Morfessor Documentation

Release 2.0.1

Sami Virpioja and Peter Smit

August 21, 2014

Contents

Note: The Morfessor 2.0 documentation is still a work in progress and contains some unfinished parts

Contents:

CHAPTER 1

License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CHAPTER 2

General

2.1 Morfessor 2.0 Technical Report

The work done in Morfessor 2.0 is described in detail in the Morfessor 2.0 Technical Report [TechRep]. The report is available for download from http://urn.fi/URN:ISBN:978-952-60-5501-5.

2.2 Terminology

Unlike previous Morfessor implementations, Morfessor 2.0 is, in principle, applicable to any string segmentation task. Thus we use terms that are not specific to morphological segmentation task.

The task of the algorithm is to find a set of *constructions* that describe the provided training corpus efficiently and accurately. The training corpus contains a collection of *compounds*, which are the largest sequences that a single construction can hold. The smallest pieces of constructions and compounds are called *atoms*.

For example, in morphological segmentation, compounds are word forms, constructions are morphs, and atoms are characters. In chunking, compounds are sentences, constructions are phrases, and atoms are words.

2.3 Citing

The authors do kindly ask that you cite the Morfessor 2.0 techical report [TechRep] when using this tool in academic publications.

In addition, when you refer to the Morfessor algorithms, you should cite the respective publications where they have been introduced. For example, the first Morfessor algorithm was published in [Creutz2002] and the semi-supervised extension in [Kohonen2010]. See [TechRep] for further information on the relevant publications.

CHAPTER 3

Installation instructions

Morfessor 2.0 is installed using setuptools library for Python. Morfessor can be installed from the packages available on the Morpho project homepage and the Morfessor Github page, or can be directly installed from the Python Package Index (PyPI).

The Morfessor packages are created using the current Python packaging standards, as described on http://docs.python.org/install/. Morfessor packages are fully compatible with, and recommended to run in, virtual environments as described on http://virtualenv.org.

3.1 Installation from tarball or zip file

The Morfessor 2.0 tarball and zip files can be downloaded from the Morpho project homepage (latest stable version) or from the Morfessor Github page (all versions).

The tarball can be installed in two different ways. The first is to unpack the tarball or zip file and run:

```
python setup.py install
```

A second method is to use the tool pip on the tarball or zip file directly:

```
pip install morfessor-VERSION.tar.gz
```

3.2 Installation from PyPI

Morfessor 2.0 is also distributed through the Python Package Index (PyPI). This means that tools like pip and easy_install can automatically download and install the latest version of Morfessor.

Simply type:

```
pip install morfessor
```

or:

```
easy_install morfessor
```

To install the morfessor library and tools.

CHAPTER 4

Morfessor file types

4.1 Binary model

The standard format for Morfessor 2.0 is a binary model, generated by pickling the *BaselineModel* object. This ensures that all training-data, annotation-data and weights are exactly the same as when the model was saved.

4.2 Morfessor 1.0 style text model

Morfessor 2.0 also supports the text model files that are used in Morfessor 1.0. These files consists of one segmentation per line, preceded by a count, where the constructions are separated by ' + '.

Specification:

```
<int><space><CONSTRUCTION>[<space>+<space><CONSTRUCTION>]*
```

Example:

```
10 kahvi + kakku
5 kahvi + kilo + n
24 kahvi + kone + emme
```

4.3 Text corpus file

A text corpus file is a free format text-file. All lines are split into compounds using the compound-separator (default <space>). The compounds then are split into atoms using the atom-separator. Compounds can occur multiple times and will be counted as such.

Example:

```
kavhikakku kahvikilon kahvikilon
kahvikoneemme kahvikakku
```

4.4 Word list file

A word list corpus file contains one compound per line, possibly preceded by a count. If multiple entries of the same word occur there counts are summed. If no count is given, a count of one is assumed (per entry).

Specification:

```
[<int><space>]<COMPOUND>
```

Example 1:

```
10 kahvikakku
5 kahvikilon
24 kahvikoneemme
```

Example 2:

```
kahvikakku
kahvikilon
kahvikoneemme
```

4.5 Annotation file

An annotation file contains one compound and one or more annotations per compound on each line. The separators between the annotations (default ‘, ‘) and between the constructions (default ‘ ‘) are configurable.

