- `collapsePreserving(text, keep, keepReplacement)` collapses runs but emits `keepReplacement` for any run containing a kept code point, which is how you squish horizontal whitespace while keeping line breaks.
- `collapseParagraphPreserving(text, lineBreaks, paragraphReplacement)` collapses runs but emits `paragraphReplacement` only when a run contains two or more logical line breaks (a single break, including `CRLF`, becomes the usual replacement). This is the hard-wrap unwrap used by `ParagraphPreservingWhitespaceCharSequenceNormalizer`.

So the replacement is your choice and the method picks the behavior. Folding tabs and newlines to a single newline, for example, is one configured class:

```
CharClass lineFold = CharClass.of(CodePointSet.of('\n', '\t'), '\n');
lineFold.collapse("\n\n\n\t\n");   // "\n"  (the whole run folds to one newline)

CharClass ws = CharClass.whitespace();
ws.collapsePreserving(text, CodePointSet.of('\n'), '\n');   // squish spaces, keep paragraph breaks
```

When you need the normalized form together with a map back to the original, the `normalizeAligned`, `collapseAligned`, `collapsePreservingAligned`, `collapseParagraphPreservingAligned`, `trimAligned`, and `removeAllAligned` variants return an `AlignedText` that carries an `Alignment`. The alignment maps spans between the two forms with `toOriginalSpan` and `toNormalizedSpan`, staying correct across deletions and length-changing folds, and composes with `andThen`.

A `CodePointSet` can be built explicitly, as a range, by union, or loaded from a user definitions file so that delimiters can be extended without a code change. The file is line oriented and parsed with the same cursor approach (no regular expression): a `[name]` line opens a section, a # begins a comment, and each remaining line is a hex code point or an inclusive range.

```
[whitespace]
U+00A0          # no-break space
U+2000-U+200A   # typographic spaces

[dash]
U+2E5D          # oblique hyphen
```

```
CodePointSet extra = CodePointSet.fromFile(path, "whitespace");
CharClass wsPlus = CharClass.whitespace().withAdditional(extra);
```

The layered term model

`TermAnalyzer` tokenizes text and gives each token a *stack* of normalization layers while keeping its source span. It is the offset-preserving entry point for matching: the normalized form is what you index or query, and the span ties every layer back to the original text for highlighting, even when normalization changes a token's length. A `Term` is one token projected through an ordered chain of `Dimensions`: original, NFC, NFKC, whitespace, dash, case fold, full case fold, accent fold, emoji fold, confusable fold, stem, and lemma. The order is fixed because

the transforms do not commute (case folding then accent folding differs from the reverse). The original is always kept, so aggressive folding stays safe and a match on any layer maps back to the source through the token's `Span`.

```
TermAnalyzer analyzer = TermAnalyzer.builder()
    .caseFold()
    .stem(new PorterStemmer())
    .build();

Term term = analyzer.analyze("Running").get(0);
// term.original()          -> "Running"
// term.normalized()        -> "run"     (the final configured dimension, here STEM)
// term.peel()              -> "running"  (the layer below the top, O(1))
// term.at(Dimension.NFC)   -> computed lazily on first request, then cached
```

Segmentation uses the the section called “Unicode Word Segmentation (UAX #29)” word tokenizer, so the input does not need to be pre-tokenized. The dimensions named in the builder are computed eagerly; any other dimension is computed on first request, applied on top of the final form, and cached, so querying a configured layer or peeling the last one is O(1) and adding an unrequested dimension costs one transform. The character-level dimensions have built-in defaults; `STEM` and `LEMMA` require a `Stemmer` or `Lemmatizer` (and `LEMMA` a part-of-speech tag); requesting either without them throws an `IllegalStateException`. An analyzer is immutable and safe to share across threads when its configured transforms are. The built-in character normalizers are stateless and `SnowballStemmer` is thread-safe, so they can be shared as is. A `Stemmer` passed to `stem(Stemmer)` is used directly, so a stateful one such as `PorterStemmer` confines the analyzer to one thread. Passing a `StemmerFactory` to `stem(StemmerFactory)` instead wraps it in a `CachingStemmer` (see the section called “Sharing and caching”), which gives each thread its own stemmer and cache, so the analyzer is shareable even when the factory creates stateful stemmers.

Each dimension's transform is configurable on the builder. Beyond the no-argument methods that enable a dimension with its default, there are convenience methods for common options, and the general method `transform(dimension, normalizer)` for any character-level dimension:

```
TermAnalyzer analyzer = TermAnalyzer.builder()
    .whitespace(CharClass.of(CodePointSet.of('\n', '\t'), '\n')::collapse) // custom target/behavior
    .caseFold(Locale.forLanguageTag("tr"))                  // Turkish case rules
    .accentFold(Set.of(Character.UnicodeScript.LATIN), false)   // fold only Latin
    .maxTokenLength(255)                                    // tokenizer chopping
    .build();
```

The whitespace and dash methods take any `CharSequenceNormalizer`, so a `CharClass` method reference (`::normalize` for one-for-one, `::collapse` for run-collapsing) selects both the fold target and the behavior. The case-fold method takes a `Locale` for language-specific rules such as the Turkish dotted/dotless i, and the accent-fold method takes the scripts to fold and whether to fold stroke letters.

Emoji annotations

`EmojiAnnotator` provides optional metadata for a pictograph: a name, a coarse sentiment, an entity type, a document category and, for flags, a region. It assembles one `EmojiAnnotation` record per symbol from three separate sources: the bundled, project-authored `emoji-annotations.txt` table; values computed from the code point sequence itself (`EmojiFlags` decodes a regional indicator pair to its ISO 3166-1 alpha-2 code and a subdivision tag sequence to its ISO 3166-2 code, with no data file); and optional values resolved at run time through an `EmojiAnnotationJoin` hook against user-installed data such as a gazetteer. Every attribute value records its source. The bundled table contains no gazetteer identifiers. `EmojiEntityType` includes `HAND`, `SKY_AND_WEATHER` and `TOOL` for Unicode Emoji 18.0 subgroups; U+1F6D9 lighthouse is tagged `LANDMARK` as a documented exception to the upstream `transport-water` subgroup.

```
EmojiAnnotator annotator = new EmojiAnnotator();
TermAnalyzer analyzer = TermAnalyzer.builder().emojiFold().build();

// "Berlin <German flag> <red heart>"
String text = "Berlin \uD83C\uDDE9\uD83C\uDDEA \u2764\uFE0F";
for (Term term : analyzer.analyze(text)) {
  annotator.annotate(term).ifPresent(a -> {
    a.name();       // "red heart" for the heart (CLDR short name)
    a.sentiment();  // 2 for the heart (project-authored coarse score, -2..2)
    a.entityType(); // FLAG for the flag, HEART for the heart
    a.category();   // FLAGS / SMILEYS_AND_EMOTION
    a.isoRegion();  // "DE" for the flag, decoded from the sequence itself
  });
}
```

Annotations are per-symbol *metadata*, not text transforms, and are therefore not `Dimension`s: `annotate(Term)` reads the token's original layer, the symbol as written, even after a fold dimension has rewritten it. It accepts any token: a non-emoji token yields an empty result, and malformed flag-shaped text yields no region. The strict decoder `EmojiFlags.isoRegion()` throws an `IllegalArgumentException` on malformed flag sequences. The bundled table's sentiment scores are original project judgments (tagged `UNSPECIFIED`); its names and its entity-type/category assignments derive from the Unicode-licensed CLDR short names and `emoji-test.txt` group headers, and no third-party sentiment data set is included.

The annotations are optional feature inputs to three components, each added through an existing extension point, so default behavior and existing models are unchanged: the name finder (and any consumer of feature generation descriptors) activates `EmojiAnnotationFeatureGenerator` by naming `EmojiAnnotationFeatureGeneratorFactory` in a descriptor, the document categorizer takes `EmojiFeatureGenerator` alongside its defaults through the `DoccatFactory` feature generator array, and sentiment analysis trains with `EmojiSentimentFactory`, whose context generator adds the emoji evidence and is re-created from the model manifest at prediction time. A flag token then yields the features `emojiType=FLAG` and `emojiRegion=DE` without a dictionary, and a heart or crying face contributes its coarse polarity directly.

Confusable (homoglyph) folding

`Confusables` reduces text to its Unicode confusable *skeleton* following UTS #39 [https://www.unicode.org/reports/tr39/]: it decomposes the text, replaces each code point with its prototype, and decomposes again. Two strings are confusable exactly when their skeletons are equal, which catches spoofing where Cyrillic or Greek letters imitate Latin ones.

```
// "paypal" with Cyrillic a's looks identical but is a different code-point sequence
Confusables.confusable("paypal", spoofed);   // true
Confusables.skeleton("paypal");              // a matching key, not readable text
```

The skeleton changes length and offsets, so like accent folding it is a derived, matching-only form. It is also available as `ConfusableSkeletonCharSequenceNormalizer` and as the `CONFUSABLE_FOLD` term dimension. The mapping comes from the bundled Unicode security data file `confusables.txt`.

Per-language profiles

`NormalizationProfiles` selects per-language settings the same way OpenNLP already selects a Snowball stemmer by language: ask for a language, or detect it with a `LanguageDetector` when it is unspecified. Each `NormalizationProfile` pairs a language with its Snowball stemmer and the diacritic fold appropriate for that language, and builds a search-oriented `TermAnalyzer`.

```
NormalizationProfile german = NormalizationProfiles.forLanguage("de").orElseThrow();
TermAnalyzer analyzer = german.matchingAnalyzer();   // NFC, case fold, German fold, German stemmer
// "Mueller" and "Müller" both reduce to the same matching term

// detect the language when it is not known
NormalizationProfiles.detect(text, languageDetector).map(NormalizationProfile::matchingAnalyzer);
```

The diacritic fold is the generic accent fold for English and the major Romance languages, the German-specific fold (a-umlaut to `ae`, eszett to `ss`, following DIN 5007-2) for German, and none for the Nordic languages and non-Latin scripts, where folding distinct letters is language-wrong. As stated in the section called “Diacritic folding and multilingual safety”, this is a search-recall choice, not linguistic correctness; a caller that wants different behavior builds a `TermAnalyzer` directly.

Reference data

The underlying Unicode data is also available directly as immutable reference tables, with O(1) membership tests that match the Unicode standard:

- `UnicodeWhitespace` lists the 25 characters carrying the `White_Space` property, plus the related look-alike format characters (zero width space, byte order mark, ...) that are *not* whitespace. It exposes `isWhitespace(int)`, `byCodePoint(int)`, and helpers for the line breaks and the non-breaking spaces.
- `UnicodeDash` lists every code point carrying the `Dash` property, distinguishing the mathematical minus signs that are excluded from the default normalization set.

Chapter 7. Stopword Filtering

Overview

Stopwords are very common words (such as articles, prepositions, pronouns or conjunctions) that carry little semantic content of their own and are frequently removed prior to downstream processing such as indexing, classification, topic modeling, keyword extraction or similarity computation. Filtering stopwords can reduce noise, lower memory footprint and improve the signal-to-noise ratio of subsequent NLP components.

OpenNLP provides a small, dependency-free stopword filtering API in the `opennlp.tools.stopword` package. The central abstraction is the `StopwordFilter` interface, with a `Dictionary`-backed default implementation `DictionaryStopwordFilter`. Bundled stopword lists are available for eleven languages and can be loaded by ISO 639-1 code via the `StopwordLists` factory. Users may also load custom lists from any `InputStream`, mix them with the bundled defaults and add or remove individual entries at runtime.

OpenNLP further offers a tokenizer decorator (`StopwordFilteringTokenizer`) and an `ObjectStream` adapter (`StopwordFilterStream`) to plug stopword filtering into existing tokenization or training data pipelines, plus a command-line tool for ad-hoc filtering.

Loading a bundled list

A bundled stopword list can be obtained from the `StopwordLists` factory by passing the desired ISO 639-1 language code. The returned filter is a `DictionaryStopwordFilter` that is case-insensitive by default.

```
StopwordFilter filter = StopwordLists.forLanguage("en");

boolean isStop = filter.isStopword("the");       // true
String[] kept  = filter.filter(new String[] {
    "the", "quick", "brown", "fox"
});
// kept => { "quick", "brown", "fox" }
```

The set of supported languages can be queried at runtime:

```
Set<String> supported = StopwordLists.supportedLanguages();
```

Loading a custom list

Custom stopword lists can be loaded from any `InputStream` via `StopwordLists.load`. The method takes the input stream, a character set and a flag indicating whether matching should be case-sensitive.

```
try (InputStream in = new FileInputStream("my-stopwords.txt")) {
  StopwordFilter filter = StopwordLists.load(
      in, StandardCharsets.UTF_8, false /* case-insensitive */);
}
```

The expected file format is a plain UTF-8 text file with the following conventions:

- One stopword entry per line.

- Whitespace on a line separates the tokens of a multi-word entry, i.e. a line containing more than one token defines an n-gram stopword.
- Blank lines are ignored.
- Lines starting with # are treated as comments and ignored.

A small example file:

```
# my custom English stopwords
the
a
an

# multi-word entries are defined by whitespace
of the
in spite of
```

Extending or overriding defaults

A `DictionaryStopwordFilter` is immutable once constructed. To tailor a bundled list to a specific domain — for example to add project-specific noise terms or to retain a particular word that would otherwise be filtered — use the nested `DictionaryStopwordFilter.Builder`. The builder loads the bundled resource, layers user-supplied `add` / `remove` operations on top, and produces a fresh immutable filter from `build()`.

```
InputStream bundled =
    StopwordLists.class.getResourceAsStream("/opennlp/tools/stopword/en.txt");

StopwordFilter filter = DictionaryStopwordFilter.builder()
    .load(bundled, StandardCharsets.UTF_8)
    .add("foo")          // mark a new word as a stopword
    .remove("the")       // keep "the" in the output
    .build();
```

Bulk operations are also available via `addAll(Collection<String[]>)` and `removeAll(Collection<String[]>)`, which accept multi-token entries. `build()` applies all queued additions first, then all queued removals.

Multi-word stopwords

Beyond single tokens, a `StopwordFilter` can match n-grams. The overloaded `isStopword(String...)` method accepts a sequence of tokens and returns `true` if the entire sequence is registered as a multi-word stopword entry.

```
StopwordFilter filter = DictionaryStopwordFilter.builder()
    .add("of", "the")
    .add("in", "spite", "of")
    .build();

filter.isStopword("of", "the");       // true
filter.isStopword("in", "spite", "of"); // true
filter.isStopword("of");              // false
```

Multi-word entries are matched as exact, contiguous sequences and are subject to the same case-sensitivity setting as the rest of the dictionary.

The `filter(String[])` method honors both 1-gram and n-gram entries. It performs a greedy, left-to-right window scan: at each position the longest registered window is tried first; if it matches, the entire window is dropped and scanning resumes after it. Otherwise the head token is kept and scanning advances by one. Tokens that are `null` are kept in place and never participate in a window match.

```
StopwordFilter filter = DictionaryStopwordFilter.builder()
    .add("the")              // 1-gram
    .add("of", "the")        // 2-gram
    .add("in", "spite", "of") // 3-gram
    .build();

String[] kept = filter.filter(new String[] {
    "the", "king", "of", "the", "hill",
    "won", "in", "spite", "of", "rain"
});
// kept => { "king", "hill", "won", "rain" }
```

When both a shorter and a longer registered entry could match at the same position, the longer match wins. If the longest window does not match, shorter windows down to length one are tried in turn.

Decorating a Tokenizer

Any existing `Tokenizer` implementation can be wrapped in a `StopwordFilteringTokenizer` to transparently remove stopwords from its output. The decorator delegates tokenization to the wrapped instance and then runs the resulting token array through the provided `StopwordFilter`.

```
StopwordFilter filter = StopwordLists.forLanguage("en");
Tokenizer tokenizer = new StopwordFilteringTokenizer(
    SimpleTokenizer.INSTANCE, filter);

String[] tokens = tokenizer.tokenize("The quick brown fox jumps over the lazy dog");
// tokens => { "quick", "brown", "fox", "jumps", "lazy", "dog" }
```

The decorated tokenizer can be used anywhere a `Tokenizer` is expected, including in downstream OpenNLP pipelines.

Streaming usage

For training-data and bulk-processing scenarios that already operate on `ObjectStream<String[]>` (for example tokenized sentences), the `StopwordFilterStream` adapter provides a drop-in filter that strips stopwords from each emitted token array.

```
StopwordFilter filter = StopwordLists.forLanguage("en");

try (ObjectStream<String[]> tokenStream = ...;
    ObjectStream<String[]> filtered = new StopwordFilterStream(tokenStream, filter)) {

  String[] tokens;
  while ((tokens = filtered.read()) != null) {
    // tokens no longer contains any stopwords
  }
}
```

Empty arrays produced by removing every token of a short sentence are preserved so that the stream's record boundaries remain aligned with the source.

Command-line usage

Stopword filtering is also exposed as a CLI tool. The `StopwordFilter` command reads whitespace-separated tokens from standard input and writes the non-stopword tokens to standard output. The single argument is either the ISO 639-1 code of a bundled list, or a path to a custom stopword list file (same format as the Java API: one entry per line, with # comments and blank lines ignored, loaded case-insensitively).

```
$ echo "the quick brown fox" | opennlp StopwordFilter en
quick brown fox
```

To use a custom list, pass its file path instead of a language code:

```
$ echo "the quick brown fox" | opennlp StopwordFilter my-stopwords.txt
```

A bundled language code takes precedence over a file of the same name; to force loading a file whose name happens to be a language code, qualify it with a path, e.g. `./en`. The tool is intended for quick interactive checks and for use inside shell pipelines, for example chained behind a tokenizer:

```
$ opennlp SimpleTokenizer < article.txt | opennlp StopwordFilter en > filtered.txt
```

Bundled languages

The following stopword lists are bundled with OpenNLP and can be loaded by their ISO 639-1 code through `StopwordLists.forLanguage`:

ISO 639-1 code	Language
bg	Bulgarian
da	Danish
de	German
en	English
es	Spanish
fi	Finnish

ISO 639-1 code	Language
fr	French
it	Italian
nl	Dutch
pt	Portuguese
ru	Russian

The bundled lists are derived from the snowball stopword files shipped with Apache Lucene and retain their original BSD license; see the `NOTICE` and `LICENSE` files for the attribution and license text.

Thread-safety

`DictionaryStopwordFilter` is immutable once constructed and is therefore safe to share across threads without external synchronization. A filter returned by `StopwordLists.forLanguage(...)` or assembled via the `Builder` can be stored in a static field and accessed concurrently from any number of readers.

`StopwordFilteringTokenizer` and `StopwordFilterStream` are also stateless decorators with only `final` fields, so they inherit the thread-safety of the components they wrap. When paired with a `DictionaryStopwordFilter` and a thread-safe delegate tokenizer (e.g. `SimpleTokenizer.INSTANCE` or `WhitespaceTokenizer.INSTANCE`) the resulting pipeline is fully thread-safe.

Chapter 8. Name Finder

Named Entity Recognition

The Name Finder can detect named entities and numbers in text. To be able to detect entities the Name Finder needs a model. The model is dependent on the language and entity type it was trained for. The OpenNLP projects offers a number of pre-trained name finder models which are trained on various freely available corpora. They can be downloaded at our model download page. To find names in raw text the text must be segmented into tokens and sentences. A detailed description is given in the sentence detector and tokenizer tutorial. It is important that the tokenization for the training data and the input text is identical.

Name Finder Tool

The easiest way to try out the Name Finder is the command line tool. The tool is only intended for demonstration and testing. Download the English person model and start the Name Finder Tool with this command:

```
$ opennlp TokenNameFinder en-ner-person.bin
```

The name finder now reads a tokenized sentence per line from stdin, an empty line indicates a document boundary and resets the adaptive feature generators. Just copy this text to the terminal:

```
Pierre Vinken , 61 years old , will join the board as a nonexecutive director Nov. 29 .
Mr . Vinken is chairman of Elsevier N.V. , the Dutch publishing group .
Rudolph Agnew , 55 years old and former chairman of Consolidated Gold Fields PLC , was named
 a director of this British industrial conglomerate .
```

the name finder will now output the text with markup for person names:

```
<START:person> Pierre Vinken <END> , 61 years old , will join the board as a nonexecutive director Nov. 29 .
Mr . <START:person> Vinken <END> is chairman of Elsevier N.V. , the Dutch publishing group .
<START:person> Rudolph Agnew <END> , 55 years old and former chairman of Consolidated Gold Fields PLC ,
 was named a director of this British industrial conglomerate .
```

Name Finder API

To use the Name Finder in a production system it is strongly recommended to embed it directly into the application instead of using the command line interface. First the name finder model must be loaded into memory from disk or another source. In the sample below it is loaded from disk.

```
try (InputStream modelIn = new FileInputStream("en-ner-person.bin")){
  TokenNameFinderModel model = new TokenNameFinderModel(modelIn);
}
```

There is a number of reasons why the model loading can fail:

- Issues with the underlying I/O
- The version of the model is not compatible with the OpenNLP version

- The model is loaded into the wrong component, for example a tokenizer model is loaded with TokenNameFinderModel class.
- The model content is not valid for some other reason

After the model is loaded the NameFinderME can be instantiated.

```
NameFinderME nameFinder = new NameFinderME(model);
```

The initialization is now finished and the Name Finder can be used. As of OpenNLP 3.0.0 the NameFinderME class is thread safe: a single instance can be shared across multiple threads, which is the recommended deployment pattern because the underlying model is held only once in memory. The legacy ThreadSafeNameFinderME wrapper is retained for backward compatibility and is now deprecated. Note that the per-document adaptive feature generator state is cleared per thread by clearAdaptiveData, so each thread should process its own document stream end-to-end before invoking clearAdaptiveData. The input text should be segmented into documents, sentences and tokens. To perform entity detection an application calls the find method for every sentence in the document. After every document clearAdaptiveData must be called to clear the adaptive data in the feature generators. Not calling clearAdaptiveData can lead to a sharp drop in the detection rate after a few documents. The following code illustrates that:

```
for (String document[][] : documents) {
  for (String[] sentence : document) {
    Span nameSpans[] = nameFinder.find(sentence);
    // do something with the names
  }
  nameFinder.clearAdaptiveData()
}
```

the following snippet shows a call to find

```
String[] sentence = new String[]{
    "Pierre",
    "Vinken",
    "is",
    "61",
    "years"
    "old",
    "."
    };

Span[] nameSpans = nameFinder.find(sentence);
```

The nameSpans arrays contains now exactly one Span which marks the name Pierre Vinken. The elements between the start and end offsets are the name tokens. In this case the start offset is 0 and the end offset is 2. The Span object also knows the type of the entity. In this case it is person (defined by the model). It can be retrieved with a call to Span.getType(). Additionally to the statistical Name Finder, OpenNLP also offers a dictionary and a regular expression name finder implementation.

Using an ONNX Model

Using an ONNX model is similar, except we will utilize the `NameFinderDL` class instead. You must provide the path to the model file and the vocabulary file to the name finder. The vocabulary file is a `vocab.txt`, a `vocab.json`, or the `tokenizer.json` of a WordPiece model, as described for the document categorizer in the section called “Using an ONNX Model”. (There is no need to load the model as an InputStream as in the previous example.) The name finder requires a tokenized list of strings as input. The output will be an array of spans.

```
File model = new File("/path/to/model.onnx");
File vocab = new File("/path/to/vocab.txt");
// Maps every model output index to its BIO label. This must be exhaustive over the model's
// output indices; an unmapped predicted index raises IllegalStateException during find().
Map<Integer, String> ids2Labels = new HashMap<>();
ids2Labels.put(0, "O");
ids2Labels.put(1, "B-PER");
ids2Labels.put(2, "I-PER");
ids2Labels.put(3, "B-ORG");
ids2Labels.put(4, "I-ORG");
ids2Labels.put(5, "B-LOC");
ids2Labels.put(6, "I-LOC");
ids2Labels.put(7, "B-MISC");
ids2Labels.put(8, "I-MISC");
SentenceDetector sentenceDetector =
    new SentenceDetectorME(new SentenceModel(
        new File("/path/to/opennlp-en-ud-ewt-sentence-1.3-2.5.4.bin")));
String[] tokens = {"George", "Washington", "was", "president", "of", "the", "United", "States", "."};
NameFinderDL nameFinderDL = new NameFinderDL(model, vocab, ids2Labels, sentenceDetector);
// findInOriginal returns spans in the original input's coordinates.
Span[] spans = nameFinderDL.findInOriginal(tokens);
```

For additional examples, refer to the `NameFinderDLEval` class.

Long input text is split into overlapping chunks on the full Unicode `White_Space` set before WordPiece tokenization, so spacing such as a no-break space or the CJK ideographic space is recognized as a delimiter. After inference, reconstructed entity text is matched back to the caller's original input with a Unicode-aware cursor scan, so `Span#getCoveredText(...)` returns the source text even when WordPiece rejoins sub-tokens with spaces or when the source uses non-ASCII whitespace between tokens.

`findInOriginal` is declared by the `OffsetMappingNameFinder` capability interface that `NameFinderDL` implements, so a caller holding a plain `TokenNameFinder` can detect the offset-mapping capability with a `finder instanceof OffsetMappingNameFinder` check (no reflection) and fall back to token-index spans otherwise.

Optional preprocessing of the joined input text is available through `InferenceOptions` and is off by default: `setNormalizeWhitespace(true)` folds each Unicode whitespace character to an ASCII space, and `setNormalizeDashes(true)` folds Unicode dashes to the ASCII hyphen-minus. Whitespace folding is one code point to one character and preserves offsets, and so is dash folding for Basic Multilingual Plane dashes; a supplementary-plane dash shrinks from two UTF-16 units to one and shifts later offsets, which `findInOriginal` maps back across (see the section called “Use in DL components”). Full details, the underlying `CharClass` engine, and the broader normalization pipeline are documented in Chapter 6, *Text Normalization*.

The following listing reuses `model`, `vocab`, `ids2Labels`, `sentenceDetector`, and `tokens` from the example above and only shows the `InferenceOptions` wiring:

```
InferenceOptions options = new InferenceOptions();
options.setNormalizeWhitespace(true);
options.setNormalizeDashes(true);
NameFinderDL finder = new NameFinderDL(model, vocab, ids2Labels, options, sentenceDetector);
Span[] spans = finder.findInOriginal(tokens);
```

Name Finder Training

The pre-trained models might not be available for a desired language, can not detect important entities or the performance is not good enough outside the news domain. These are the typical reason to do custom training of

the name finder on a new corpus or on a corpus which is extended by private training data taken from the data which should be analyzed.

Training Tool

OpenNLP has a command line tool which is used to train the models available from the model download page on various corpora.

Note

ONNX model support is not available through the command line tool. The models that can be trained using the tool are OpenNLP models. ONNX models are trained through deep learning frameworks and then utilized by OpenNLP.

The data can be converted to the OpenNLP name finder training format. Which is one sentence per line. Some other formats are available as well. The sentence must be tokenized and contain spans which mark the entities. Documents are separated by empty lines which trigger the reset of the adaptive feature generators. A training file can contain multiple types. If the training file contains multiple types the created model will also be able to detect these multiple types.

Sample sentence of the data:

```
<START:person> Pierre Vinken <END> , 61 years old , will join the board as a nonexecutive director Nov. 29 .
Mr . <START:person> Vinken <END> is chairman of Elsevier N.V. , the Dutch publishing group .
```

The training data should contain at least 15000 sentences to create a model which performs well. Usage of the tool:

```
$ opennlp TokenNameFinderTrainer
Usage: opennlp TokenNameFinderTrainer[.evalita|.ad|.conll03|.bionlp2004|.conll02|.muc6|.ontonotes|.brat] \
[-featuregen featuregenFile] [-nameTypes types] [-sequenceCodec codec] [-factory factoryName] \
[-resources resourcesDir] [-type typeOverride] [-params paramsFile] -lang language \
-model modelFile -data sampleData [-encoding charsetName]

Arguments description:
	-featuregen featuregenFile
		The feature generator descriptor file
	-nameTypes types
		name types to use for training
	-sequenceCodec codec
		sequence codec used to code name spans
	-factory factoryName
		A sub-class of TokenNameFinderFactory
	-resources resourcesDir
		The resources directory
	-type typeOverride
		Overrides the type parameter in the provided samples
	-params paramsFile
		training parameters file.
	-lang language
		language which is being processed.
	-model modelFile
		output model file.
	-data sampleData
		data to be used, usually a file name.
	-encoding charsetName
		encoding for reading and writing text, if absent the system default is used.
```

It is now assumed that the english person name finder model should be trained from a file called 'en-ner-person.train' which is encoded as UTF-8. The following command will train the name finder and write the model to en-ner-person.bin:

```
$ opennlp TokenNameFinderTrainer -model en-ner-person.bin -lang en -data en-ner-person.train -encoding UTF-8
```

The example above will train models with a pre-defined feature set. It is also possible to use the -resources parameter to generate features based on external knowledge such as those based on word representation (clustering) features. The external resources must all be placed in a resource directory which is then passed as a parameter. If this option is used it is then required to pass, via the -featuregen parameter, an XML custom feature generator which includes some clustering features shipped with the TokenNameFinder. Currently, three formats of clustering lexicons are accepted:

- Space separated two column file specifying the token and the cluster class as generated by toolkits such as word2vec [https://code.google.com/p/word2vec/].
- Space separated three column file specifying the token, clustering class and weight as such as Clark's clusters [https://github.com/ninjin/clark_pos_induction].
- Tab separated three column Brown clusters as generated by Liang's toolkit [https://github.com/percyliang/brown-cluster].

Additionally it is possible to specify the number of iterations, the cutoff and to overwrite all types in the training data with a single type. Finally, the -sequenceCodec parameter allows to specify a BIO (Begin, Inside, Out) or BILOU (Begin, Inside, Last, Out, Unit) encoding to represent the Named Entities. An example of one such command would be as follows:

```
$ opennlp TokenNameFinderTrainer -featuregen brown.xml -sequenceCodec BILOU -resources clusters/ \
-params PerceptronTrainerParams.txt -lang en -model ner-test.bin -data en-train.opennlp -encoding UTF-8
```

Training API

To train the name finder from within an application it is recommended to use the training API instead of the command line tool. Basically three steps are necessary to train it:

- The application must open a sample data stream
- Call the NameFinderME.train method
- Save the TokenNameFinderModel to a file

The three steps are illustrated by the following sample code:

```
TokenNameFinderFactory factory = TokenNameFinderFactory.create(null, null, Collections.emptyMap(), new BioCodec());
File trainingFile = new File("en-ner-person.train");
ObjectStream<String> lineStream =
    new PlainTextByLineStream(new MarkableFileInputStreamFactory(trainingFile), StandardCharsets.UTF_8);

TokenNameFinderModel trainedModel;
try (ObjectStream<NameSample> sampleStream = new NameSampleDataStream(lineStream)) {
 trainedModel = NameFinderME.train("eng", "person", sampleStream, TrainingParameters.defaultParams(), factory);
}

File modelFile = new File("en-ner-person.bin");
try (OutputStream modelOut = new BufferedOutputStream(new FileOutputStream(modelFile))) {
 trainedModel.serialize(modelOut);
}
```

Custom Feature Generation

OpenNLP defines a default feature generation which is used when no custom feature generation is specified. Users which want to experiment with the feature generation can provide a custom feature generator. Either via an API or via a xml descriptor file.

