

EMORES – Empirical Morphological Reasoning

Toni Arnold, Zürich

December 3, 2007

Copyright (C) 2006-2007 by Toni Arnold, Zurich

This file is part of emores.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Abstract

Emores is an abductive morphological reasoning engine based on the Stuttgart Finite State Transducer Tools SFST. Its aim is to facilitate a guided brute force attack on a specific problem of word morphology in computational linguistics: extending the lexicon from corpus data. For a particular inflected natural language, it requires a hand crafted SFST finite state transducer and a seed lexicon covering all of its regular inflection classes. When fed with new word forms from a corpus text, it guesses which lemmas could have generated it (induction) and what other word forms could be explained with that lemmas (deduction). This generated data is written to the database and is analysed using SQL to measure the explanatory power of the guessed lemmas with respect to the corpora (abduction).

This document is a working draft, as emores is still in alpha stage.

Contents

1	The Concept	4
1.1	Empirical evidence	4
1.2	The SFST morphology	4
1.3	Inflectional classes	5
1.4	Examples	5
1.5	Lemma guessing	6
1.5.1	Extracting wildcard matches	7
1.5.2	Guessing in Morph-it!	7
1.5.3	Guessing in Hunmorph	8
1.5.4	Guessing in emores	8
1.5.5	The algebraic solution to generate the guesser morpholgy.	8
1.6	The emores software architecture	11
2	Using Emores	13
2.1	Software dependencies	13
2.2	Configuration	13
2.2.1	Environment variables	13
2.2.2	emores.cfg	13
2.2.3	logging.cfg	14
2.3	Database setup	14
2.4	Running the tests	14
2.4.1	Test interdependence problems	14
2.4.2	Test repeatability	15
2.5	Preparing a morphology	16
2.5.1	Use case: The morphology engineer	16
2.6	Using a morphology	17
2.6.1	Use case: The lexicon extender	17
2.7	Use cases scripts	18
2.7.1	Use case doctest for English	18
2.7.2	Use case doctest for German	19
3	The Build Scripts	22
3.1	Makefiles	22
3.1.1	Main Makefile	22
3.1.2	Documentation Makefile	22
3.1.3	Doctests Makefiles	23
3.2	Scripts Overview	23
3.2.1	prefix	23

3.2.2	db_create_database	24
3.2.3	db_create_tables	24
3.2.4	cp_tablemodule	24
3.2.5	test	24
3.2.6	emores	24
3.3	Statistical data processing	25
3.3.1	The Makefile	25
3.3.2	Generating diagrams	25
3.4	Installing emores	25
4	Code Documentation	27
4.1	Character encoding	27
4.2	SFST sources	27
4.2.1	inflectionclass sources	27
4.3	Python overview	28
4.3.1	Namespace layout	28
4.3.2	Synonymic main/db modules	28
4.4	The test layout	29
4.5	Main modules	30
4.5.1	utility.py	30
4.5.2	config.py	31
4.5.3	text.py	32
4.5.4	token.py	32
4.6	SFST-related modules	33
4.6.1	morphology.py	33
4.6.2	inflectionclass.py	34
4.6.3	transducer.py	34
4.6.4	guesser.py	36
4.7	Python DB ORM classes	36
4.7.1	Common characteristics	36
4.7.2	The insert methods	37
4.7.3	Delayed flushing to the database	37
4.7.4	db/db.py	38
4.7.5	db/language.py	38
4.7.6	db/corpus.py	39
4.7.7	db/text.py	39
4.7.8	db/tokentype.py	39
4.7.9	db/token_.py	40
4.7.10	db/texttoken.py	40
4.7.11	db/morphology.py	40
4.7.12	db/lexicon.py	41
4.7.13	db/inflectionclass.py	41
4.7.14	db/lemma.py	42
5	The Emores Database	43
5.1	Database model	43
5.2	Special columns	43
5.2.1	Extending the database	43
5.3	Database Programming	44
5.3.1	SQL functions accessed by db/db.py	44

6 Empirical Results	48
6.1 The data	48
6.2 Computational performance	48
6.2.1 Possible optimisations	48
6.3 Induction	49
6.4 Deduction	49
6.4.1 Clustering lemmas by their generative power	50
6.4.2 English (en_X)	51
6.4.3 German (de_DE)	51
6.5 Abduction (de_DE)	52
6.5.1 Inherent abduction limitations	53
A Big diagrams	54
Glossary	61
Bibliography	63

Chapter 1

The Concept

1.1 Empirical evidence

Native speakers of a language don't need a morphological lexicon to learn to use new words and to inflect them the correct way: It suffices to read these words in some specific inflected forms in texts written by others. In German, it is even common to name the inflection classes of nouns after some suffixes needed to identify a particular class, written e.g. as `<NMASC_ES_EN>` in the lexicon of a SMOR morphology (Schmid et al., 2004). If a native speaker identifies the same words with the correct gender and the two suffixes “es” and “en”, (s)he has learnt the word and can produce more word forms just by regularly inflecting it.

There is no secret magic behind this learning process. Emores aims to logically model this process of extending the lexicon in a formal way. The logical model implemented in emores is naive with respect to at least two additional sources of information humans use habitually: It normally is easy to guess the word class of a word form in the context of a text, and it sometimes is possible to guess the inflection class with just one word form with some probability using the combination of trailing letters in the base stem.

The reason for not implementing the former reasoning aid is the lack of free and well trained stochastic tagger software for non-English languages. The reason for the lack of the latter logic is the fact that it is a special linguistic knowledge not implemented in the currently available Schmid et al. implementation. But there exists a free implementation in a different context: Hermann has implemented such an automatic inflection device for his German translation of Nelson's inform library (version 6). Both omissions are thus purely pragmatical and in no way theoretically founded.

1.2 The SFST morphology

Emores treats the SFST morphology more or less as a black box. There should be as little linguistic knowledge in the emores code as possible. However, there are many languages in the world, and it might (and sooner or later will) turn out that emores makes assumptions which hold only for Indo-European or similar languages.

An elaborated language specific morphology is a precondition for emores to work, not a result of applying emores on text corpora: The linguistic work needs to be done in advance. Consult the `data/XMOR/HOWTO` in the SFST package for details on how

to accomplish this for a new language. Of course, emores can be used as a tool for validating a concrete implementation of a language morphology for improving it.

Emores tries to be as non-intrusive with respect to the morphology as possible. The SFST morphologies distributed together with emores differ in two important ways from the original: The trivial one is the Unicode encoding of the originally Latin-1 encoded SFST-PL source code. The less trivial difference is the additional `*_guesser.fst` main file which is free of disjunctions in the composition path containing the lexicon. This makes the morphology algebraically solveable for a lemma which is the precondition for reusing a given morphology as a guesser. For details see the algebraic solution on pg. 8.

1.3 Inflectional classes

The classical linguistic definition of inflectional classes does often generalise over morphological details like changing umlauts which are handled as a separate theoretical concept. The emores definition of an inflectional class is purely technical and based on a hand-elaborated seed lexicon: Each different combination of lexical annotation like `<AdjReg>` and additional character mapping rules (like `o:i` in the lexical entry for “mouse”/“mice”) counts as an inflectional class.

The inflectional classes are derived from this lexicon by replacing the lexeme (and preserving the character mapping rules) from each lemma with the `.+` Dot wildcard. Each unique string derived that way counts as a single inflectional class.

A stem is the basic word form that the analyser returns together with the analysis annotation as the analysis when fed with different inflected word forms. Several lexemes can be analysed as one stem. This naming convention differs from the one used in SFST, where expression “stem” is used to denote what is called a lexeme in emores.