Specification:

```
<compound> <analysis1construction1>[ <analysis1constructionN>][, <analysis2construction1> [<analysis2
```

Example:

```
kahvikakku kahvi kakku, kahvi kak ku
kahvikilon kahvi kilon
kahvikoneemme kahvi konee mme, kah vi ko nee mme
```

CHAPTER 5

Command line tools

The installation process installs 4 scripts in the appropriate PATH.

5.1 morfessor

The morfessor command is a full-featured script for training, updating models and segmenting test data.

5.1.1 Loading existing model

-l <file> load *Binary model*

-L <file> load *Morfessor 1.0 style text model*

5.1.2 Loading data

-t <file>, --traindata <file> Input corpus file(s) for training (text or bz2/gzipped text; use '-' for standard input; add several times in order to append multiple files). Standard, all sentences are split on whitespace and the tokens are used as compounds. The `--traindata-list` option can be used to read all input files as a list of compounds, one compound per line optionally prefixed by a count. See *Data format command line options* for changing the delimiters used for separating compounds and atoms.

--traindata-list Interpret all training files as list files instead of corpus files. A list file contains one compound per line with optionally a count as prefix.

-T <file>, --testdata <file> Input corpus file(s) to analyze (text or bz2/gzipped text; use '-' for standard input; add several times in order to append multiple files). The file is read in the same manner as an input corpus file. See *Data format command line options* for changing the delimiters used for separating compounds and atoms.

5.1.3 Training model options

-m <mode>, --mode <mode> Morfessor can run in different modes, each doing different actions on the model. The modes are:

none Do initialize or train a model. Can be used when just loading a model for segmenting new data

init Create new model and load input data. Does not train the model

batch Loads an existing model (which is already initialized with training data) and run *Batch training*

init+batch Create a new model, load input data and run *Batch training*. **Default**

online Create a new model, read and train the model concurrently as described in *Online training*

online+batch First read and train the model concurrently as described in *Online training* and after that retrain the model using *Batch training*

-a <algorithm>, --algorithm <algorithm> Algorithm to use for training:

recursive Recursive as descirbed in *Recursive training* **Default**

viterbi Viterbi as described in *Local Viterbi training*

-d <type>, --dampening <type> Method for changing the compound counts in the input data. Options:

none Do not alter the counts of compounds (token based training)

log Change the count x of a compound to $\log(x)$ (log-token based training)

ones Treat all compounds as if they only occured once (type based training)

-f <list>, --forcesplit <list> A list of atoms that would always cause the compound to be split. By default only hyphens (-) would force a split. Note the notation of the argument list. To have no force split characters, use as an empty string as argument (`-f ""`). To split, for example, both hyphen (-) and apostrophe (') use `-f "-'"`

-F <float>, --finish-threshold <float> Stopping threshold. Training stops when the decrease in model cost of the last iteration is smaller then finish_threshold * #boundaries; (default '0.005')

-r <seed>, --randseed <seed> Seed for random number generator

-R <float>, --randsplit <float> Initialize new words by random splitting using the given split probability (default no splitting). See *Random initialization*

--skips Use random skips for frequently seen compounds to speed up training. See *Random initialization*

--batch-minfreq <int> Compound frequency threshold for batch training (default 1)

--max-epochs <int> Hard maximum of epochs in training

--nosplit-re <regexp> If the expression matches the two surrounding characters, do not allow splitting (default None)

--online-epochint <int> Epoch interval for online training (default 10000)

--viterbi-smoothing <float> Additive smoothing parameter for Viterbi training and segmentation (default 0).

--viterbi-maxlen <int> Maximum construction length in Viterbi training and segmentation (default 30)

5.1.4 Saving model

-s <file> save *Binary model*

-S <file> save *Morfessor 1.0 style text model*

5.1.5 Examples

Training a model from inputdata.txt, saving a *Morfessor 1.0 style text model* and segmenting the test.txt set:

```
morfessor -t inputdata.txt -S model.segm -T test.txt
```

5.2 morfessor-train

The morfessor-train command is a convenience command that enables easier training for morfessor models.