Feature Generation defined by API

The custom generator must be used for training and for detecting the names. If the feature generation during training time and detection time is different the name finder might not be able to detect names. The following lines show how to construct a custom feature generator

```
AdaptiveFeatureGenerator featureGenerator = new CachedFeatureGenerator(
  new AdaptiveFeatureGenerator[]{
   new WindowFeatureGenerator(new TokenFeatureGenerator(), 2, 2),
   new WindowFeatureGenerator(new TokenClassFeatureGenerator(true), 2, 2),
   new OutcomePriorFeatureGenerator(),
   new PreviousMapFeatureGenerator(),
   new BigramNameFeatureGenerator(),
   new SentenceFeatureGenerator(true, false),
   new BrownTokenFeatureGenerator(BrownCluster dictResource)
  });
```

which is similar to the default feature generator but with a BrownTokenFeature added. The javadoc of the feature generator classes explain what the individual feature generators do. To write a custom feature generator please implement the AdaptiveFeatureGenerator interface or if it must not be adaptive extend the FeatureGeneratorAdapter. The train method which should be used is defined as

```
public static TokenNameFinderModel train(String languageCode, String type,
        ObjectStream<NameSample> samples, TrainingParameters trainParams,
        TokenNameFinderFactory factory) throws IOException
```

where the TokenNameFinderFactory allows to specify a custom feature generator. To detect names the model which was returned from the train method must be passed to the NameFinderME constructor.

```
new NameFinderME(model);
```

Feature Generation defined by XML Descriptor

OpenNLP can also use a xml descriptor file to configure the feature generation. The descriptor file is stored inside the model after training and the feature generators are configured correctly when the name finder is instantiated. The following sample shows a xml descriptor which contains the default feature generator plus several types of clustering features:

```
<featureGenerators cache="true" name="nameFinder">
 <generator class="opennlp.tools.util.featuregen.WindowFeatureGeneratorFactory">
  <int name="prevLength">2</int>
  <int name="nextLength">2</int>
  <generator class="opennlp.tools.util.featuregen.TokenClassFeatureGeneratorFactory"/>
 </generator>
 <generator class="opennlp.tools.util.featuregen.WindowFeatureGeneratorFactory">
  <int name="prevLength">2</int>
  <int name="nextLength">2</int>
  <generator class="opennlp.tools.util.featuregen.TokenFeatureGeneratorFactory"/>
 </generator>
 <generator class="opennlp.tools.util.featuregen.DefinitionFeatureGeneratorFactory"/>
 <generator class="opennlp.tools.util.featuregen.PreviousMapFeatureGeneratorFactory"/>
 <generator class="opennlp.tools.util.featuregen.BigramNameFeatureGeneratorFactory"/>
 <generator class="opennlp.tools.util.featuregen.SentenceFeatureGeneratorFactory">
  <bool name="begin">true</bool>
  <bool name="end">false</bool>
 </generator>
 <generator class="opennlp.tools.util.featuregen.WindowFeatureGeneratorFactory">
  <int name="prevLength">2</int>
  <int name="nextLength">2</int>
  <generator class="opennlp.tools.util.featuregen.BrownClusterTokenClassFeatureGeneratorFactory">
   <str name="dict">brownCluster</str>
  </generator>
 </generator>
 <generator class="opennlp.tools.util.featuregen.BrownClusterTokenFeatureGeneratorFactory">
  <str name="dict">brownCluster</str>
 </generator>
 <generator class="opennlp.tools.util.featuregen.BrownClusterBigramFeatureGeneratorFactory">
  <str name="dict">brownCluster</str>
 </generator>
 <generator class="opennlp.tools.util.featuregen.WordClusterFeatureGeneratorFactory">
  <str name="dict">word2vec.cluster</str>
 </generator>
 <generator class="opennlp.tools.util.featuregen.WordClusterFeatureGeneratorFactory">
  <str name="dict">clark.cluster</str>
 </generator>
</featureGenerators>
```

The root element must be featureGenerators, each sub-element adds a feature generator to the configuration. The sample xml contains additional feature generators with respect to the API defined above.

The following table shows the supported feature generators (you must specify the Factory's FQDN):

Table 8.1. Feature Generators

Feature Generator	Parameters
CharacterNgramFeatureGeneratorFactory	*min* and *max* specify the length of the generated character ngrams
DefinitionFeatureGeneratorFactory	none
DictionaryFeatureGeneratorFactory	*dict* is the key of the dictionary resource to use, and *prefix* is a feature prefix string
PreviousMapFeatureGeneratorFactory	none
SentenceFeatureGeneratorFactory	*begin* and *end* to generate begin or end features, both are optional and are boolean values
TokenClassFeatureGeneratorFactory	none
TokenFeatureGeneratorFactory	none

Feature Generator	Parameters
BigramNameFeatureGeneratorFactory	none
TokenPatternFeatureGeneratorFactory	none
POSTaggerNameFeatureGeneratorFactory	*model* is the file name of the POS Tagger model to use
WordClusterFeatureGeneratorFactory	*dict* is the key of the clustering resource to use
BrownClusterTokenFeatureGeneratorFactory	*dict* is the key of the clustering resource to use
BrownClusterTokenClassFeatureGeneratorFactory	*dict* is the key of the clustering resource to use
BrownClusterBigramFeatureGeneratorFactory	*dict* is the key of the clustering resource to use
WindowFeatureGeneratorFactory	*prevLength* and *nextLength* must be integers and specify the window size

Window feature generator can contain other generators.

Evaluation

The built in evaluation can measure the named entity recognition performance of the name finder. The performance is either measured on a test dataset or via cross validation.

Evaluation Tool

The following command shows how the tool can be run:

```
$ opennlp TokenNameFinderEvaluator -model en-ner-person.bin -data en-ner-person.test -encoding UTF-8

Precision: 0.8005071889818507
Recall: 0.7450581122145297
F-Measure: 0.7717879983140168
```

Note

The command line interface does not support cross evaluation in the current version.

Evaluation API

The evaluation can be performed on a pre-trained model and a test dataset or via cross validation. In the first case the model must be loaded and a NameSample ObjectStream must be created (see code samples above), assuming these two objects exist the following code shows how to perform the evaluation:

```
TokenNameFinderEvaluator evaluator = new TokenNameFinderEvaluator(new NameFinderME(model));
evaluator.evaluate(sampleStream);

FMeasure result = evaluator.getFMeasure();
System.out.println(result.toString());
```

In the cross validation case all the training arguments must be provided (see the Training API section above). To perform cross validation the ObjectStream must be resettable.

```
InputStreamFactory dataIn = new MarkableFileInputStreamFactory(new File("en-ner-person.train"));
ObjectStream<NameSample> sampleStream = new NameSampleDataStream(
    new PlainTextByLineStream(dataIn, StandardCharsets.UTF_8));
TokenNameFinderCrossValidator evaluator = new TokenNameFinderCrossValidator("eng",
    null, TrainingParameters.defaultParams(), null, (TokenNameFinderEvaluationMonitor) null);
evaluator.evaluate(sampleStream, 10);

FMeasure result = evaluator.getFMeasure();
System.out.println(result.toString());
```

Named Entity Annotation Guidelines

Annotation guidelines define what should be labeled as an entity. To build a private corpus it is important to know these guidelines and maybe write a custom one. Here is a list of publicly available annotation guidelines:

- MUC6 [https://cs.nyu.edu/cs/faculty/grishman/NEtask20.book_1.html]
- MUC7 [http://acl.ldc.upenn.edu/muc7/ne_task.html]
- ACE [https://www.ldc.upenn.edu/sites/www.ldc.upenn.edu/files/english-entities-guidelines-v6.6.pdf]
- CONLL 2002 [https://www.clips.uantwerpen.be/conll2002/ner/]
- CONLL 2003 [https://www.clips.uantwerpen.be/conll2003/ner/]

Chapter 9. Document Categorizer

Classifying

The OpenNLP Document Categorizer can classify text into pre-defined categories. It supports multiple classification backends: Maximum Entropy (default), Support Vector Machines (SVM) via the `opennlp-ml-libsvm` module, and ONNX-based deep learning models via the `opennlp-dl` module. For someone interested in Gross Margin, the sample text given below could be classified as GMDecrease

```
Major acquisitions that have a lower gross margin than the existing network
also had a negative impact on the overall gross margin, but it should improve
following the implementation of its integration strategies.
```

and the text below could be classified as GMIncrease

```
The upward movement of gross margin resulted from amounts pursuant to
adjustments to obligations towards dealers.
```

To be able to classify a text, the document categorizer needs a model. The classifications are requirements-specific and hence there is no pre-built model for document categorizer under OpenNLP project.

Document Categorizer Tool

Note that ONNX model support is not available through the command line tool. The models that can be trained using the tool are OpenNLP models. ONNX models are trained through deep learning frameworks and then utilized by OpenNLP.

The easiest way to try out the document categorizer is the command line tool. The tool is only intended for demonstration and testing. The following command shows how to use the document categorizer tool.

```
$ opennlp Doccat model
```

The input is read from standard input and output is written to standard output, unless they are redirected or piped. As with most components in OpenNLP, document categorizer expects input which is segmented into sentences.

Document Categorizer API

To perform classification you will need a maxent model - these are encapsulated in the DoccatModel class of OpenNLP tools - or an ONNX model trained for document classification.

Using an OpenNLP model, first you need to grab the bytes from the serialized model on an InputStream:

```
InputStream is = ...
DoccatModel m = new DoccatModel(is);
```

With the DoccatModel in hand we are just about there:

```
String inputText = ...
String[] textTokens = inputText.split(" "); // split by whitespace
DocumentCategorizerME myCategorizer = new DocumentCategorizerME(m);
double[] outcomes = myCategorizer.categorize(textTokens);
String category = myCategorizer.getBestCategory(outcomes);
```

Using an SVM Model

OpenNLP provides SVM-based document categorization through the `DocumentCategorizerSVM` class in the `opennlp-ml-libsvm` module. This implementation uses the zlibsvm [https://github.com/rzo1/zlibsvm] library, an object-oriented Java binding for LIBSVM.

SVM text classification supports configurable term weighting strategies (binary, TF, TF-IDF, log-normalized TF), feature selection strategies (information gain, chi-square, term frequency, document frequency), feature scaling, and all SVM parameters (kernel type, cost, gamma, etc.).

To use SVM-based categorization, first add the dependency:

```
<dependency>
   <groupId>org.apache.opennlp</groupId>
   <artifactId>opennlp-ml-libsvm</artifactId>
   <version>CURRENT_OPENNLP_VERSION</version>
</dependency>
```

Training an SVM model from `DocumentSample` data:

```
// Configure the SVM categorizer
SvmDoccatConfiguration config = new SvmDoccatConfiguration.Builder()
   .setSvmConfiguration(new SvmConfigurationImpl.Builder()
      .setKernelType(KernelType.LINEAR)
      .setProbability(true)
      .build())
   .setTermWeightingStrategy(TermWeightingStrategy.TF_IDF)
   .setFeatureSelectionStrategy(FeatureSelectionStrategy.INFORMATION_GAIN)
   .setMaxFeatures(1000)
   .setScaleFeatures(true)
   .setScaleRange(0.0, 1.0)
   .build();

// Train from a DocumentSample stream with a bag-of-words feature generator
FeatureGenerator featureGenerator = new BagOfWordsFeatureGenerator();
SvmDoccatModel model = DocumentCategorizerSVM.train("eng", sampleStream, config, featureGenerator);

// Serialize the model
try (OutputStream out = new FileOutputStream("doccat-svm.bin")) {
   model.serialize(out);
}
```

Classifying text with the trained model:

```
// Load the model
SvmDoccatModel model;
try (InputStream in = new FileInputStream("doccat-svm.bin")) {
   model = SvmDoccatModel.deserialize(in);
}

// Classify
DocumentCategorizer categorizer = new DocumentCategorizerSVM(model, new BagOfWordsFeatureGenerator());
double[] outcomes = categorizer.categorize(new String[]{"tokens", "of", "text"});
String bestCategory = categorizer.getBestCategory(outcomes);
```

Using an ONNX Model

Using an ONNX model is similar, except we will utilize the `DocumentCategorizerDL` class instead. You must provide the path to the model file and the vocabulary file to the document categorizer. (There is no need to load the model as an InputStream as in the previous example.)

```
File model = new File("/path/to/model.onnx");
File vocab = new File("/path/to/vocab.txt");
Map<Integer, String> categories = new HashMap<>();
String[] inputText = new String[]{"My input text is great."};
final DocumentCategorizerDL myCategorizer = new DocumentCategorizerDL(
   model, vocab, categories, new AverageClassificationScoringStrategy(), new InferenceOptions());
double[] outcomes = myCategorizer.categorize(inputText);
String category = myCategorizer.getBestCategory(outcomes);
```

The vocabulary file is either plain text with one token per line, as in `vocab.txt`, or a JSON object mapping each token to an integer ID, as in `vocab.json`. A Hugging Face `tokenizer.json` of a WordPiece model is also accepted: `model.vocab` supplies the tokens, and an entry of `added_tokens` that `model.vocab` lacks is added with its ID. A `tokenizer.json` of another model type, such as BPE or Unigram, is rejected, and so is an added token whose ID or token conflicts with `model.vocab`. When the file sets `normalizer.lowercase`, it must agree with the lower casing the component is configured with; the other tokenizer settings of the file are not read. An unreadable vocabulary is rejected with an `InvalidFormatException`. The same vocabulary files are accepted by `NameFinderDL` and `SentenceVectorsDL`.

When a configuration file is given in place of the categories map, its top-level `id2label` object maps each output index to a category label. A configuration that is not well-formed JSON, or whose `id2label` is not an object of strings keyed by integers, is rejected with an `InvalidFormatException`.

Like `NameFinderDL`, long input is split into overlapping chunks on the full Unicode `White_Space` set rather than Java's `\s`, so text copied from PDFs, the web, or multilingual sources tokenizes consistently. Optional preprocessing through `InferenceOptions` is off by default: `setNormalizeWhitespace(true)` maps each Unicode whitespace code point to an ASCII space, and `setNormalizeDashes(true)` maps Unicode dashes to the ASCII hyphen-minus. Whitespace folding is a one-to-one replacement that preserves character offsets, and so is dash folding for Basic Multilingual Plane dashes; a supplementary-plane dash shrinks from two UTF-16 units to one and shifts later offsets (see the section called “Use in DL components”). See Chapter 6, *Text Normalization* for the shared `CharClass` engine and the full normalization library.

```
InferenceOptions options = new InferenceOptions();
options.setNormalizeWhitespace(true);
options.setNormalizeDashes(true);
DocumentCategorizerDL categorizer = new DocumentCategorizerDL(
   model, vocab, categories, new AverageClassificationScoringStrategy(), options);
```

Training

The Document Categorizer can be trained on annotated training material. The data can be in OpenNLP Document Categorizer training format. This is one document per line, containing category and text separated by a whitespace. Other formats can also be available. The following sample shows the sample from above in the required format. Here GMDecrease and GMIncrease are the categories.

```
MDecrease Major acquisitions that have a lower gross margin than the existing network also \
  had a negative impact on the overall gross margin, but it should improve following \
  the implementation of its integration strategies .
GMIncrease The upward movement of gross margin resulted from amounts pursuant to adjustments \
  to obligations towards dealers .
```

Note

The line breaks marked with a backslash are just inserted for formatting purposes and must not be included in the training data.

Training Tool

The following command will train the document categorizer and write the model to en-doccat.bin:

```
$ opennlp DoccatTrainer -model en-doccat.bin -lang en -data en-doccat.train -encoding UTF-8
```

Additionally it is possible to specify the number of iterations, and the cutoff.

Training API

So, naturally you will need some access to many pre-classified events to train your model. The class opennlp.tools.doccat.DocumentSample encapsulates a text document and its classification. DocumentSample has two constructors. Each take the text's category as one argument. The other argument can either be raw text, or an array of tokens. By default, the raw text will be split into tokens by whitespace. So, let's say your training data was contained in a text file, where the format is as described above. Then you might want to write something like this to create a collection of DocumentSamples:

```
DoccatModel model = null;
try {
  ObjectStream<String> lineStream =
  new PlainTextByLineStream(new MarkableFileInputStreamFactory(new File("en-sentiment.train")),
 StandardCharsets.UTF_8);

  ObjectStream<DocumentSample> sampleStream = new DocumentSampleStream(lineStream);

  model = DocumentCategorizerME.train("eng", sampleStream,
      TrainingParameters.defaultParams(), new DoccatFactory());
} catch (IOException e) {
  e.printStackTrace();
}
```

Now might be a good time to cruise over to Hulu or something, because this could take a while if you've got a large training set. You may see a lot of output as well. Once you're done, you can pretty quickly step to classification directly, but first we'll cover serialization. Feel free to skim.

```
try (OutputStream modelOut = new BufferedOutputStream(new FileOutputStream(modelFile))) {
  model.serialize(modelOut);
}
```

Training an SVM Model

The SVM-based document categorizer uses the same `DocumentSample` training format as the Maximum Entropy categorizer. Training is done through the `DocumentCategorizerSVM.train()` method with a `SvmDoccatConfiguration` that controls the SVM parameters, term weighting, feature selection, and feature scaling.

A typical SVM text classification pipeline uses a LINEAR kernel with binary or TF-IDF term weighting and information gain or chi-square feature selection:

```
SvmDoccatConfiguration config = new SvmDoccatConfiguration.Builder()
    .setSvmConfiguration(new SvmConfigurationImpl.Builder()
        .setKernelType(KernelType.LINEAR)
        .setProbability(true)
        .build())
    .setTermWeightingStrategy(TermWeightingStrategy.TF_IDF)
    .setFeatureSelectionStrategy(FeatureSelectionStrategy.INFORMATION_GAIN)
    .setMaxFeatures(1000)
    .setScaleFeatures(true)
    .build();

SvmDoccatModel model = DocumentCategorizerSVM.train("eng", sampleStream, config,
    new BagOfWordsFeatureGenerator());
```

The resulting `SvmDoccatModel` can be serialized and deserialized using Java serialization:

```
// Save
model.serialize(new FileOutputStream("svm-doccat.bin"));

// Load
SvmDoccatModel loaded = SvmDoccatModel.deserialize(new FileInputStream("svm-doccat.bin"));
```

Chapter 10. Sentiment Analysis

Classifying

The OpenNLP Sentiment Analyzer can classify text into sentiment categories such as "positive" or "negative". It is based on the maximum entropy framework. For example, the text below could be classified as *positive*:

```
I love this product it is absolutely wonderful and amazing
```

and the text below could be classified as *negative*:

```
Terrible experience the worst customer service I have ever had
```

To be able to classify text, the sentiment analyzer needs a model. The sentiment categories are requirements-specific and defined by the training data. There are no pre-built models for sentiment analysis under the OpenNLP project.

Sentiment Analysis Tool

The easiest way to try out the sentiment analyzer is the command line tool. The tool is only intended for demonstration and testing. The following command shows how to use the sentiment analysis tool:

```
$ opennlp Sentiment model
```

The input is read from standard input and the predicted sentiment is written to standard output.

Sentiment Analysis API

To perform sentiment classification you will need a model encapsulated in the `SentimentModel` class. First, load the model from an `InputStream`:

```
InputStream is = ...
SentimentModel model = new SentimentModel(is);
```

With the `SentimentModel` in hand, create a `SentimentME` instance and predict sentiments:

```
SentimentME sentiment = new SentimentME(model);

// Predict from a raw sentence string (tokenized internally)
String result = sentiment.predict("I love this product");

// Or predict from pre-tokenized input
String[] tokens = new String[]{"I", "love", "this", "product"};
String result2 = sentiment.predict(tokens);

// Access the probability distribution over sentiment categories
double[] probabilities = sentiment.probabilities(tokens);
String bestSentiment = sentiment.getBestSentiment(probabilities);
```

Training

The Sentiment Analyzer can be trained on annotated training material. The data format is one sample per line, containing the sentiment category followed by the text tokens, all separated by whitespace. The following sample shows the required format:

```
positive I love this movie it is absolutely wonderful and amazing
positive This product is great and I am very happy with it
negative I hate this product it broke after one day of use
negative Terrible experience the worst customer service I have ever had
```

Training Tool

The following command will train the sentiment analyzer and write the model to `en-sentiment.bin`:

```
$ opennlp SentimentTrainer -model en-sentiment.bin -lang en -data en-sentiment.train -encoding UTF-8
```

Training API

To train a sentiment model programmatically, prepare an `ObjectStream` of `SentimentSample` objects and call the `SentimentME.train()` method:

```
SentimentModel model;

InputStreamFactory dataIn = new MarkableFileInputStreamFactory(
    new File("en-sentiment.train"));

ObjectStream<String> lineStream =
    new PlainTextByLineStream(dataIn, StandardCharsets.UTF_8);
ObjectStream<SentimentSample> sampleStream =
    new SentimentSampleStream(lineStream);

model = SentimentME.train("eng", sampleStream,
    TrainingParameters.defaultParams(), new SentimentFactory());
```

Once trained, the model can be serialized for later use:

```
try (OutputStream modelOut = new BufferedOutputStream(
    new FileOutputStream("en-sentiment.bin"))) {
  model.serialize(modelOut);
}
```

Evaluation

Evaluation Tool

The sentiment analyzer can be evaluated against test data using the command line tool:

```
$ opennlp SentimentEvaluator -model en-sentiment.bin -data en-sentiment.test -encoding UTF-8
```

This will output precision, recall, and F-measure statistics.

Cross Validation

K-fold cross validation can be performed to evaluate the model without a separate test set:

```
$ opennlp SentimentCrossValidator -lang en -data en-sentiment.train -encoding UTF-8 -folds 10
```

Evaluation API

The evaluation API allows programmatic evaluation against a set of `SentimentSample` references:

```
SentimentME sentiment = new SentimentME(model);
SentimentEvaluator evaluator = new SentimentEvaluator(sentiment);
evaluator.evaluate(testSampleStream);

System.out.println(evaluator.getFMeasure());
```

Chapter 11. Part-of-Speech Tagger

Tagging

The Part of Speech Tagger marks tokens with their corresponding word type based on the token itself and the context of the token. A token might have multiple pos tags depending on the token and the context. The OpenNLP POS Tagger uses a probability model to predict the correct pos tag out of the tag set. To limit the possible tags for a token a tag dictionary can be used which increases the tagging and runtime performance of the tagger.

POS Tagger Tool

The easiest way to try out the POS Tagger is the command line tool. The tool is only intended for demonstration and testing. Download the English maxent pos model and start the POS Tagger Tool with this command:

```
$ opennlp POSTagger opennlp-en-ud-ewt-pos-1.3-2.5.4.bin
```

The POS Tagger now reads a tokenized sentence per line from stdin. Copy these two sentences to the console:

```
Pierre Vinken , 61 years old , will join the board as a nonexecutive director Nov. 29 .
Mr. Vinken is chairman of Elsevier N.V. , the Dutch publishing group .
```

The POS Tagger will now echo the sentences with pos tags to the console:

```
Pierre_PROPN Vinken_PROPN ,_PUNCT 61_NUM years_NOUN old_ADJ ,_PUNCT will_AUX join_VERB the_DET
 board_NOUN as_ADP
  a_DET nonexecutive_ADJ director_NOUN Nov._PROPN 29_NUM ._PUNCT
Mr._PROPN Vinken_PROPN is_AUX chairman_NOUN of_ADP Elsevier_ADJ N.V._PROPN ,_PUNCT the_DET
 Dutch_PROPN publishing_VERB group_NOUN .
```

The tag set used by the English pos model is the Penn Treebank tag set [https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html].

POS Tagger API

The POS Tagger can be embedded into an application via its API. First the pos model must be loaded into memory from disk or another source. In the sample below it is loaded from disk.

```
try (InputStream modelIn = new FileInputStream("opennlp-en-ud-ewt-pos-1.3-2.5.4.bin"){
  POSModel model = new POSModel(modelIn);
}
```

After the model is loaded the POSTaggerME can be instantiated.

```
POSTaggerME tagger = new POSTaggerME(model);
```

The POS Tagger instance is now ready to tag data. It expects a tokenized sentence as input, which is represented as a String array, each String object in the array is one token. As of OpenNLP 3.0.0 POSTaggerME is thread safe

and a single instance can be shared across threads; the legacy ThreadSafePOSTaggerME wrapper is retained for backward compatibility and is now deprecated.

The following code shows how to determine the most likely pos tag sequence for a sentence.

```
String[] sent = new String[]{"Most", "large", "cities", "in", "the", "US", "had",
                             "morning", "and", "afternoon", "newspapers", "."};
String[] tags = tagger.tag(sent);
```

The tags array contains one part-of-speech tag for each token in the input array. The corresponding tag can be found at the same index as the token has in the input array. The confidence scores for the returned tags can be easily retrieved from a POSTaggerME with the following method call:

```
double[] probs = tagger.probs();
```

The call to probs is stateful and will always return the probabilities of the last tagged sentence. When the tagger is shared across threads, the "last tagged sentence" is tracked per thread, so each thread must call tag before calling probs on that same thread. The probs method should only be called when the tag method was called before on the current thread, otherwise the behavior is undefined.

Some applications need to retrieve the n-best pos tag sequences and not only the best sequence. The topKSequences method is capable of returning the top sequences. It can be called in a similar way as tag.

```
Sequence[] topSequences = tagger.topKSequences(sent);
```

Each Sequence object contains one sequence. The sequence can be retrieved via Sequence.getOutcomes() which returns a tags array and Sequence.getProbs() returns the probability array for this sequence.

Training

The POS Tagger can be trained on annotated training material. The training material is a collection of tokenized sentences where each token has the assigned part-of-speech tag. The native POS Tagger training material looks like this:

```
About_ADV 10_NUM Euro_PROPN ,_PUNCT I_PRON reckon._PUNCT
That_PRON sounds_VERB good_ADJ ._PUNCT
```

Each sentence must be in one line. The token/tag pairs are combined with "_". The token/tag pairs are whitespace separated. The data format does not define a document boundary. If a document boundary should be included in the training material it is suggested to use an empty line.

The Part-of-Speech Tagger can either be trained with a command line tool, or via a training API.

Training Tool

OpenNLP has a command line tool which is used to train the models available from the model download page on various corpora.

Usage of the tool:

```
$ opennlp POSTaggerTrainer
Usage: opennlp POSTaggerTrainer[.conllx] [-type maxent|perceptron|perceptron_sequence] \
            [-dict dictionaryPath] [-ngram cutoff] [-params paramsFile] [-iterations num] \
            [-cutoff num] -model modelFile -lang language -data sampleData \
            [-encoding charsetName]

Arguments description:
      -type maxent|perceptron|perceptron_sequence
            The type of the token name finder model. One of maxent|perceptron|perceptron_sequence.
      -dict dictionaryPath
            The XML tag dictionary file
      -ngram cutoff
            NGram cutoff. If not specified will not create ngram dictionary.
      -params paramsFile
            training parameters file.
      -iterations num
            number of training iterations, ignored if -params is used.
      -cutoff num
            minimal number of times a feature must be seen, ignored if -params is used.
      -model modelFile
            output model file.
      -lang language
            language which is being processed.
      -data sampleData
            data to be used, usually a file name.
      -encoding charsetName
            encoding for reading and writing text, if absent the system default is used.
```

The following command illustrates how an English part-of-speech model can be trained:

```
$ opennlp POSTaggerTrainer -type maxent -model en-custom-pos-maxent.bin \
                    -lang en -data en-custom-pos.train -encoding UTF-8
```

Training API

The Part-of-Speech Tagger training API supports the training of a new pos model. Basically three steps are necessary to train it:

- The application must open a sample data stream
- Call the 'POSTagger.train' method
- Save the POSModel to a file

The following code illustrates that:

```
POSModel model = null;

try {
  ObjectStream<String> lineStream = new PlainTextByLineStream(
   new MarkableFileInputStreamFactory(new File("en-custom-pos-maxent.bin")), StandardCharsets.UTF_8);

  ObjectStream<POSSample> sampleStream = new WordTagSampleStream(lineStream);

  model = POSTaggerME.train("eng", sampleStream, TrainingParameters.defaultParams(), new POSTaggerFactory());
} catch (IOException e) {
  e.printStackTrace();
}
```

The above code performs the first two steps, opening the data and training the model. The trained model must still be saved into an OutputStream, in the sample below it is written into a file.

```
try (OutputStream modelOut = new BufferedOutputStream(new FileOutputStream(modelFile))){
  model.serialize(modelOut);
}
```

Tag Dictionary

The tag dictionary is a word dictionary which specifies which tags a specific token can have. Using a tag dictionary has two advantages, inappropriate tags can not been assigned to tokens in the dictionary and the beam search algorithm has to consider fewer possibilities and can search faster.

The dictionary is defined in a xml format and can be created and stored with the POSDictionary class. Below is an example to train a custom model using a tag dictionary.

Sample POS Training material (file : en-custom-pos.train)

```
It_PRON is_OTHER spring_PROPN season_NOUN. The_DET flowers_NOUN are_OTHER red_ADJ and_CCONJ
 yellow_ADJ ._PUNCT
Red_NOUN is_OTHER my_DET favourite_ADJ colour_NOUN ._PUNCT
```

Sample Tag Dictionary (file : dictionary.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary case_sensitive="false">
  <entry tags="PRON">
    <token>It</token>
  </entry>
  <entry tags="OTHER">
    <token>is</token>
  </entry>
  <entry tags="PROPN">
    <token>Spring</token>
  </entry>
  <entry tags="NOUN">
    <token>season</token>
  </entry>
  <entry tags="DET">
    <token>the</token>
  </entry>
  <entry tags="NOUN">
    <token>flowers</token>
  </entry>
  <entry tags="OTHER">
    <token>are</token>
  </entry>
  <entry tags="NOUN">
    <token>red</token>
  </entry>
  <entry tags="CCONJ">
    <token>and</token>
  </entry>
  <entry tags="NOUN">
    <token>yellow</token>
  </entry>
  <entry tags="PRON">
    <token>my</token>
  </entry>
  <entry tags="ADJ">
    <token>favourite</token>
  </entry>
  <entry tags="NOUN">
    <token>colour</token>
  </entry>
  <entry tags="PUNCT">
    <token>.</token>
  </entry>
</dictionary>
```

Sample code to train a model using above tag dictionary

```
POSModel model = null;
 try {
  ObjectStream<String> lineStream = new PlainTextByLineStream(
    new MarkableFileInputStreamFactory(new File("en-custom-pos.train")), StandardCharsets.UTF_8);

  ObjectStream<POSSample> sampleStream = new WordTagSampleStream(lineStream);

  TrainingParameters params = ModelUtil.createDefaultTrainingParameters();
  params.put(TrainingParameters.CUTOFF_PARAM, 0);

  POSTaggerFactory factory = new POSTaggerFactory();
  TagDictionary dict = factory.createTagDictionary(new File("dictionary.xml"));
  factory.setTagDictionary(dict);

  model = POSTaggerME.train("eng", sampleStream, params, factory);

  OutputStream modelOut = new BufferedOutputStream(new FileOutputStream("en-custom-pos-maxent.bin"));
  model.serialize(modelOut);

 } catch (IOException e) {
  e.printStackTrace();
 }
```

The custom model is then used to tag a sequence.

```
String[] sent = new String[]{"Spring", "is", "my", "favourite", "season", "."};
String[] tags = tagger.tag(sent);
Arrays.stream(tags).forEach(k -> System.out.print(k + " "));
```

```
  Input
     Sentence: Spring is my favourite season.