The handling of different character mapping rules as distinct inflectional classes is very rudimental, as in practice, they’re used to map special word forms to the same word stem stem by using two lemmas for the same stem, as in the example below. To account for that, emores would need to generate the analysis to be able to prefer such pairs of lemmas.

1.4 Examples

The Schmid et al. lexicon contains two lexical entries for the English foreign word “Switch”, one for the singular and one for the plural form:

```
<Base_Stems>Switch<>:e<>:s<NN><base><fremd><NMasc/Pl>
<Base_Stems>Switch<NN><base><fremd><NMasc/Sg_es>
```

When analysed with `fst-mor`, SFST returns the same stem `Switch` for the singular and the plural word forms:

```
analyze> Switch
Switch<+NN><Masc><Akk><Sg>
Switch<+NN><Masc><Nom><Sg>
Switch<+NN><Masc><Dat><Sg>
analyze> Switches
Switch<+NN><Masc><Akk><Pl>
Switch<+NN><Masc><Gen><Pl>
```

```
Switch<+NN><Masc><Gen><Sg>
Switch<+NN><Masc><Nom><Pl>
Switch<+NN><Masc><Dat><Pl>
```

This is an example for the German verb “lernen” (“to learn”), where the stem differs from the lexeme:

```
<ge><Base_Stems>lern<V><base><nativ><VVReg>
```

```
analyze> lernst
lernen<+V><2><Sg><Pres><Ind>
```

The following table applies the terms for the plural lexical entry for “Switch”:

Term	Example
inflectional class	<Base_Stems>.+<>:e<>:s<NN><base><fremd><NMasc/Pl>
lemma	<Base_Stems>Switch<>:e<>:s<NN><base><fremd><NMasc/Pl>
lexical annotation	<Base_Stems><NN><base><fremd><NMasc/Pl>
lexeme	Switch<>:e<>:s
word form	Switches
analysis	Switch<+NN><Masc><Nom><Pl>
analysis annotation	<+NN><Masc><Nom><Pl>
stem	Switch

1.5 Lemma guessing

Constructing a lemma guesser using SFST turned out to be difficult and susceptible to implementation detours. From a very abstract point of view, a morphological transducer is some machine code program, the lexicon together with the morphology is its source code, and emores needs to deduce (the lexical part of) the source code (concretely the explicit lemmas based on the wildcard-dotted inflectional classes) from the input (word forms) and the intended output (the analysis strings) of the running program. My first attempts to implement a lemma guesser afterwards reminded me of that story from the early seventies Papadimoulis told about an attempt to recover a lost source code deck: “What sounded good in Albert’s mind didn’t work out so well in reality. It turns out that placing an object deck in the compiler caused the mainframe to grind to a halt and flash lots of warning lights”.

There is no facile mapping from the inflectional class based analysis (the output of the program) to the concrete lemma that could have been used to analyse the word form in that way: First of all, the annotation symbols are different. The mapping from lexical annotation symbols to analysis annotation symbols is part of the morphology (e.g. `map1.fst` and `map2.fst` in XMOR) and should not be repeated in the emores code — but such a mapping is required to guess the lemma for a word form from the analysis string. Even worse, there is no facile mapping from the stem to the lexeme, which is more obvious for a German verb example: the stem is `lernen`, but the lexeme used in the lemma is `lern`: The infinitive suffix of the stem is added by the morphology. In theory, they could differ in any way implemented deeply in an SFST morphology. For the first published emores version 0.0.1 I more or less consciously ignored that problem and used the guessed stem to construct the lexeme.

1.5.1 Extracting wildcard matches

It is common for standard regular expressions to be able to retrieve a string matched by Dot wildcards using parentheses like `(.+)`. But SFST doesn't provide such functionality because finite state matching works completely different from regular expression matching e.g. in Python: The latter uses a recursive algorithm ("Secret Labs' Regular Expression Engine", see `Python-2.4.4/Lib/sre.py`) which allows the extraction of matching groups, but SFST uses a standard technique for finite state transducers by performing a functional application of the word form to the analysis transducer, minimises the result and generates all strings the resulting transducer can generate, as this code excerpt from SFST illustrates (`fst.C`, ln. 510):

```
bool Transducer::analyze_string( char *string, FILE *file
    , bool with_brackets )

{
    Transducer a1(string, &alphabet, false);
    Transducer *a2=&>(*this || a1);
    Transducer *a3=&(a2->lower_level());
    delete a2;
    a2 = &a3->minimise();
    delete a3;

    a2->alphabet.copy(alphabet);
    bool result = a2->print_strings( file, with_brackets );
    delete a2;
    return result;
}
```

There is no obvious place to implement wildcard submatch extraction, and a theoretical contemplation confirms the impression: "The extraction of submatch boundaries has been mostly ignored by computer science theorists, and it is perhaps the most compelling argument for using recursive backtracking (Cox on Finite State Automata), or, as Carlson appositely put it in a mail on the [Python-ideas] list, we would need something like "groups, named references, etc., which are a PITA for non-recursive engines".

So unlike Albert, we haven't lost the source code we need, we even don't have it at all — the purpose of `emores` is to semi-automatically extend the lexicon with unknown lemmas. So let's first have a look how others might have addressed that problem:

1.5.2 Guessing in Morph-it!

The Morph-it! project used a separately constructed guesser from the the finite state utilities written by Daciuk, but in the Morph-it! documentation they conclude "that it would probably be better to couple Morph-it! with a morphological guesser than to invest too much on expanding it". (Zanchetta and Baroni, 2005, pg. 10)

But the idea behind `emores` was from the beginning to reuse the language-specific morphology code written for the normal analyser to avoid having to maintain two distinct morphologies, one for analysing known words and one for guessing lemmas for unknown ones.

1.5.3 Guessing in Hunmorph

In a paper about Hunmorph written by Trón et al., they mention that jmorph includes a special guesser: “Two major additional features of jmorph are its capability of morphological synthesis as well as acting as a guesser (hypothesizing lemmas)”. Jmorph, a Java version of their morphological analyzer, “records the full parse tree of rule applications. By offering various ways of serializing this data structure, it allows for more structured information in the outputs than would be possible by simple concatenation of the tag chunks associated with the rules”. (pg. 4)

Besides the fact that I didn’t find a public, documented version of jmorph, the SFST toolkit, solely using finite state transducers, doesn’t have a “parser” which could output any trees, so it didn’t seem that emores could benefit from it.

1.5.4 Guessing in emores

The basic axiom for starting with the emores project was the idea to use dotted inflectional classes to analyse unknown word forms. They’re obviously suited for analysing unknown word forms (albeit they overgenerate, but that problem would be addressed with abductive result ranking), but originally there was no way to automatically map the analysis to the fully specified lemma which would have produced that very analysis. I desperately needed some idea. And in the deepest despair, there suddenly emerged hope. The achievement of the algebraic method used in emores is best explained in the corresponding Python doctest:

1.5.5 The algebraic solution to generate the guesser morpholgy.

For version 0.0.1 I tried a naive solution by omitting map1 (which deletes the “unwanted symbols in the analysis”) to guess the lemma. If that approach had been feasible, it would have made unnecessary the hassle with the logging char dead end explained below. Therefore I constructed a minimal artificial toy morphology to expose the problems and to deduce a proposed solution:

```
>>> import sfst
>>> src = '''
... % Experimental 'expose the problem' pseudo morphology
...
... % any symbols must pass trough the $MAP1$ wildcards
... ALPHABET = [a-z]<form><stem><reginf>
...
... % delete unwanted symbols in the analysis:
... % the 1st .* matches the lexeme, the 2nd .* matches the <form> tag
... $MAP1$ = <>:<stem> .* <>:<reginf> .*
...
... % A "regular inflection" lemma
... $LEX$ = <stem>lexeme<reginf>
...
... % delete unwanted symbols on the "surface"
... % lexeme< -> <stem>lexeme<reginf>
... $MAP2$ = <stem>:<> .* <reginf>:<>
...
... $LEX$ = $MAP1$ || $LEX$ || $MAP2$
...
... % The "regular" morphology:
... $REGMOR$ = $LEX$ || {lexeme}:{surface}
...
... % A dummy "inflection"
```

```

... $INFL$ = <form>:<>
...
... % Problem culprit: analyser somehow mapping "lexeme" to a
... % completely different "analysis"
... $ANALYSIS$ = .* {analysis}:{lexeme} .*
...
... % Put together the morphology
... $MORPH$ = $ANALYSIS$ || $REGMOR$ $INFL$
...
... $MORPH$
... '''
>>> morph = sfst.compile(src)