The basic command structure is:

```
morfessor-train [arguments] traindata-file [traindata-file ...]
```

The arguments are identical to the ones for the morfessor command. The most relevant are:

-s <file> save binary model

-S <file> save Morfessor 1.0 style model

5.2.1 Examples

Train a morfessor model from a wordcount list in ISO_8859-15, doing type based training, writing the log to logfile and saving them model as model.bin:

```
morfessor-train --encoding=ISO_8859-15 --traindata-list --logfile=log.log -s model.bin -d ones trainc
```

5.3 morfessor-segment

The morfessor-segment command is a convenience command that enables easier segmentation of test data with a morfessor model.

The basic command structure is:

```
morfessor-segment [arguments] testcorpus-file [testcorpus-file ...]
```

The arguments are identical to the ones for the morfessor command. The most relevant are:

-l <file> load binary model

-L <file> load Morfessor 1.0 style model

5.3.1 Examples

Loading a binary model and segmenting the words in testdata.txt:

```
morfessor-segment -l model.bin testdata.txt
```

5.4 morfessor-evaluate

The morfessor-evaluate command is used for evaluating a morfessor model against a gold-standard. If multiple models are evaluated, it reports statistical significant differences between them.

The basic command structure is:

```
morfessor-evaluate [arguments] <goldstandard> <model> [<model> ...]
```

5.4.1 Positional arguments

<goldstandard> gold standard file in standard annotation format

<model> model files to segment (either binary or Morfessor 1.0 style segmentation models).

5.4.2 Optional arguments

-t TEST_SEGMENTATIONS, --testsegmentation TEST_SEGMENTATIONS

Segmentation of the test set. Note that all words in the gold-standard must be segmented

--num-samples <int> number of samples to take for testing

--sample-size <int> size of each testing samples

--format-string <format> Python new style format string used to report evaluation results. The following variables are a value and and action separated with and underscore. E.g. fscore_avg for the average f-score. The available values are "precision", "recall", "fscore", "samplesize" and the available actions: "avg", "max", "min", "values", "count". A last meta-data variable (without action) is "name", the filename of the model. See also the format-template option for predefined strings.

--format-template <template> Uses a template string for the format-string options. Available templates are: default, table and latex. If format-string is defined this option is ignored.

5.4.3 Examples

Evaluating three different models against a golden standard, outputting the results in latex table format::

```
morfessor-evaluate --format-template=latex goldstd.txt model1.bin model2.segm model3.bin
```

5.5 Data format command line options

--encoding <encoding> Encoding of input and output files (if none is given, both the local encoding and UTF-8 are tried).

--lowercase lowercase input data

--traindata-list input file(s) for batch training are lists (one compound per line, optionally count as a prefix)

--atom-separator <regexp> atom separator regexp (default None)

--compound-separator <regexp> compound separator regexp (default 's+')

--analysis-separator <str> separator for different analyses in an annotation file. Use NONE for only allowing one analysis per line

--output-format <format> format string for –output file (default: '{analysis}\n'). Valid keywords are: `{analysis}` = constructions of the compound, `{compound}` = compound string, {count} = count of the compound (currently always 1), `{logprob}` = log-probability of the analysis, and `{clogprob}` = log-probability of the compound. Valid escape sequences are `\n` (newline) and `\t` (tabular)

--output-format-separator <str> construction separator for analysis in –output file (default: ' ')

--output-newlines for each newline in input, print newline in –output file (default: 'False')

5.6 Universal command line options

--verbose <int> -v verbose level; controls what is written to the standard error stream or log file (default 1)

--logfile <file> write log messages to file in addition to standard error stream

--progressbar Force the progressbar to be displayed (possibly lowers the log level for the standard error stream)

--help -h show this help message and exit

--version show version number and exit

CHAPTER 6

Morfessor features

All features below are described in a short format, mainly to guide making the right choice for a certain parameter. These features are explained in detail in the *Morfessor 2.0 Technical Report*.