  Output
     POS Tags using the custom model (en-custom-pos-maxent.bin): \
    PROPN OTHER PRON ADJ NOUN PUNCT

  Output with the default model
     POS Tags using the default model (opennlp-en-ud-ewt-pos-1.3-2.5.4.bin): \
    NOUN AUX PRON ADJ NOUN PUNCT
```

Evaluation

The built-in evaluation can measure the accuracy of the pos tagger. The accuracy can be measured on a test data set or via cross validation.

Evaluation Tool

There is a command line tool to evaluate a given model on a test data set. The following command shows how the tool can be run:

```
$ opennlp POSTaggerEvaluator -model pt.postagger.bin -data pt.postagger.test -encoding utf-8
```

This will display the resulting accuracy score, e.g.:

```
Loading model ... done
Evaluating ... done

Accuracy: 0.9659110277825124
```

There is a command line tool for cross-validation of the test data set. The following command shows how the tool can be run:

```
$ opennlp POSTaggerCrossValidator -lang pt -data pt.postagger.test -encoding utf-8
```

This will display the resulting accuracy score, e.g.:

```
Accuracy: 0.9659110277825124
```

Chapter 12. Lemmatizer

Lemmatizer Tool

A lemmatizer returns, for a given word form (token) and Part of Speech tag, the dictionary form of a word, which is usually referred to as its lemma. A token could ambiguously be derived from several basic forms or dictionary words which is why the postag of the word is required to find the lemma. For example, the form `show' may refer to either the verb "to show" or to the noun "show". Currently, OpenNLP implement statistical and dictionary-based lemmatizers.

The easiest way to try out the Lemmatizer is the command line tool, which provides access to the statistical lemmatizer. Note that the tool is only intended for demonstration and testing.

Once you have trained a lemmatizer model (see below for instructions), you can start the Lemmatizer Tool with this command:

```
$ opennlp LemmatizerME opennlp-en-ud-ewt-lemmas-1.3-2.5.4.bin < sentences
```

The Lemmatizer now reads a pos tagged sentence(s) per line from standard input. For example, you can copy this sentence to the console:

```
Rockwell_PROPN International_ADJ Corp_NOUN 's_PUNCT Tulsa_PROPN unit_NOUN said_VERB it_PRON
signed_VERB a_DET tentative_NOUN agreement_NOUN extending_VERB its_PRON contract_NOUN
with_ADP Boeing_PROPN Co._NOUN to_PART provide_VERB structural_ADJ parts_NOUN for_ADP
Boeing_PROPN 's_PUNCT 747_NUM jetliners_NOUN ._PUNCT
```

The Lemmatizer will now echo the lemmas for each word postag pair to the console:

```
Rockwell PROPN rockwell
International ADJ international
Corp NOUN corp
's PUNCT 's
Tulsa PROPN tulsa
unit NOUN unit
said VERB say
it PRON it
signed VERB sign
...
```

Lemmatizer API

The Lemmatizer can be embedded into an application via its API. Currently, a statistical and DictionaryLemmatizer are available. Note that these two methods are complementary and the DictionaryLemmatizer can also be used as a way of post-processing the output of the statistical lemmatizer.

The statistical lemmatizer requires that a trained model is loaded into memory from disk or from another source. In the example below it is loaded from disk:

```
LemmatizerModel model = null;
try (InputStream modelIn = new FileInputStream("opennlp-en-ud-ewt-lemmas-1.3-2.5.4.bin")) {
  model = new LemmatizerModel(modelIn);
}
```

Once the model is loaded, a LemmatizerME can be instantiated.

```
LemmatizerME lemmatizer = new LemmatizerME(model);
```

The Lemmatizer instance is now ready to lemmatize data. It expects a tokenized sentence as input, which is represented as a String array, each String object in the array is one token, and the POS tags associated with each token. As of OpenNLP 3.0.0 LemmatizerME is thread safe and a single instance can be shared across threads; the legacy ThreadSafeLemmatizerME wrapper is retained for backward compatibility and is now deprecated.

The following code shows how to determine the most likely lemma for a sentence.

```
String[] tokens = new String[] { "Rockwell", "International", "Corp.", "'s",
    "Tulsa", "unit", "said", "it", "signed", "a", "tentative", "agreement",
    "extending", "its", "contract", "with", "Boeing", "Co.", "to",
    "provide", "structural", "parts", "for", "Boeing", "'s", "747",
    "jetliners", "." };

String[] postags = new String[] { "PROPN", "ADJ", "NOUN", "PUNCT", "PROPN", "NOUN",
    "VERB", "PRON", "VERB", "DET", "NOUN", "NOUN", "VERB", "PRON", "NOUN", "ADP",
    "PROPN", "NOUN", "PART", "VERB", "ADJ", "NOUN", "ADP", "PROPN", "PUNCT", "NUM", "NOUN",
    "PUNCT" };

String[] lemmas = lemmatizer.lemmatize(tokens, postags);
```

The lemmas array contains one lemma for each token in the input array. The corresponding tag and lemma can be found at the same index as the token has in the input array.

The DictionaryLemmatizer is constructed by passing the InputStream of a lemmatizer dictionary. Such dictionary consists of a text file containing, for each row, a word, its postag and the corresponding lemma, each column separated by a tab character.

```
show  NOUN show
showcase NOUN showcase
showcases NOUN showcase
showdown NOUN showdown
showdowns NOUN showdown
shower  NOUN shower
showers  NOUN shower
showman  NOUN showman
showmanship NOUN showmanship
showmen  NOUN showman
showroom NOUN showroom
showrooms NOUN showroom
shows  NOUN show
shrapnel NOUN shrapnel
```

Alternatively, if a (word,postag) pair can output multiple lemmas, the the lemmatizer dictionary would consist of a text file containing, for each row, a word, its postag and the corresponding lemmas separated by "#":

```
muestras NOUN muestra
cantaba  VERB cantar
fue  VERB ir#ser
entramos VERB entrar
```

First the dictionary must be loaded into memory from disk or another source. In the sample below it is loaded from disk.

```
InputStream dictLemmatizer = null;
try (dictLemmatizer = new FileInputStream("english-dict-lemmatizer.txt")) {

}
```

After the dictionary is loaded the DictionaryLemmatizer can be instantiated.

```
DictionaryLemmatizer lemmatizer = new DictionaryLemmatizer(dictLemmatizer);
```

The DictionaryLemmatizer instance is now ready. It expects two String arrays as input, containing the tokens and their respective postags. The following code shows how to find a lemma using a DictionaryLemmatizer.

```
String[] tokens = new String[]{"Most", "large", "cities", "in", "the", "US", "had",
                    "morning", "and", "afternoon", "newspapers", "."};
String[] tags = tagger.tag(sent);
String[] lemmas = lemmatizer.lemmatize(tokens, tags);
```

The tags array contains one part-of-speech tag for each token in the input array. The corresponding tag and lemmas can be found at the same index as the token has in the input array.

Lemmatizer Training

The training data consist of three columns separated by tabs. Each word has been put on a separate line and there is an empty line after each sentence. The first column contains the current word, the second its part-of-speech tag and the third its lemma. Here is an example of the file format, with a sample sentence of the training data:

```
He       PRON  he
reckons   VERB  reckon
the      DET   the
current   ADJ   current
accounts  NOUN  account
deficit   NOUN   deficit
will     AUX   will
narrow    VERB   narrow
to       PART   to
only     ADV   only
#        #    #
1.8      NUM   1.8
millions  NOUN   million
in       ADP   in
September PROPN  september
.        PUNCT   O
```

The Universal Dependencies Treebank and the CoNLL 2009 datasets distribute training data for many languages.

Training Tool

OpenNLP has a command line tool which is used to train the models on various corpora.

Usage of the tool:

```
$ opennlp LemmatizerTrainerME
Usage: opennlp LemmatizerTrainerME [-factory factoryName] [-params paramsFile] -lang language -model modelFile -data
 sampleData [-encoding charsetName]

Arguments description:
  -factory factoryName
    A sub-class of LemmatizerFactory where to get implementation and resources.
  -params paramsFile
    training parameters file.
  -lang language
    language which is being processed.
  -model modelFile
    output model file.
  -data sampleData
    data to be used, usually a file name.
  -encoding charsetName
    encoding for reading and writing text, if absent the system default is used.
```

Its now assumed that the english lemmatizer model should be trained from a file called 'en-custom-lemmatizer.train' which is encoded as UTF-8. The following command will train the lemmatizer and write the model to en-custom-lemmatizer.bin:

```
$ opennlp LemmatizerTrainerME -model en-custom-lemmatizer.bin -params PerceptronTrainerParams.txt \
                    -lang en -data en-custom-lemmatizer.train -encoding UTF-8
```

Training API

The Lemmatizer offers an API to train a new lemmatizer model. First a training parameters file needs to be instantiated:

```
TrainingParameters mlParams = CmdLineUtil.loadTrainingParameters(params.getParams(), false);
if (mlParams == null) {
  mlParams = ModelUtil.createDefaultTrainingParameters();
}
```

Then we read the training data:

```
InputStreamFactory inputStreamFactory = null;
try {
  inputStreamFactory = new MarkableFileInputStreamFactory(new File(en-custom-lemmatizer.train));
} catch (FileNotFoundException e) {
  e.printStackTrace();
}
ObjectStream<String> lineStream = null;
LemmaSampleStream lemmaStream = null;
try {
  lineStream = new PlainTextByLineStream((inputStreamFactory), StandardCharsets.UTF_8);
  lemmaStream = new LemmaSampleStream(lineStream);
} catch (IOException e) {
  CmdLineUtil.handleCreateObjectStreamError(e);
}
```

The following step proceeds to train the model:

```
LemmatizerModel model;
try {
  LemmatizerFactory lemmatizerFactory = LemmatizerFactory.create(params.getFactory());
  model = LemmatizerME.train(params.getLang(), lemmaStream, mlParams, lemmatizerFactory);
} catch (IOException e) {
  throw new TerminateToolException(-1, "IO error while reading training data or indexing data: " + e.getMessage(), e);
} finally {
  try {
    sampleStream.close();
  } catch (IOException e) {
  }
}
```

Lemmatizer Evaluation

The built in evaluation can measure the accuracy of the statistical lemmatizer. The accuracy can be measured on a test data set.

There is a command line tool to evaluate a given model on a test data set. The following command shows how the tool can be run:

```
$ opennlp LemmatizerEvaluator -model en-custom-lemmatizer.bin -data en-custom-lemmatizer.test -encoding utf-8
```

This will display the resulting accuracy score, e.g.:

```
Loading model ... done
Evaluating ... done

Accuracy: 0.9659110277825124
```

Chapter 13. Stemmer

Introduction

A stemmer reduces a word form to a shorter root. OpenNLP ships rule-based engines such as Snowball and Porter through the `opennlp.tools.stemmer.Stemmer` interface. Stemming is language-specific and lossy: it is useful for search and feature reduction, not as a substitute for lemmatization when a dictionary form is required.

StemmerFactory

`StemmerFactory` captures a stemmer configuration once and mints configured `Stemmer` instances on demand. It is a plain supplier of stemmers, not a `BaseToolFactory` loaded from a model manifest. The English Snowball factory needs no external data:

```
StemmerFactory factory = new SnowballStemmerFactory(SnowballStemmer.ALGORITHM.ENGLISH);
Stemmer stemmer = factory.newStemmer();

stemmer.stem("running");       // "run"
stemmer.stemAll("running");    // ["run"]
```

Sharing and caching

Some stemmer implementations keep mutable engine state and must not be called from several threads at once. `SharingStemmer` wraps a `StemmerFactory` and routes each call to a per-thread delegate, so one instance can be shared safely. `CachingStemmer` adds a bounded per-thread LRU cache of word-to-stem mappings on top of the same factory pattern. Both wrappers return the same stem as a fresh factory stemmer for the same word:

```
StemmerFactory factory = new SnowballStemmerFactory(SnowballStemmer.ALGORITHM.ENGLISH);

new SharingStemmer(factory).stem("running");   // "run"
new CachingStemmer(factory).stem("running");   // "run"
```

Thread-safe stemmers such as the shareable `SnowballStemmer` class do not need `SharingStemmer`. Long-running containers should call `clearThreadLocalState()` when a pooled thread no longer uses a sharing or caching stemmer.

Hunspell dictionaries

`opennlp.tools.stemmer.hunspell` loads a user-supplied `.aff` affix file and its `.dic` word list. OpenNLP bundles no dictionary data; dictionaries are downloaded separately, and each provides license information. Retain the dictionary's copyright notices and license text with downloaded files. OpenNLP's Apache License does not relicense those files. The dictionary is immutable and safe to share; `HunspellStemmerFactory` creates a fresh stemmer per call.

```
HunspellDictionary dictionary = HunspellDictionary.load(
    Path.of("dictionary.aff"), Path.of("dictionary.dic"));
Stemmer stemmer = new HunspellStemmerFactory(dictionary).newStemmer();

stemmer.stem("workers");   // "work"
stemmer.stem("table");     // "table" (unknown vocabulary is unchanged)
```

The stems above assume a dictionary that lists `work` with an agentive and a plural suffix. Which stem a published dictionary yields for a given form is decided by that dictionary. An opt-in catalog download accepts an application-supplied `DictionaryCatalog`, needs `-Dopennlp.download.remote=true`, and fetches through the digest-verified `ResourceInstaller` path. A file that already exists in the target is not replaced. Remove old files before refreshing a dictionary. Loading uses `HunspellDictionary.LoadMode.STRICT` by default. Unsupported affix directives cause an `IOException` with the directive name and source line. File-based loading also includes the path. Unknown directive names are rejected. Metadata and suggestion settings that do not affect stemming or analysis are ignored. Applications can select `ALLOW_PARTIAL` to use the supported rules and inspect skipped directives:

```
HunspellDictionary partial = HunspellDictionary.load(
    Path.of("dictionary.aff"), Path.of("dictionary.dic"),
    HunspellDictionary.LoadMode.ALLOW_PARTIAL);
for (HunspellDictionary.UnsupportedDirective diagnostic : partial.getUnsupportedDirectives()) {
  System.err.println(diagnostic.directive() + " at "
      + diagnostic.source() + ":" + diagnostic.lineNumber());
}
```

The diagnostic list is immutable and contains the first source location for each unsupported directive, in file order. Partial loading does not apply behavior from skipped directives. Strict affix loading does not establish complete Hunspell compatibility. `ICONV` and `OCONV` convert input and output; `IGNORE` removes configured characters. `KEEPCASE`, `CHECKSHARPS`, and language-specific lowercase mappings govern case variants. `COMPLEXPREFIXES` permits 2 prefixes and 1 suffix instead of 1 prefix and 2 suffixes. Compound rules, boundary checks, simplified junctions, and recursive `BREAK` separators return recognized part stems in order, without duplicates. Morphological `st:` fields supply explicit stems, `ds:` suffixes make the derived form the stem, and `sp:` fields prepend surface material. `AM` expands morphology aliases. Rules that add and strip no material are undone like any other rule. The first listed homonym decides whether a spelling is forbidden, and a forbidden direct or affixed reading also blocks compound and `BREAK` readings. `SYLLABLENUM` applies Hungarian compound syllable adjustments. The deprecated `LEMMA_PRESENT` directive is accepted without changing results. OpenNLP does not generate inflected forms or spelling suggestions, and the morphological fields of an analysis are not part of the public API; the stemmer returns stems.

Native Hunspell's spelling, stemming, and analysis operations can return different results from one another. OpenNLP has one engine, so it follows the spell checker for recognition and the analyzer for stems and fields wherever the two agree, and resolves each disagreement the same way for every caller. The results deviate from the native library in the following cases.

- Compound stems are the stems of the recognized parts, in order. The native library concatenates the parts and stems the last one, so `riverboats` gives `river` and `boat` here and `riverboat` natively. The native form is the input with its last part replaced by that part's stem.

- Purely numeric tokens such as `1.5` or `42-42` are accepted by the native spell checker before any dictionary lookup. Here they are unknown input and are returned unchanged. Recognize numbers in the tokenizer or before stemming.

- A word starting with a dotted capital I is lowercased with Unicode mapping outside the Turkic languages, so `#nvite` is recognized when `invite` is listed. The native checker never tries the lowercase form of such a word without a Turkic `LANG`. Declare `LANG tr`, `az`, or `crh` for Turkic data, where both implementations keep the dotted and dotless letters apart.

- A compound of three or more parts that is one edit away from a listed word, such as `sunsetlamp` next to `bunsetlamp`, is rejected natively by running the spelling suggester, which OpenNLP does not provide. Such compounds are recognized here. An application that needs the rejection can validate multi-part compounds with a spelling checker, or limit `COMPOUNDWORDMAX` in the affix file where the language allows.

- Where the native spell checker and analyzer disagree, recognition follows the spell checker and output follows the analyzer. This affects all-uppercase Turkish words such as `#PEK`, which the native checker rejects while its analyzer stems them; mixed-case words with a capitalized initial such as `RaMarket` and `BREAK` forms, which the native analyzer does not handle; and forbidden affixed compounds, which the native analyzer stems. In each case OpenNLP recognizes the word when the native checker does and returns the analyzer's stems.

- An affix file without `SET` is read as UTF-8 and rejected when it is not valid UTF-8; the native default is ISO-8859-1. Add `SET ISO8859-1` to such a file.

- Malformed declarations are rejected where the native parser guesses: a multi-character flag under the default flag mode, an `AF` or `AM` table whose entry count does not match its header, and a numeric flag outside 1 through 65535. Correct the declaration; `ALLOW_PARTIAL` skips unsupported directives, not malformed ones.

`FLAG` and `AF` declarations apply to the complete affix file, including rules listed before those declarations. Parsing rejects malformed text in parsed rules, invalid counts, numeric flags outside the range 1 through 65535 and out-of-range `COMPOUNDMIN` values. Compound length and boundary checks count Unicode code points. A rule that strips a whole stem applies only when the affix file declares `FULLSTRIP`, as in Hunspell itself. Each affix or dictionary stream is rejected when it exceeds `HunspellDictionary.MAX_STREAM_BYTES` (64 MiB). Compound and word-break search is bounded, so an input with very many candidate analyses may not yield all of them.

Chapter 14. Document Annotation Container

Introduction

The task APIs of OpenNLP each return one kind of result: a tokenizer returns spans, a tagger returns tags, a name finder returns names. An application that runs several of them over one text has to keep those results aligned itself, and structured results such as coreference chains or dependency arcs, which refer to other results, have no place to live at all. The document container gives them one. It adds a small API next to the task APIs and replaces none of them: a task that needs only one component keeps calling it directly.

The package `opennlp.tools.document` provides an immutable container that carries the original text of one document together with any number of annotation layers over it. A layer is a list of `Annotation` values, each pairing a `Span` with a value. A span's offsets count Java chars (UTF-16 units, so a supplementary-plane character counts as two) from the beginning of the text exactly as the caller supplied it, never a normalized or derived form, so every annotation can be highlighted in the source text. The value's Java type is part of the layer's identity: a token layer reads back as `List<Annotation<String>>`, for example the token `The` paired with the span `[0..3)`.

Example 14.1. A document with three annotation layers

```
text:     The dog barks. It naps.
          0        1         2
          01234567890123456789012

sentences: [0..14) "The dog barks."   [15..23) "It naps."
tokens:    [0..3) "The" [4..7) "dog" [8..14) "barks." [15..17) "It" [18..23) "naps."
pos-tags:  [0..3) "DT"  [4..7) "NN"  [8..14) "VBZ"    [15..17) "PRP" [18..23) "VBZ"
```

Each layer in the figure above is identified by a `LayerKey` that names the layer and declares the type of its values: `Layers.TOKENS` is a `LayerKey<String>`, so `document.get(Layers.TOKENS)` returns the token annotations with string values. Two keys are equal when their id, their value type, and their scope are equal, so independently created constants for the same layer interoperate. The keys of the core linguistic layers live in `Layers`; a capability-specific layer's key lives on the annotator that provides it, and adding a capability never changes the container.

Key ids are namespaced. Keys OpenNLP defines start with the `opennlp:` prefix; a user extension must use its own prefix. A bare id, like the `token-lengths` key below, is legal for an application-local layer. Toolkit keys are created through `Layers.key` and `Layers.documentKey`, which apply the prefix.

A document is never modified in place: adding a layer returns a new document that shares the unchanged layers with its ancestor. Documents created through `Document.of` capture their text at construction and are safe to share between threads. Annotations reference other annotations by index within their layer, never by object identity. Three invariants make those index references sound: a layer keeps its insertion order, a layer is immutable once added and detached from the caller's input list, and providing a layer that already exists is rejected, so an index reference stays valid for the lifetime of the document.

A key declares its scope. Keys are positional by default: every annotation carries a span, and consumers never null-check it. A document-scoped key carries whole-document values without spans, which is where a language id, a category distribution, or provenance belongs:

```
LayerKey<String> LANGUAGE = LayerKey.document("app:language", String.class);

Document tagged = document.with(LANGUAGE, List.of(Annotation.of("eng")));
String language = tagged.get(LANGUAGE).get(0).value(); // "eng", span is null
```

A corpus may carry a hand-annotated version of a layer beside a produced one. The convention is a `gold:` id prefix on the same key scheme, for example `gold:opennlp:tokens` beside `opennlp:tokens`; since adding a layer is once-only, competing versions always live under distinct keys and never replace each other.

What the layers contain

A layer has one value type and one annotation per unit, anchored on the span of that unit in the original text. The toolkit's own adapters provide these layers:

Table 14.1. Layers provided by the toolkit's adapters

Key	Value	One annotation per	Provided by
`opennlp:sentences`	`String`, the sentence text	sentence	`SentenceDetectorAnnotator`
`opennlp:tokens`	`String`, the token	token	`TokenizerAnnotator`
`opennlp:pos`	`String`, the tag	token, on the token's span	`POSTaggerAnnotator`
`opennlp:lemmas`	`String`, the lemma	token, on the token's span	`LemmatizerAnnotator`
`opennlp:stems`	`String`, the stem	token, on the token's span	`StemmerAnnotator`
`opennlp:entities`	`String`, the entity type	name, on the name's span	`NameFinderAnnotator`
`opennlp:chunks`	`String`, the chunk type	chunk	`ChunkerAnnotator`
`opennlp:phrases`	`Phrase`: the label and the head span	constituent	`ParserAnnotator`
`opennlp:termvectors`	`TermVector`: the term, its frequency and, except in `SCORING_ONLY` mode, its occurrence spans	distinct term; document-scoped, without a span	`TermVectorAnnotator`

The keys of the sentences, tokens, pos and entities layers live in `Layers` (`SENTENCES`, `TOKENS`, `POS_TAGS` and `ENTITIES`); the others live on the annotator that provides them, which is also where an extension puts its key. Layers that refer to other layers do so by index into them, which the invariants above keep valid: a coreference layer, for example, puts one annotation on each mention's span with the chain it belongs to, and a dependency layer puts one annotation on each token's span naming the head token by its index in `opennlp:tokens`. Adding such a layer requires no change in this package; the section below shows how it is done.

Building a pipeline

A pipeline step implements `DocumentAnnotator`: it reads layers from a document and returns a new document with its own layers added. An annotator declares the layers it requires and provides, and a `DocumentAnalyzer` validates those declarations when the pipeline is assembled: every required layer must be provided by an earlier annotator, and no two annotators may provide the same layer, so a misordered pipeline fails at build time rather than midway through a document.

Adapters for the toolkit's own components are provided: `SentenceDetectorAnnotator`, `TokenizerAnnotator`, `POSTaggerAnnotator`, `NameFinderAnnotator`, `ChunkerAnnotator`, `ParserAnno-`

`tator`, `LemmatizerAnnotator`, and `StemmerAnnotator`. Each wraps one of the existing component APIs; an application that needs only one task can keep using that component's API directly. The following pipeline combines three adapters with one custom annotator and analyzes the text from the introduction's figure; the components behind the adapters are any `SentenceDetector`, `Tokenizer`, and `POSTagger`, for example the ME implementations loaded from models:

```
DocumentAnalyzer analyzer = DocumentAnalyzer.builder()
    .add(new SentenceDetectorAnnotator(sentenceDetector))
    .add(new TokenizerAnnotator(tokenizer))
    .add(new POSTaggerAnnotator(tagger))
    .add(new TokenLengthAnnotator())
    .build();

Document document = analyzer.analyze("The dog barks. It naps.");
```

The resulting document carries exactly the four layers the pipeline provides: the figure's three plus the custom token-lengths layer. The sentence layer holds two annotations, `[0..14)` covering `The dog barks.` and `[15..23)` covering `It naps.`. The token layer holds five tokens whose spans refer to the document text even inside the second sentence, so `It` is `[15..17)` and `naps.` is `[18..23)`. Layers produced per token stay aligned with the token layer by position:

```
List<Annotation<String>> tokens = document.get(Layers.TOKENS);
List<Annotation<String>> tags = document.get(Layers.POS_TAGS);
for (int i = 0; i < tags.size(); i++) {
  // each tag sits on its token's span, e.g. "DT" on [0..3) for "The"
  Span span = tags.get(i).span();
}

// every span refers to the original text, so covered text round-trips
for (Annotation<String> token : tokens) {
  CharSequence covered = token.span().getCoveredText(document.text());
}
```

Because a document is immutable, independent pipelines can process the same text in parallel, each starting from its own `Document.of(text)`, and their results can be joined afterwards: `merge` returns a new document carrying the layers of both. The texts must match, and by default a layer key present on both documents is rejected, so two branches that both run a tokenizer collide on `opennlp:tokens`. When the branches rebuild a shared prefix identically, passing `DuplicateLayerPolicy.KEEP_EQUAL` keeps one copy of each agreeing layer; copies whose contents differ are still rejected.

Writing a custom annotator

The annotator below defines its own key in its own code, reads the token layer, and provides one integer annotation per token; the analyzer's build-time validation guarantees a tokenizer ran earlier:

```
static final LayerKey<Integer> TOKEN_LENGTHS =
    LayerKey.of("token-lengths", Integer.class);

class TokenLengthAnnotator implements DocumentAnnotator {

  @Override
  public Document annotate(Document document) {
    DocumentAnnotators.requireLayers(document, Layers.TOKENS);
    List<Annotation<String>> tokens = document.get(Layers.TOKENS);
    List<Annotation<Integer>> lengths = new ArrayList<>(tokens.size());
    for (Annotation<String> token : tokens) {
      lengths.add(new Annotation<>(token.span(), token.value().length()));
    }
    return document.with(TOKEN_LENGTHS, lengths);
  }

  @Override
  public Set<LayerKey<?>> requires() {
    return Set.of(Layers.TOKENS);
  }

  @Override
  public Set<LayerKey<?>> provides() {
    return Set.of(TOKEN_LENGTHS);
  }
}
```

Because the key declares the value type, the values come back as numbers without a cast; for the text above the five values are `3, 3, 6, 2, 5`, each on its token's span:

```
List<Annotation<Integer>> lengths = document.get(TOKEN_LENGTHS);
int firstTokenLength = lengths.get(0).value(); // 3, for "The" at [0..3)
```

Required layers must be present, but they may be empty: an empty token layer yields the annotator's provided layers present but empty, so a pipeline degrades gracefully on documents without content. An absent required layer is rejected with an `IllegalArgumentException` naming the layer, because a missing pipeline stage is an assembly error, not an empty document. `DocumentAnnotators.requireLayers` performs exactly that rejection, including the null check on the document, and is what the toolkit's own adapters use.

Term vectors

`TermVectorAnnotator` rolls the token layer up into a document-scoped term vector layer for index consumers: one `TermVector` per distinct term, carrying the term string, its occurrence count, and the occurrence spans in original text coordinates. Term identity comes from the annotator's inputs: without a normalizer the term is the token's covered text as-is; with a plain `CharSequenceNormalizer`, the general path, each token's covered text is normalized on its own to become the term; with an `OffsetAwareNormalizer` the document text is normalized once and each token's covered normalized text becomes the term. On every path tokens differing only by a normalization fold group together while their spans keep pointing into the original text. A token whose normalized form is empty is omitted from the layer. The layer preserves first-occurrence order.

```
DocumentAnalyzer analyzer = DocumentAnalyzer.builder()
   .add(new TokenizerAnnotator(tokenizer))
   .add(new TermVectorAnnotator())
   .build();

Document document = analyzer.analyze("The dog barks. The dog naps.");
List<Annotation<TermVector>> vectors = document.get(TermVectorAnnotator.TERM_VECTORS);
// vectors.get(1).value() is ("dog", 2, [[4..7), [19..22)])
```

The normalized-term workflow passes any shipped `CharSequenceNormalizer` to the plain-normalizer constructor: the normalizer is applied to each token's covered text to produce the term, so the case, NFC, and accent folds that `buildAligned()` rejects all work here, and every occurrence span stays the token's own span in the original text.

```
CharSequenceNormalizer folder = TextNormalizer.builder().caseFold().build();
DocumentAnalyzer analyzer = DocumentAnalyzer.builder()
   .add(new TokenizerAnnotator(WhitespaceTokenizer.INSTANCE))
   .add(new TermVectorAnnotator(folder))
   .build();

Document document = analyzer.analyze("Word word WORD");
List<Annotation<TermVector>> vectors = document.get(TermVectorAnnotator.TERM_VECTORS);
// vectors.get(0).value() is ("word", 3, [[0..4), [5..9), [10..14)])
```

Scoring-only consumers that never read offsets can skip storing them: `new TermVectorAnnotator(TermVectorAnnotator.Mode.SCORING_ONLY)` emits vectors that carry counts but no spans. `TermVector.count(term, frequency)` creates such a span-free vector; `TermVector.withSpans(term, spans)` creates a full one.

Chapter 15. SpellChecker

Overview

The OpenNLP SpellChecker is a native, language-agnostic spelling corrector built on the *SymSpell* (Symmetric Delete) algorithm. Given a (possibly misspelled) term it proposes corrections drawn from a frequency dictionary, ranked by edit distance and corpus frequency. It can also correct whole phrases, repairing wrongly inserted or omitted spaces (word splits and merges) with the help of a bigram language model.

SymSpell is fast because it pre-computes a deletes-only index: for every dictionary term it derives all strings obtained by deleting up to a small number of symbols and maps each such delete back to the originating terms. A query is answered by generating the deletes of the query and intersecting them with the index, which turns the costly fuzzy search into a handful of hash-map lookups. Candidates are then verified with an edit-distance metric. Because only deletes are generated (no insertions, substitutions or transpositions), candidate generation is orders of magnitude cheaper than a naïve search over the whole dictionary.

The SpellChecker ships as the optional `opennlp-spellcheck` extension module (package root `opennlp.spellcheck`). It is a good fit when you need to:

- clean up noisy user input (search queries, form fields, OCR output) before tokenization or classification;
- normalize a training or evaluation corpus as part of a preprocessing pipeline;
- offer interactive "did you mean" suggestions for a single term.

It is *not* a grammar checker and it does not require a trained statistical model – correction quality is determined entirely by the supplied frequency dictionary.

> **Note**
>
> The algorithm and its compound-correction heuristic are an independent Java rewrite of the SymSpell reference implementation by Wolf Garbe (MIT). Attribution is recorded in the project NOTICE file.

SpellChecker Tools

The extension provides two command line tools, dispatched through the `bin/spellcheck` launcher of the binary distribution: `SpellCheckModelBuilder` builds a binary model from plain-text frequency dictionaries, and `CorrectText` corrects text using such a model. Each tool prints its usage when invoked with the `help` parameter.

Building a Model

A model is built from a unigram frequency list and, optionally, a bigram list (see the section called “Dictionary and Model Format” for the file format). Usage of the tool:

```
$ bin/spellcheck SpellCheckModelBuilder help
Usage: opennlp-spellcheck SpellCheckModelBuilder -unigrams in [-bigrams in] -model out \
  [-maxEditDistance num] [-countThreshold num] [-distance damerau-osa|levenshtein] \
  [-corpusWordCount num] -lang lang [-prefixLength num] [-encoding charsetName]

Arguments description:
 -lang lang
  The language tag of the model (e.g. en), stored as model.language.
 -unigrams in
  The unigram frequency dictionary file (word<TAB>count per line).
 -bigrams in
  An optional bigram frequency dictionary file (w1 w2<TAB>count per line).
 -model out
  The output file the binary SymSpell model is written to.
 -maxEditDistance num
  The largest precomputed dictionary edit distance. (default 2)
 -prefixLength num
  Number of leading symbols used for delete generation (must be > maxEditDistance). (default 7)
 -countThreshold num
  Minimum corpus count for a term to be indexed. (default 1)
 -distance damerau-osa|levenshtein
  The edit-distance metric used for verification. (default damerau-osa)
 -corpusWordCount num
  Corpus size N used for compound-correction scoring; 0 (default) derives N from the
  dictionary's summed counts.
 -encoding charsetName
  encoding for reading and writing text, if absent the system default is used.
```

The following command builds an English model from a unigram and a bigram dictionary, using the default Damerau-OSA edit distance:

```
$ bin/spellcheck SpellCheckModelBuilder -lang en \
  -unigrams en-unigrams.txt -bigrams en-bigrams.txt \
  -model en-spellcheck.bin -encoding UTF-8
```

Correcting Text

Once a model has been built (or a packaged model is on the classpath), the `CorrectText` tool corrects text line by line. Usage of the tool:

```
$ bin/spellcheck CorrectText help
Usage: opennlp-spellcheck CorrectText [-model model] [-maxEditDistance num] [-lang lang] \
  [-suggest true|false] [-verbosity TOP|CLOSEST|ALL] [-compound true|false] \
  [-inputFile in] [-outputFile out] [-encoding charsetName]

Arguments description:
 -model model
  A binary SymSpell model file. Either -model or -lang must be given.
 -lang lang
  Resolve a packaged spellcheck model for this language from the classpath.
  Either -model or -lang must be given.
 -inputFile in
  The input text file (read line by line). If absent, standard input is used.
 -outputFile out
  The output text file. If absent, standard output is used.
 -compound true|false
  Correct each line as a whole phrase, repairing wrong word splits/merges. (default false)
 -maxEditDistance num
  The maximum edit distance considered per token. (default 2)
 -suggest true|false
  List the suggestions for each token (honoring -verbosity) instead of emitting
  corrected text. (default false)
 -verbosity TOP|CLOSEST|ALL
  How many suggestions to list per token; only used with -suggest true.
  TOP=best only, CLOSEST=all at the smallest distance, ALL=all within -maxEditDistance.
  (default TOP)
 -encoding charsetName
  encoding for reading and writing text, if absent the system default is used.
```

By default each line is corrected token by token, reading from standard input and writing to standard output:

```
$ echo "the qick brwn fox" | bin/spellcheck CorrectText -model en-spellcheck.bin
the quick brown fox
```

With `-compound true` the whole line is corrected as a phrase, additionally repairing wrongly inserted or omitted spaces:

```
$ echo "thequick brownfox" | bin/spellcheck CorrectText -model en-spellcheck.bin -compound true
the quick brown fox
```

With `-suggest true` the tool lists the candidate suggestions for every token (honoring `-verbosity`) instead of emitting corrected text:

```
$ echo "teh qick" | bin/spellcheck CorrectText -model en-spellcheck.bin -suggest true -verbosity CLOSEST
teh => [the]
qick => [quick, quack]
```

Instead of an explicit `-model` file you can resolve a packaged model from the classpath by language with `-lang` (see the section called “Multi-language Models”):

```
$ bin/spellcheck CorrectText -lang en -inputFile noisy.txt -outputFile clean.txt -encoding UTF-8
```

SpellChecker API

The central abstraction is the `SpellChecker` interface; its default implementation is the `SymSpell` engine. The interface exposes two ways to correct input:

```
public interface SpellChecker {

  // Suggestions for a single term, bounded by maxEditDistance.
  List<SuggestItem> lookup(String term, Verbosity verbosity, int maxEditDistance);

  // Convenience overload: Verbosity.TOP and the configured max edit distance.
  List<SuggestItem> lookup(String term);

  // Corrects a whole phrase, repairing word splits and merges.
  List<SuggestItem> lookupCompound(String input, int maxEditDistance);
}
```

Each suggestion is an immutable `SuggestItem` record carrying the corrected term, the edit distance to the queried term and the corpus frequency of the suggestion. Its natural ordering sorts the best candidate first (ascending edit distance, then descending frequency), so `get(0)` is the top suggestion.