```

The whole morphology just analyses one word, “surface”, with a lemma containing the lexeme “lexeme” to generate the analysis “analysis”:

```

>>> morph.analyze('surface')
['analysis<form>']

```

Of course, the huge difference between the lexeme and the analysis is artificial, but the crux is that a real morphology in fact performs such transformations, see the example for the German verb “lernen” on pg. 6.

So let’s first transform that transducer to a guesser transducer by replacing the lemma with an inflectional class pattern. (The trailing numbers of the symbols express the patch level.) It analyses our “surface” equally:

```

>>> guesser_src_1 = src.replace( \
...     '<stem>lexeme<reginf>',
...     '<stem>.+<reginf>')
>>> guesser_morph_1 = sfst.compile(guesser_src_1)
>>> guesser_morph_1.analyze('surface')
['analysis<form>']

```

Now let’s remove map1:

```

>>> guesser_src_2 = guesser_src_1.replace( \
...     '$LEX$ = $MAP1$ || $LEX$ || $MAP2$',
...     '$LEX$ = $LEX$ || $MAP2$')
>>> guesser_morph_2 = sfst.compile(guesser_src_2)
>>> guesser_morph_2.analyze('surface')
['<stem>analysis<reginf><form>']

```

We now get the additional tagging of the inflectional class lemma, (together with the analysis annotation), which makes the analysis somehow look like a lemma. The example morphology was constructed with the objective to make obvious why the alleged lemma is in fact an useless artifact: the lexeme was itself inflected by the \$ANALYSIS\$ transducer. This pattern seems to be common in SFST morphologies. Look at the `smor.fst` main file of the German Schmid et al. morphology to get an idea how complex it can become.

The false pretence that constructing a guesser would be easy was caused by pure ignorance: I didn’t take into consideration that the pure lexicon cannot easily be exposed to the surface, because it is usually nested deeply into the morphology. In our example, we can simply remove the \$ANALYSIS\$ transducer to omit the pseudo-morphological part:

```

>>> guesser_src_3 = guesser_src_2.replace( \
...     '$MORPH$ = $ANALYSIS$ || $REGMOR$ $INFL$',
...     '$MORPH$ = $REGMOR$ $INFL$')
>>> guesser_morph_3 = sfst.compile(guesser_src_3)
>>> guesser_morph_3.analyze('surface')
['<stem>lexeme<reginf><form>']

```

This looks more like a lemma. But above transformation is only simple because we're using a toy morphology. But our toy is perfectly suited for demonstrating that the wildcard logging mechanism proposed in `pysfst-1.0.2` doesn't work for a real morphology. First lets look whether the mechanism still is present in the `pysfst` version used:

```
>>> lsrc = '''
... ALPHABET=[a-zA]
... $W$ = .....
... $T1$ = . [aL]:[eL] ...
... $T2$ = $T1$ || $W$
... $T2$
... '''
>>> lt = sfst.compile(lsrc)
>>> lt.analyze_log('hello', 'L')
(['hallo'], 'hello')
```

The wildcard matched the test input string, and the logger coincidentally logged the character sequence that `W` actually matched (the second element of the result tuple), as this patch fully specifying the wildcards justifies:

```
>>> lsrc_1 = lsrc.replace( \
...     '$W$ = .....',
...     '$W$ = hello')
>>> lt_1 = sfst.compile(lsrc_1)
>>> lt_1.analyze('hello')
['hallo']
```

Now let's apply that mechanism to our toy morphology by patching the `ALPHABET` (we're reusing the simple guesser from patch level 1):

```
>>> guesser_src_4 = guesser_src_1.replace( \
...     'ALPHABET = [a-z]<form><stem><reginf>',
...     'ALPHABET = [a-zA]<form><stem><reginf>')
>>> guesser_morph_4 = sfst.compile(guesser_src_4)
>>> guesser_morph_4.analyze_log('surface', 'L')
(['analysis<form>'], '')
```

The analysis is as expected, but the log has become empty. The reason for this behaviour is the fact that the various `minimise()` calls in the SFST engine cancelled out the logging char `L` — it would have been necessary to carry it through the whole functional application chain, or, in other words, patching the whole “morphology”. I took this as a proof that my arduously implemented logging mechanism is merely the result of some sort of mental blackout.

Considering the mathematical beauty of finite state transducers, we could try it with an algebraic transformation of the source to bring the original `LEX` to the surface. Note that this is impossible on the level of the binary transducer, again because `minimise()` does formal transformations, too and thus hides the original `LEX` pattern within the whole result transducer.

Automatically generating a guesser out of a real size linguistic analyser would require some sort of computer algebra system (CAS) tailored for dealing with SFST sources — something way beyond the scope of `emores`.

After having written that very sentence, I felt badly stuck. Then, in an almost unconscious thinking process, I became convinced that it somehow must be possible. Then I implemented the `Algebra` class for `pysfst-1.1.1`, which actually has made it possible. To demonstrate, let's start with the original source again (patch level 0):

```
>>> solved_src = sfst.solve(src, '$LEX$')
```

```
>>> solved_src.split('\n')[-1:][0]
'(_$LEX$) (_$INFL$) || ^_ ( $ANALYSIS$ || $MAP1$ (_$INFL$) ) ||
  $ANALYSIS$ || $MAP1$ (_$INFL$) || $LEX$ (_$INFL$) || $MAP2$ (
  _$INFL$) || {lexeme}:{surface} $INFL$'
```

Doesn't this look nicely "CAS'y"? Let's try whether it works:

```
>>> guesser = sfst.compile(solved_src)
>>> guesses = guesser.analyze('surface')
>>> print sorted(guesses)
['<stem>lexeme<reginf><form>']
```

Besides the fact that analysis symbols seemingly need to be passed through (with `_$INFL$`), this finally is our aimed lemma guess! Starting with emores version 0.0.2, this algebraic technique is used for the guesser.

1.6 The emores software architecture

The plan to implement emores started with an offhanded sketch of the database diagram (see pg. 55), as a relational model is well suited to statically represent the logical structure of the intended reasoning. Furthermore, a contemporary database engine should impose little physical restrictions on the manageable amount of data generated in the process of reasoning by "brute force".

As the concrete implementation of the abduction logic is intended to be part of the research process, it seemed important to separate the comparatively primitive, but very time-consuming inductive/deductive generation of guesses and the abductive quest for the "best explanation" of the empirical data within that guesses. By using an SQL database, that quest becomes a matter of data mining using a well established, interactively usable 4GL query language instead of tedious walks through traces of algorithmic implementations in a 3GL language (see the chapters "Database Programming" on pg. 44 and "Empirical Results" on pg. 48).