6.1 Batch training

In batch training, each epoch consists of an iteration over the full training data. Epochs are repeated until the model cost is converged. All training data needed in the training needs to be loaded before the training starts.

6.2 Online training

In online training the model is updated while the data is being added. This allows for rapid testing and prototyping. All data is only processed once, hence it is advisable to run *Batch training* afterwards. The size of an epoch is a fixed, predefined number of compounds processed. The only use of an epoch for online training is to select the best annotations in semi-supervised training.

6.3 Recursive training

In recursive training, each compound is processed in the following manner. The current split for the compound is removed from the model and its constructions are updated accordingly. After this, all possible splits are tried, by choosing one split and running the algorithm recursively on the created constructions.

In the end, the best split is selected and the training continues with the next compound.

6.4 Local Viterbi training

In Local Viterbi training the compounds are processed sequentially. Each compound is removed from the corpus and afterwards segmented using Viterbi segmentation. The result is put back into the model.

In order to allow new constructions to be created, the smoothing parameter must be given some non-zero value.

6.5 Random skips

In Random skips, frequently seen compounds are skipped in training with a random probability. As shown in the *Morfessor 2.0 Technical Report* this speeds up the training considerably with only a minor loss in model performance.

6.6 Random initialization

In random initialization all compounds are split randomly. Each possible boundary is made a split with the given probability.

Selecting a good random initialization parameter helps in finding local optima as long as the split probability is high enough.

CHAPTER 7

Python library interface to Morfessor

Morfessor 2.0 contains a library interface in order to be integrated in other python applications. The public members are documented below and should remain relatively the same between Morfessor versions. Private members are documented in the code and can change anytime in releases.

The classes are documented below.

7.1 IO class

class morfessor.io.**MorfessorIO**(*encoding=None*, *construction_separator=' + '*, *comment_start='#'*, *compound_separator='\s+'*, *atom_separator=None*, *lowercase=False*)

Definition for all input and output files. Also handles all encoding issues.

The only state this class has is the separators used in the data. Therefore, the same class instance can be used for initializing multiple files.

read_annotations_file(*file_name*, *construction_separator=' '*, *analysis_sep=','*)

Read a annotations file.

Each line has the format: <compound> <constr1> <constr2>... <constrN>, <constr1>...<constrN>, ...

Yield tuples (compound, list(analyses)).

read_any_model(*file_name*)

Read a file that is either a binary model or a Morfessor 1.0 style model segmentation. This method can not be used on standard input as data might need to be read multiple times

read_binary_model_file(*file_name*)

Read a pickled model from file.

read_corpus_file(*file_name*)

Read one corpus file.

For each compound, yield (1, compound, compound_atoms). After each line, yield (0, "n", ()).

read_corpus_files(*file_names*)

Read one or more corpus files.

Yield for each compound found (1, compound, compound_atoms).

read_corpus_list_file(*file_name*)

Read a corpus list file.

Each line has the format: <count> <compound>

Yield tuples (count, compound, compound_atoms) for each compound.

read_corpus_list_files (*file_names*)
Read one or more corpus list files.

Yield for each compound found (count, compound, compound_atoms).

read_segmentation_file (*file_name*, *has_counts=True*, ***kwargs*)
Read segmentation file.

File format: <count> <construction1><sep><construction2><sep>...<constructionN>

write_binary_model_file (*file_name*, *model*)
Pickle a model to a file.

write_lexicon_file (*file_name*, *lexicon*)
Write to a Lexicon file all constructions and their counts.

write_segmentation_file (*file_name*, *segmentations*, ***kwargs*)
Write segmentation file.

File format: <count> <construction1><sep><construction2><sep>...<constructionN>

7.2 Model classes

class morfessor.baseline.**AnnotatedCorpusEncoding** (*corpus_coding*, *weight=None*, *penalty=-9999.9*)
Encoding the cost of an Annotated Corpus.