```
public record SuggestItem(String term, int editDistance, long frequency)
    implements Comparable<SuggestItem> { }
```

The `Verbosity` enum controls how many suggestions a `lookup` call returns:

- `TOP` – the single best suggestion (cheapest to compute);
- `CLOSEST` – all suggestions sharing the smallest edit distance found;
- `ALL` – every suggestion within the requested maximum edit distance.

An engine is created from a `SymSpellConfig` and populated through `add` (unigrams) and `addBigram` (bigrams). In practice the engine is built from a dictionary or loaded from a model (see the section called “Dictionary and Model Format”), but it can also be filled programmatically:

```
SymSpellConfig config = SymSpellConfig.builder()
    .maxDictionaryEditDistance(2)   // largest precomputed edit distance (default 2)
    .prefixLength(7)                // leading symbols used for deletes (default 7)
    .countThreshold(1)              // minimum count to index a term (default 1)
    .editDistance(DamerauOSADistance.INSTANCE) // verification metric (default)
    .build();

SymSpell symSpell = new SymSpell(config);   // or: new SymSpell() for the defaults
symSpell.add("quick", 1_000);
symSpell.add("brown", 800);
symSpell.addBigram("quick", "brown", 120);
```

Once populated, the engine is safe for concurrent `lookup` calls. Looking up a single term and reading the best suggestion:

```
List<SuggestItem> suggestions = symSpell.lookup("qick", Verbosity.CLOSEST, 2);
for (SuggestItem item : suggestions) {
  System.out.printf("%s (distance=%d, frequency=%d)%n",
     item.term(), item.editDistance(), item.frequency());
}
```

Correcting a whole phrase, including wrongly split or merged words, with `lookupCompound` (which always returns a singleton list holding the best correction):

```
String corrected = symSpell.lookupCompound("thequick brwn fox", 2).get(0).term();
// -> "the quick brown fox"
```

Choosing an Edit Distance

The verification metric is pluggable behind the `EditDistance` interface and selected through `SymSpellConfig`. Two implementations are provided:

- `DamerauOSADistance` (the *default*) – an Optimal String Alignment (restricted Damerau-Levenshtein) metric in which a transposition of two adjacent symbols counts as a single edit. It is Unicode-aware, operating on code points, and matches the variant used by the SymSpell reference implementation. Prefer it for natural-language text where transpositions ("teh" $\rightarrow$ "the") are common typos.
- `LevenshteinDistance` – a thin adapter over Apache Commons Text in which only insertions, deletions and substitutions count, so a transposition costs two edits. It operates on UTF-16 `char` units. Use it when you want plain Levenshtein semantics or to match another system's behavior.

Both expose a shared, stateless `INSTANCE` and are passed to the builder:

```
SymSpellConfig config = SymSpellConfig.builder()
   .editDistance(LevenshteinDistance.INSTANCE)
   .build();
```

Dictionary and Model Format

A SpellChecker is driven by plain-text frequency dictionaries, loaded into an engine by the `FrequencyDictionaryLoader`. Two UTF-8 (by default) line-based formats are supported. In both, columns are separated by TABs or spaces, so the space-delimited SymSpell reference dictionaries [https://github.com/wolfgarbe/SymSpell#dictionary-file-format] load as they are. Blank lines and lines starting with # (comments) are skipped.

The *unigram* dictionary lists one `word<TAB>count` per line, where `count` is the term's corpus frequency, a nonnegative integer:

```
the 23135851162
of 13151942776
and 12997637966
quick 1885979
brown 1019234
fox 271289
```

The optional *bigram* dictionary lists one `word1 word2<TAB>count` per line (the two words separated by a space, then a TAB and the count). Bigrams improve compound correction:

```
quick brown 12046
brown fox 5634
the quick 98765
```

A model is the built engine together with its source dictionaries and the metadata needed to identify it. The `SymSpellModels` facade builds, serializes and deserializes models, and writes the packaged form consumed by the model-resolver. The loaded model exposes a ready-to-query engine via `getSymSpell()`:

```
InputStreamFactory unigrams = new MarkableFileInputStreamFactory(new File("en-unigrams.txt"));
InputStreamFactory bigrams = new MarkableFileInputStreamFactory(new File("en-bigrams.txt"));

SymSpellModel model = SymSpellModels.buildModel(
    "en", SymSpellConfig.defaultConfig(), StandardCharsets.UTF_8, unigrams, bigrams);

// Persist the binary model.
try (OutputStream out = Files.newOutputStream(Path.of("en-spellcheck.bin"))) {
  SymSpellModels.serialize(model, out);
}

// Reload it later and query the engine.
try (InputStream in = Files.newInputStream(Path.of("en-spellcheck.bin"))) {
  SymSpellModel loaded = SymSpellModels.deserialize(in);
  SpellChecker checker = loaded.getSymSpell();
  List<SuggestItem> suggestions = checker.lookup("qick");
}
```

Loading just the dictionaries into an existing engine – for example to add domain terms – can be done directly with the loader:

```
SymSpell engine = new SymSpell(SymSpellConfig.defaultConfig());
FrequencyDictionaryLoader loader = new FrequencyDictionaryLoader(StandardCharsets.UTF_8);
loader.loadUnigrams(engine, unigrams);
loader.loadBigrams(engine, bigrams);
```

Multi-language Models

Production multi-language dictionaries are large and are therefore *not* bundled in the `opennlp-spellcheck` jar. Instead, each language ships as a separate Maven artifact named `opennlp-models-spellcheck-{lang}` (for example `opennlp-models-spellcheck-en`), packaged as a binary model plus a matching `model.properties` descriptor (`model.language`, `model.name`, `model.version`, `model.sha256`).

Add the model artifact for the language you need to your classpath:

```
<dependency>
   <groupId>org.apache.opennlp</groupId>
   <artifactId>opennlp-models-spellcheck-en</artifactId>
   <version>CURRENT_OPENNLP_VERSION</version>
</dependency>
```

The `SymSpellModelResolver` then discovers and loads the matching model using the standard OpenNLP model-resolver machinery (see Chapter 21, *Classpath Loading of OpenNLP Models*). It returns an `Optional` that is empty when no matching model jar is present:

```
Optional<SymSpellModel> model = new SymSpellModelResolver().resolveByLanguage("en");
SpellChecker checker = model.orElseThrow().getSymSpell();
List<SuggestItem> suggestions = checker.lookup("qick");
```

On the command line the same resolution happens when you pass `-lang` instead of `-model` to `CorrectText`.

Pipeline Integration

The SpellChecker integrates into OpenNLP preprocessing through the standard normalizer and stream abstractions, so correction can be slotted into an existing chain without bespoke glue code.

`SpellCheckingCharSequenceNormalizer` implements `opennlp.tools.util.normalizer.CharSequenceNormalizer` and can be composed with the other normalizers (URL, emoji, shrink, ...) inside an `AggregateCharSequenceNormalizer`. It works in one of two modes: `Mode.PER_TOKEN` (the default) corrects each whitespace-delimited token independently and preserves the line's whitespace and punctuation, while `Mode.COMPOUND` corrects the whole input as a phrase, repairing word splits and merges. It guards against over-correction (skipping short tokens, numbers and URL/email-like tokens, and never changing a word the dictionary already contains) and preserves the original casing pattern of each token.

```
SymSpellModel model = SymSpellModels.deserialize(in);

CharSequenceNormalizer normalizer = SpellCheckingCharSequenceNormalizer.builder(model)
   .mode(SpellCheckingCharSequenceNormalizer.Mode.PER_TOKEN)
   .maxEditDistance(2)
   .build();

CharSequence clean = normalizer.normalize("the qick brwn fox"); // -> "the quick brown fox"
```

In per-token mode, letter and capitalization checks include supplementary Unicode code points. A lowercase letter prevents all-uppercase correction. An uppercase token of one code point uses initial-capital correction. Known dictionary words keep their original casing. This example uses U+10428, a lowercase Deseret letter:

```
SymSpell engine = new SymSpell();
engine.add("\uD801\uDC28word", 1000);
CharSequenceNormalizer normalizer = new SpellCheckingCharSequenceNormalizer(engine);

String corrected = normalizer.normalize("\uD801\uDC28WROLD").toString();
// corrected is "\uD801\uDC28word"
```

For stream-based preprocessing two `FilterObjectStream<String, String>` implementations wrap an existing `ObjectStream<String>` and correct each element as it is read:

- `SpellCorrectingObjectStream` corrects each line; in compound mode it may split or merge tokens. Use it for free text.
- `SpellCorrectingTokenStream` corrects tokenized data token-by-token and re-joins with the same delimiter. It is *token-count preserving*, so the corrected element stays aligned with any parallel annotation (tags, spans).

Dropping the corrector into a line-based reading chain:

```
ObjectStream<String> lines =
    new PlainTextByLineStream(inputStreamFactory, StandardCharsets.UTF_8);
ObjectStream<String> corrected = new SpellCorrectingObjectStream(lines, model);

String line;
while ((line = corrected.read()) != null) {
  // ... consume corrected lines ...
}
```

Chapter 16. Chunker

Chunking

Text chunking consists of dividing a text in syntactically correlated parts of words, like noun groups, verb groups, but does not specify their internal structure, nor their role in the main sentence.

Chunker Tool

The easiest way to try out the Chunker is the command line tool. The tool is only intended for demonstration and testing.

Download the English maxent chunker model from the website and start the Chunker Tool with this command:

```
$ opennlp ChunkerME en-chunker.bin
```

The Chunker now reads a pos tagged sentence per line from stdin. Copy these two sentences to the console:

```
Rockwell_NNP International_NNP Corp._NNP 's_POS Tulsa_NNP unit_NN said_VBD it_PRP signed_VBD
   a_DT tentative_JJ agreement_NN extending_VBG its_PRP$ contract_NN with_IN Boeing_NNP Co._NNP
   to_TO provide_VB structural_JJ parts_NNS for_IN Boeing_NNP 's_POS 747_CD jetliners_NNS ._.
Rockwell_NNP said_VBD the_DT agreement_NN calls_VBZ for_IN it_PRP to_TO supply_VB 200_CD
   additional_JJ so-called_JJ shipsets_NNS for_IN the_DT planes_NNS ._.
```

The Chunker will now echo the sentences grouped tokens to the console:

```
[NP Rockwell_NNP International_NNP Corp._NNP ] [NP 's_POS Tulsa_NNP unit_NN ] [VP said_VBD ]
   [NP it_PRP ] [VP signed_VBD ] [NP a_DT tentative_JJ agreement_NN ] [VP extending_VBG ]
   [NP its_PRP$ contract_NN ] [PP with_IN ] [NP Boeing_NNP Co._NNP ] [VP to_TO provide_VB ]
   [NP structural_JJ parts_NNS ] [PP for_IN ] [NP Boeing_NNP ] [NP 's_POS 747_CD jetliners_NNS ] ._.
[NP Rockwell_NNP ] [VP said_VBD ] [NP the_DT agreement_NN ] [VP calls_VBZ ] [SBAR for_IN ]
   [NP it_PRP ] [VP to_TO supply_VB ] [NP 200_CD additional_JJ so-called_JJ shipsets_NNS ]
   [PP for_IN ] [NP the_DT planes_NNS ] ._.
```

The example uses Penn Treebank part-of-speech tags [https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html] and CoNLL-2000 IOB chunk tags [https://aclanthology.org/W00-0726/].

Chunking API

The Chunker can be embedded into an application via its API. First the chunker model must be loaded into memory from disk or another source. In the sample below it is loaded from disk.

```
InputStream modelIn = null;
ChunkerModel model = null;

try (modelIn = new FileInputStream("en-chunker.bin")){
  model = new ChunkerModel(modelIn);
}
```

After the model is loaded a Chunker can be instantiated.

```
ChunkerME chunker = new ChunkerME(model);
```

The Chunker instance is now ready to tag data. It expects a tokenized sentence as input, which is represented as a String array, each String object in the array is one token, and the POS tags associated with each token. As of OpenNLP 3.0.0 ChunkerME is thread safe and a single instance can be shared across threads; the legacy ThreadSafeChunkerME wrapper is retained for backward compatibility and is now deprecated.

The following code shows how to determine the most likely chunk tag sequence for a sentence.

```
String[] sent = new String[] { "Rockwell", "International", "Corp.", "'s",
    "Tulsa", "unit", "said", "it", "signed", "a", "tentative", "agreement",
    "extending", "its", "contract", "with", "Boeing", "Co.", "to",
    "provide", "structural", "parts", "for", "Boeing", "'s", "747",
    "jetliners", "." };

String[] pos = new String[] { "NNP", "NNP", "NNP", "POS", "NNP", "NN",
    "VBD", "PRP", "VBD", "DT", "JJ", "NN", "VBG", "PRP$", "NN", "IN",
    "NNP", "NNP", "TO", "VB", "JJ", "NNS", "IN", "NNP", "POS", "CD", "NNS",
    "." };

String[] tag = chunker.chunk(sent, pos);
```

The tags array contains one chunk tag for each token in the input array. The corresponding tag can be found at the same index as the token has in the input array. The confidence scores for the returned tags can be easily retrieved from a ChunkerME with the following method call:

```
double[] probs = chunker.probs();
```

The call to probs is stateful and will always return the probabilities of the last tagged sentence. When the chunker is shared across threads, the "last tagged sentence" is tracked per thread, so each thread must call chunk before calling probs on that same thread. The probs method should only be called when the chunk method was called before on the current thread, otherwise the behavior is undefined.

Some applications need to retrieve the n-best chunk tag sequences and not only the best sequence. The topKSequences method is capable of returning the top sequences. It can be called in a similar way as chunk.

```
Sequence[] topSequences = chunk.topKSequences(sent, pos);
```

Each Sequence object contains one sequence. The sequence can be retrieved via Sequence.getOutcomes() which returns a tags array and Sequence.getProbs() returns the probability array for this sequence.

Chunker Training

The pre-trained models might not be available for a desired language, can not detect important entities or the performance is not good enough outside the news domain.

These are the typical reason to do custom training of the chunker on a new corpus or on a corpus which is extended by private training data taken from the data which should be analyzed.

The training data can be converted to the OpenNLP chunker training format, which is based on CoNLL2000 [https://www.cnts.ua.ac.be/conll2000/chunking]. Other formats may also be available. The training data consist

of three columns separated one single space. Each word has been put on a separate line and there is an empty line after each sentence. The first column contains the current word, the second its part-of-speech tag and the third its chunk tag. The chunk tags contain the name of the chunk type, for example I-NP for noun phrase words and I-VP for verb phrase words. Most chunk types have two types of chunk tags, B-CHUNK for the first word of the chunk and I-CHUNK for each other word in the chunk. Here is an example of the file format:

Sample sentence of the training data:

```
He        PRP  B-NP
reckons   VBZ  B-VP
the       DT   B-NP
current   JJ   I-NP
account   NN   I-NP
deficit   NN   I-NP
will      MD   B-VP
narrow    VB   I-VP
to        TO   B-PP
only      RB   B-NP
#         #    I-NP
1.8       CD   I-NP
billion   CD   I-NP
in        IN   B-PP
September NNP  B-NP
.         .    O
```

Note

For improved visualization the example above uses tabs instead of a single space as column separator.

Training Tool

OpenNLP has a command line tool which is used to train the models available from the model download page on various corpora.

Usage of the tool:

```
$ opennlp ChunkerTrainerME
Usage: opennlp ChunkerTrainerME[.ad] [-params paramsFile] [-iterations num] [-cutoff num] \
            -model modelFile -lang language -data sampleData [-encoding charsetName]

Arguments description:
      -params paramsFile
            training parameters file.
      -iterations num
            number of training iterations, ignored if -params is used.
      -cutoff num
            minimal number of times a feature must be seen, ignored if -params is used.
      -model modelFile
            output model file.
      -lang language
            language which is being processed.
      -data sampleData
            data to be used, usually a file name.
      -encoding charsetName
            encoding for reading and writing text, if absent the system default is used.
```

Its now assumed that the English chunker model should be trained from a file called en-chunker.train which is encoded as UTF-8. The following command will train the name finder and write the model to en-chunker.bin:

```
$ opennlp ChunkerTrainerME -model en-chunker.bin -lang en -data en-chunker.train -encoding UTF-8
```

Additionally its possible to specify the number of iterations, the cutoff and to overwrite all types in the training data with a single type.

Training API

The Chunker offers an API to train a new chunker model. The following sample code illustrates how to do it:

```
ObjectStream<String> lineStream =
    new PlainTextByLineStream(new MarkableFileInputStreamFactory(new File("en-chunker.train")),
 StandardCharsets.UTF_8);

ChunkerModel model;
try(ObjectStream<ChunkSample> sampleStream = new ChunkSampleStream(lineStream)) {
   model = ChunkerME.train("eng", sampleStream, TrainingParameters.defaultParams(), new ChunkerFactory());
}

try (OutputStream modelOut = new BufferedOutputStream(new FileOutputStream(modelFile))) {
   model.serialize(modelOut);
}
```

Chunker Evaluation

The built-in evaluation can measure the chunker performance. The performance is either measured on a test dataset or via cross validation.

Chunker Evaluation Tool

The following command shows how the tool can be run:

```
$ opennlp ChunkerEvaluator
Usage: opennlp ChunkerEvaluator[.ad] -model model [-misclassified true|false] \
            [-detailedF true|false] -lang language -data sampleData [-encoding charsetName]
```

A sample of the command considering you have a data sample named en-chunker.eval and you trained a model called en-chunker.bin:

```
$ opennlp ChunkerEvaluator -model en-chunker.bin -data en-chunker.eval -encoding UTF-8
```

and here is a sample output:

```
Precision: 0.9255923572240226
Recall: 0.9220610430991112
F-Measure: 0.9238233255623465
```

You can also use the tool to perform 10-fold cross validation of the Chunker. The following command shows how the tool can be run:

```
$ opennlp ChunkerCrossValidator
Usage: opennlp ChunkerCrossValidator[.ad] [-params paramsFile] [-iterations num] [-cutoff num] \
            [-misclassified true|false] [-folds num] [-detailedF true|false] \
            -lang language -data sampleData [-encoding charsetName]

Arguments description:
      -params paramsFile
            training parameters file.
      -iterations num
            number of training iterations, ignored if -params is used.
      -cutoff num
            minimal number of times a feature must be seen, ignored if -params is used.
      -misclassified true|false
            if true will print false negatives and false positives.
      -folds num
            number of folds, default is 10.
      -detailedF true|false
            if true will print detailed FMeasure results.
      -lang language
            language which is being processed.
      -data sampleData
            data to be used, usually a file name.
      -encoding charsetName
            encoding for reading and writing text, if absent the system default is used.
```

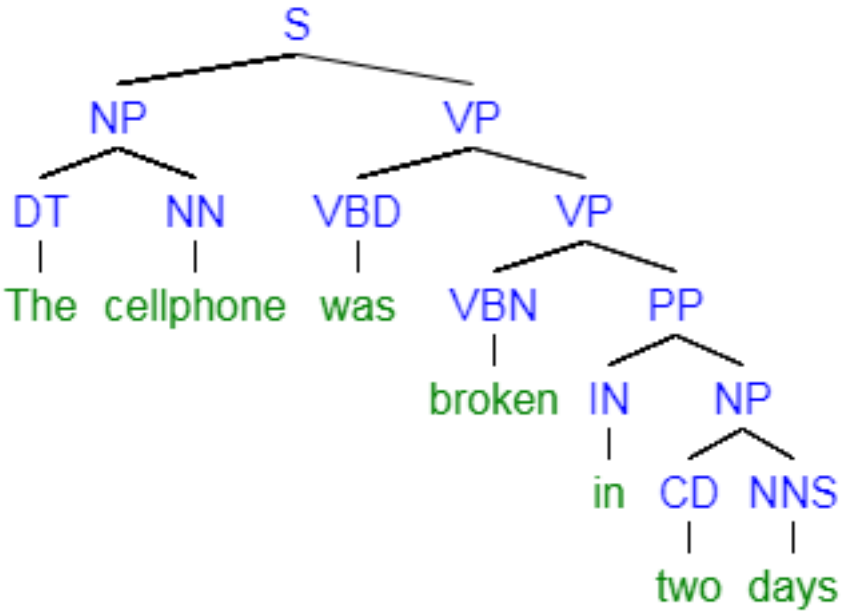
It is not necessary to pass a model. The tool will automatically split the data to train and evaluate:

```
$ opennlp ChunkerCrossValidator -lang pt -data en-chunker.cross -encoding UTF-8
```

Chapter 17. Parser

Parsing

A parser returns a parse tree from a sentence according to a phrase structure grammar. A parse tree specifies the internal structure of a sentence. For example, the following image represents a parse tree for the sentence 'The cellphone was broken in two days':



A parse tree can be used to determine the role of subtrees or constituents in the sentence. For example, it is possible to know that 'The cellphone' is the subject of the sentence and the verb (action) is 'was broken.'

Parser Tool

The easiest way to try out the Parser is the command line tool. The tool is only intended for demonstration and testing. Download the English chunking parser model from the website and start the Parse Tool with the following command.

```
$ opennlp Parser en-parser-chunking.bin
```

Loading the big parser model can take several seconds, be patient. Copy this sample sentence to the console.

```
The cellphone was broken in two days .
```

The parser should now print the following to the console.

```
(TOP (S (NP (DT The) (NN cellphone)) (VP (VBD was) (VP (VBN broken) (PP (IN in) (NP (CD two) (NNS days))))) (. .)))
```

With the following command the input can be read from a file and be written to an output file.

```
$ opennlp Parser en-parser-chunking.bin < article-tokenized.txt > article-parsed.txt.
```

The article-tokenized.txt file must contain one sentence per line which is tokenized with the English tokenizer model from our website. See the Tokenizer documentation for further details.

Before a line is tokenized, the tool separates round and curly brackets from the text around them, because the parser expects each bracket as a token of its own. A bracket that is already surrounded by whitespace is left as it is, so no doubled spaces are introduced. Brackets that directly follow each other are each separated: `a))b` becomes `a ) ) b` and `foo(bar){baz}` becomes `foo ( bar ) { baz }`. Whether a character counts as whitespace follows the toolkit whitespace mode, see `StringUtil.isWhitespace`; under the default mode any Unicode whitespace counts, tabs and no-break spaces included.

Parsing API

The Parser can be easily integrated into an application via its API. To instantiate a Parser the parser model must be loaded first.

```
InputStream modelIn = new FileInputStream("en-parser-chunking.bin");
try {
  ParserModel model = new ParserModel(modelIn);
} catch (IOException e) {
  e.printStackTrace();
} finally {
  if (modelIn != null) {
    try {
      modelIn.close();
   }
   catch (IOException e) {
   }
}
```

Unlike the other components to instantiate the Parser a factory method should be used instead of creating the Parser via the new operator. The parser model is either trained for the chunking parser or the tree insert parser the parser implementation must be chosen correctly. The factory method will read a type parameter from the model and create an instance of the corresponding parser implementation.

```
Parser parser = ParserFactory.create(model);
```

The parser can tag and chunk with components the application supplies instead of the ones stored in the model, for example a POS tagger trained on domain text. This factory method takes the build, attach and check models and the head rules from the parser model and does not construct the model's own tagger or chunker. To replace only one of them, pass the model's component for the other.

```
POSTagger tagger = new POSTaggerME(customPosModel);
Chunker chunker = new ChunkerME(model.getParserChunkerModel());
Parser parser = ParserFactory.create(model, tagger, chunker,
   AbstractBottomUpParser.defaultBeamSize, AbstractBottomUpParser.defaultAdvancePercentage);
```

Right now the tree insert parser is still experimental and there is no pre-trained model for it. The parser expect a whitespace tokenized sentence. A utility method from the command line tool can parse the sentence String. The following code shows how the parser can be called.

```
String sentence = "The quick brown fox jumps over the lazy dog .";
Parse[] topParses = ParserTool.parseLine(sentence, parser, 1);
```

The topParses array only contains one parse because the number of parses is set to 1. The Parse object contains the parse tree. To display the parse tree call the show method. It either prints the parse to the console or into a provided StringBuffer. Similar to Exception.printStackTrace.

TODO: Extend this section with more information about the Parse object.

Parser Training

The OpenNLP offers two different parser implementations, the chunking parser and the treeinsert parser. The later one is still experimental and not recommended for production use. (TODO: Add a section which explains the two different approaches) The training can either be done with the command line tool or the training API. In the first case the training data must be available in the OpenNLP format. Which is the Penn Treebank format, but with the limitation of a sentence per line.

```
(TOP (S (NP-SBJ (DT Some) )(VP (VBP say) (NP (NNP November) ))(. .) ))
(TOP (S (NP-SBJ (PRP I) )(VP (VBP say) (NP (CD 1992) ))(. .) (" ") ))
```

Penn Treebank annotation guidelines can be found on the Penn Treebank home page [https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html]. A parser model also contains a pos tagger model, depending on the amount of available training data it is recommended to switch the tagger model against a tagger model which was trained on a larger corpus. The pre-trained parser model provided on the website is doing this to achieve a better performance. (TODO: On which data is the model on the website trained, and say on which data the tagger model is trained)

Training Tool

OpenNLP has a command line tool which is used to train the models available from the model download page on various corpora. The data must be converted to the OpenNLP parser training format, which is shortly explained above. To train the parser a head rules file is also needed. (TODO: Add documentation about the head rules file) Usage of the tool:

```
$ opennlp ParserTrainer
Usage: opennlp ParserTrainer -headRules headRulesFile [-parserType CHUNKING|TREEINSERT] \
                        [-params paramsFile] [-iterations num] [-cutoff num] \
                        -model modelFile -lang language -data sampleData \
                        [-encoding charsetName]

Arguments description:
  -headRules headRulesFile
    head rules file.
  -parserType CHUNKING|TREEINSERT
    one of CHUNKING or TREEINSERT, default is CHUNKING.
  -params paramsFile
    training parameters file.
  -iterations num
    number of training iterations, ignored if -params is used.
  -cutoff num
    minimal number of times a feature must be seen, ignored if -params is used.
  -model modelFile
    output model file.
  -format formatName
    data format, might have its own parameters.
  -encoding charsetName
    encoding for reading and writing text, if absent the system default is used.
  -lang language
    language which is being processed.
  -data sampleData
    data to be used, usually a file name.
  -encoding charsetName
    encoding for reading and writing text, if absent the system default is used.
```

The model on the website was trained with the following command:

```
$ opennlp ParserTrainer -model en-parser-chunking.bin -parserType CHUNKING \
                    -headRules head_rules \
                    -lang en -data train.all -encoding ISO-8859-1
```

Its also possible to specify the cutoff and the number of iterations, these parameters are used for all trained models. The -parserType parameter is an optional parameter, to use the tree insertion parser, specify TREEINSERT as type. The TaggerModelReplacer tool replaces the tagger model inside the parser model with a new one.

Note

The original parser model will be overwritten with the new parser model which contains the replaced tagger model.

```
$ opennlp TaggerModelReplacer en-parser-chunking.bin en-pos-maxent.bin
```

Additionally there are tools to just retrain the build or the check model.

Training API

The Parser training API supports the training of a new parser model. Four steps are necessary to train it:

- A HeadRules class needs to be instantiated: currently EnglishHeadRules and AncoraSpanishHeadRules are available.

- The application must open a sample data stream.
- Call a Parser train method: This can be either the CHUNKING or the TREEINSERT parser.
- Save the ParseModel to a file

The following code snippet shows how to instantiate the HeadRules:

```
static HeadRules createHeadRules(TrainerToolParams params) throws IOException {
   ArtifactSerializer headRulesSerializer = null;
   if (params.getHeadRulesSerializerImpl() != null) {
      headRulesSerializer = ExtensionLoader.instantiateExtension(ArtifactSerializer.class,
            params.getHeadRulesSerializerImpl());
   } else {
     if ("eng".equals(params.getLang())) {
        headRulesSerializer = new opennlp.tools.parser.lang.en.HeadRules.HeadRulesSerializer();
     } else if ("es".equals(params.getLang())) {
        headRulesSerializer = new opennlp.tools.parser.lang.es.AncoraSpanishHeadRules.HeadRulesSerializer();
     } else {
        // default for now, this case should probably cause an error ...
        headRulesSerializer = new opennlp.tools.parser.lang.en.HeadRules.HeadRulesSerializer();
     }
   }

   Object headRulesObject = headRulesSerializer.create(new FileInputStream(params.getHeadRules()));
   if (headRulesObject instanceof HeadRules) {
     return (HeadRules) headRulesObject;
   } else {
     throw new TerminateToolException(-1, "HeadRules Artifact Serializer must create an object of type HeadRules!");
   }
}
```

The following code illustrates the three other steps, namely, opening the data, training the model and saving the ParserModel into an output file.