For me it was immediately clear that this could hardly been efficiently done solely in C++, the language SFST is implemented in. It was crucial that the implementation language is GPL-compatible Free Software for combining it with the GPL-ed SFST and that there are well established libraries for SQL database access. I chose Python because it is very prevalent today, there are sophisticated test tools (unittest, doctest), it is easy to use interactively and there are well established database access libraries like pycopg2 for direct SQL access and SQLAlchemy for automated ORM mapping, the latter mainly used for inserting data into the database.

Therefore pysfst, the Python interface for SFST was a precondition which had to be implemented before emores. Because emores would need to automatically compile "trial balloon" transducers, it had to be a "deep" interface including the compiler itself and not solely an interface to transducer instances. In fact, the needs and demands of emores shaped the actually published pysfst implementation.

The component diagram visualises the components emores is composed of in terms of the underlying technology and the interfaces used to interact with them. Note that the Python interfaces are attached to the components in the diagram, but they're in fact ordinary Python modules, partially coded in C or C++. The "source files interface" in the XMOR/SMOR component is not an interface in the strictest sense, but it describes accurately *how* the depending components interact with it: by compiling (SFST) or manipulating (Python) the source text.

The R component is somewhat isolated, it is attached to the emores software only with a Makefile operating with psql on the filesystem. The statistical plots in the chapter “Empirical Results” (pg. 48) are generated with R.

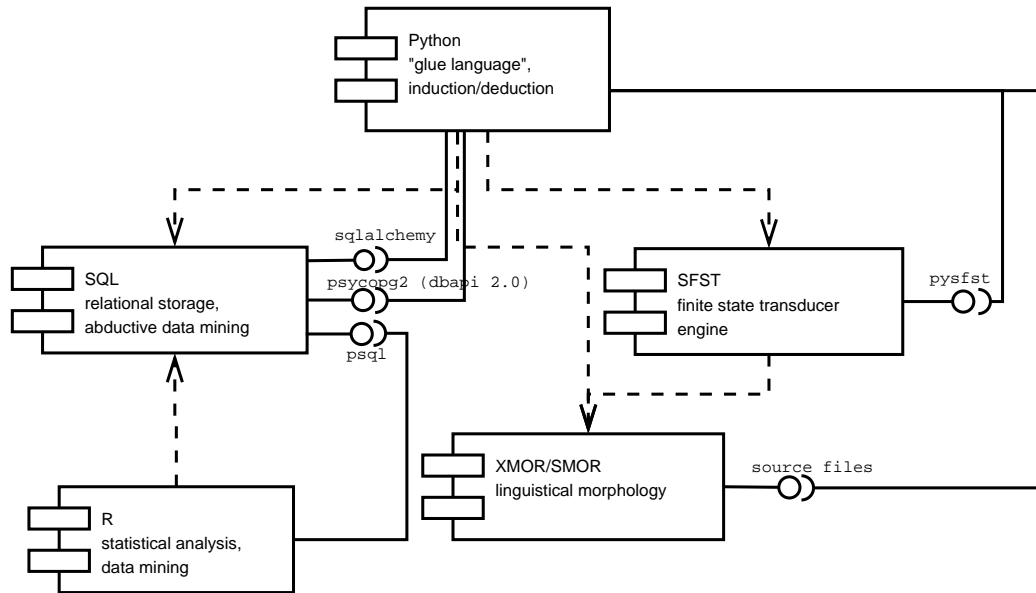


Figure 1.1: Emores component diagram

Chapter 2

Using Emores

2.1 Software dependencies

Any software involved in the emores project is Free Software.

Emores itself is written in Python (version 2.4), some code cleanup has been done using pylint.

The morphological engine is SFST, the Stuttgart Finite State Transducer Tools. It is accessed from Python via the pysfst interface.

The reasoning engine is based on the relational database postgresql. The ORM interface is provided by SQLAlchemy, straight SQL is done with psycopg2 (in Python). The database diagrams was drawn with Dia and converted to SQL scripts with tedia2sql.

This article was written with the Kile editor and LaTeX on a Gentoo Linux box.

2.2 Configuration

2.2.1 Environment variables

- `EMORES_LANG` is the only *mandatory* environment variable for emores to run and points to the directory where the morphologies reside. For a system-wide installation, the `emores` startup script sets it to `/usr/share/emores/lang`.
- `EMORES_PREFIX` is intended for doing local emores development, and it should point to the directory containing the `emores` source tree.
- `EMORES_CONF` can be set to point to a specific `emores.cfg` configuration file (see below).

2.2.2 `emores.cfg`

The main configuration file in `./etc/emores.cfg` gets globally installed with emores. To enable local customization, emores looks for the first found configuration file in this order:

1. from the filename in the environment variable `EMORES_CONF`
2. from the user's home directory in `~/.emores/emores.cfg`

3. from the global `/etc/emores/emores.cfg`

2.2.3 logging.cfg

Configures the logging depth for emores and SQLAlchemy. Per default, emores is configured to log anything (DEBUG) and sqlalchemy only for ERROR.

2.3 Database setup

Emores needs a running postgresql server to work. The database setup scripts `db_create_database` and `db_create_tables` can be found in the `./scripts` directory. When called with the command line argument `test`, they will create the `emores_test` database. The default emores user has a weak password. If you need to change it for your environment, adjust the plain text password in the `emores.cfg` file, too.

Running the scripts a second time will delete all previously stored data.

2.4 Running the tests

The general test layout and the reasons behind the way it is constructed is discussed in the corresponding section on pg. 29.

There is a `test` script which is intended to run the tests during development. It sets up some environment variables and runs the `test.py` Python script. Because some tests take a really long time to execute, it accepts a filter pattern to restrict the test runner to a particular test class if the tests are run interactively. Test names are defined as full module name (including the local namespace). Examples:

For testing the class `./db/language`, enter

```
> ./test db.language
```

For testing the whole database module, enter

```
> ./test db
```

For testing anything, just enter

```
> ./test
```

Alternatively, you can use

```
make test
```

in the emores main directory.

2.4.1 Test interdependence problems

As all tests are run within the same suite, the test order becomes unpredictable (allegedly alphabetical, but it would be foolish to verify that and even rely on it), and Python doesn't seem to provide a simple way to violate the test order independence doctrine of unit testing. This is a problem for database dependent testing as long as mocking up the entire db is not an option (do you feel like mocking up an SQL interpreter?) and setting up the data preconditions is expensive, as it can require to run whole module clusters which have their own tests, too. While that approach literally would satisfy the test independence requirement, it violates the (mostly unwritten) test usability requirement that errors in module A should be reported in the test module for module A and not in the `setUp` of another module B requiring module A (which thus is failing to perform its task).

Not mocking up the database can even cause much nastier problems, e.g. if db constraints get violated: The violating INSERT clause obtrusively sticks in the SQLAlchemy session, causing any subsequent tests trying to flush it to fail with the first error. Oh holy test independence. . . And searching the web for unit test database reveals rather reassurance that the problem hurts many people than enlightenment.

There is a contemporary trend in OO software development to consider the SQL language as a legacy and therefore to completely replace it with OO constructs. A proponent of that concept is Microsoft with its LINQ project. That kind of objects could be simpler to mock up than a set of database table content. But this is not the path the emores project travels on, as it has major drawbacks (see the remarks about ORM mapping in general on pg. 44).

Some literature about unit testing tends to put testing higher level classes with a lot of prerequisites and dependencies in the “functional testing” → “out of topic” basket and therefore is of little help for overcoming the problems when “just don’t do it” respective “tacitly limit the test to the trivial methods, for the record” (a common practice in the corporate IT world, I suppose) is not a seminal option.