In this encoding constructions that are missing are penalized.

get_cost ()
Return the cost of the Annotation Corpus.

set_constructions (*constructions*)
Method for re-initializing the constructions. The count of the constructions must still be set with a call to set_count

set_count (*construction*, *count*)
Set an initial count for each construction. Missing constructions are penalized

update_count (*construction*, *old_count*, *new_count*)
Update the counts in the Encoding, setting (or removing) a penalty for missing constructions

update_weight ()
Update the weight of the Encoding by taking the ratio of the corpus boundaries and annotated boundaries

class morfessor.baseline.**AnnotationsModelUpdate** (*data*, *model*)
Class for using development annotations to update the corpus weight during batch training

update_model (*epochs*)
Tune model corpus weight based on the precision and recall of the development data, trying to keep them equal

class morfessor.baseline.**BaselineModel** (*forcesplit_list=None*, *corpusweight=1.0*, *use_skips=False*, *nosplit_re=None*)
Morfessor Baseline model class.

Implements training of and segmenting with a Morfessor model. The model is complete agnostic to whether it is used with lists of strings (finding phrases in sentences) or strings of characters (finding morphs in words).

forward_logprob(*compound*)
Find log-probability of a compound using the forward algorithm.

Parameters **compound** – compound to process

Returns the (negative) log-probability of the compound. If the probability is zero, returns a number that is larger than the value defined by the penalty attribute of the model object.

get_compounds()
Return the compound types stored by the model.

get_constructions()
Return a list of the present constructions and their counts.

get_cost()
Return current model encoding cost.

get_segmentations()
Retrieve segmentations for all compounds encoded by the model.

load_data(*data*, *freqthreshold=1*, *count_modifier=None*, *init_rand_split=None*)
Load data to initialize the model for batch training.

Parameters

- **data** – iterator/generator of (count, compound, atoms) tuples
- **corpus** – corpus instance
- **freqthreshold** – discard compounds that occur less than given times in the corpus (default 1)
- **count_modifier** – function for adjusting the counts of each compound
- **init_rand_split** – If given, random split the word with init_rand_split as the probability for each split

Adds the compounds in the corpus to the model lexicon. Returns the total cost.

load_segmentations(*segmentations*)
Load model from existing segmentations.

The argument should be an iterator providing a count and a segmentation.

segment(*compound*)
Segment the compound by looking it up in the model analyses.

Raises KeyError if compound is not present in the training data. For segmenting new words, use viterbi_segment(compound).

set_annotations(*annotations*, *annotatedcorpusweight*)
Prepare model for semi-supervised learning with given annotations.

tokens
Return the number of construction tokens.

train_batch(*algorithm='recursive'*, *algorithm_params=()*, *devel_annotations=None*, *finish_threshold=0.005*, *max_epochs=None*)
Train the model in batch fashion.

The model is trained with the data already loaded into the model (by using an existing model or calling one of the load_ methods).

In each iteration (epoch) all compounds in the training data are optimized once, in a random order. If applicable, corpus weight, annotation cost, and random split counters are recalculated after each iteration.

Parameters

- **algorithm** – string in ('recursive', 'viterbi') that indicates the splitting algorithm used.
- **algorithm_params** – parameters passed to the splitting algorithm.
- **devel_annotations** – an annotated dataset (iterator of (compound, [analyses]) tuples) used for controlling the weight of the corpus encoding.
- **finish_threshold** – the stopping threshold. Training stops when the improvement of the last iteration is smaller then finish_threshold * #boundaries

train_online(*data*, *count_modifier=None*, *epoch_interval=10000*, *algorithm='recursive'*, *algorithm_params=()*, *init_rand_split=None*, *max_epochs=None*)

Train the model in online fashion.

The model is trained with the data provided in the data argument. As example the data could come from a generator linked to standard in for live monitoring of the splitting.

All compounds from data are only optimized once. After online training, batch training could be used for further optimization.

Epochs are defined as a fixed number of compounds. After each epoch (like in batch training), the annotation cost, and random split counters are recalculated if applicable.