```
ParserModel model = null;
File modelOutFile = params.getModel();
CmdLineUtil.checkOutputFile("parser model", modelOutFile);

try {
  HeadRules rules = createHeadRules(params);
  InputStreamFactory inputStreamFactory = new MarkableFileInputStreamFactory(new File("parsing.train"));
  ObjectStream<String> stringStream = new PlainTextByLineStream(inputStreamFactory, StandardCharsets.UTF_8);
  ObjectStream<Parse> sampleStream = new ParseSample(stringStream);
  ParserType type = parseParserType(params.getParserType());
  if (ParserType.CHUNKING.equals(type)) {
   model = opennlp.tools.parser.chunking.Parser.train(
       params.getLang(), sampleStream, rules,
       mlParams);
  } else if (ParserType.TREEINSERT.equals(type)) {
   model = opennlp.tools.parser.treeinsert.Parser.train(params.getLang(), sampleStream, rules,
       mlParams);
   }
  } catch (IOException e) {
    throw new TerminateToolException(-1, "IO error while reading training data or indexing data: " + e.getMessage(), e);
  } finally {
    try {
     sampleStream.close();
    }
    catch (IOException e) {
     // sorry that this can fail
    }
  }
  CmdLineUtil.writeModel("parser", modelOutFile, model);
```

Parser Evaluation

The built in evaluation can measure the parser performance. The performance is measured on a test dataset.

Parser Evaluation Tool

The following command shows how the tool can be run:

```
$ opennlp ParserEvaluator
Usage: opennlp ParserEvaluator[.ontonotes|frenchtreebank] [-misclassified true|false] -model model \
            -data sampleData [-encoding charsetName]
```

A sample of the command considering you have a data sample named en-parser-chunking.eval, and you trained a model called en-parser-chunking.bin:

```
$ opennlp ParserEvaluator -model en-parser-chunking.bin -data en-parser-chunking.eval -encoding UTF-8
```

and here is a sample output:

```
Precision: 0.9009744742967609
Recall: 0.8962012400910446
F-Measure: 0.8985815184245214
```

The Parser Evaluation tool reimplements the PARSEVAL scoring method as implemented by the EVALB [http://nlp.cs.nyu.edu/evalb/] script, which is the most widely used evaluation tool for constituent parsing.

Note

Currently the Parser Evaluation tool does not allow to make exceptions in the constituents to be evaluated, in the way Collins or Bikel usually do. Any contributions are very welcome. If you want to contribute please contact us on the mailing list or open an issue on GitHub [https://github.com/apache/opennlp/issues] (see also the historical issue OPENNLP-688 [https://issues.apache.org/jira/browse/OPENNLP-688]).

Evaluation API

The evaluation can be performed on a pre-trained model and a test dataset or via cross validation. In the first case the model must be loaded and a Parse ObjectStream must be created (see code samples above), assuming these two objects exist the following code shows how to perform the evaluation:

```
Parser parser = ParserFactory.create(model);
ParserEvaluator evaluator = new ParserEvaluator(parser);
evaluator.evaluate(sampleStream);

FMeasure result = evaluator.getFMeasure();
System.out.println(result.toString());
```

In the cross validation case all the training arguments must be provided (see the Training API section above). To perform cross validation the ObjectStream must be resettable.

```
InputStreamFactory inputStreamFactory = new MarkableFileInputStreamFactory(new File("parsing.train"));
ObjectStream<String> stringStream = new PlainTextByLineStream(inputStreamFactory, StandardCharsets.UTF_8);
ObjectStream<Parse> sampleStream = new ParseSample(stringStream);
ParserCrossValidator evaluator = new ParserCrossValidator("eng", trainParameters, headRules, \
parserType, listeners.toArray(new ParserEvaluationMonitor[listeners.size()])));
evaluator.evaluate(sampleStream, 10);

FMeasure result = evaluator.getFMeasure();
System.out.println(result.toString());
```

Chapter 18. Dependency Parsing

Introduction

Dependency parsing assigns each token a syntactic head and a relation label. Two parsers implement `DependencyParser`: `DependencyParserME` and `FeedforwardDependencyParser`. Both accept tokens with aligned part-of-speech tags and return a `DependencyTree`. Both choose the next arc-standard transition for each parser state: `DependencyParserME` scores the transitions with a maximum entropy model and `FeedforwardDependencyParser` with a feedforward network. The command-line tools cover both parsers.

Training data

`ConlluDependencySampleStream` reads the basic `HEAD` and `DEPREL` columns of a CoNLL-U treebank. Select `ConlluTagset.U` for universal tags or `ConlluTagset.X` for language-specific tags. The reader retains syntactic words from multiword tokens and skips sentences without a complete dependency tree or the selected POS tags.

The arc-standard transition system only builds projective trees, where no two arcs cross. Training skips non-projective sentences, and the parser never predicts a non-projective tree.

Training and loading a model

Train and save the standard OpenNLP model with `TrainingParameters`:

```
InputStreamFactory trainingData =
    () -> Files.newInputStream(Path.of("train.conllu"));

TrainingParameters parameters = TrainingParameters.defaultParams();
DependencyModel model;
try (ObjectStream<DependencySample> samples =
    new ConlluDependencySampleStream(trainingData, ConlluTagset.U)) {
  model = DependencyParserME.train("eng", samples, parameters);
}
model.serialize(Path.of("en-depparse.bin"));

DependencyParser parser =
    new DependencyParserME(new DependencyModel(Path.of("en-depparse.bin")));
```

Command-line tools

Training, evaluation, and cross validation read UTF-8 CoNLL-U files. An `-encoding` value other than UTF-8 is rejected. Use `-tagset u` (the default) for UPOS or `-tagset x` for XPOS. Use the same tagset for training and inference. An optional `-params` file supplies the standard training parameters. For example:

```
opennlp DependencyParserTrainerME.conllu -lang eng -data train.conllu -model en-depparse.bin
opennlp DependencyParserEvaluator.conllu -model en-depparse.bin -data dev.conllu
opennlp DependencyParserCrossValidator.conllu -lang eng -data train.conllu -folds 5
opennlp DependencyParserME en-depparse.bin < tagged-sentences.txt
```

Inference reads one sentence per line in the POS tagger's `token_TAG` format, for example `dogs_NOUN run_VERB`. It writes ten-column CoNLL-U output with one-based heads and `0` for the root. Input tags are retained in XPOS; UPOS and other unavailable annotations are _. The evaluation and cross-validation tools report UAS and LAS over all scored tokens and over the tokens that are not punctuation. With `-misclassified true` they also print each misparsed sentence next to its gold tree.

Feedforward parser

`FeedforwardDependencyTrainer` trains a neural model from the same samples and `TrainingParameters`; absent keys take the trainer's defaults. This example trains, saves, and reloads a model and parses with a beam of four:

```
FeedforwardDependencyModel model;
try (ObjectStream<DependencySample> samples =
    new ConlluDependencySampleStream(trainingData, ConlluTagset.U)) {
  model = FeedforwardDependencyTrainer.train("eng", samples, new TrainingParameters());
}
model.serialize(Path.of("en-depparse-ff.bin"));

DependencyParser parser = new FeedforwardDependencyParser(
    new FeedforwardDependencyModel(Path.of("en-depparse-ff.bin")), 4);
```

A beam size of one decodes greedily; a larger beam keeps that many transition sequences per step and scores each of them. Without a beam size the parser uses the beam size the model was refined with, or decodes greedily if the model was not refined. One parser instance can be shared between threads.

Command-line tools

As with the tools in the section called “Command-line tools”, the feedforward trainer and evaluator read CoNLL-U in UTF-8, and `-tagset` selects the POS column. The parser reads `token_TAG` lines and writes CoNLL-U, and the evaluator logs the same scores. The `-params` file holds the `FeedforwardDependencyTrainer` keys, for example `EmbeddingSize`, `HiddenSize`, `Iterations`, and `Dropout`; absent keys, or a missing file, take the trainer's defaults. A `-refineParams` file refines the trained model with beam search, with `BeamSize` (at least two) setting the beam. With `-baseModel` the trainer refines an existing model instead of training a new one; its `-lang` must match the model. For example:

```
opennlp FeedforwardDependencyParserTrainer.conllu -lang eng -data train.conllu \
    -model en-depparse-ff.bin -params ff.params -refineParams refine.params
opennlp FeedforwardDependencyParserEvaluator.conllu -model en-depparse-ff.bin -data dev.conllu
opennlp FeedforwardDependencyParser -beamSize 4 en-depparse-ff.bin < tagged-sentences.txt
```

The parser and evaluator take an optional `-beamSize`. Without it they use the beam size a refined model records, and parse a model that was not refined greedily. There is no feedforward cross-validation tool.

Parsing and evaluation

Tree indices refer to the input token array:

```
String[] tokens = {"Alice", "sent", "Bob", "a", "message", "."};
String[] tags = {"PROPN", "VERB", "PROPN", "DET", "NOUN", "PUNCT"};
DependencyTree tree = parser.parse(tokens, tags);

for (DependencyArc arc : tree.arcs()) {
  String dependent = tokens[arc.dependent()];
  String head = arc.head() == DependencyArc.ROOT_HEAD
      ? "ROOT" : tokens[arc.head()];
  System.out.println(dependent + " -> " + head + " (" + arc.relation() + ")");
}
```

`DependencyEvaluator` reports the unlabeled attachment score (the fraction of tokens with the correct head) and the labeled attachment score (the fraction with the correct head and relation label), both over all tokens and, through the `ExcludingPunctuation` getters, over the tokens whose gold tag is not `PUNCT`. `DependencyCrossValidator` splits a sample stream into folds, trains a parser on the other folds with the given training parameters, and scores the held-out fold:

```
DependencyCrossValidator validator =
    new DependencyCrossValidator("eng", parameters);
validator.evaluate(trainingSamples, 4);
double uas = validator.getUas();
double lasExcludingPunctuation = validator.getLasExcludingPunctuation();
```

The provided treebank evaluation, `UniversalDependencyParserEval` in `opennlp-eval-tests`, trains each parser from scratch on a Universal Dependencies 2.0 training split and pins all four scores on data the parser has not seen. `DependencyParserME` is scored on the English, German, Spanish AnCora, and French development splits and a five-fold cross validation of the English training split; `FeedforwardDependencyParser` is scored greedily and with a beam of four on the English and Spanish AnCora development splits. It uses gold sentence boundaries, tokens, and tags, so it measures the dependency parser rather than a complete text pipeline. Evaluation uses the existing `opennlp-data` treebanks.

Chapter 19. Entity Linker

Overview

The Entity Linker API lets an application associate named entities with records in an external data source. For example, a location found by a Name Finder can be matched to a gazetteer entry, or a person can be matched to a directory record. OpenNLP provides the linking interfaces and data types; applications supply the implementation and data source.

An `EntityLinker` receives the document text, sentence and token spans, and the name spans produced for each sentence. It returns application-defined spans that hold the links. `LinkedSpan` and `BaseLink` are support classes for this result; they are not linker implementations.

Preparing Input Spans

The four arguments passed to `find` use two coordinate systems. Sentence and token spans are character offsets in the complete document. Name spans are token indexes within their sentence, as returned by `NameFinderME`. The outer indexes of `tokensBySentence` and `namesBySentence` must correspond to the same entry in `sentences`.

The following method prepares those arrays. The sentence detector, tokenizer, and name finder are accepted as interfaces so the same code works with their standard OpenNLP implementations.

```
static List<? extends Span> linkEntities(String documentText,
    SentenceDetector sentenceDetector, Tokenizer tokenizer,
    TokenNameFinder nameFinder, EntityLinker<? extends Span> linker) {

  Span[] sentences = sentenceDetector.sentPosDetect(documentText);
  Span[][] tokensBySentence = new Span[sentences.length][];
  Span[][] namesBySentence = new Span[sentences.length][];

  for (int i = 0; i < sentences.length; i++) {
    Span sentence = sentences[i];
    String sentenceText = sentence.getCoveredText(documentText).toString();
    Span[] relativeTokenSpans = tokenizer.tokenizePos(sentenceText);
    String[] tokenTexts = Span.spansToStrings(relativeTokenSpans, sentenceText);

    namesBySentence[i] = nameFinder.find(tokenTexts);
    tokensBySentence[i] = new Span[relativeTokenSpans.length];
    for (int j = 0; j < relativeTokenSpans.length; j++) {
      // Tokenizer offsets are sentence-relative; EntityLinker expects document offsets.
      tokensBySentence[i][j] = new Span(relativeTokenSpans[j], sentence.getStart());
    }
  }

  return linker.find(documentText, sentences, tokensBySentence, namesBySentence);
}
```

Implementing an Entity Linker

Implement `EntityLinker<T>`, where `T` extends `Span`. Load reusable resources in `init`. Implement the document overload of `find` by visiting each sentence, and implement the sentence overload with the actual lookup logic.

This example uses `LinkedSpan` for its results and a small concrete subclass of `BaseLink` for gazetteer records:

```
public final class GazetteerLink extends BaseLink {

  public GazetteerLink(String id, String name) {
    super(null, id, name, "location");
  }
}

public final class GazetteerLinker
    implements EntityLinker<LinkedSpan<GazetteerLink>> {

  private Map<String, GazetteerLink> entries;

  @Override
  public void init(EntityLinkerProperties properties) throws IOException {
    String path = properties.getProperty("gazetteer.path", "gazetteer.tsv");
    entries = loadEntries(Path.of(path));
  }

  @Override
  public List<LinkedSpan<GazetteerLink>> find(String documentText,
      Span[] sentences, Span[][] tokensBySentence, Span[][] namesBySentence) {
    List<LinkedSpan<GazetteerLink>> results = new ArrayList<>();
    for (int i = 0; i < sentences.length; i++) {
      results.addAll(find(documentText, sentences, tokensBySentence,
          namesBySentence, i));
    }
    return results;
  }

  @Override
  public List<LinkedSpan<GazetteerLink>> find(String documentText,
      Span[] sentences, Span[][] tokensBySentence, Span[][] namesBySentence,
      int sentenceIndex) {
    List<LinkedSpan<GazetteerLink>> results = new ArrayList<>();
    Span[] tokens = tokensBySentence[sentenceIndex];

    for (Span name : namesBySentence[sentenceIndex]) {
      // Name spans contain token indexes, so map their boundaries through the token spans.
      int start = tokens[name.getStart()].getStart();
      int end = tokens[name.getEnd() - 1].getEnd();
      String searchTerm = documentText.substring(start, end);
      GazetteerLink match = entries.get(searchTerm);

      if (match != null) {
        ArrayList<GazetteerLink> matches = new ArrayList<>();
        matches.add(match);
        LinkedSpan<GazetteerLink> linkedSpan =
            new LinkedSpan<>(matches, start, end, name.getType());
        linkedSpan.setSentenceid(sentenceIndex);
        linkedSpan.setSearchTerm(searchTerm);
        results.add(linkedSpan);
      }
    }
    return results;
  }

  private static Map<String, GazetteerLink> loadEntries(Path path)
      throws IOException {
    // Parse the application's data source here.
    return Map.of();
  }
}
```

A production implementation should validate array lengths and name boundaries before indexing into `tokens`, normalize lookup keys consistently, and define how multiple candidate records are scored and ordered.

Loading an Implementation

`EntityLinkerFactory` loads an implementation through OpenNLP's extension loader and then calls its `init` method. With the overload that accepts an entity type, the key is `linker.<entityType>`:

```
linker.location=com.example.GazetteerLinker
gazetteer.path=/data/locations.tsv
```

Load the properties and request the configured linker as follows:

```
EntityLinkerProperties properties =
    new EntityLinkerProperties(new File("entity-linker.properties"));
EntityLinker<?> linker = EntityLinkerFactory.getLinker("location", properties);
```

The overload without an entity type reads the key `linker` instead.

Command Line Tool

The **EntityLinker** command loads the `linker.location` implementation from a properties file. It reads blank-line-separated documents in the native Name Finder sample format from standard input and writes each linked span to standard output.

```
opennlp EntityLinker entity-linker.properties < names.txt
```

Each non-empty input line is a tokenized sentence whose names use `<START:type>` and `<END>` markers. For example:

```
The office is in <START:location> New York <END> .

The second document starts after the blank line .
```

Chapter 20. Coreference Resolution

The OpenNLP Coreference Resolution system links multiple mentions of an entity in a document together. The OpenNLP implementation is currently limited to noun phrase mentions, other mention types cannot be resolved.

TODO: Write more documentation about the coref component. Any contributions are very welcome. If you want to contribute please contact us on the mailing list or open an issue on GitHub [https://github.com/apache/opennlp/issues] (see also the historical issue OPENNLP-48 [https://issues.apache.org/jira/browse/OPENNLP-48]).

Chapter 21. Classpath Loading of OpenNLP Models

Background

Since version 2.4.0, OpenNLP supports the ability to load pre-trained OpenNLP models from the classpath. It relies on either a simple implementation using the application's classpath or on the classgraph [https://github.com/classgraph/classgraph] library to locate OpenNLP model JAR files. Our pre-trained models are bundled from the OpenNLP Models repository [https://github.com/apache/opennlp-models]. This chapter describes

- how to load and use a pre-trained OpenNLP model available in the classpath.
- how to bundle a custom OpenNLP model to be loadable as a JAR file from the classpath.

Loading a pre-trained model

First, you need to add the following dependency to your classpath:

```
<dependency>
  <groupId>org.apache.opennlp</groupId>
  <artifactId>opennlp-model-resolver</artifactId>
  <version>CURRENT_OPENNLP_VERSION</version>
</dependency>
```

by using our pre-trained models or by building custom models as described later in this section. If you need advanced classpath scanning capabilities, you can also add the classgraph library to your classpath.

```
<dependency>
   <groupId>io.github.classgraph</groupId>
   <artifactId>classgraph</artifactId>
   <version>CURRENT_CLASSGRAPH_VERSION</version>
</dependency>
```

Make sure you replace the placeholders with the appropriate version values. Next, you can search for such a model and load it from the classpath:

```
final ClassgraphModelFinder finder = new ClassgraphModelFinder(); // or use: new SimpleClassPathModelFinder()
final ClassPathModelLoader loader = new ClassPathModelLoader();
final Set<ClassPathModelEntry> models = finder.findModels(false);
for(ClassPathModelEntry entry : models) {

    final ClassPathModel model = loader.load(entry);
    if(model != null) {
        System.out.println(model.getModelName());
        System.out.println(model.getModelSHA256());
        System.out.println(model.getModelVersion());
        System.out.println(model.getModelLanguage());
        // do something with the model by consuming the byte array
    }
}
```

Moreover, certain OpenNLP models can be obtained via a *ClassPathModelProvider*, such as OpenNLP's built-in *DefaultClassPathModelProvider* class. It allows direct use of models available under a certain locale, given that those are present in the classpath and can be loaded.

```
final ClassPathModelProvider provider = new DefaultClassPathModelProvider(finder, loader);
// Here: SentenceModel, other model types accordingly
final SentenceModel sm = provider.load("en", opennlp.tools.models.ModelType.SENTENCE_DETECTOR,
 SentenceModel.class);
if(sm != null) {
    // do something with the (sentence) model
}
```

In this example, the finder and loader objects can be created or re-used as shown in the previous code example.

Note

When running on Java 21+ (the minimum supported version since OpenNLP 3.0.0), the JVM argument

```
--add-opens java.base/jdk.internal.loader=ALL-UNNAMED
```

may be required. Without this parameter, OpenNLP uses the JVM bootstrap classpath to locate models rather than the UCP class loader. For more advanced or non-standard class loading scenarios, using ClassGraph and implementing a custom provider may cover additional cases beyond the default UCP class loader or JVM bootstrap class path behavior.

Bundling a self-trained model

If you intend to provide your own custom trained OpenNLP models as JAR files for classpath discovery, we recommend that you have a look at our setup in the OpenNLP Models repository [https://github.com/apache/opennlp-models]. We recommend to bundle one model per JAR file. Make sure you add a *model.properties* file with the following content:

```
model.name=${model.name}
model.version=${model.version}
model.sha256=${model.sha256}
model.language=${model.language}
```

Make sure to replace the values accordingly and configure your build tool to include the binary model and the *model.properties* in the resulting JAR file. To load such a custom model, you may need to adjust the pattern for classpath scanning. For example, if you name the model "custom-opennlp-model", you need the following code to successfully find and load it:

```
final ClassgraphModelFinder finder = new ClassgraphModelFinder("custom-opennlp-model.jar"); // or use: new
 SimpleClassPathModelFinder("custom-opennlp-model.jar")
final ClassPathModelLoader loader = new ClassPathModelLoader();
final Set<ClassPathModelEntry> models = finder.findModels(false);
for(ClassPathModelEntry entry : models) {
   final ClassPathModel model = loader.load(entry);
   if(model != null) {
     System.out.println(model.getModelName());
     System.out.println(model.getModelSHA256());
     System.out.println(model.getModelVersion());
     System.out.println(model.getModelLanguage());
     // do something with the model by consuming the byte array
   }
}
```

For `SimpleClassPathModelFinder` and `DirectoryModelFinder`, the search mask is a wildcard, not a regular expression. An asterisk matches any run of characters, including none, a question mark matches exactly one character, and every other character, including percent and plus signs, stands for itself. There is no escape character. The mask is compared case-sensitively with the whole decoded file part of each classpath URL, and the finders put an asterisk in front of it, so `custom-opennlp-model.jar` matches that jar in any directory. Use the actual file name: `model-?.jar` matches `model-##.jar`, and a space is written as a space, not as `%20`. A character outside the Basic Multilingual Plane counts as one character for the question mark. The model and properties URIs returned by the finders remain escaped for opening the resources. `ClassgraphModelFinder` uses ClassGraph's own wildcard rules.

Custom subclasses of `AbstractClassPathModelFinder` match model file names with `matchesWildcard(url, "*.bin")`.

Installing third-party resources

`ResourceInstaller` copies a supplied `http`, `https`, or `file` URI into a target directory. Http and https sources require a SHA-256 or SHA-512 checksum. File sources may omit it. Gzip-compressed tar and zip archives are unpacked, plain gzip files are decompressed, and other content is stored under its source name. OpenNLP `*.bin` model files remain packed.

```
Path result = ResourceInstaller.install(
   corpusTarGz.toUri(), targetDirectory, expectedSha256);
// result is targetDirectory; the archived files appear under their archive paths
```

Content is unpacked in a staging directory on the target filesystem. Archive paths cannot escape that directory, duplicate file paths are rejected, and existing target files are not replaced. A checksum, download, or unpacking error promotes no files. Promotion also rejects symbolic links below the target. Concurrent changes to the target directory are outside this guarantee.

Http and https requests use connection and read timeouts, follow up to five redirects by default, and reject an https-to-http downgrade. Default limits are 1 GiB downloaded, 4 GiB expanded, and 100000 archive entries; compressed content may expand to at most 100 times its compressed size, with a 1 MiB floor for small sources. These defaults can be set at JVM startup with `opennlp.download.max.bytes`, `opennlp.install.max.total.bytes`, `opennlp.install.max.entries`, and `opennlp.install.max.expansion.ratio`. Use `ResourceInstaller.Limits` to set limits for one call.

The expansion ratio and the expansion limit are enforced independently and the tighter one applies, so raising `opennlp.install.max.total.bytes` alone does not admit an archive that exceeds the ratio. A

resource that legitimately compresses better than 100 to 1, such as a corpus of highly repetitive text, needs `opennlp.install.max.expansion.ratio` raised as well.

```
ResourceInstaller.Limits limits = ResourceInstaller.Limits.builder()
    .readTimeout(Duration.ofSeconds(10))
    .maxDownloadBytes(1024L * 1024)   // download limit in bytes
    .maxExpandedBytes(1024L * 1024)   // expansion limit in bytes
    .maxExpansionRatio(500)           // expanded bytes per compressed byte
    .build();
Path result = ResourceInstaller.install(
    corpusTarGz.toUri(), targetDirectory, expectedSha256, limits);
```

Chapter 22. Extending OpenNLP

OpenNLP extensions allow customization of existing components. Most components define a factory class that can be implemented to control their creation. The implementation class must implement the required interface and have a public no-argument constructor.

Extension Class Allowlist

Extension classes are loaded by `ExtensionLoader` via reflection. Only classes whose fully-qualified name starts with a registered package prefix are permitted. The default prefix is `opennlp.`, covering all built-in factories and serializers. The same check applies to the model reader and writer classes that `GenericModelReader` and `GenericModelWriter` load for a model type.

To use a custom class outside `opennlp.`, register its package before loading any model that references it:

```
ExtensionLoader.registerAllowedPackage("com.example.nlp");
```

Alternatively, set the system property at JVM startup:

```
-DOPENNLP_EXT_ALLOWED_PACKAGES=com.example.nlp.,com.other.factories.
```

A registered prefix can be removed with `ExtensionLoader.unregisterAllowedPackage(String)`.

Chapter 23. Corpora

OpenNLP has built-in support to convert into the native training format or directly use various corpora needed by the different trainable components.

CONLL

CoNLL stands for the Conference on Computational Natural Language Learning and is not a single project but a consortium of developers attempting to broaden the computing environment. More information about the entire conference series can be obtained here for CoNLL.

CONLL 2000

The shared task of CoNLL-2000 is Chunking.

Getting the data

CoNLL-2000 made available training and test data for the Chunk task in English. The data consists of the same partitions of the Wall Street Journal corpus (WSJ) as the widely used data for noun phrase chunking: sections 15-18 as training data (211727 tokens) and section 20 as test data (47377 tokens). The annotation of the data has been derived from the WSJ corpus by a program written by Sabine Buchholz from Tilburg University, The Netherlands. Both training and test data can be obtained from https://www.clips.uantwerpen.be/conll2000/chunking/ [https://www.clips.uantwerpen.be/conll2000/chunking/].

Converting the data

The data don't need to be transformed because Apache OpenNLP Chunker follows the CONLL 2000 format for training. Check the section called “Chunker Training” section to learn more.

Training

We can train the model for the Chunker using the train.txt available at CONLL 2000:

```
$ opennlp ChunkerTrainerME -model en-chunker.bin -iterations 500 \
                    -lang en -data train.txt -encoding UTF-8
```

```
Indexing events using cutoff of 5

 Computing event counts...  done. 211727 events
 Indexing...  done.
Sorting and merging events... done. Reduced 211727 events to 197252.
Done indexing.
Incorporating indexed data for training...
done.
 Number of Event Tokens: 197252
     Number of Outcomes: 22
   Number of Predicates: 107838
...done.
Computing model parameters...
Performing 500 iterations.
  1:  .. loglikelihood=-654457.1455212828 0.2601510435608118
  2:  .. loglikelihood=-239513.5583724216 0.9260037690044255
  3:  .. loglikelihood=-141313.1386347238 0.9443387003074715
  4:  .. loglikelihood=-101083.50853437989 0.954375209585929
... cut lots of iterations ...
498:  .. loglikelihood=-1710.8874647317095 0.9995040783650645
499:  .. loglikelihood=-1708.0908900815848 0.9995040783650645
500:  .. loglikelihood=-1705.3045902366732 0.9995040783650645
Writing chunker model ... done (4.019s)

Wrote chunker model to path: .\en-chunker.bin
```

Evaluating

We evaluate the model using the file test.txt available at CONLL 2000:

```
$ opennlp ChunkerEvaluator -model en-chunker.bin -lang en -encoding utf8 -data test.txt
```

```
Loading Chunker model ... done (0,665s)
current: 85,8 sent/s avg: 85,8 sent/s total: 86 sent
current: 88,1 sent/s avg: 87,0 sent/s total: 174 sent
current: 156,2 sent/s avg: 110,0 sent/s total: 330 sent
current: 192,2 sent/s avg: 130,5 sent/s total: 522 sent
current: 167,2 sent/s avg: 137,8 sent/s total: 689 sent
current: 179,2 sent/s avg: 144,6 sent/s total: 868 sent
current: 183,2 sent/s avg: 150,3 sent/s total: 1052 sent
current: 183,2 sent/s avg: 154,4 sent/s total: 1235 sent
current: 169,2 sent/s avg: 156,0 sent/s total: 1404 sent
current: 178,2 sent/s avg: 158,2 sent/s total: 1582 sent
current: 172,2 sent/s avg: 159,4 sent/s total: 1754 sent
current: 177,2 sent/s avg: 160,9 sent/s total: 1931 sent


Average: 161,6 sent/s
Total: 2013 sent
Runtime: 12.457s

Precision: 0.9244354736974896
Recall: 0.9216837162502096
F-Measure: 0.9230575441395671
```

CONLL 2002

The shared task of CoNLL-2002 is language independent named entity recognition for Spanish and Dutch.

Getting the data

The data consists of three files per language: one training file and two test files testa and testb. The first test file will be used in the development phase for finding good parameters for the learning system. The second test file will be used for the final evaluation. Currently, there are data files available for two languages: Spanish and Dutch.

The Spanish data is a collection of newswire articles made available by the Spanish EFE News Agency. The articles are from May 2000. The annotation was carried out by the TALP Research Center [https://www.talp.cat/] of the Technical University of Catalonia (UPC) and the Center of Language and Computation (CLiC) [https://web.archive.org/web/20220516042208/http://clic.ub.edu/]of the University of Barcelona (UB), and funded by the European Commission through the NAMIC project (IST-1999-12392).

The Dutch data consist of four editions of the Belgian newspaper "De Morgen" of 2000 (June 2, July 1, August 1 and September 1). The data was annotated as a part of the Atranos [https://atranos.esat.kuleuven.ac.be/] project at the University of Antwerp.

You can find the Spanish files here: https://www.lsi.upc.edu/~nlp/tools/nerc/nerc.html You must download esp.train.gz, unzip it and you will see the file esp.train.

You can find the Dutch files here: https://www.cnts.ua.ac.be/conll2002/ner.tgz You must unzip it and go to /ner/data/ned.train.gz, so you unzip it too, and you will see the file ned.train.

Converting the data

I will use Spanish data as reference, but it would be the same operations to Dutch. You just must remember change “-lang es” to “-lang nl” and use the correct training files. So to convert the information to the OpenNLP format:

```
$ opennlp TokenNameFinderConverter conll02 -data esp.train -lang es -types per > es_corpus_train_persons.txt
```

Optionally, you can convert the training test samples as well.

```
$ opennlp TokenNameFinderConverter conll02 -data esp.testa -lang es -types per > corpus_testa.txt
$ opennlp TokenNameFinderConverter conll02 -data esp.testb -lang es -types per > corpus_testb.txt
```

Training with Spanish data

To train the model for the name finder:

```
\bin\opennlp TokenNameFinderTrainer -lang es -encoding u
tf8 -iterations 500 -data es_corpus_train_persons.txt -model es_ner_person.bin

Indexing events using cutoff of 5

      Computing event counts...  done. 264715 events
      Indexing...  done.
Sorting and merging events... done. Reduced 264715 events to 222660.
Done indexing.
Incorporating indexed data for training...
done.
      Number of Event Tokens: 222660
        Number of Outcomes: 3
       Number of Predicates: 71514
...done.
Computing model parameters ...
Performing 500 iterations.
  1:  ... loglikelihood=-290819.1519958615      0.9689326256540053
  2:  ... loglikelihood=-37097.17676455632      0.9689326256540053
  3:  ... loglikelihood=-22910.372489660916     0.9706476776911017
  4:  ... loglikelihood=-17091.547325669497     0.9777874317662392
  5:  ... loglikelihood=-13797.620926769372     0.9833821279489262
  6:  ... loglikelihood=-11715.806710780415     0.9867140131839903
  7:  ... loglikelihood=-10289.222078246517     0.9886859452618855
  8:  ... loglikelihood=-9249.208318314624      0.9902310031543358
  9:  ... loglikelihood=-8454.169590899777      0.9913227433277298
 10:  ... loglikelihood=-7823.742997451327      0.9921953799369133
 11:  ... loglikelihood=-7309.375882641964      0.9928224694482746
 12:  ... loglikelihood=-6880.131972149693      0.9932946754056249
 13:  ... loglikelihood=-6515.3828767792365     0.993638441342576
 14:  ... loglikelihood=-6200.82723154046       0.9939595413935742
 15:  ... loglikelihood=-5926.213730444915      0.994269308501596
 16:  ... loglikelihood=-5683.9821840753275     0.9945299661900534
 17:  ... loglikelihood=-5468.4211798176075     0.9948246227074401
 18:  ... loglikelihood=-5275.127017232056      0.9950286156810154

... cut lots of iterations ...