During the development of emores, I had to change the policy from informal test interdependencies to expensive `setUp` routines as the codebase and therefore the number of tests grew. Non-explicit interdependencies proved to be unmaintainable (which was to be expected anyway). The high computational costs of setting up the same preconditions for several tests relatively decreased as the overall test costs increased: Whether running all tests takes 1 or 11/2 hours doesn’t really matter in usability. As a compromise, I didn’t re-generate the preconditions on each `setUp` (which is called for each test method of a `TestCase` instance), but used a global flag to set up the prerequisites only once per test module.

As the unit- and doctests don’t use mock objects, but heavily rely on the database, it is possible that a defective module inserts insane data another module cannot properly handle. Even when anything works as expected, one test module can insert too much data for another to process in a reasonable amount of time. One such vulnerable test is the top level `induction` module, reasoning over previously inserted data. Its unit test first deactivates the ‘Kafka’ example corpus in its `setUp` to reduce the amount of data to process for the case when another test module has fetched the example text, but not fully processed it.

In case of trouble, it should always be possible to first re-generate an initial database with the `db_create_tables` test script and then to run the failing test in isolation to fix the module in question until it works as expected.

The single one doctest where the lack of some implementation of test interdependence constraints in Python hurts most is the `sql_functions.txt` doctest (pg. 44), as a database summarisation test needs something to summarise if one wants to display some expedient results — but to check the results for correctness, it needs to control which data is created before, which it therefore can’t.

2.4.2 Test repeatability

Related to the problem of test interdependence is the problem of test repeatability. If all the tests always would completely clean up, this would be no problem. For simple low level tests, e.g. for the db access classes, cleaning up is comparatively simple, but for higher level classes, cleaning up is perhaps not even desired at all.

It would presumably be feasible to enforce repeatability by wrapping all tests in a database transaction which is always rolled back at the end of the test. The performance

penalty probably would be huge if this would be done in the `setUp/tearDown` pair.

A not so obvious drawback of isolating all tests within transactions is the fact that then the tests are run in an environment which is too clean to find defects in the software with regard to reliability and robustness. ¹

For now, I consider repeatability as desirable, which was the reason to add much more cleanup code in many tests. But I won't enforce it throughout, e.g. the `test_deduction.py` is not repeatable: If the example text has been completely computed, it remains as such in the database, and the assertion `self.assert_(len(new_lemmas) > 0)` fails.

2.5 Preparing a morphology

2.5.1 Use case: The morphology engineer

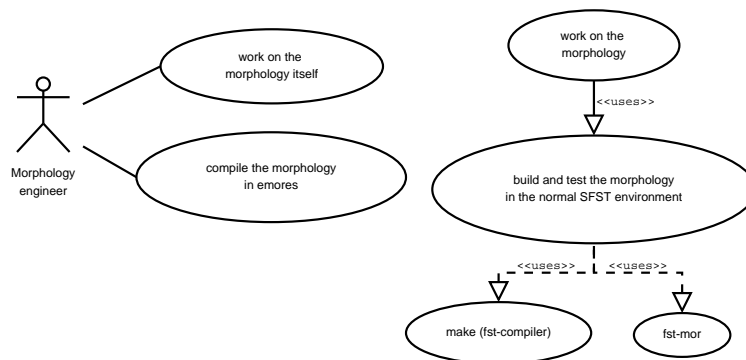


Figure 2.1: Work on the SFST morphology

Emores needs a morphology for any particular language. For the Python modules to work, the included transducers need to be compiled in advance. This is done by issuing `make` in the corresponding `./lang/[language-code]/morphology` directory (using the Makefile shipped with SFST).

The morphology gets compiled and inserted into the database with the `new()` method. It extracts all inflectional classes from the (seed) lexicon contained within the morphology and compiles a separate guesser transducer for each of them. For test examples see the “morphology.py” section under “SFST-related modules” on pg. 33. For setting up a working production environment, see the doctest section “Use cases scripts” on pg. 18

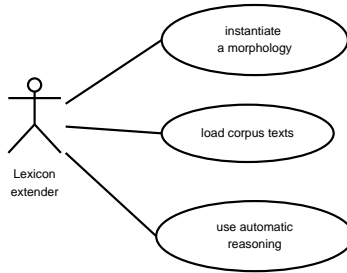


Figure 2.2: Extend the lexicon with emores

2.6 Using a morphology

2.6.1 Use case: The lexicon extender

To load a pre-compiled morphology for the configured language (passed in by the constructor), use the `load()` method. In both cases, the following object structure gets created:

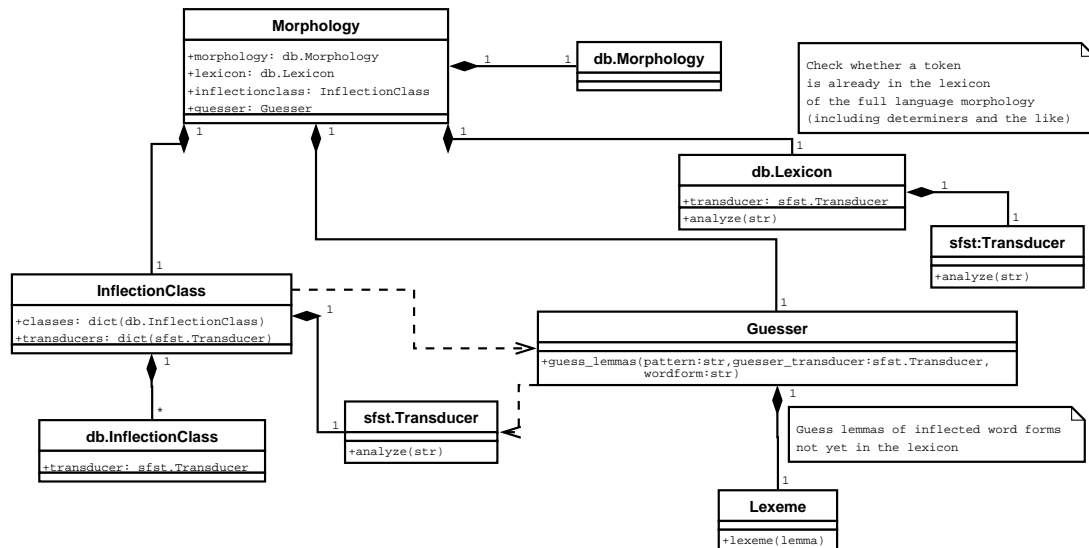


Figure 2.3: A fully instantiated morphology

The object graph only distantly resembles the relevant part of the database model (pg. 55). The significant difference might be viewed as an example of the object-relational impedance mismatch, as Neward described it in detail in “Object/Relational Mapping is the Vietnam of Computer Science” — or as intellectual laziness on my side.

¹One such defect was introduced with an SQL function which was used to simplify the query for the induction. The first test run succeeded, but repeating the failing test reliably reproduced a duplicate primary key constraint violation for the induction results. The problem and the symptom were rather distant, as the SQL function unexpectedly returned duplicate token when the same token was present in multiple texts. If all the transactions had been rolled back, that rather severe defect had silently lurked around much longer.

2.7 Use cases scripts

The intention of this section is to document a complete initialization and usage example for emores for the en_X and de_DE language each. Because the concrete sequence of steps matters, the complete scripts were written as one doctest.

Running `make` in the SFST language directories `./lang/en_X/morphology/` and `./lang/de_DE/morphology/` has been omitted due to unpredictable output (for the Python doctests), depending on whether there was something to build or not — it therefore is a precondition for proceeding.

These doctests are also the only ones where the proceeding from induction over deduction to abduction can be demonstrated, as the latter logically depend on the former. Isolated doctests for these modules thus have been omitted.