Parameters

- **data** – iterator/generator of (_, _, compound) tuples. The first two arguments are ignored, as every occurence of the compound is taken with count 1
- **count_modifier** – function for adjusting the counts of each compound
- **epoch_interval** – number of compounds to process before starting a new epoch
- **algorithm** – string in ('recursive', 'viterbi') that indicates the splitting algorithm used.
- **algorithm_params** – parameters passed to the splitting algorithm.
- **init_rand_split** – probability for random splitting a compound to at any point for initializing the model. None or 0 means no random splitting.

types

Return the number of construction types.

viterbi_nbest(*compound*, *n*, *addcount=1.0*, *maxlen=30*)

Find top-n optimal segmentations using the Viterbi algorithm.

Parameters

- **compound** – compound to be segmented
- **addcount** – constant for additive smoothing (0 = no smoothing)
- **maxlen** – maximum length for the constructions

If additive smoothing is applied, new complex construction types can be selected during the search. Without smoothing, only new single-atom constructions can be selected.

Returns the n most probable segmentations and their log-probabilities.

viterbi_segment(*compound*, *addcount=1.0*, *maxlen=30*)

Find optimal segmentation using the Viterbi algorithm.

Parameters

- **compound** – compound to be segmented
- **addcount** – constant for additive smoothing (0 = no smoothing)

- **maxlen** – maximum length for the constructions

If additive smoothing is applied, new complex construction types can be selected during the search. Without smoothing, only new single-atom constructions can be selected.

Returns the most probable segmentation and its log-probability.

class morfessor.baseline.**ConstrNode**
ConstrNode(rcount, count, splitloc)

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

__repr__()
Return a nicely formatted representation string

count
Alias for field number 1

rcount
Alias for field number 0

splitloc
Alias for field number 2

class morfessor.baseline.**CorpusEncoding**(*lexicon_encoding*, *weight=1.0*)
Encoding the corpus class

The basic difference to a normal encoding is that the number of types is not stored directly but fetched from the lexicon encoding. Also does the cost function not contain any permutation cost.

frequency_distribution_cost()
Calculate -log[(M - 1)! (N - M)! / (N - 1)!] for M types and N tokens.

get_cost()
Override for the Encoding get_cost function. A corpus does not have a permutation cost

types
Return the number of types of the corpus, which is the same as the number of boundaries in the lexicon + 1

class morfessor.baseline.**Encoding**(*weight=1.0*)
Base class for calculating the entropy (encoding length) of a corpus or lexicon.

Commonly subclassed to redefine specific methods.

frequency_distribution_cost()
Calculate -log[(u - 1)! (v - u)! / (v - 1)!]

v is the number of tokens+boundaries and u the number of types

get_cost()
Calculate the cost for encoding the corpus/lexicon

permutations_cost()
The permutations cost for the encoding.

types
Define number of types as 0. types is made a property method to ensure easy redefinition in subclasses

update_count(*construction*, *old_count*, *new_count*)
Update the counts in the encoding.

class morfessor.baseline.**LexiconEncoding**
Class for calculating the encoding cost for the Lexicon

add(*construction*)
Add a construction to the lexicon, updating automatically the count for its atoms

get_codelength(*construction*)
Return an approximate codelength for new construction.

remove(*construction*)
Remove construction from the lexicon, updating automatically the count for its atoms

types
Return the number of different atoms in the lexicon + 1 for the compound-end-token

7.3 Evaluation classes

class morfessor.evaluation.**EvaluationConfig**
EvaluationConfig(num_samples, sample_size)

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

__repr__()
Return a nicely formatted representation string

num_samples
Alias for field number 0

sample_size
Alias for field number 1

class morfessor.evaluation.**MorfessorEvaluation**(*reference_annotations*)
Do the evaluation of one model, on one testset. The basic procedure is to create, in a stable manner, a number of samples and evaluate them independently. The stable selection of samples makes it possible to use the resulting values for Pair-wise statistical significance testing.

reference_annotations is a standard annotation dictionary: {compound => ([annoation1],..) }

evaluate_model(*model*, *configuration=EvaluationConfig(num_samples=10*, *sample_size=1000)*, *meta_data=None*)
Get the prediction of the test samples from the model and do the evaluation

The meta_data object has preferably at least the key 'name'.

evaluate_segmentation(*segmentation*, *configuration=EvaluationConfig(num_samples=10*, *sample_size=1000)*, *meta_data=None*)
Method for evaluating an existing segmentation

get_samples(*configuration=EvaluationConfig(num_samples=10*, *sample_size=1000)*)
Get a list of samples. A sample is a list of compounds.