491:  ... loglikelihood=-1174.8485558758211     0.998983812779782
492:  ... loglikelihood=-1173.9971776942477     0.998983812779782
493:  ... loglikelihood=-1173.1482915871768     0.998983812779782
494:  ... loglikelihood=-1172.3018855781158     0.998983812779782
495:  ... loglikelihood=-1171.457947774544      0.998983812779782
496:  ... loglikelihood=-1170.6164663670502     0.998983812779782
497:  ... loglikelihood=-1169.7774296286693     0.998983812779782
498:  ... loglikelihood=-1168.94082591387       0.998983812779782
499:  ... loglikelihood=-1168.1066436580463     0.9989875904274408
500:  ... loglikelihood=-1167.2748713765225     0.9989875904274408
Writing name finder model ... done (2,168s)

Wrote name finder model to
path: .\es_ner_person.bin
```

CONLL 2003

The shared task of CoNLL-2003 is language independent named entity recognition for English and German.

Getting the data

The English data is the Reuters Corpus, which is a collection of newswire articles. The Reuters Corpus can be obtained free of charges from the NIST for research purposes: https://trec.nist.gov/data/reuters/reuters.html

The German data is a collection of articles from the German newspaper Frankfurter Rundschau. The articles are part of the ECI Multilingual Text Corpus which can be obtained for 75$ (2010) from the Linguistic Data Consortium: https://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC94T5 [https://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC94T5]

After one of the corpora is available the data must be transformed as explained in the README file to the CONLL format. The transformed data can be read by the OpenNLP CONLL03 converter. Note that for CoNLL-2003 corpora, the -lang argument should either be "eng" or "deu", instead of "en" or "de".

Converting the data (optional)

To convert the information to the OpenNLP format:

```
$ opennlp TokenNameFinderConverter conll03 -lang eng -types per -data corpus_train.txt > eng.train
```

Optionally, you can convert the training test samples as well.

```
$ opennlp TokenNameFinderConverter conll03 -lang eng -types per -data corpus_testa.txt > eng.testa
$ opennlp TokenNameFinderConverter conll03 -lang eng -types per -data corpus_testb.txt > eng.testb
```

Training with English data

You can train the model for the name finder this way:

```
$ opennlp TokenNameFinderTrainer.conll03 -model en_ner_person.bin \
                        -lang eng -types per -data eng.train -encoding utf8
```

If you have converted the data, then you can train the model for the name finder this way:

```
$ opennlp TokenNameFinderTrainer.conll03 -model en_ner_person.bin \
                        -lang eng -types per -data corpus_train.txt -encoding utf8
```

Either way you should see the following output during the training process:

```
Indexing events with TwoPass using cutoff of 0

 Computing event counts...  done. 203621 events
 Indexing...  done.
Collecting events... Done indexing in 6,01 s.
Incorporating indexed data for training...
done.
 Number of Event Tokens: 203621
     Number of Outcomes: 3
   Number of Predicates: 442041
Computing model parameters...
Performing 300 iterations.
  1:  . (201717/203621) 0.9906492945226671
  2:  . (202770/203621) 0.9958206668270955
  3:  . (203129/203621) 0.9975837462737144
  4:  . (203261/203621) 0.9982320094685715
  5:  . (203381/203621) 0.9988213396457143
  6:  . (203429/203621) 0.9990570717165714
  7:  . (203454/203621) 0.9991798488368095
  8:  . (203494/203621) 0.9993762922291906
  9:  . (203509/203621) 0.9994499585013333
 10:  . (203533/203621) 0.999567824536762
 20:  . (203592/203621) 0.9998575785405238
 30:  . (203613/203621) 0.9999607113215239
Stopping: change in training set accuracy less than 1.0E-5
Stats: (203621/203621) 1.0
...done.

Training data summary:
#Sentences: 14041
#Tokens: 203621
#person entities: 6600

Writing name finder model ... Compressed 442041 parameters to 29538
4 outcome patterns
done (0,395s)

Wrote name finder model to
path: ./en_ner_person.bin

Execution time: 11,498 seconds
```

Evaluating with English data

You can evaluate the model for the name finder this way:

```
$ opennlp TokenNameFinderEvaluator.conll03 -model en_ner_person.bin \
                        -lang eng -types per -data eng.testa -encoding utf8
```

If you converted the test A and B files above, you can use them to evaluate the model.

```
$ opennlp TokenNameFinderEvaluator.conll03 -model en_ner_person.bin \
                        -lang eng -types per -data corpus_testa.txt -encoding utf8
```

Either way you should see the following output:

```
Loading Token Name Finder model ... done (0,176s)
current: 1805,4 sent/s avg: 1805,4 sent/s total: 1961 sent

Average: 2298,1 sent/s
Total: 3454 sent
Runtime: 1.503s

Evaluated 3453 samples with 1617 entities; found: 1472 entities; correct: 1370.
      TOTAL: precision:   93,07%;  recall:   84,72%; F1:   88,70%.
     person: precision:   93,07%;  recall:   84,72%; F1:   88,70%. [target: 1617; tp: 1370; fp: 102]

Execution time: 1,955 seconds
```

CoNLL-U

The CoNLL-U [https://universaldependencies.org/format.html] format of the Universal Dependencies treebanks can be used directly by the tokenizer, sentence detector, POS tagger, and lemmatizer trainers through the `conllu` format name, for example `opennlp TokenizerTrainer.conllu`, and converted with the matching converter tools.

The reader interprets the format as follows. A multiword token line has a range id such as `3-4`; the range must be two runs of ASCII digits without a leading zero, joined by a single hyphen, and its end must not be smaller than its start. An id that does not follow this form, such as `3-`, `-4`, `3--4`, `1-2-3`, or `4-3`, stops the read with an `InvalidFormatException` that names the id. The language of a sentence is taken from `text_xx` and `text_xxx` comment keys, where the code consists of two or three lowercase ASCII letters; when three letters follow, all three form the code.

CoNLL-U multiword token lines

Each id in a CoNLL-U multiword token range must have its corresponding word line in the same sentence. For example, a range `1-3` requires lines `1`, `2`, and `3`. A missing word line stops reading with `InvalidFormatException` naming the range and missing id. Correct the input before resuming conversion or training; missing words are not fabricated.

Arvores Deitadas

The Portuguese corpora available at Floresta Sintá(c)tica [https://www.linguateca.pt] project follow the Arvores Deitadas (AD) format. Apache OpenNLP includes tools to convert from AD format to native format.

Getting the data

The Corpus can be downloaded from here: https://www.linguateca.pt/floresta/corpus.html

The Name Finder models were trained using the Amazonia corpus: amazonia.ad [https://www.linguateca.pt/floresta/ficheiros/gz/amazonia.ad.gz]. The Chunker models were trained using the Bosque_CF_8.0.ad [https://www.linguateca.pt/floresta/ficheiros/gz/Bosque_CF_8.0.ad.txt.gz].

Converting the data (optional)

To extract NameFinder training data from Amazonia corpus:

```
$ opennlp TokenNameFinderConverter ad -lang pt -encoding ISO-8859-1 -data amazonia.ad > corpus.txt
```

To extract Chunker training data from Bosque_CF_8.0.ad corpus:

```
$ opennlp ChunkerConverter ad -lang pt -data Bosque_CF_8.0.ad.txt -encoding ISO-8859-1 > bosque-chunk
```

A quoted lemma ends at a matching quote followed by a field boundary. Embedded apostrophes and quotes within a word remain part of the lemma. Quotes in subsequent annotation fields or in the surface text do not extend the lemma or remove part of the token text.

The AD reader separates the parts of a line on whitespace as the Unicode White_Space property defines it, independent of the toolkit whitespace mode: tabs, no-break spaces, the next line character (U+0085), and the line and paragraph separators (U+2028 and U+2029) are handled like ordinary spaces, while the ASCII separator controls U+001C to U+001F are ordinary characters. A multiword name joined with underscores, such as `Rio_de_Janeiro`, is split into its words, and repeated or leading underscores do not produce empty words. A POS tag that contains whitespace is joined with `=` so that it remains a single token. The text id of a sentence is the run of ASCII digits that directly follows the leading ASCII letters and hyphens of its metadata, and the paragraph id is the digit run after the first `p=` that a digit follows; an id that does not fit into a 32-bit integer is invalid metadata. A punctuation line has a lexeme without any letter or digit of any script, so an accented or non-Latin word is not treated as punctuation.

Markup lines are recognized by their exact tag name. The tags `<s>`, `<p>`, `<t>`, `<caixa>`, and `<ext>` may have attributes after whitespace. A tag with a name that only starts with one of these names, such as `<sx>`, and a self-closing tag such as `<s/>` or `<s id="1"/>` are not sentence tags. A leaf line of a tree has its lexeme after the closing parenthesis, separated by whitespace; a line with no text or only whitespace after the parenthesis has no lexeme. Inside a quoted lemma, a secondary tag, or the lexeme, a line separator or next line character is kept as part of the text. A tree line that starts with equals signs but has no tag keeps all of the text after the equals signs as its lexeme. Such a line is skipped when that text is empty, or when it is a word of any script followed by a period or an angle bracket, and a line separator inside the text does not end the word.

End of file in AD input

An opening `<s>` tag with no following content at end of file ends the stream without producing a sample. A sentence with content but without its closing `</s>` is returned from the lines read so far.

AD leaf tags with embedded equals signs

Leaf tags may contain equals signs after the indentation, as in `=H==CJT:num`. The reader takes the functional tag after the last colon, so this example has the tag `num`. A chunk leaf without a functional tag uses its lexeme as the tag instead of a null value. This permits FlorestaVirgem chunk conversion to continue and changes the POS labels of the affected tagged leaves to their functional tags, such as `num` or `prop`. Re-evaluate models trained from affected samples.

Text boundaries in AD name samples

During a pass over LIT or CIE input, consecutive sentences with the same text name share a text id. When the name changes, the first name sample of the next text sets `clearAdaptiveData`, so name-finder context does not carry across text boundaries. Resetting the stream clears this boundary state, so the first sample of a new pass also clears adaptive data. Amazonia input continues to use its numeric text id. This changes training and evaluation samples for LIT and CIE corpora; evaluate models with representative data from those corpora.

Training and Evaluation

To perform the evaluation the corpus was split into a training and a test part.

```
$ sed '1,55172d' corpus.txt > corpus_train.txt
$ sed '55172,100000000d' corpus.txt > corpus_test.txt
```

```
$ opennlp TokenNameFinderTrainer -model pt-ner.bin -cutoff 20 -lang PT -data corpus_train.txt -encoding UTF-8
...
$ opennlp TokenNameFinderEvaluator -model pt-ner.bin -lang PT -data corpus_train.txt -encoding UTF-8

Precision: 0.8005071889818507
Recall: 0.7450581122145297
F-Measure: 0.7717879983140168
```

OntoNotes Release 4.0

"OntoNotes Release 4.0, Linguistic Data Consortium (LDC) catalog number LDC2011T03 and isbn 1-58563-574-X, was developed as part of the OntoNotes project, a collaborative effort between BBN Technologies, the University of Colorado, the University of Pennsylvania and the University of Southern Californias Information Sciences Institute. The goal of the project is to annotate a large corpus comprising various genres of text (news, conversational telephone speech, weblogs, usenet newsgroups, broadcast, talk shows) in three languages (English, Chinese, and Arabic) with structural information (syntax and predicate argument structure) and shallow semantics (word sense linked to an ontology and coreference). OntoNotes Release 4.0 is supported by the Defense Advance Research Project Agency, GALE Program Contract No. HR0011-06-C-0022.

OntoNotes Release 4.0 contains the content of earlier releases -- OntoNotes Release 1.0 LDC2007T21, OntoNotes Release 2.0 LDC2008T04 and OntoNotes Release 3.0 LDC2009T24 -- and adds newswire, broadcast news, broadcast conversation and web data in English and Chinese and newswire data in Arabic. This cumulative publication consists of 2.4 million words as follows: 300k words of Arabic newswire 250k words of Chinese newswire, 250k words of Chinese broadcast news, 150k words of Chinese broadcast conversation and 150k words of Chinese web text and 600k words of English newswire, 200k word of English broadcast news, 200k words of English broadcast conversation and 300k words of English web text.

The OntoNotes project builds on two time-tested resources, following the Penn Treebank for syntax and the Penn PropBank for predicate-argument structure. Its semantic representation will include word sense disambiguation for nouns and verbs, with each word sense connected to an ontology, and co-reference. The current goals call for annotation of over a million words each of English and Chinese, and half a million words of Arabic over five years." (https://catalog.ldc.upenn.edu/LDC2011T03)

Name Finder Training

The OntoNotes corpus can be used to train the Name Finder. The corpus contains different name types to train a model for a specific type only the built-in type filter option should be used.

The sample shows how to train a model to detect person names.

```
$ bin/opennlp TokenNameFinderTrainer.ontonotes -lang en -model en-ontonotes.bin \
    -nameTypes person -ontoNotesDir ontonotes-release-4.0/data/files/data/english/

Indexing events using cutoff of 5

 Computing event counts...  done. 1953446 events
 Indexing...  done.
Sorting and merging events... done. Reduced 1953446 events to 1822037.
Done indexing.
Incorporating indexed data for training...
done.
 Number of Event Tokens: 1822037
     Number of Outcomes: 3
   Number of Predicates: 298263
...done.
Computing model parameters ...
Performing 100 iterations.
  1:  ... loglikelihood=-2146079.7808976253 0.976677625078963
  2:  ... loglikelihood=-195016.59754190338 0.976677625078963
... cut lots of iterations ...
 99:  ... loglikelihood=-10269.902459614596 0.9987299367374374
100:  ... loglikelihood=-10227.160010853702 0.9987314724850341
Writing name finder model ... done (2.315s)

Wrote name finder model to
path: /dev/opennlp/trunk/opennlp-tools/en-ontonotes.bin
```

BioNLP/NLPBA 2004

The BioNLP/NLPBA 2004 shared task uses biomedical abstracts derived from the GENIA corpus for named entity recognition. It defines five entity types: `DNA`, `RNA`, `protein`, `cell_type`, and `cell_line`. The shared task paper [https://aclanthology.org/W04-1213/] describes the corpus and annotation task. OpenNLP identifies this input format as `bionlp2004`.

Getting the data

A copy of the original training and evaluation files is available in the `original-data` directory of the JNLPBA corpus repository [https://github.com/spyysalo/jnlpba]. The corpus is not distributed with OpenNLP. Review the terms supplied with the corpus before using or redistributing it.

Input format

The converter reads UTF-8 text with one token and one IOB tag per line, separated by a tab. A blank line ends a sentence. `B-type` starts an entity, `I-type` continues it, and `O` marks a token outside an entity. Lines beginning with `###MEDLINE:` mark document boundaries in the original corpus. For example:

```
IL-2 B-DNA
receptor I-DNA
activates O
T B-cell_type
cells I-cell_type
. O
```

Converting the data

Pass the desired entity types to `-types` as a comma-separated list. The values are case-sensitive and must use the names shown above. This command selects all five types and writes native OpenNLP name finder samples to `jnlpba.train`:

```
$ opennlp TokenNameFinderConverter bionlp2004 \
  -data original-data/train/Genia4ERtask1.iob2 \
  -types DNA,protein,cell_type,cell_line,RNA \
  > jnlpba.train
```

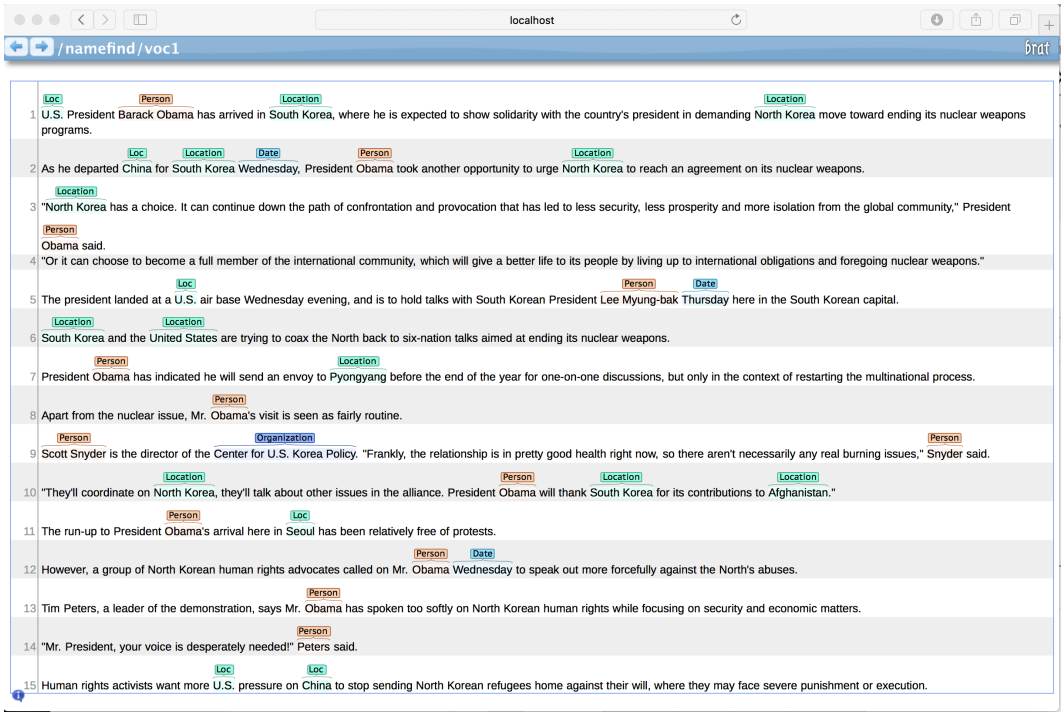
The input example from the previous section is converted to:

```
<START:DNA> IL-2 receptor <END> activates <START:cell_type> T cells <END> .
```

To convert only a subset, list only those types, for example `-types protein,cell_type`. The resulting file can be passed to the standard **TokenNameFinderTrainer**. See the section called “Name Finder Training” for training parameters and data recommendations.

Brat Format Support

The brat annotation tool is an online environment for collaborative text annotation and supports labeling documents with named entities. The best performance of a name finder can only be achieved if it was trained on documents similar to the documents it will process. For that reason it is often necessary to manually label a large number of documents and build a custom corpus. This is where brat comes in handy.



OpenNLP can directly be trained and evaluated on labeled data in the brat format. Instructions on how to use, download and install brat can be found on the project website: https://brat.nlplab.org Configuration of brat, including setting up the different entities and relations can be found at: https://brat.nlplab.org/configuration.html

Training

To train your namefinder model using your brat annotated files you can either use the opennlp command line tool or call opennlp.tools.cmdline.CLI main class from your preferred IDE. Calling opennlp TokenNameFinder.brat without arguments gives you a list of all the arguments you can use. Obviously some combinations are not valid. E.g. you should not provide a token model and also define a rule based tokenizer.

```
$ opennlp TokenNameFinderTrainer.brat
Usage: opennlp TokenNameFinderTrainer.brat [-factory factoryName] [-resources resourcesDir] [-type modelType]
[-featuregen featuregenFile] [-nameTypes types] [-sequenceCodec codec] [-params paramsFile] -lang language
-model modelFile [-tokenizerModel modelFile] [-ruleBasedTokenizer name] -annotationConfig annConfFile
-bratDataDir bratDataDir [-recursive value] [-sentenceDetectorModel modelFile]

Arguments description:
  -factory factoryName
    A sub-class of TokenNameFinderFactory
  -resources resourcesDir
    The resources directory
  -type modelType
    The type of the token name finder model
  -featuregen featuregenFile
    The feature generator descriptor file
  -nameTypes types
    name types to use for training
  -sequenceCodec codec
    sequence codec used to code name spans
  -params paramsFile
    training parameters file.
  -lang language
    language which is being processed.
  -model modelFile
    output model file.
  -tokenizerModel modelFile
  -ruleBasedTokenizer name
  -annotationConfig annConfFile
  -bratDataDir bratDataDir
    location of brat data dir
  -recursive value
  -sentenceDetectorModel modelFile
```

The following command will train a danish organization name finder model.

```
$ opennlp TokenNameFinderTrainer.brat -resources conf/resources \
-featuregen conf/resources/fg-da-org.xml -nameTypes Organization \
-params conf/resources/TrainerParams.txt -lang da \
-model models/da-org.bin -ruleBasedTokenizer simple \
-annotationConfig data/annotation.conf -bratDataDir data/gold/da/train \
-recursive true -sentenceDetectorModel models/da-sent.bin

Indexing events using cutoff of 0

Computing event counts...
done. 620738 events
Indexing...  done.
Collecting events... Done indexing.
Incorporating indexed data for training...
done.
 Number of Event Tokens: 620738
     Number of Outcomes: 3
   Number of Predicates: 1403655
Computing model parameters...
Performing 100 iterations.
  1:  . (614536/620738) 0.9900086671027067
  2:  . (617590/620738) 0.9949286172265915
  3:  . (618615/620738) 0.9965798775006525
  4:  . (619263/620738) 0.9976237961909856
  5:  . (619509/620738) 0.9980200986567602
  6:  . (619830/620738) 0.9985372250450271
  7:  . (619968/620738) 0.9987595410624128
  8:  . (620110/620738) 0.9989883010223315
  9:  . (620200/620738) 0.9991332897293222
 10:  . (620266/620738) 0.9992396147811153
 20:  . (620538/620738) 0.999677802873354
 30:  . (620641/620738) 0.9998437343935767
 40:  . (620653/620738) 0.9998630662211755
Stopping: change in training set accuracy less than 1.0E-5
Stats: (620594/620738) 0.9997680180688149
...done.

Writing name finder model ... Training data summary:
#Sentences: 26133
#Tokens: 620738
#Organization entities: 13053

Compressed 1403655 parameters to 116378
4 outcome patterns
done (11.099s)

Wrote name finder model to
path: models/da-org.bin
```

Evaluation

To evaluate you name finder model opennlp provides an evaluator that works with your brat annotated data. Normally you would partition your data in a training set and a test set e.g. 70% training and 30% test. The training set is of cause only used for training the model and should never be used for evaluation. The test set is only used for evaluation. In order to avoid overfitting, it is preferable if the training set and test set is somewhat balanced so that both sets represents a broad variety of the entities it should be able to identify. Shuffling the data before splitting is most likely sufficient in many cases.

```
$ opennlp TokenNameFinderEvaluator.brat -model models/da-org.bin \
-ruleBasedTokenizer simple -annotationConfig data/annotation.conf \
-bratDataDir data/gold/da/test -recursive true \
-sentenceDetectorModel models/da-sent.bin

Loading Token Name Finder model ... done (12.395s)

Average: 610.7 sent/s
Total: 6133 sent
Runtime: 10.043s

Precision: 0.7321974661424203
Recall: 0.25176505933603727
F-Measure: 0.3746926000447127
```

Cross Validation

You can also use the cross validation to evaluate you model. This can come in handy when you do not have enough data to divide it into a proper training and test set. Running cross validation with the misclassified attribute set to true can also be helpful because it will identify missed annotations as they will pop up as false positives in the text output.

```
$ opennlp TokenNameFinderCrossValidator.brat -resources conf/resources \
-featuregen conf/resources/fg-da-org.xml -nameTypes Organization \
-params conf/resources/TrainerParams.txt -lang da -misclassified true \
-folds 10 -detailedF true -ruleBasedTokenizer simple -annotationConfig data/annotation.conf \
-bratDataDir data/gold/da -recursive true -sentenceDetectorModel models/da-sent.bin

Indexing events using cutoff of 0

Computing event counts...
done. 555858 events
   Indexing...  done.
Collecting events... Done indexing.
Incorporating indexed data for training...
done.
 Number of Event Tokens: 555858
     Number of Outcomes: 3
   Number of Predicates: 1302740
Computing model parameters...
Performing 100 iterations.
  1:  . (550095/555858) 0.9896322442062541
  2:  . (552971/555858) 0.9948062274897546
...
...
... (training and evaluating x 10)
...
done

Evaluated 26133 samples with 13053 entities; found: 12174 entities; correct: 10361.
       TOTAL: precision:   85.11%;  recall:   79.38%; F1:   82.14%.
Organization: precision:   85.11%;  recall:   79.38%; F1:   82.14%. [target: 13053; tp: 10361; fp: 1813]
```

MASC

The Manually Annotated Sub-Corpus [https://anc.org/data/masc/] (MASC) of the Open American National Corpus is about 500,000 words of American English with hand-validated annotations. Each document is a text file and a header file (`.hdr`) that names its standoff annotation files in the GrAF XML format: sentences, word segmentation, Penn Treebank tokens with part-of-speech tags, and named entities. OpenNLP reads them through the `masc` format of the sentence detector, tokenizer, POS tagger and name finder tools. The reader looks for header files in the given directory, recursively when requested, reads only the files with the filter string in their name, and logs and skips a document it cannot read.

For example, to train a sentence detector on the written part of the corpus:

```
$ opennlp SentenceDetectorTrainer.masc -lang en -model en-sent-masc.bin \
-data masc/data/written -recurrentSearch true -fileFilter "" -sentencesPerSample 1
```

An identifier is the prefix `seg-r`, `penn-n` or `ne-n` followed by ASCII digits, and a region has two nonnegative offsets in order, each written with ASCII digits only and no sign. Lists of identifiers and offsets are separated by space, tab, carriage return or line feed. A document with another value in an annotation is skipped, and the log names the value.

Chapter 24. Machine Learning

Maximum Entropy

To explain what maximum entropy is, it will be simplest to quote from Manning and Schütze* (p. 589): “ Maximum entropy modeling is a framework for integrating information from many heterogeneous information sources for classification. The data for a classification problem is described as a (potentially large) number of features. These features can be quite complex and allow the experimenter to make use of prior knowledge about what types of information are expected to be important for classification. Each feature corresponds to a constraint on the model. We then compute the maximum entropy model, the model with the maximum entropy of all the models that satisfy the constraints. This term may seem perverse, since we have spent most of the book trying to minimize the (cross) entropy of models, but the idea is that we do not want to go beyond the data. If we chose a model with less entropy, we would add `information' constraints to the model that are not justified by the empirical evidence available to us. Choosing the maximum entropy model is motivated by the desire to preserve as much uncertainty as possible. ”

So that gives a rough idea of what the maximum entropy framework is. Don't assume anything about your probability distribution other than what you have observed.

On the engineering level, using maxent is an excellent way of creating programs which perform very difficult classification tasks very well. For example, precision and recall figures for programs using maxent models have reached (or are) the state of the art on tasks like part of speech tagging, sentence detection, prepositional phrase attachment, and named entity recognition. On the engineering level, an added benefit is that the person creating a maxent model only needs to inform the training procedure of the event space, and need not worry about independence between features.

While the authors of this implementation of maximum entropy are generally interested using maxent models in natural language processing, the framework is certainly quite general and useful for a much wider variety of fields. In fact, maximum entropy modeling was originally developed for statistical physics.

For a very in-depth discussion of how maxent can be used in natural language processing, try reading Adwait Ratnaparkhi's dissertation. Also, check out Berger, Della Pietra, and Della Pietra's paper A Maximum Entropy Approach to Natural Language Processing, which provides an excellent introduction and discussion of the framework.

*Foundations of statistical natural language processing. Christopher D. Manning, Hinrich Schütze. Cambridge, Mass.: MIT Press, c1999.

Implementation

We have tried to make the opennlp.maxent implementation easy to use. To create a model, one needs (of course) the training data, and then implementations of two interfaces in the opennlp.maxent package, EventStream and ContextGenerator. These have fairly simple specifications, and example implementations can be found in the OpenNLP Tools preprocessing components.

`BasicContextGenerator` turns one string of contextual predicates into the predicate array. Without arguments, runs of Unicode whitespace (the White_Space property, which includes no-break and ideographic spaces) separate predicates, independent of the `opennlp.whitespace.mode` setting. A separator given to the constructor is literal text, not a regular expression, and a separator containing a backslash is rejected with an `IllegalArgumentException`. Empty predicates are not returned.

Table 24.1. Separators written for the regular expression of 2.x

Separator in 2.x	In 3.x
`"\\|", "\\."`	`"|", "."`
`"\\s+", " +"`	the constructor without arguments

Separator in 2.x	In 3.x
`"[,;]"`	one separator, `","` or `";"`

We have also set in place some interfaces and code to make it easier to automate the training and evaluation process (the Evaluable interface and the TrainEval class). It is not necessary to use this functionality, but if you do you'll find it much easier to see how well your models are doing. The opennlp.grok.preprocess.namefind package is an example of a maximum entropy component which uses this functionality.

`FileEventStream` reads events from a text file, one event per line, and `RealValueFileEventStream` and `RealBasicEventStream` read events with real-valued features. The first field of a line is the outcome and each further field is a context, optionally followed by `=` and its value after the last `=`. Space, tab, carriage return, line feed and form feed separate fields; other characters, including no-break spaces, stay inside feature names. A line with only an outcome is an event without contexts, and a blank line is an error. For example:

```
other wc=ic=1.0 w&c=he,ic=2.0 n1wc=lc=3.0
org-start wc=ic=1.0 w&c=apache,ic=2.0 p1wc=lc=3.0
```

`SimpleEventStreamBuilder` takes `outcome/context1 context2` or `outcome/context1;value1 context2;value2`.

We have managed to use several techniques to reduce the size of the models when writing them to disk, which also means that reading in a model for use is much quicker than with less compact encodings of the model. This was especially important to us since we use many maxent models in the Grok library, and we wanted the start up time and the physical size of the library to be as minimal as possible. As of version 1.2.0, maxent has an io package which greatly simplifies the process of loading and saving models in different formats.

Text embedding providers

`TextEmbedder` in `opennlp-api` accepts text and returns vectors. `TextEmbedderProvider` is its provider interface, resolved through `Providers` as described in the section called “Service providers”. The API names no backend: `opennlp-dl` registers the `onnx` provider. The binary distribution does not include the `opennlp-dl` or `opennlp-dl-gpu` jars, and `opennlp-tools` does not depend on them, so add one of them to the application to use it.

An embedder is safe for concurrent use until it is closed. `embedAll` checks every text before embedding any of them and forms its own batches, so it accepts a list of any size. No inference of `SentenceVectorsDL` exceeds 16,384 token positions, except that a single text whose encoding is longer runs as an inference of its own. Inference failures are reported as `EmbeddingException`.