2.7.1 Use case doctest for English

Python imports and setup:

```
>>> import emores
>>> from emores import config
>>> from emores.morphology import Morphology
>>> from emores.text import Text
>>> from emores.induction import Induction
>>> from emores.deduction import Deduction
>>> from emores.abduction import Abduction
```

We deliberately set the language, regardless of its preselection as default language:

```
>>> conf = config.LanguageConfig()
>>> conf.set_language('en_X')
>>> conf.language
'en_X'
```

Preparing the morphology

The central access point for the compilation of a morphology is the `Morphology` class. After running `make`, `new` compiles anything and stores it in the database.

```
>>> morph = Morphology(conf)
>>> morph.new('usecase_en_X', '1.0.0')
>>> morph = None          # make sure to reload the object
```

Now the morphology engineer has finished his work and passes the laptop over to the lexicon extender to process a corpus text.

Using the morphology

First, we load the morphology just stored.

```
>>> morph = Morphology(conf)
>>> morph.load('usecase_en_X')
```

Then we retrieve additional corpus data. Multiple instances of the same text (corpus, name and url identically) are not supported, in that case the text gets retrieved only once. This is a rather slow operation because of the one-by-one filtering of any characters not belonging to the language (try to use finite state complementation with Unicode...).

```
>>> text = Text(conf)
>>> text.new('Kafka', 'Metamorphosis', 'http://www.gutenberg.org/files
/5200/5200.zip')
```

Now we perform the induction step: For any newly found token which isn't yet analyseable with the initial lexicon, all lemmas are guessed and inserted into the database. Because emores operates somewhat slowly, we limit the amount of data processed.

```
>>> induct = Induction(morph)
>>> induct.induce(limit=100)
```

This step really takes a long time. To spot progress, connect to the `emores_test` database using a tool like `pgadmin3` and compare the count of token to the count of asserted single inductions:

```
SELECT COUNT(*) FROM token;
SELECT COUNT(*) FROM induction;
```

If the induction has finished, there are induction database entries for each token that somehow was analyseable. After the induction has taken place, the deduction step can happen, also taking a lot of time:

```
>>> deduct = Deduction(morph)
>>> deduct.deducte()
```

Now there are deduction database entries for each lemma guessed in the induction step and the results can be examined by analysing the database directly using SQL or by using canned queries provided by the abduction module:

```
>>> abduct = Abduction(morph)
>>> saturated = abduct.saturation()
```

Now we audit one guessed lemma for the verb “to consider”:

```
>>> for row in saturated:
...     if row[0].find(u'consider<V>') > 0:
...         row_consider = row
>>> print row_consider
(u'<Stem>consider<V><base><native><VerbReg>', 0.75, 4L, 5L)
```

The lemma is not completely saturated, as the corresponding column is not 1, but only 0.75 on a generative lemma productivity of 4. To investigate which tokens lead to that saturation level of the lemma, we use the `deduction` reporting method of the `Abduction` class:

```
>>> sorted(abduct.deduction(u'<Stem>consider<V><base><native><VerbReg>'
))
[(u'consider', 3), (u'considered', 1), (u'considering', 1), (u'
considers', 0)]
```

There were 4 word forms found in the deduction range, but the fourth, “considers”, did not occur in our corpus, so the saturation level remains 3/4.

2.7.2 Use case doctest for German

Python imports and setup:

```
>>> import emores
>>> from emores import config
>>> from emores.morphology import Morphology
>>> from emores.text import Text
>>> from emores.induction import Induction
>>> from emores.deduction import Deduction
>>> from emores.abduction import Abduction
```

This time we really need to explicitly set the language:

```
>>> conf = config.LanguageConfig()
>>> conf.set_language('de_DE')
>>> conf.language
'de_DE'
```

Preparing the morphology

Because the SMOR morphology used for de_DE is very complex, the initialisation process really takes a long time. You can watch `tail -F /var/log/emores/emores.log` to see what's happening.

```
>>> morph = Morphology(conf)
>>> morph.new('usecase_de_DE', '1.0.0')
>>> morph = None # make sure to reload the object
```

Using the morphology

First, we load the morphology just stored.

```
>>> morph = Morphology(conf)
>>> morph.load('usecase_de_DE')
```

This is the same text as for the en_X example, but in the original language:

```
>>> text = Text(conf)
>>> text.new('Kafka', 'Die Verwandlung', 'http://www.gutenberg.org/
files/22367/22367-8.zip')
```

If the induction (and hence deduction and abduction) was limited as in en_X to 100 tokens to process, it would take ages (around 50 days) if run in the test suite. Therefore it is restricted to one single token which is saturable:

```
>>> induct = Induction(morph)
>>> induct.induce(token='Schritt')
```

```
>>> deduct = Deduction(morph)
>>> deduct.deduce()
```

For the abduction, we restrict the `saturation` and `deduction_count` levels accordingly to get the most likely lemmas. A saturation of 1.0 means that all generateable tokens by the lemma have been found in the corpus, and `deduction_count` value of 4 was found by taking the empirical maximum of the unrestricted results.

```
>>> abduct = Abduction(morph)
>>> saturated = abduct.saturation(token='Schritt',
...                               saturation=1.0, deduction_count=4)
>>> for row in saturated:
...     print row[0]
<Base_Stems>Schritt<NN><base><frei><NMasc-s/sse>
<Base_Stems>Schritt<NN><base><fremd><NMasc-s/sse>
<Base_Stems>Schritt<NN><base><nativ><NMasc-s/sse>
<Base_Stems>Schritt<NN><base><nativ><NNeut-s/sse>
<Base_Stems>schritt<NN><base><frei><NMasc-s/sse>
<Base_Stems>schritt<NN><base><fremd><NMasc-s/sse>
<Base_Stems>schritt<NN><base><nativ><NMasc-s/sse>
<Base_Stems>schritt<NN><base><nativ><NNeut-s/sse>
```

This is the maximal restriction we theoretically can get from the linguistically agnostic part of the emores machinery. But compared to the `deduction_binary_crosstab.de_DE.dat` file (the data for the example cluster dendrogram), the abductive narrowing mechanism is fairly powerful: it cuts the original guess of 307 lemmas down to 8, and half of them could be ruled out by the simple linguistic (and hence language specific) convention to lemmatise nouns in upper case in German.

A quick look at the deduction for the correct lemma gives evidence that it indeed produces all word forms:

```
>>> sorted(abduct.deduction(u'<Base_Stems>Schritt<NN><base><nativ><
    NMasc-s/sse>'))
[(u'Schritt', 2), (u'Schritte', 2), (u'Schritten', 1), (u'Schrittess',
    1)]
```

For a native speaker however, the result is still questionable: The genitive word form “Schrittss” would be possible, too, although the form with an “e” is preferable to avoid a sequence of three consonants (Duden, 1995, pg. 222). In that case, the SFST morphology seems to be too restrictive. But as this choice is a matter of style, it is very unlikely to find both forms in the same text or even in different texts from the same author.

Chapter 3

The Build Scripts

3.1 Makefiles

3.1.1 Main Makefile

The main Makefile located in the top directory defines these explicit targets:

- `db_scripts` Generates the `emores_db` database creation SQL-scripts out of of the Dia model using `tedia2sql`.
- `doc` Generates the pdf documentation.
- `texsrc` Generates the \LaTeX source from the doctests.
- `website` Generates the emores home page with plot thumbnails.
- `module` Builds the Python module.
- `clean` Removes the python module build directory and recursivel the doctest tex files.

3.1.2 Documentation Makefile

The documentation makefile is located in the `./doc/src/build` directory and defines these important targets:

- `bibliography` Generates the bibliography using `bibtex`.
- `glossary` Generates the glossary using the `makeglos.pl` script from the `glossary` package.
- `diagrams` Draws the UML diagrams with Dia.
- `rplots, tabular` Copies statistical results from the `./stats` directory.
- `chktex` Calls `chktex` to spot \LaTeX problems.
- `hyphen` Calls `hyphen_show` to spot hyphenation problems.