This method is stable, so each time it is called with a specific test_set and configuration it will return the same samples. Also this method caches the samples in the _samples variable.

class morfessor.evaluation.**MorfessorEvaluationResult**(*meta_data=None*)
A MorfessorEvaluationResult is returned by a MorfessorEvaluation object. It's purpose is to store the evaluation data and provide nice formatting options.

Each MorfessorEvaluationResult contains the data of 1 evaluation (which can have multiple samples).

__getitem__(*item*)
Provide dict style interface for all values (standard values and metadata)

__str__()
Method for default visualization

add_data_point(*precision*, *recall*, *f_score*, *sample_size*)
Method used by MorfessorEvaluation to add the results of a single sample to the object

format(*format_string*)
Format this object. The format string can contain all variables, e.g. fscore_avg, precision_values or any item from metadata

class morfessor.evaluation.**WilcoxonSignedRank**
Class for doing statistical signficance testing with the Wilcoxon Signed-Rank test

It implements the Pratt method for handling zero-differences and applies a 0.5 continuity correction for the z-statistic.

static print_table(*results*)
Nicely format a results table as returned by significance_test

significance_test(*evaluations*, *val_property='fscore_values'*, *name_property='name'*)
Takes a set of evaluations (which should have the same test-configuration) and calculates the p-value for the Wilcoxon signed rank test

Returns a dictionary with (name1,name2) keys and p-values as values.

CHAPTER 8

Code Examples for using library interface

8.1 Segmenting new data using an existing model

```
import morfessor

io = morfessor.MorfessorIO()

model = io.read_binary_model_file('model.bin')

words = ['words', 'segmenting', 'morfessor', 'unsupervised']

for word in words:
    print(model.viterbi_segment(word))
```

8.2 Testing type vs token models

```
import morfessor

io = morfessor.MorfessorIO()

train_data = list(io.read_corpus_file('training_data'))

model_types = morfessor.BaselineModel()
model_logtokens = morfessor.BaselineModel()
model_tokens = morfessor.BaselineModel()

model_types.load_data(train_data, count_modifier=lambda x: 1)
def log_func(x):
    return int(round(math.log(x + 1, 2)))
model_logtokens.load_data(train_data, count_modifier=log_func)
model_tokens.load_data(train_data)

models = [model_types, model_logtokens, model_tokens]

for model in models:
    model.train_batch()

goldstd_data = io.read_annotations_file('gold_std')
ev = morfessor.MorfessorEvaluation(goldstd_data)
results = [ev.evaluate_model(m) for m in models]
```

```
wsr = morfessor.WilcoxonSignedRank()
r = wsr.significance_test(results)
WilcoxonSignedRank.print_table(r)
```

The equivalent of this on the command line would be:

```
morfessor-train -s model_types -d ones training_data
morfessor-train -s model_logtokens -d log training_data
morfessor-train -s model_tokens training_data

morfessor-evaluate gold_std morfessor-train morfessor-train morfessor-train
```

8.3 Testing different amounts of supervision data

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*

Bibliography

[TechRep] Sami Virpioja, Peter Smit, Stig-Arne Grönroos, and Mikko Kurimo. Morfessor 2.0: Python Implementation and Extensions for Morfessor Baseline. Aalto University publication series SCIENCE + TECHNOLOGY, 25/2013. Aalto University, Helsinki, 2013. ISBN 978-952-60-5501-5.

[Creutz2002] Mathias Creutz and Krista Lagus. Unsupervised discovery of morphemes. In Proceedings of the Workshop on Morphological and Phonological Learning of ACL-02, pages 21-30, Philadelphia, Pennsylvania, 11 July, 2002.

[Kohonen2010] Oskar Kohonen, Sami Virpioja and Krista Lagus. Semi-supervised learning of concatenative morphology. In Proceedings of the 11th Meeting of the ACL Special Interest Group on Computational Morphology and Phonology, pages 78-86, Uppsala, Sweden, July 2010. Association for Computational Linguistics.

m