```
ProviderSpec spec = ProviderSpec.of(Path.of("model.onnx"),
    Map.of("vocabulary", "vocab.txt"));
Providers<TextEmbedderProvider> providers = Providers.of(TextEmbedderProvider.class);
try (TextEmbedder embedder = providers.select(spec).create(spec)) {
  float[][] vectors = embedder.embedAll(List.of("first sentence", "second sentence"));
}
```

The `onnx` provider supports a local model file named `*.onnx`. It requires the option `vocabulary`, resolved against the directory of the model if it is relative, which names a `vocab.txt`, a `vocab.json`, or the `tokenizer.json` of a WordPiece model, as described for the document categorizer in the section called “Using an ONNX Model”, and accepts `lowerCase=true` (default) or `lowerCase=false`, `pooling=mean` (default) or `pooling=cls`, `normalize=true` (default) or `normalize=false`, and `maxLength`, the maximum number of tokens per input (512 by default); longer input is truncated. The option `padding` sets how `embedAll` batches texts of different lengths. Values are accepted in any letter case. `auto` (default) selects the device and model default; the provider uses a CPU session, so it selects `exact_length`. `exact_length` does not pad and runs one inference per tokenized length, `longest` pads each batch to its longest text, and `max_length`

pads every text to `maxLength`, single `embed` calls included. If the output dimension is unknown, the constructor also probes at `maxLength`, which must fit the model's position limit. The vectors match up to floating-point rounding under every strategy, provided the model honors the attention mask; a `sentence_embedding` output that ignores the mask changes with padding. Padding needs a `[PAD]` token in the vocabulary, or `<pad>` for a RoBERTa vocabulary. The defaults match sentence-transformers models such as all-MiniLM-L6-v2. A model with a `sentence_embedding` output is used as it is, without pooling. A spec with another option is not supported. The key `opennlp.provider.opennlp.tools.embeddings.TextEmbedderProvider` selects a provider by name. Without `opennlp-dl` or `opennlp-dl-gpu` the API has no provider until another one is installed. Existing `SentenceVectorsDL` constructors remain available.

To compute the vectors on a GPU, depend on `opennlp-dl-gpu` in place of `opennlp-dl` and pass `InferenceOptions` to `SentenceVectorsDL`. `opennlp-dl-gpu` supports NVIDIA GPUs on linux-x64 and win-x64 only. On macOS, use `opennlp-dl`. The machine needs the CUDA and cuDNN versions that the ONNX Runtime release requires, see the CUDA execution provider requirements [https://onnxruntime.ai/docs/execution-providers/CUDA-ExecutionProvider.html#requirements]. The `onnx` provider spec has no device option, so an embedder created through it runs on the CPU. If CUDA or the configured device is not available, the constructor throws an `OrtException`; ONNX Runtime does not fall back to the CPU. A negative device id is rejected with an `IllegalArgumentException`. A lower case setting in the options takes precedence over the `lowerCase` argument. On a GPU, `embedAll` pads each batch to its longest text by default, if the vocabulary has a padding token, because one wide inference is faster there than many narrow ones; a model with a `sentence_embedding` output is not padded by default, because it may ignore the attention mask and then padding changes the vectors. Pass `SentenceVectorsOptions` to select pooling, normalization, the maximum token length and an explicit `PaddingStrategy`. `PaddingStrategy.AUTO` selects the device and model default; `null` is rejected. Padded batches can introduce small floating-point differences when the model honors the attention mask. Select `PaddingStrategy.EXACT_LENGTH` to retain unpadded batches on a GPU. `InferenceOptions` supplies the session settings separately.

```
InferenceOptions options = new InferenceOptions();
options.setGpu(true);
options.setGpuDeviceId(0);
SentenceVectorsOptions vectorOptions = new SentenceVectorsOptions();
try (SentenceVectorsDL vectors = new SentenceVectorsDL(model, vocab, true, vectorOptions, options)) {
  float[] vector = vectors.embed("first sentence");
}
```

Chapter 25. UIMA Integration

The UIMA Integration module wraps the OpenNLP components as UIMA Analysis Engines. These annotators can be used in any UIMA pipeline to automatically annotate text with sentences, tokens, named entities, part-of-speech tags, chunks, and parse trees. The module is located in the `opennlp-uima` artifact.

Adding the Dependency

To use the OpenNLP UIMA annotators, add the following dependency to your project:

```
<dependency>
  <groupId>org.apache.opennlp</groupId>
  <artifactId>opennlp-uima</artifactId>
  <version>${opennlp.version}</version>
</dependency>
```

This module depends on Apache UIMA and the OpenNLP runtime. The UIMA framework dependency is included transitively.

Type System

The module ships with a default type system defined in `TypeSystem.xml` inside the descriptors directory. This type system defines the following annotation types:

- `opennlp.uima.Sentence` - Sentence boundary annotations
- `opennlp.uima.Token` - Token annotations with a `pos` feature for part-of-speech tags
- `opennlp.uima.Chunk` - Chunk annotations with a `chunkType` feature (e.g. NP, VP)
- `opennlp.uima.Person | Organization | Location` - Named entity types
- `opennlp.uima.Date | Time | Money | Percentage` - Additional named entity types
- `opennlp.uima.Parse` - Parse tree node annotations with `parseType`, `children`, and `prob` features

The default type system can be replaced with a custom type system. To do so, update the type references in the analysis engine descriptors to point to your custom types and import your custom type system instead of `TypeSystem.xml`.

Descriptor Structure

Each OpenNLP UIMA annotator is configured through a UIMA analysis engine descriptor XML file. It specifies:

- The annotator implementation class
- Configuration parameters (e.g. which type system types to use)
- An external resource dependency for the OpenNLP model file
- A reference to the type system

Models are loaded through the UIMA external resource mechanism. Each ML-based annotator declares a dependency on a model resource with the key `opennlp.uima.ModelName`. The model file is bound to this key through the resource manager configuration. For example, to configure the sentence detector model:

```
<externalResourceDependencies>
  <externalResourceDependency>
    <key>opennlp.uima.ModelName</key>
    <interfaceName>opennlp.uima.sentdetect.SentenceModelResource</interfaceName>
  </externalResourceDependency>
</externalResourceDependencies>

<resourceManagerConfiguration>
  <externalResources>
    <externalResource>
      <name>SentenceModel</name>
      <fileResourceSpecifier>
        <fileUrl>file:en-sent.bin</fileUrl>
      </fileResourceSpecifier>
      <implementationName>opennlp.uima.sentdetect.SentenceModelResourceImpl</implementationName>
    </externalResource>
  </externalResources>
  <externalResourceBindings>
    <externalResourceBinding>
      <key>opennlp.uima.ModelName</key>
      <resourceName>SentenceModel</resourceName>
    </externalResourceBinding>
  </externalResourceBindings>
</resourceManagerConfiguration>
```

Sentence Detector

The `opennlp.uima.sentdetect.SentenceDetector` annotator detects sentence boundaries and creates sentence annotations in the CAS.

Configuration Parameters:

- `opennlp.uima.SentenceType` (mandatory) - The full name of the sentence annotation type. Default: `opennlp.uima.Sentence`
- `opennlp.uima.ContainerType` (optional) - If set, sentence detection is restricted to within annotations of this type. Useful for detecting sentences only inside specific regions of a document (e.g. paragraphs).
- `opennlp.uima.ProbabilityFeature` (optional) - Feature name for storing the detection confidence score on each sentence annotation.

Model Resource Interface: `opennlp.uima.sentdetect.SentenceModelResource`

Example Descriptor: See `descriptors/SentenceDetector.xml` in the opennlp-uima module.

Tokenizer

Three tokenizer implementations are available as UIMA annotators. All tokenizers require sentence annotations to already be present in the CAS.

Learnable Tokenizer

The `opennlp.uima.tokenize.Tokenizer` annotator uses a maximum entropy model to identify token boundaries.

Configuration Parameters:

- `opennlp.uima.SentenceType` (mandatory) - The sentence annotation type. Default: `opennlp.uima.Sentence`
- `opennlp.uima.TokenType` (mandatory) - The token annotation type. Default: `opennlp.uima.Token`
- `opennlp.uima.tokenizer.IsAlphaNumericOptimization` (optional) - If set, enables an optimization that treats purely alphanumeric sequences as single tokens without consulting the model.
- `opennlp.uima.ProbabilityFeature` (optional) - Feature name for storing token probability scores.

Model Resource Interface: `opennlp.uima.tokenize.TokenizerModelResource`

Simple Tokenizer

The `opennlp.uima.tokenize.SimpleTokenizer` annotator is a rule-based tokenizer that splits text by character class boundaries. It requires no model.

Configuration Parameters:

- `opennlp.uima.SentenceType` (mandatory) - The sentence annotation type.
- `opennlp.uima.TokenType` (mandatory) - The token annotation type.

Whitespace Tokenizer

The `opennlp.uima.tokenize.WhitespaceTokenizer` annotator splits text at whitespace boundaries. It requires no model.

Configuration Parameters:

- `opennlp.uima.SentenceType` (mandatory) - The sentence annotation type.
- `opennlp.uima.TokenType` (mandatory) - The token annotation type.

Name Finder

Two named entity recognition annotators are provided: a machine learning-based annotator and a dictionary-based annotator. Both require sentence and token annotations to already be present in the CAS.

Learnable Name Finder

The `opennlp.uima.namefind.NameFinder` annotator uses a maximum entropy model to detect named entities such as person names, organizations, and locations.

Configuration Parameters:

- `opennlp.uima.SentenceType` (mandatory) - The sentence annotation type.
- `opennlp.uima.TokenType` (mandatory) - The token annotation type.
- `opennlp.uima.NameType` (mandatory) - The annotation type for detected entities (e.g. `opennlp.uima.Person`).
- `opennlp.uima.ProbabilityFeature` (optional) - Feature name for storing entity probability scores.
- `opennlp.uima.BeamSize` (optional) - Beam size for the beam search.
- `opennlp.uima.DocumentConfidenceType` (optional) - Annotation type for storing document-level confidence information.

Model Resource Interface: `opennlp.uima.namefind.TokenNameFinderModelResource`

To detect multiple entity types, configure one Name Finder annotator per entity type, each with its own model. The provided descriptors include pre-configured annotators for person, organization, location, date, time, money, and percentage entities.

Dictionary Name Finder

The `opennlp.uima.namefind.DictionaryNameFinder` annotator performs dictionary-based named entity recognition. It matches token sequences against entries in an OpenNLP dictionary file. No machine learning model is required.

Configuration Parameters:

- `opennlp.uima.SentenceType` (mandatory) - The sentence annotation type.
- `opennlp.uima.TokenType` (mandatory) - The token annotation type.
- `opennlp.uima.NameType` (mandatory) - The annotation type for detected entities.
- `opennlp.uima.Dictionary` (mandatory) - External resource key for the OpenNLP dictionary file to use for matching.

POS Tagger

The `opennlp.uima.postag.POSTagger` annotator assigns part-of-speech tags to tokens. It requires sentence and token annotations to already be present in the CAS.

Configuration Parameters:

- `opennlp.uima.SentenceType` (mandatory) - The sentence annotation type. Default: `opennlp.uima.Sentence`
- `opennlp.uima.TokenType` (mandatory) - The token annotation type. Default: `opennlp.uima.Token`
- `opennlp.uima.POSFeature` (mandatory) - The feature name on the token type where the POS tag will be stored. Default: `pos`
- `opennlp.uima.ProbabilityFeature` (optional) - Feature name for storing tagging probability scores.
- `opennlp.uima.BeamSize` (optional) - Beam size for the beam search.
- `opennlp.uima.DictionaryName` (optional) - External resource key for a tag dictionary that constrains possible tags for known words.

Model Resource Interface: `opennlp.uima.postag.POSModelResource`

Chunker

The `opennlp.uima.chunker.Chunker` annotator identifies non-recursive syntactic phrases (chunks) such as noun phrases (NP) and verb phrases (VP). It requires sentence and token annotations with POS tags to already be present in the CAS.

Configuration Parameters:

- `opennlp.uima.SentenceType` (mandatory) - The sentence annotation type. Default: `opennlp.uima.Sentence`
- `opennlp.uima.TokenType` (mandatory) - The token annotation type. Default: `opennlp.uima.Token`
- `opennlp.uima.POSFeature` (mandatory) - The feature name for reading POS tags from tokens. Default: `pos`

- `opennlp.uima.ChunkType` (mandatory) - The annotation type for chunk annotations. Default: `opennlp.uima.Chunk`
- `opennlp.uima.ChunkTagFeature` (mandatory) - The feature name on the chunk type where the chunk tag (e.g. NP, VP) will be stored. Default: `chunkType`
- `opennlp.uima.BeamSize` (optional) - Beam size for the beam search.

Model Resource Interface: `opennlp.uima.chunker.ChunkerModelResource`

Parser

The `opennlp.uima.parser.Parser` annotator performs full syntactic parsing and creates a hierarchical parse tree structure in the CAS. It requires sentence and token annotations to already be present in the CAS.

Configuration Parameters:

- `opennlp.uima.SentenceType` (mandatory) - The sentence annotation type. Default: `opennlp.uima.Sentence`
- `opennlp.uima.TokenType` (mandatory) - The token annotation type. Default: `opennlp.uima.Token`
- `opennlp.uima.ParseType` (mandatory) - The annotation type for parse tree nodes. Default: `opennlp.uima.Parse`
- `opennlp.uima.TypeFeature` (mandatory) - The feature name for storing the parse node type (e.g. S, NP, VP). Default: `parseType`
- `opennlp.uima.ChildrenFeature` (mandatory) - The feature name for storing references to child parse nodes. Default: `children`
- `opennlp.uima.ProbabilityFeature` (optional) - Feature name for storing parse probability scores. Default: `prob`
- `opennlp.uima.BeamSize` (optional) - Beam size for the beam search.

Model Resource Interface: `opennlp.uima.parser.ParserModelResource`

Document Categorizer

The `opennlp.uima.doccat.DocumentCategorizer` annotator classifies document text into categories using a trained document categorization model.

Configuration Parameters:

- `opennlp.uima.doccat.CategoryType` (mandatory) - The annotation type for the category result.
- `opennlp.uima.doccat.CategoryFeature` (mandatory) - The feature name on the category type where the classification result is stored.

Model Resource Interface: `opennlp.uima.doccat.DoccatModelResource`

Language Detector

The `opennlp.uima.doccat.LanguageDetector` annotator identifies the language of the document text and sets the CAS document language accordingly.

Configuration Parameters:

- `opennlp.uima.SentenceType` (mandatory) - The sentence annotation type. Default: `opennlp.uima.Sentence`

Model Resource Interface: `opennlp.uima.doccat.DoccatModelResource`

Example Descriptor: See `descriptors/LanguageDetector.xml` in the opennlp-uima module.

Normalizer

The `opennlp.uima.normalizer.Normalizer` annotator extracts structured data from named entity annotations. It can convert the covered text of a named entity into typed values (e.g. parsing a money amount into a numeric value) and optionally look up normalized forms in a dictionary.

Configuration Parameters:

- `opennlp.uima.NameType` (mandatory) - The named entity annotation type to normalize.
- `opennlp.uima.normalizer.StructureFeature` (mandatory) - The feature name where the normalized value is stored.
- `opennlp.uima.Dictionary` (optional) - External resource key for a dictionary used to look up normalized forms.

The normalizer supports the following target feature types: `String`, `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double`. Number parsing is locale-aware and uses the CAS document language.

Building an Aggregate Pipeline

The annotators are designed to be composed into an aggregate analysis engine where each annotator builds on the annotations produced by earlier ones. The standard processing order is:

- Sentence Detector (produces sentence annotations)
- Tokenizer (produces token annotations within sentences)
- Name Finders (produce entity annotations from tokens)
- POS Tagger (adds POS tags to tokens)
- Chunker (produces chunk annotations from POS-tagged tokens)
- Parser (produces parse tree from tokens within sentences)

The module includes a pre-configured aggregate descriptor `descriptors/OpenNlpTextAnalyzer.xml` that chains sentence detection, tokenization, multiple name finders (person, organization, location, date, time, money, percentage), POS tagging, chunking, and parsing in the correct order.

This aggregate descriptor demonstrates how to bind models for all annotators in one place using the resource manager configuration. Each annotator's model key follows the pattern `AnnotatorKey/opennlp.uima.ModelName`, for example:

```
<externalResourceBinding>
  <key>SentenceDetector/opennlp.uima.ModelName</key>
  <resourceName>SentenceModel</resourceName>
</externalResourceBinding>
<externalResourceBinding>
  <key>Tokenizer/opennlp.uima.ModelName</key>
  <resourceName>TokenModel</resourceName>
</externalResourceBinding>
```

Below is a complete example showing how to create and run an aggregate pipeline programmatically using the UIMA framework APIs:

```
// Load the aggregate analysis engine descriptor
XMLInputSource in = new XMLInputSource("descriptors/OpenNlpTextAnalyzer.xml");
ResourceSpecifier specifier = UIMAFramework.getXMLParser()
    .parseResourceSpecifier(in);

// Create the analysis engine
AnalysisEngine ae = UIMAFramework.produceAnalysisEngine(specifier);

// Create a CAS and set the document text
CAS cas = ae.newCAS();
cas.setDocumentText("Pierre Vinken, 61 years old, will join the board "
    + "as a nonexecutive director Nov. 29. Mr. Vinken is chairman "
    + "of Elsevier N.V., the Dutch publishing group.");
cas.setDocumentLanguage("en");

// Run the pipeline
ae.process(cas);

// Iterate over detected sentences
Type sentenceType = cas.getTypeSystem().getType("opennlp.uima.Sentence");
for (AnnotationFS sentence : cas.getAnnotationIndex(sentenceType)) {
  System.out.println("Sentence: " + sentence.getCoveredText());
}

// Iterate over detected tokens
Type tokenType = cas.getTypeSystem().getType("opennlp.uima.Token");
Feature posFeature = tokenType.getFeatureByBaseName("pos");
for (AnnotationFS token : cas.getAnnotationIndex(tokenType)) {
  System.out.println("Token: " + token.getCoveredText()
      + " POS: " + token.getStringValue(posFeature));
}

// Iterate over detected person names
Type personType = cas.getTypeSystem().getType("opennlp.uima.Person");
for (AnnotationFS person : cas.getAnnotationIndex(personType)) {
  System.out.println("Person: " + person.getCoveredText());
}

// Clean up
cas.release();
ae.destroy();
```

Using Custom Type Systems

The default type system can be replaced with your own custom types. This is useful when integrating OpenNLP annotators into an existing UIMA pipeline that already defines its own type system.

To use custom types:

- Create your own type system descriptor with the annotation types you need.
- Update the annotator descriptor to import your custom type system instead of `TypeSystem.xml`.
- Set the configuration parameters (e.g. `opennlp.uima.SentenceType`, `opennlp.uima.TokenType`) to reference your custom type names.

For example, if your type system defines sentences as `my.types.Sentence` and tokens as `my.types.Token`, update the descriptor:

```
<configurationParameterSettings>
  <nameValuePair>
    <name>opennlp.uima.SentenceType</name>
    <value>
      <string>my.types.Sentence</string>
    </value>
  </nameValuePair>
  <nameValuePair>
    <name>opennlp.uima.TokenType</name>
    <value>
      <string>my.types.Token</string>
    </value>
  </nameValuePair>
</configurationParameterSettings>
```

Running the PEAR Sample in CVD

The CAS Visual Debugger (CVD) is shipped as part of the UIMA distribution and is a tool which can run the OpenNLP UIMA Annotators and display their analysis results. The source distribution comes with a script which can create a sample UIMA application. This includes the sentence detector, tokenizer, POS tagger, chunker, and name finders for English. This sample application is packaged in the PEAR format and must be installed with the PEAR installer before it can be run by CVD. Please consult the UIMA documentation for further information about the PEAR installer.

After the PEAR is installed, start the CAS Visual Debugger shipped with the UIMA framework and click on Tools -> Load AE. Then select the `opennlp.uima.OpenNlpTextAnalyzer_pear.xml` file in the file dialog. Now enter some text and start the analysis engine with "Run -> Run OpenNLPTextAnalyzer". Afterwards the results will be displayed. You should see sentences, tokens, chunks, POS tags, and possibly some named entities. Remember the input text must be written in English.

Chapter 26. Morfologik Addon

Morfologik [https://github.com/morfologik/morfologik-stemming] provides tools for finite state automata (FSA) construction and dictionary-based morphological dictionaries.

The Morfologik Addon implements OpenNLP interfaces and extensions to allow the use of FSA Morfologik dictionary tools.

Morfologik Integration

To allow for an easy integration with OpenNLP, the following implementations are provided:

- The `MorfologikPOSTaggerFactory` extends `POSTaggerFactory`, which helps create a POSTagger model with an embedded FSA TagDictionary.
- The `MorfologikTagDictionary` implements an FSA based `TagDictionary`, allowing for much smaller files than the default XML based with improved memory consumption.
- The `MorfologikLemmatizer` implements an FSA based `Lemmatizer` dictionaries.

`MorfologikTagDictionary` supports concurrent lookups on a shared instance. Each thread keeps its own lookup; call `clearThreadLocalState()` when a pooled thread is done with the dictionary. Entries without a POS tag are ignored; a word with no tagged entries returns `null`.

The first two implementations can be used directly from command line, as in the example bellow. Having an FSA Morfologik dictionary (see next section how to build one), you can train a POS Tagger model with an embedded FSA dictionary.

The example trains a POSTagger with a CONLL corpus named `portuguese_bosque_train.conll` and an FSA dictionary named `pt-morfologik.dict`. It will output a model named `pos-pt_fsadic.model`.

```
$ bin/opennlp POSTaggerTrainer -type perceptron -lang pt -model pos-pt_fsadic.model \
                -data portuguese_bosque_train.conll -encoding UTF-8 \
                -factory opennlp.morfologik.tagdict.MorfologikPOSTaggerFactory \
                -dict pt-morfologik.dict
```

Another example follows. It shows how to use the `MorfologikLemmatizer`. You will need a lemma dictionary and info file, in this example, we will use a very small Portuguese dictionary. Its syntax is `lemma,lexeme,postag`.

File `lemmaDictionary.txt`:

```
casa,casa,NOUN
casar,casa,V
casar,casar,V-INF
Casa,Casa,PROP
casa,casinha,NOUN
casa,casona,NOUN
menino,menina,NOUN
menino,menino,NOUN
menino,meninão,NOUN
menino,menininho,NOUN
carro,carro,NOUN
```

Mandatory metadata file, which must have the same name but .info extension `lemmaDictionary.info`:

```
#
# REQUIRED PROPERTIES
#

# Column (lemma, inflected, tag) separator. This must be a single byte in the target encoding.
fsa.dict.separator=,

# The charset in which the input is encoded. UTF-8 is strongly recommended.
fsa.dict.encoding=UTF-8

# The type of lemma-inflected form encoding compression that precedes automaton
# construction. Allowed values: [suffix, infix, prefix, none].
# Details are in Daciuk's paper and in the code.
# Leave at 'prefix' if not sure.
fsa.dict.encoder=prefix
```

The following code creates a binary FSA Morfologik dictionary, loads it in MorfologikLemmatizer and uses it to find the lemma the word "casa" noun and verb.

```
// Part 1: compile an FSA lemma dictionary

// we need the tabular dictionary. It is mandatory to have info
//  file with same name, but .info extension
Path textLemmaDictionary = Paths.get("dictionaryWithLemma.txt");

// this will build a binary dictionary located in compiledLemmaDictionary
Path compiledLemmaDictionary = new MorfologikDictionayBuilder()
    .build(textLemmaDictionary);

// Part 2: load a MorfologikLemmatizer and use it
MorfologikLemmatizer lemmatizer = new MorfologikLemmatizer(compiledLemmaDictionary);

String[] toks = {"casa", "casa"};
String[] tags = {"NOUN", "V"};

String[] lemmas = lemmatizer.lemmatize(toks, tags);
System.out.println(Arrays.toString(lemmas)); // outputs [casa, casar]
```

Morfologik CLI Tools

The Morfologik addon provides a command line tool. `XMLDictionaryToTable` makes easy to convert from an OpenNLP XML based dictionary to a tabular format. `MorfologikDictionaryBuilder` can take a tabular dictionary and output a binary Morfologik FSA dictionary.

```
$ sh bin/morfologik-addon
OpenNLP Morfologik Addon. Usage: opennlp-morfologik-addon TOOL
where TOOL is one of:
  MorfologikDictionaryBuilder    builds a binary POS Dictionary using Morfologik
  XMLDictionaryToTable           reads an OpenNLP XML tag dictionary and outputs it in a tabular file
All tools print help when invoked with help parameter
Example: opennlp-morfologik-addon POSDictionaryBuilder help
```

Chapter 27. The Command Line Interface

This section details the available tools and parameters of the Command Line Interface. For an introduction in its usage please refer to the section called “Command line interface (CLI)”.

Doccat

Doccat

Learned document categorizer

```
Usage: opennlp Doccat model < documents
```

DoccatTrainer

Trainer for the learnable document categorizer

```
Usage: opennlp DoccatTrainer[.leipzig] [-factory factoryName] [-featureGenerators fg] [-tokenizer tokenizer]
        [-params paramsFile] -lang language -model modelFile -data sampleData [-encoding charsetName]
  Arguments description:
    -factory factoryName
      A sub-class of DoccatFactory where to get implementation and resources.
    -featureGenerators fg
      Comma separated feature generator classes. Bag of words is used if not specified.
    -tokenizer tokenizer
      Tokenizer implementation. WhitespaceTokenizer is used if not specified.
    -params paramsFile
      training parameters file.
    -lang language
      language which is being processed.
    -model modelFile
      output model file.
    -data sampleData
      data to be used, usually a file name.
    -encoding charsetName
      encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
leipzig	sentencesDir	sentencesDir	No	Dir with Leipzig sentences to be used
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

DoccatEvaluator

Measures the performance of the Doccat model with the reference data

```
Usage: opennlp DoccatEvaluator[.leipzig] -model model [-misclassified true|false] [-reportOutputFile
        outputFile] -data sampleData [-encoding charsetName]
 Arguments description:
   -model model
     the model file to be evaluated.
   -misclassified true|false
     if true will print false negatives and false positives.
   -reportOutputFile outputFile
     the path of the fine-grained report file.
   -data sampleData
     data to be used, usually a file name.
   -encoding charsetName
     encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
leipzig	sentencesDir	sentencesDir	No	Dir with Leipzig sentences to be used
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

DoccatCrossValidator

K-fold cross validator for the learnable Document Categorizer

```
Usage: opennlp DoccatCrossValidator[.leipzig] [-misclassified true|false] [-folds num] [-factory factoryName]
        [-featureGenerators fg] [-tokenizer tokenizer] [-params paramsFile] -lang language [-reportOutputFile
        outputFile] -data sampleData [-encoding charsetName]
 Arguments description:
   -misclassified true|false
     if true will print false negatives and false positives.
   -folds num
     number of folds, default is 10.
   -factory factoryName
     A sub-class of DoccatFactory where to get implementation and resources.
   -featureGenerators fg
     Comma separated feature generator classes. Bag of words is used if not specified.
   -tokenizer tokenizer
     Tokenizer implementation. WhitespaceTokenizer is used if not specified.
   -params paramsFile
     training parameters file.
   -lang language
     language which is being processed.
   -reportOutputFile outputFile
     the path of the fine-grained report file.
   -data sampleData
     data to be used, usually a file name.
   -encoding charsetName
     encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
leipzig	sentencesDir	sentencesDir	No	Dir with Leipzig sentences to be used
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

DoccatConverter

Converts leipzig data format to native OpenNLP format

```
Usage: opennlp DoccatConverter help|leipzig [help|options...]
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
leipzig	sentencesDir	sentencesDir	No	Dir with Leipzig sentences to be used
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

Langdetect

LanguageDetector

Learned language detector

```
Usage: opennlp LanguageDetector model < documents
```

LanguageDetectorTrainer

Trainer for the learnable language detector

```
Usage: opennlp LanguageDetectorTrainer[.leipzig] -model modelFile [-params paramsFile] [-factory factoryName]
        -data sampleData [-encoding charsetName]
  Arguments description:
    -model modelFile
      output model file.
    -params paramsFile
      training parameters file.
    -factory factoryName
      A sub-class of LanguageDetectorFactory where to get implementation and resources.
    -data sampleData
      data to be used, usually a file name.
    -encoding charsetName
      encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
leipzig	sentencesDir	sentencesDir	No	Dir with Leipzig sentences to be used
	sentencesPerSample	sentencesPerSample	No	Number of sentences per sample
	samplesPerLanguage	samplesPerLanguage	No	Number of samples per language
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

LanguageDetectorConverter

Converts leipzig data format to native OpenNLP format

```
Usage: opennlp LanguageDetectorConverter help|leipzig [help|options...]
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
leipzig	sentencesDir	sentencesDir	No	Dir with Leipzig sentences to be used
	sentencesPerSample	sentencesPerSample	No	Number of sentences per sample
	samplesPerLanguage	samplesPerLanguage	No	Number of samples per language
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

LanguageDetectorCrossValidator

K-fold cross validator for the learnable Language Detector

```
Usage: opennlp LanguageDetectorCrossValidator[.leipzig] [-misclassified true|false] [-folds num] [-factory
        factoryName] [-params paramsFile] [-reportOutputFile outputFile] -data sampleData [-encoding
        charsetName]
 Arguments description:
   -misclassified true|false
     if true will print false negatives and false positives.
   -folds num
     number of folds, default is 10.
   -factory factoryName
     A sub-class of LanguageDetectorFactory where to get implementation and resources.
   -params paramsFile
     training parameters file.
   -reportOutputFile outputFile
     the path of the fine-grained report file.
   -data sampleData
     data to be used, usually a file name.
   -encoding charsetName
     encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
leipzig	sentencesDir	sentencesDir	No	Dir with Leipzig sentences to be used
	sentencesPerSample	sentencesPerSample	No	Number of sentences per sample
	samplesPerLanguage	samplesPerLanguage	No	Number of samples per language
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

LanguageDetectorEvaluator

Measures the performance of the Language Detector model with the reference data

```
Usage: opennlp LanguageDetectorEvaluator[.leipzig] -model model [-misclassified true|false]
        [-reportOutputFile outputFile] -data sampleData [-encoding charsetName]
 Arguments description:
   -model model
     the model file to be evaluated.
   -misclassified true|false
     if true will print false negatives and false positives.
   -reportOutputFile outputFile
     the path of the fine-grained report file.
   -data sampleData
     data to be used, usually a file name.
   -encoding charsetName
     encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
leipzig	sentencesDir	sentencesDir	No	Dir with Leipzig sentences to be used

Format	Argument	Value	Optional	Description
	sentencesPerSample	sentencesPerSample	No	Number of sentences per sample
	samplesPerLanguage	samplesPerLanguage	No	Number of samples per language
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

Dictionary

DictionaryBuilder

Builds a new dictionary

```
Usage: opennlp DictionaryBuilder -outputFile out -inputFile in [-encoding charsetName]

 Arguments description:
  -outputFile out
    The dictionary file.
  -inputFile in
    Plain file with one entry per line
  -encoding charsetName
    encoding for reading and writing text, if absent the system default is used.
```

Tokenizer

SimpleTokenizer

Character class tokenizer

```
Usage: opennlp SimpleTokenizer < sentences
```

TokenizerME

Learnable tokenizer

```
Usage: opennlp TokenizerME model < sentences
```

TokenizerTrainer

Trainer for the learnable tokenizer

```
Usage: opennlp TokenizerTrainer[.irishsentencebank|.ad|.pos|.conllx|.namefinder|.parse|.conllu] [-factory
        factoryName] [-abbDict path] [-alphaNumOpt isAlphaNumOpt] [-params paramsFile] -lang language -model
        modelFile -data sampleData [-encoding charsetName]
 Arguments description:
   -factory factoryName
     A sub-class of TokenizerFactory where to get implementation and resources.
   -abbDict path
     abbreviation dictionary in XML format.
   -alphaNumOpt isAlphaNumOpt
     Optimization flag to skip alpha numeric tokens for further tokenization
   -params paramsFile
     training parameters file.
   -lang language
     language which is being processed.
   -model modelFile
     output model file.
   -data sampleData
     data to be used, usually a file name.
   -encoding charsetName
     encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
irishsentencebank	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ad	encoding	charsetName	No	Encoding for reading and writing text, if absent the system default is used.
	lang	language	No	Language which is being processed.
	data	sampleData	No	Data to be used, usually a file name.
	splitHyphenatedTokens	split	Yes	If true all hyphenated tokens will be separated (default true)
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
pos	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.