3.1.3 Doctests Makefiles

Each `./src/[...]doctest/` subfolder contains a Makefile for generating the \LaTeX sources included in this documentation. The Makefile first generates an XML representation of the doctest using the `rst2xml.py` script from the Docutils package. This XML representation afterwards is converted into a \LaTeX section using the `./doc/xslt/doctest.xsl` stylesheet with the `xsltproc` utility from the libxslt package. Its output needs to be postprocessed by the `./doc/xslt/unescape.py` script. The reason for this indirection is the fact that the complete \LaTeX document generated by the `rst2latex.py` script is not suitable for including in the main documentation.

3.2 Scripts Overview

The `./script` directory contains various build scripts for different parts of emores. Any of these scripts can be run from the working folder and will return there just before it finishes.

- `prefix`: Common include setting up local development paths to emores.
- `build`: Encapsulates the emores Python module build process.
- `db_create_database`: Calls the SQL script to create the emores database and activates PL/pgSQL and the `tablefunc` extension.
- `db_create_tables`: Calls the SQL script to create the tables in the emores database and the SQL functions.
- `text2unicode.py`: Utility script to convert the original SFST sources from Latin-1 to unicode.
- `test`: Runs the unit tests.
- `runpylint`: Does some static Python code checking.
- `cp_tablemodule`: Clones ORM class modules.
- `emores`: Startup emores within the same environment as for test.

3.2.1 prefix

The `prefix` script is included by the other scripts and defines these important environment variables:

- `PREFIX` Path prefix for the local development emores installation for all scripts, including the emores folder itself.
- `EMROES_LANG` Path to the `lang` morphology directory.
- `DEVEL` If not commented out, it causes tests to be run from the build directory `$PREFIX/build/lib/emores`
- `SFST_PATH` For `pysfst` development purposes: If not commented out, the `pysfst` module is loaded from the given path instead of the local installation.

3.2.2 db_create_database

Depending on the optional parameter `test` this script creates the postgresql database by first calling the `create_emores_user.sql` script and then the `create_emores_database.sql` or the `create_emores_database_test.sql` script respectively. It first creates the user `emores` and then the database itself. Running the script twice generates an error (which can be ignored) as the user cannot be dropped while one of the databases still exists.

3.2.3 db_create_tables

Depending on the optional parameter `test` this script creates the tables in the database determined by the optional parameter `test`. It first executes the `create_emores_tables.sql` and then the `grant_emores_privileges.sql` script which are generated by the main Makefile.

Afterwards it executes the manually written `populate_emores_tables.sql` script to fill the tables with some initial content the Python module tests need to successfully run.

3.2.4 cp_tablemodule

`cp_tablemodule [source_table] [destination_table]` copies a db ORM table module and its accompanying test module and adjust the module class/method/variables names appropriately. Terminates with an error if the `destination_table` already exists.

For example `cp_tablemodule text morphology` creates the files `./emores/src/db/morphology.py` and `./emores/src/db/tests/test_morphology.py`.

Any specialised `insert` and `select` methods need to be adjusted by hand. Additionally, the test must be activated in the `test.py` script. If some test data is needed, the corresponding `INSERT` statements need to be added in the `populate_emores_tables.sql` script.

See also the section “Extending the database” on pg. 43.

3.2.5 test

Builds `emores` and executes any unit tests.

Synopsis: `./test [ddd|dbg|pdb] [test-regexp]`

The optional `ddd|dbg|pdb` arguments control whether the tests are run within the GNU `ddd` GUI respective just using a custom-compiled debug-Python version outside of any debugger (`dbg`) or within the `winpdb` Python debugger (`pdb`). For more details see the script itself.

3.2.6 emores

Synopsis: `./test [ddd|dbg]`

`emores` works similarly as the `test` startup script. It executes the customised `import_emores.py` Python script to instantiate any modules needed for the current development activities. This script therefore changes often. There are two versions of `emores`, one in the `./scripts` subdirectory (designed for debugging, respecting

ddd|dbg) and a simpler one in the `./scripts/deploy` subdirectory, intended to be installed to `/usr/bin/emores` for starting up emores.

3.3 Statistical data processing

The `./stats` directory contains the scripts used to generate the diagrams and tables used in the chapter “Empirical Results” (pg. 48). By nature, a precondition for obtaining any results is some data. As producing that data is a computationally expensive and therefore very slow process, the statistical processing is not included in the main build action. Therefore the result diagrams are included in the emores source to make the documentation buildable without having to wait endlessly until “Die Verwandlung” eventually would have been completely processed for bootstrapping the results.

3.3.1 The Makefile

The `./stats/` working directory contains a `Makefile` linking the various script files together. Although in theory the main `all` target could be built provided that the database has been initialised in a way meeting the result descriptions in this documentation, it is usually used on a per-result file basis for a single task, invoked e.g. as `make lemma_count.de_DE.eps`. This is especially convenient for fine-tuning single diagrams.

3.3.2 Generating diagrams

For the diagrams, there are a bunch of files involved in creating them:

Suffix	Description
<code>.sql</code>	SQL views and selects generating the data required for R
<code>.py</code>	Python filter for isolating or formatting SQL result tables
<code>.dat</code>	Data frame produced by above SQL view
<code>.tex</code>	Data frame formatted for L ^A T _E X(optional)
<code>.R</code>	GNU R batch reading the data frame and producing the diagram
<code>.Rout</code>	Textual output of R processing the data (not used further)
<code>.eps</code>	Result diagram as encapsulated postscript file

For statistical analysis and generating the plots, first the data is fetched with plain SQL with the help of some global PL/pgSQL functions out of the postgresql database, eventually filtered with a Python filter script and then processed with R, the GNU package for statistical computing.

3.4 Installing emores

There are two scripts for installing emores: the Gentoo `ebuild` in the `./ebuild/app-tex/emores` directory and the `setup.py` script for the Python module installer. They create the following directory structure:

Path prefix	Description
/etc/emores/	Configuration files
/usr/bin/	Startup script (emores)
/usr/lib/emores/	Auxiliary scripts
/usr/lib/emores/sql/	Database setup SQL scripts
/usr/lib/python2.4/site-packages/emores/	Python emores module
/usr/share/emores/lang/	SFST morphology files
/usr/share/doc/emores-<version>/	Documentation
/var/log/emores/	Logging directory emores.log

Only the last two directories (var and doc) are created by the ebuild, any other files are installed by the `setup.py` script and therefore by Python. For customised packages on other platforms, consider therefore to patch this script.

Chapter 4

Code Documentation

4.1 Character encoding

As emores is all about natural languages, but tries to be language neutral by itself, the chosen character encoding of any language data is UTF-8. On the other hand, coding anything in UTF-8 (implying that the extended characters would actually be used) would introduce a strong dependency on the underlying operating system, e.g. supporting UTF-8 file paths if the config file allows it. Therefore the following encoding convention was used:

Data Type	ASCII	UTF-8
SFST sources		x
Python modules		x
Python scripts	x	
SQL database		x
SQL scripts		x
Dia diagrams	x	
Config files	x	
Shell scripts	x	
R batch files	x	

4.2 SFST sources

Any language specific information should be stored in the `./lang/[language-code]` directory. They all contain two subdirectories `./inflectionclass` and `morphology`. The latter contains the normal SFST morphology for that particular language, the former defines additional constructs exclusively used by emores.