Format	Argument	Value	Optional	Description
conllx	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
namefinder	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
parse	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
conllu	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

TokenizerMEEvaluator

Evaluator for the learnable tokenizer

```
Usage: opennlp TokenizerMEEvaluator[.irishsentencebank|.ad|.pos|.conllx|.namefinder|.parse|.conllu] -model
        model [-misclassified true|false] -data sampleData [-encoding charsetName]
 Arguments description:
   -model model
     the model file to be evaluated.
   -misclassified true|false
     if true will print false negatives and false positives.
   -data sampleData
     data to be used, usually a file name.
   -encoding charsetName
     encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
irishsentencebank	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ad	encoding	charsetName	No	Encoding for reading and writing text, if absent the system default is used.
	lang	language	No	Language which is being processed.
	data	sampleData	No	Data to be used, usually a file name.
	splitHyphenatedTokens	split	Yes	If true all hyphenated tokens will be separated (default true)
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
pos	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
conllx	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
namefinder	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
parse	data	sampleData	No	Data to be used, usually a file name.

Format	Argument	Value	Optional	Description
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
conllu	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

TokenizerCrossValidator

K-fold cross validator for the learnable tokenizer

```
Usage: opennlp TokenizerCrossValidator[.irishsentencebank|.ad|.pos|.conllx|.namefinder|.parse|.conllu]
        [-misclassified true|false] [-folds num] [-factory factoryName] [-abbDict path] [-alphaNumOpt
        isAlphaNumOpt] [-params paramsFile] -lang language -data sampleData [-encoding charsetName]
 Arguments description:
   -misclassified true|false
     if true will print false negatives and false positives.
   -folds num
     number of folds, default is 10.
   -factory factoryName
     A sub-class of TokenizerFactory where to get implementation and resources.
   -abbDict path
     abbreviation dictionary in XML format.
   -alphaNumOpt isAlphaNumOpt
     Optimization flag to skip alpha numeric tokens for further tokenization
   -params paramsFile
     training parameters file.
   -lang language
     language which is being processed.
   -data sampleData
     data to be used, usually a file name.
   -encoding charsetName
     encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
irishsentencebank	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ad	encoding	charsetName	No	Encoding for reading and writing text, if absent the system default is used.

Format	Argument	Value	Optional	Description
	lang	language	No	Language which is being processed.
	data	sampleData	No	Data to be used, usually a file name.
	splitHyphenatedTokens	split	Yes	If true all hyphenated tokens will be separated (default true)
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
pos	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
conllx	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
namefinder	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
parse	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
conllu	data	sampleData	No	Data to be used, usually a file name.

Format	Argument	Value	Optional	Description
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

TokenizerConverter

Converts foreign data formats (irishsentencebank,ad,pos,conllx,namefinder,parse,conllu) to native OpenNLP format

```
Usage: opennlp TokenizerConverter help|irishsentencebank|ad|pos|conllx|namefinder|parse|conllu
        [help|options...]
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
irishsentencebank	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ad	encoding	charsetName	No	Encoding for reading and writing text, if absent the system default is used.
	lang	language	No	Language which is being processed.
	data	sampleData	No	Data to be used, usually a file name.
	splitHyphenatedTokens	split	Yes	If true all hyphenated tokens will be separated (default true)
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
pos	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
conllx	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text,

Format	Argument	Value	Optional	Description
				if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
namefinder	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
parse	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
conllu	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

DictionaryDetokenizer

```
Usage: opennlp DictionaryDetokenizer detokenizerDictionary
```

Sentdetect

SentenceDetector

Learnable sentence detector

```
Usage: opennlp SentenceDetector model < sentences
```

SentenceDetectorTrainer

Trainer for the learnable sentence detector

```
Usage: opennlp
        SentenceDetectorTrainer[.irishsentencebank|.ad|.pos|.conllx|.namefinder|.parse|.moses|.conllu|.letsmt]
        [-factory factoryName] [-eosChars string] [-abbDict path] [-params paramsFile] -lang language -model
        modelFile -data sampleData [-encoding charsetName]
 Arguments description:
   -factory factoryName
     A sub-class of SentenceDetectorFactory where to get implementation and resources.
   -eosChars string
     EOS characters.
   -abbDict path
     abbreviation dictionary in XML format.
   -params paramsFile
     training parameters file.
   -lang language
     language which is being processed.
   -model modelFile
     output model file.
   -data sampleData
     data to be used, usually a file name.
   -encoding charsetName
     encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
irishsentencebank	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ad	encoding	charsetName	No	Encoding for reading and writing text.
	lang	language	No	Language which is being processed.
	data	sampleData	No	Data to be used, usually a file name.
	includeTitles	includeTitles	Yes	If true will include sentences marked as headlines.
pos	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
conllx	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text,

Format	Argument	Value	Optional	Description
				if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
namefinder	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
parse	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
moses	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
conllu	sentencesPerSample	sentencesPerSample	No	Number of sentences per sample
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
letsmt	detokenizer	dictionary	Yes	Specifies the file with detokenizer dictionary.
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

SentenceDetectorEvaluator

Evaluator for the learnable sentence detector

```
Usage: opennlp
        SentenceDetectorEvaluator[.irishsentencebank|.ad|.pos|.conllx|.namefinder|.parse|.moses|.conllu|.letsmt]
        -model model [-misclassified true|false] -data sampleData [-encoding charsetName]
 Arguments description:
   -model model
     the model file to be evaluated.
   -misclassified true|false
     if true will print false negatives and false positives.
   -data sampleData
     data to be used, usually a file name.
   -encoding charsetName
     encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
irishsentencebank	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ad	encoding	charsetName	No	Encoding for reading and writing text.
	lang	language	No	Language which is being processed.
	data	sampleData	No	Data to be used, usually a file name.
	includeTitles	includeTitles	Yes	If true will include sentences marked as headlines.
pos	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
conllx	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
namefinder	data	sampleData	No	Data to be used, usually a file name.

Format	Argument	Value	Optional	Description
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
parse	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
moses	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
conllu	sentencesPerSample	sentencesPerSample	No	Number of sentences per sample
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
letsmt	detokenizer	dictionary	Yes	Specifies the file with detokenizer dictionary.
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

SentenceDetectorCrossValidator

K-fold cross validator for the learnable sentence detector

```
Usage: opennlp
        SentenceDetectorCrossValidator[.irishsentencebank|.ad|.pos|.conllx|.namefinder|.parse|.moses|.conllu|.letsmt]
        [-factory factoryName] [-eosChars string] [-abbDict path] [-params paramsFile] -lang language
        [-misclassified true|false] [-folds num] -data sampleData [-encoding charsetName]
 Arguments description:
   -factory factoryName
     A sub-class of SentenceDetectorFactory where to get implementation and resources.
   -eosChars string
     EOS characters.
   -abbDict path
     abbreviation dictionary in XML format.
   -params paramsFile
     training parameters file.
   -lang language
     language which is being processed.
   -misclassified true|false
     if true will print false negatives and false positives.
   -folds num
     number of folds, default is 10.
   -data sampleData
     data to be used, usually a file name.
   -encoding charsetName
     encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
irishsentencebank	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ad	encoding	charsetName	No	Encoding for reading and writing text.
	lang	language	No	Language which is being processed.
	data	sampleData	No	Data to be used, usually a file name.
	includeTitles	includeTitles	Yes	If true will include sentences marked as headlines.
pos	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
conllx	data	sampleData	No	Data to be used, usually a file name.

Format	Argument	Value	Optional	Description
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
namefinder	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
parse	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
moses	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
conllu	sentencesPerSample	sentencesPerSample	No	Number of sentences per sample
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
letsmt	detokenizer	dictionary	Yes	Specifies the file with detokenizer dictionary.
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

SentenceDetectorConverter

Converts foreign data formats (irishsentencebank,ad,pos,conllx,namefinder,parse,moses,conllu,letsmt) to native OpenNLP format

```
Usage: opennlp SentenceDetectorConverter
        help|irishsentencebank|ad|pos|conllx|namefinder|parse|moses|conllu|letsmt [help|options...]
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
irishsentencebank	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ad	encoding	charsetName	No	Encoding for reading and writing text.
	lang	language	No	Language which is being processed.
	data	sampleData	No	Data to be used, usually a file name.
	includeTitles	includeTitles	Yes	If true will include sentences marked as headlines.
pos	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
conllx	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
namefinder	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

Format	Argument	Value	Optional	Description
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
parse	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
	detokenizer	dictionary	No	Specifies the file with detokenizer dictionary.
moses	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
conllu	sentencesPerSample	sentencesPerSample	No	Number of sentences per sample
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
letsmt	detokenizer	dictionary	Yes	Specifies the file with detokenizer dictionary.
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

Namefind

TokenNameFinder

Learnable name finder

```
Usage: opennlp TokenNameFinder model1 model2 ... modelN < sentences
```

TokenNameFinderTrainer

Trainer for the learnable name finder

```
Usage: opennlp TokenNameFinderTrainer[.evalita|.ad|.conll03|.bionlp2004|.conll02|.muc6|.ontonotes|.brat]
        [-factory factoryName] [-resources resourcesDir] [-type modelType] [-featuregen featuregenFile]
        [-nameTypes types] [-sequenceCodec codec] [-params paramsFile] -lang language -model modelFile -data
        sampleData [-encoding charsetName]
 Arguments description:
   -factory factoryName
     A sub-class of TokenNameFinderFactory
   -resources resourcesDir
     The resources directory
   -type modelType
     The type of the token name finder model
   -featuregen featuregenFile
     The feature generator descriptor file
   -nameTypes types
     name types to use for training
   -sequenceCodec codec
     sequence codec used to code name spans
   -params paramsFile
     training parameters file.
   -lang language
     language which is being processed.
   -model modelFile
     output model file.
   -data sampleData
     data to be used, usually a file name.
   -encoding charsetName
     encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
evalita	lang	it	No	
	types	per,loc,org,gpe	No	
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ad	encoding	charsetName	No	Encoding for reading and writing text, if absent the system default is used.
	lang	language	No	Language which is being processed.
	data	sampleData	No	Data to be used, usually a file name.
	splitHyphenatedTokens	split	Yes	If true all hyphenated tokens will be separated (default true)
conll03	lang	eng|deu	No	
	types	per,loc,org,misc	No	
	data	sampleData	No	Data to be used, usually a file name.

Format	Argument	Value	Optional	Description
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
bionlp2004	types	DNA, protein, cell_type, cell_line, RNA	No	
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
conll02	lang	spa|nld	No	
	types	per,loc,org,misc	No	
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
muc6	tokenizerModel	modelFile	No	
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ontonotes	ontoNotesDir	OntoNotes 4.0 corpus directory	No	
brat	tokenizerModel	modelFile	Yes	
	ruleBasedTokenizer	name	Yes	
	annotationConfig	annConfFile	No	
	bratDataDir	bratDataDir	No	Location of brat data dir
	recursive	value	Yes	
	sentenceDetectorModel	modelFile	Yes	

TokenNameFinderEvaluator

Measures the performance of the NameFinder model with the reference data

```
Usage: opennlp TokenNameFinderEvaluator[.evalita|.ad|.conll03|.bionlp2004|.conll02|.muc6|.ontonotes|.brat]
        [-nameTypes types] -model model [-misclassified true|false] [-detailedF true|false]
        [-reportOutputFile outputFile] -data sampleData [-encoding charsetName]
 Arguments description:
   -nameTypes types
     name types to use for evaluation
   -model model
     the model file to be evaluated.
   -misclassified true|false
     if true will print false negatives and false positives.
   -detailedF true|false
     if true (default) will print detailed FMeasure results.
   -reportOutputFile outputFile
     the path of the fine-grained report file.
   -data sampleData
     data to be used, usually a file name.
   -encoding charsetName
     encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
evalita	lang	it	No	
	types	per,loc,org,gpe	No	
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ad	encoding	charsetName	No	Encoding for reading and writing text, if absent the system default is used.
	lang	language	No	Language which is being processed.
	data	sampleData	No	Data to be used, usually a file name.
	splitHyphenatedTokens	split	Yes	If true all hyphenated tokens will be separated (default true)
conll03	lang	eng|deu	No	
	types	per,loc,org,misc	No	
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
bionlp2004	types	DNA, protein, cell_type, cell_line, RNA	No	

Format	Argument	Value	Optional	Description
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
conll02	lang	spa|nld	No	
	types	per,loc,org,misc	No	
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
muc6	tokenizerModel	modelFile	No	
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ontonotes	ontoNotesDir	OntoNotes 4.0 corpus directory	No	
brat	tokenizerModel	modelFile	Yes	
	ruleBasedTokenizer	name	Yes	
	annotationConfig	annConfFile	No	
	bratDataDir	bratDataDir	No	Location of brat data dir
	recursive	value	Yes	
	sentenceDetectorModel	modelFile	Yes	

TokenNameFinderCrossValidator

K-fold cross validator for the learnable Name Finder

```
Usage: opennlp
        TokenNameFinderCrossValidator[.evalita|.ad|.conll03|.bionlp2004|.conll02|.muc6|.ontonotes|.brat]
        [-factory factoryName] [-resources resourcesDir] [-type modelType] [-featuregen featuregenFile]
        [-nameTypes types] [-sequenceCodec codec] [-params paramsFile] -lang language [-misclassified
        true|false] [-folds num] [-detailedF true|false] [-reportOutputFile outputFile] -data sampleData
        [-encoding charsetName]
 Arguments description:
   -factory factoryName
     A sub-class of TokenNameFinderFactory
   -resources resourcesDir
     The resources directory
   -type modelType
     The type of the token name finder model
   -featuregen featuregenFile
     The feature generator descriptor file
   -nameTypes types
     name types to use for training
   -sequenceCodec codec
     sequence codec used to code name spans
   -params paramsFile
     training parameters file.
   -lang language
     language which is being processed.
   -misclassified true|false
     if true will print false negatives and false positives.
   -folds num
     number of folds, default is 10.
   -detailedF true|false
     if true (default) will print detailed FMeasure results.
   -reportOutputFile outputFile
     the path of the fine-grained report file.
   -data sampleData
     data to be used, usually a file name.
   -encoding charsetName
     encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
evalita	lang	it	No	
	types	per,loc,org,gpe	No	
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ad	encoding	charsetName	No	Encoding for reading and writing text, if absent the system default is used.
	lang	language	No	Language which is being processed.
	data	sampleData	No	Data to be used, usually a file name.
	splitHyphenatedTokens	split	Yes	If true all hyphenated tokens will be

Format	Argument	Value	Optional	Description
				separated (default true)
conll03	lang	eng|deu	No	
	types	per,loc,org,misc	No	
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
bionlp2004	types	DNA, protein, cell_type, cell_line, RNA	No	
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
conll02	lang	spa|nld	No	
	types	per,loc,org,misc	No	
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
muc6	tokenizerModel	modelFile	No	
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ontonotes	ontoNotesDir	OntoNotes 4.0 corpus directory	No	
brat	tokenizerModel	modelFile	Yes	
	ruleBasedTokenizer	name	Yes	
	annotationConfig	annConfFile	No	
	bratDataDir	bratDataDir	No	Location of brat data dir
	recursive	value	Yes	
	sentenceDetectorModel	modelFile	Yes	

TokenNameFinderConverter

Converts foreign data formats (evalita,ad,conll03,bionlp2004,conll02,muc6,ontonotes,brat) to native OpenNLP format

```
Usage: opennlp TokenNameFinderConverter help|evalita|ad|conll03|bionlp2004|conll02|muc6|ontonotes|brat
        [help|options...]
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
evalita	lang	it	No	
	types	per,loc,org,gpe	No	
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ad	encoding	charsetName	No	Encoding for reading and writing text, if absent the system default is used.
	lang	language	No	Language which is being processed.
	data	sampleData	No	Data to be used, usually a file name.
	splitHyphenatedTokens	split	Yes	If true all hyphenated tokens will be separated (default true)
conll03	lang	eng|deu	No	
	types	per,loc,org,misc	No	
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
bionlp2004	types	DNA, protein, cell_type, cell_line, RNA	No	
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
conll02	lang	spa|nld	No	
	types	per,loc,org,misc	No	

Format	Argument	Value	Optional	Description
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
muc6	tokenizerModel	modelFile	No	
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ontonotes	ontoNotesDir	OntoNotes 4.0 corpus directory	No	
brat	tokenizerModel	modelFile	Yes	
	ruleBasedTokenizer	name	Yes	
	annotationConfig	annConfFile	No	
	bratDataDir	bratDataDir	No	Location of brat data dir
	recursive	value	Yes	
	sentenceDetectorModel	modelFile	Yes	

CensusDictionaryCreator

Converts 1990 US Census names into a dictionary

```
Usage: opennlp CensusDictionaryCreator [-encoding charsetName] [-lang code] -censusData censusDict -dict dict

 Arguments description:
   -encoding charsetName
   -lang code
   -censusData censusDict
   -dict dict
```

Postag

POSTagger

Learnable part of speech tagger

```
Usage: opennlp POSTagger model < sentences
```

POSTaggerTrainer

Trains a model for the part-of-speech tagger

```
Usage: opennlp POSTaggerTrainer[.ad|.conllx|.parse|.ontonotes|.conllu] [-factory factoryName] [-resources
        resourcesDir] [-tagDictCutoff tagDictCutoff] [-featuregen featuregenFile] [-dict dictionaryPath]
        [-params paramsFile] -lang language -model modelFile -data sampleData [-encoding charsetName]
 Arguments description:
  -factory factoryName
    A sub-class of POSTaggerFactory where to get implementation and resources.
  -resources resourcesDir
    The resources directory
  -tagDictCutoff tagDictCutoff
    TagDictionary cutoff. If specified will create/expand a mutable TagDictionary
  -featuregen featuregenFile
    The feature generator descriptor file
  -dict dictionaryPath
    The XML tag dictionary file
  -params paramsFile
    training parameters file.
  -lang language
    language which is being processed.
  -model modelFile
    output model file.
  -data sampleData
    data to be used, usually a file name.
  -encoding charsetName
    encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
ad	encoding	charsetName	No	Encoding for reading and writing text, if absent the system default is used.
	lang	language	No	Language which is being processed.
	data	sampleData	No	Data to be used, usually a file name.
	expandME	expandME	Yes	Expand multiword expressions.
	includeFeatures	includeFeatures	Yes	Combine POS Tags with word features, like number and gender.
conllx	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
parse	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

Format	Argument	Value	Optional	Description
ontonotes	ontoNotesDir	OntoNotes 4.0 corpus directory	No	
conllu	tagset	tagset	Yes	U|x u for unified tags and x for language-specific part-of-speech tags
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

POSTaggerEvaluator

Measures the performance of the POS tagger model with the reference data

```
Usage: opennlp POSTaggerEvaluator[.ad|.conllx|.parse|.ontonotes|.conllu] -model model [-misclassified
        true|false] [-reportOutputFile outputFile] -data sampleData [-encoding charsetName]
  Arguments description:
    -model model
      the model file to be evaluated.
    -misclassified true|false
      if true will print false negatives and false positives.
    -reportOutputFile outputFile
      the path of the fine-grained report file.
    -data sampleData
      data to be used, usually a file name.
    -encoding charsetName
      encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
ad	encoding	charsetName	No	Encoding for reading and writing text, if absent the system default is used.
	lang	language	No	Language which is being processed.
	data	sampleData	No	Data to be used, usually a file name.
	expandME	expandME	Yes	Expand multiword expressions.
	includeFeatures	includeFeatures	Yes	Combine POS Tags with word features, like number and gender.
conllx	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text,

Format	Argument	Value	Optional	Description
				if absent the system default is used.
parse	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ontonotes	ontoNotesDir	OntoNotes 4.0 corpus directory	No	
conllu	tagset	tagset	Yes	U|x u for unified tags and x for language-specific part-of-speech tags
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

POSTaggerCrossValidator

K-fold cross validator for the learnable POS tagger

```
Usage: opennlp POSTaggerCrossValidator[.ad|.conllx|.parse|.ontonotes|.conllu] [-misclassified true|false]
        [-folds num] [-factory factoryName] [-resources resourcesDir] [-tagDictCutoff tagDictCutoff]
        [-featuregen featuregenFile] [-dict dictionaryPath] [-params paramsFile] -lang language
        [-reportOutputFile outputFile] -data sampleData [-encoding charsetName]
 Arguments description:
   -misclassified true|false
     if true will print false negatives and false positives.
   -folds num
     number of folds, default is 10.
   -factory factoryName
     A sub-class of POSTaggerFactory where to get implementation and resources.
   -resources resourcesDir
     The resources directory
   -tagDictCutoff tagDictCutoff
     TagDictionary cutoff. If specified will create/expand a mutable TagDictionary
   -featuregen featuregenFile
     The feature generator descriptor file
   -dict dictionaryPath
     The XML tag dictionary file
   -params paramsFile
     training parameters file.
   -lang language
     language which is being processed.
   -reportOutputFile outputFile
     the path of the fine-grained report file.
   -data sampleData
     data to be used, usually a file name.
   -encoding charsetName
     encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
ad	encoding	charsetName	No	Encoding for reading and writing text, if absent the system default is used.
	lang	language	No	Language which is being processed.
	data	sampleData	No	Data to be used, usually a file name.
	expandME	expandME	Yes	Expand multiword expressions.
	includeFeatures	includeFeatures	Yes	Combine POS Tags with word features, like number and gender.
conllx	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
parse	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ontonotes	ontoNotesDir	OntoNotes 4.0 corpus directory	No	
conllu	tagset	tagset	Yes	U|x u for unified tags and x for language-specific part-of-speech tags
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

POSTaggerConverter

Converts foreign data formats (ad,conllx,parse,ontonotes,conllu) to native OpenNLP format

```
Usage: opennlp POSTaggerConverter help|ad|conllx|parse|ontonotes|conllu [help|options...]
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
ad	encoding	charsetName	No	Encoding for reading and writing text, if absent the system default is used.
	lang	language	No	Language which is being processed.
	data	sampleData	No	Data to be used, usually a file name.
	expandME	expandME	Yes	Expand multiword expressions.
	includeFeatures	includeFeatures	Yes	Combine POS Tags with word features, like number and gender.
conllx	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
parse	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.
ontonotes	ontoNotesDir	OntoNotes 4.0 corpus directory	No	
conllu	tagset	tagset	Yes	U|x u for unified tags and x for language-specific part-of-speech tags
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

Lemmatizer

LemmatizerME

Learnable lemmatizer

```
Usage: opennlp LemmatizerME model < sentences
```

LemmatizerTrainerME

Trainer for the learnable lemmatizer

```
Usage: opennlp LemmatizerTrainerME[.conllu] [-factory factoryName] [-params paramsFile] -lang language -model
        modelFile -data sampleData [-encoding charsetName]
  Arguments description:
    -factory factoryName
      A sub-class of LemmatizerFactory where to get implementation and resources.
    -params paramsFile
      training parameters file.
    -lang language
      language which is being processed.
    -model modelFile
      output model file.
    -data sampleData
      data to be used, usually a file name.
    -encoding charsetName
      encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
conllu	tagset	tagset	Yes	U|x u for unified tags and x for language-specific part-of-speech tags
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

LemmatizerEvaluator

Measures the performance of the Lemmatizer model with the reference data

```
Usage: opennlp LemmatizerEvaluator[.conllu] -model model [-misclassified true|false] [-reportOutputFile
        outputFile] -data sampleData [-encoding charsetName]
  Arguments description:
    -model model
      the model file to be evaluated.
    -misclassified true|false
      if true will print false negatives and false positives.
    -reportOutputFile outputFile
      the path of the fine-grained report file.
    -data sampleData
      data to be used, usually a file name.
    -encoding charsetName
      encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
conllu	tagset	tagset	Yes	U|x u for unified tags and x for language-specific part-of-speech tags
	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

Chunker

ChunkerME

Learnable chunker

```
Usage: opennlp ChunkerME model < sentences
```

ChunkerTrainerME

Trainer for the learnable chunker

```
Usage: opennlp ChunkerTrainerME[.ad] [-factory factoryName] [-params paramsFile] -lang language -model
        modelFile -data sampleData [-encoding charsetName]
 Arguments description:
   -factory factoryName
     A sub-class of ChunkerFactory where to get implementation and resources.
   -params paramsFile
     training parameters file.
   -lang language
     language which is being processed.
   -model modelFile
     output model file.
   -data sampleData
     data to be used, usually a file name.
   -encoding charsetName
     encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
ad	encoding	charsetName	No	Encoding for reading and writing text, if absent the system default is used.
	lang	language	No	Language which is being processed.

Format	Argument	Value	Optional	Description
	end	end	Yes	Index of last sentence
	start	start	Yes	Index of first sentence
	data	sampleData	No	Data to be used, usually a file name.

ChunkerEvaluator

Measures the performance of the Chunker model with the reference data

```
Usage: opennlp ChunkerEvaluator[.ad] -model model [-misclassified true|false] [-detailedF true|false] -data
        sampleData [-encoding charsetName]
  Arguments description:
    -model model
      the model file to be evaluated.
    -misclassified true|false
      if true will print false negatives and false positives.
    -detailedF true|false
      if true (default) will print detailed FMeasure results.
    -data sampleData
      data to be used, usually a file name.
    -encoding charsetName
      encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
ad	encoding	charsetName	No	Encoding for reading and writing text, if absent the system default is used.
	lang	language	No	Language which is being processed.
	end	end	Yes	Index of last sentence
	start	start	Yes	Index of first sentence
	data	sampleData	No	Data to be used, usually a file name.

ChunkerCrossValidator

K-fold cross validator for the chunker

```
Usage: opennlp ChunkerCrossValidator[.ad] [-factory factoryName] [-params paramsFile] -lang language
        [-misclassified true|false] [-folds num] [-detailedF true|false] -data sampleData [-encoding
        charsetName]
 Arguments description:
   -factory factoryName
     A sub-class of ChunkerFactory where to get implementation and resources.
   -params paramsFile
     training parameters file.
   -lang language
     language which is being processed.
   -misclassified true|false
     if true will print false negatives and false positives.
   -folds num
     number of folds, default is 10.
   -detailedF true|false
     if true (default) will print detailed FMeasure results.
   -data sampleData
     data to be used, usually a file name.
   -encoding charsetName
     encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
ad	encoding	charsetName	No	Encoding for reading and writing text, if absent the system default is used.
	lang	language	No	Language which is being processed.
	end	end	Yes	Index of last sentence
	start	start	Yes	Index of first sentence
	data	sampleData	No	Data to be used, usually a file name.

ChunkerConverter

Converts ad data format to native OpenNLP format

```
Usage: opennlp ChunkerConverter help|ad [help|options...]
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
ad	encoding	charsetName	No	Encoding for reading and writing text, if absent the system default is used.
	lang	language	No	Language which is being processed.

Format	Argument	Value	Optional	Description
	end	end	Yes	Index of last sentence
	start	start	Yes	Index of first sentence
	data	sampleData	No	Data to be used, usually a file name.

Parser

Parser

Performs full syntactic parsing

```
Usage: opennlp Parser [-bs n -ap n -k n -tk tok_model] model < sentences
  -bs n: Use a beam size of n.
  -ap f: Advance outcomes in with at least f% of the probability mass.
  -k n: Show the top n parses.  This will also display their log-probabilities.
  -tk tok_model: Use the specified tokenizer model to tokenize the sentences. Defaults to a WhitespaceTokenizer.
```

ParserTrainer

Trains the learnable parser

```
Usage: opennlp ParserTrainer[.ontonotes|.frenchtreebank] [-headRulesSerializerImpl className] -headRules
        headRulesFile [-parserType CHUNKING|TREEINSERT] [-fun true|false] [-params paramsFile] -lang language
        -model modelFile [-encoding charsetName] -data sampleData
  Arguments description:
    -headRulesSerializerImpl className
      head rules artifact serializer class name
    -headRules headRulesFile
      head rules file.
    -parserType CHUNKING|TREEINSERT
      one of CHUNKING or TREEINSERT, default is CHUNKING.
    -fun true|false
      Learn to generate function tags.
    -params paramsFile
      training parameters file.
    -lang language
      language which is being processed.
    -model modelFile
      output model file.
    -encoding charsetName
      encoding for reading and writing text, if absent the system default is used.
    -data sampleData
      data to be used, usually a file name.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
ontonotes	ontoNotesDir	OntoNotes 4.0 corpus directory	No	

Format	Argument	Value	Optional	Description
frenchtreebank	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

ParserEvaluator

Measures the performance of the Parser model with the reference data

```
Usage: opennlp ParserEvaluator[.ontonotes|.frenchtreebank] -model model [-misclassified true|false] -data
        sampleData [-encoding charsetName]
  Arguments description:
    -model model
      the model file to be evaluated.
    -misclassified true|false
      if true will print false negatives and false positives.
    -data sampleData
      data to be used, usually a file name.
    -encoding charsetName
      encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
ontonotes	ontoNotesDir	OntoNotes 4.0 corpus directory	No	
frenchtreebank	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

ParserConverter

Converts foreign data formats (ontonotes,frenchtreebank) to native OpenNLP format

```
Usage: opennlp ParserConverter help|ontonotes|frenchtreebank [help|options...]
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
ontonotes	ontoNotesDir	OntoNotes 4.0 corpus directory	No	
frenchtreebank	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text,

Format	Argument	Value	Optional	Description
				if absent the system default is used.

BuildModelUpdater

Trains and updates the build model in a parser model

```
Usage: opennlp BuildModelUpdater[.ontonotes|.frenchtreebank] -model modelFile [-params paramsFile] -lang
       language -data sampleData [-encoding charsetName]
Arguments description:
  -model modelFile
    output model file.
  -params paramsFile
    training parameters file.
  -lang language
    language which is being processed.
  -data sampleData
    data to be used, usually a file name.
  -encoding charsetName
    encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
ontonotes	ontoNotesDir	OntoNotes 4.0 corpus directory	No	
frenchtreebank	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

CheckModelUpdater

Trains and updates the check model in a parser model

```
Usage: opennlp CheckModelUpdater[.ontonotes|.frenchtreebank] -model modelFile [-params paramsFile] -lang
       language -data sampleData [-encoding charsetName]
Arguments description:
  -model modelFile
    output model file.
  -params paramsFile
    training parameters file.
  -lang language
    language which is being processed.
  -data sampleData
    data to be used, usually a file name.
  -encoding charsetName
    encoding for reading and writing text, if absent the system default is used.
```

The supported formats and arguments are:

Format	Argument	Value	Optional	Description
ontonotes	ontoNotesDir	OntoNotes 4.0 corpus directory	No	
frenchtreebank	data	sampleData	No	Data to be used, usually a file name.
	encoding	charsetName	Yes	Encoding for reading and writing text, if absent the system default is used.

TaggerModelReplacer

Replaces the tagger model in a parser model

```
Usage: opennlp TaggerModelReplacer parser.model tagger.model
```

Entitylinker

EntityLinker

Links an entity to an external data set

The *model* argument is an Entity Linker properties file. The command loads the implementation named by `linker.location` and reads tokenized, named-entity-annotated sentences from standard input. See the Entity Linker chapter for the input format and configuration details.

```
Usage: opennlp EntityLinker model < sentences
```

Languagemodel

NGramLanguageModel

Gives the probability and most probable next token(s) of a sequence of tokens in a language model

```
Usage: opennlp NGramLanguageModel model
```

Chapter 28. Evaluation Test Data

What is it?

The evaluation test data is the data used in the tests that evaluate functionality and performance of OpenNLP. These tests ensure reliability and can help identify potential bugs, errors, or performance issues.

The evaluation tests leverage the k-fold cross-validation procedure. This technique works by dividing the evaluation data into `k` equally sized parts or folds. The algorithm is then trained on `k-1` of the folds and tested on the remaining fold. This process is repeated `k` times, so that each of the k-folds is used exactly once as the test data, and the results of each fold are combined to produce an overall estimate of the algorithm's performance.

Where is it?

OpenNLP evaluation tests data is available at https://nightlies.apache.org/opennlp/ (file name: `opennlp-data.zip`)

Here's a link to the evaluation-tests build on Jenkins [https://ci-builds.apache.org/job/OpenNLP/].

How to use the evaluation test data to run test?

The evaluation tests data can be downloaded and saved in the desired directory and can be used to run OpenNLP Evaluation Tests as below:

```
mvn test -DOPENNLP_DATA_DIR=/path/to/opennlp-eval-test-data/ -Peval-tests
```

How to change evaluation data?

OpenNLP Evaluation Tests use nightlies.apache.org [https://nightlies.apache.org/] to share data for testing and releasing candidate build. You can also upload the opennlp-data.zip to `nightlies.apache.org` as below:

```
curl -u your_asf_username -T ./opennlp-data.zip "https://nightlies.apache.org/opennlp/"
```

More information about changing the evaluation test data on `nightlies.apache.org` can be found at: https://nightlies.apache.org/authoring.html [https://nightlies.apache.org/authoring.html]