4.2.1 inflectionclass sources

alphabet.fst

The `alphabet.fst` SFST source contains various alphabets, one used to recognise natural language texts and others for analysing SFST sources.

alphabet_filter.fst

The `alphabet_filter.fst` is a trivial transducer recognizing only characters used in the natural language alphabet. It is needed by the `tokenizer` Python module.

inflectionclass.fst

The file `inflectionclass.fst` generates a transducer which is used to extract the inflectional classes from the seed lexicon.

lexeme.fst

The file `lexeme.fst` generates a transducer which is used to extract the lexeme from an inflectional class or an analysis.

4.3 Python overview

4.3.1 Namespace layout

The namespace layout under the `./src` directory (which will become the main module namespace when installed) divides the modules into functional groups:

- `./` main modules and test runner
- `./tests` tests for the top level namespace
- `./db` SQLAlchemy database table mappers
- `./db/tests` tests for the db namespace

The environment variable `EMORES_HOME` defines the root of the emores namespace and can be inserted into Python's `sys.path` before the module is loaded.

4.3.2 Synonymic main/db modules

There are modules like `text` which are defined both in the main and in the table mapper namespace. In such a case, the main module is used for higher level methods using the underlying mapper module.

Because SQLAlchemy interprets inheriting from a mapper as table inheritance (see the chapter “Mapping a Class with Table Inheritance” in its documentation), the mapping would have to be repeated in the main class. Therefore the relation is implemented as an object composition: The main class owns the mapper class.

The main logical classes intended to be interactively used therefore don't use the ORM classes directly, but a “behavioural” wrapper in the root namespace which is composed of the corresponding ORM class from the db namespace. Behavioural wrappers usually define some kind of insert method using the ORM class. If there are additional arguments, as for the `url` keyword argument in the `text.insert()` method, these are used for arbitrary actions, in this example retrieving the text from the internet.

4.4 The test layout

In for each source subfolder in the `./src` directory, there are both a `./tests` directory with one unit test file for each source class and a `./doctests` directory with tests included by this main documentation. The `test.py` script contains the test runner to execute each of the tests (see the chapter “Running the tests” on pg. 14).

Having both unit tests and doctests in parallel introduces the question of where to put a specific test routine. The chosen heuristics goes something like this: The externally used behaviour of a class goes into the doctest, and correctness tests like carrying out edge cases or making assertions about member variables not used from outside go into the unit tests.

Some modules are dependent on other modules to provide required data (see the section about test interdependence on pg. 14), namely induction, deduction and abduction. The doctests for these modules have been merged into the usecase tests on pg. 18 and pg. 19.

I extensively used doctests because I like that marriage between documentation and working code. Doctests are a variant of literate programming: interactive Python sessions are embedded with human-readable documentation in a `reStructuredText` (see the Docutils web page) and automatically executed in Python when testing the code. But the LaTeX conversion is rather hackish, for details see the chapter about the corresponding Makefile on pg. 23.

When written this way, the code documentation is more likely to be accurate and up to date. Because each doctest shows a tiny use case for a module, their explanatory power usually surpasses what can be done with automatically generated API documentation.

Especially in Python, there’s an inherent tension opposite to traditional OO apidoc generation as used in environments like Java or .NET: In these languages, putting the canonical concept of information hiding using private methods and member variables into effect feels natural, and it is common to expose the way a class is intended to be used with an explicit interface. An automated apidoc generator can easily respect that structuring of the code.

But Python equally is a terse interactive scripting and prototyping language as it is a full OO programming environment, and the additional constructs to implement information hiding and explicit interfaces at least to me mostly feel somewhat stilted — albeit that viewpoint depends on the type of project, as sweepingly using such techniques in a generalised application framework like Zope3 certainly is reasonable.

As Zope3 is intended to be used do build specialised applications on top of it by peers, features like information hiding (from the peers) and interface stability count. In contrast, for dedicated research applications like emores, the situation is deeply different: At the time a class initially is written, it is not yet fully determinable how it would be used later on, as the development process is very iterative. And the final question of overall feasibility always lurks around behind the curtains. The codebase therefore remains a hybrid between a prototype and a real application, and thus features like a minimal amount of lines of code for a particular functionality count, too. Even some indetermination in the concrete implementation (particularly with respect to symbol visibility) can be considered as a desirable feature. These characteristics conflict with automated apidoc generation, which would expose too much on the one hand (due to the general publicity of all symbols) and too little on the other hand, as they’re not suited to “tell user stories” to clarify the intention of the code parts.

4.5 Main modules

4.5.1 utility.py

Small utility classes and functions for being used by other modules

Python imports and setup:

```
>>> import os
>>> import emores
>>> from emores import utility
>>> from emores import config
>>> from emores.utility import Cwd, QuoteBracket, SourceUtility
```

Cwd class

Compiling complex SFST transducers involves including source files from the filesystem. Therefore there is a need for an easy way to temporarily change the working directory. The Cwd class maintains a stack of working directories for that purpose.

```
>>> cwd = Cwd()
>>> current = os.getcwd()
>>> cwd.cd('/home/')      # 1st cwd change
>>> cwd.cd('/tmp/')      # 2nd cwd change
>>> cwd.cd()              # return to 1st cwd
>>> cwd.cd()              # return to the original cwd
>>> os.getcwd() == current
True
```

QuoteBracket class

For analysing arbitrary SFST source strings using SFST itself, any special symbols must be quoted. The chosen SFST symbols to represent < and > are <LT> and <GT>. They must not be part of the symbol set of any morphology. This constraint is nowhere explicitly checked — and thus could become a culprit of delicate bugs one of these days...

```
>>> qb = QuoteBracket()
>>> q = qb.quote(u'<><Symbol>:e')
>>> q == u'<LT><GT><LT>Symbol<GT>:e'
True
```

SourceUtility class

An utility class which centralises common SFST source code manipulation functionality needed by several modules. It currently implements only generating a language specific alphabet definition line:

```
>>> su = SourceUtility(config.LanguageConfig())
>>> include = su.language_alphabet()
>>> include.index('#include')
0
>>> include.index('ALPHABET = [#LANGUAGE_ALPHABET#]') > 0
True
```

unique_list function

Python doesn't seem to have a built-in unique list function, therefore I implemented a simple one:

```
>>> list_with_duplicates = ['a', 'b', 'a', 'c', 'b']
>>> unique_list = sorted(utility.unique(list_with_duplicates))
>>> unique_list
['a', 'b', 'c']
```

4.5.2 config.py

Emores configuration

Python imports and setup:

```
>>> import emores
>>> from emores import config
>>> from emores.config import LanguageConfig
```

These classes read the configuration files. The Config class looks for the main emores.cfg file in these places:

1. The file in the \$EMORES_CONF environment variable
2. The dotted .emores.cfg file in the user's ~/home directory
3. The ./etc/emores.cfg file in the emores directory tree

The Config class is not intended to be used by other modules. The main purpose of emores.cfg is to define the path to and the names of the available language files. The LanguageConfig class is used to access them.

The global DBConfig instance is available as dbconfig module attribute while the LanguageConfig must be instantiated by the original caller and is passed around as argument:

```
>>> config.dbconfig.user      # the standard database user
'emores'

>>> lang_cfg = LanguageConfig()
>>> lang_cfg.language        # the default language
'en_X'
>>> lang_cfg.main[-9:]       # the SFST main file for it
'morph.fst'
```

logging

The config module also provides the logger configuration using the Python logging module. The configuration file logging.cfg is read from the same directory where emores.cfg is found. A global logger is instantiated as log.

```
>>> from emores.config import log
>>> log.debug("config.txt logging doctest")
```

The standard configuration uses a FileHandler such that the log can be watched using e.g. tail -F /var/log/emores/emores.log.

4.5.3 text.py

Read arbitrary texts from the internet

Python imports and setup:

```
>>> import random          # for generating unique text names
>>> import emores
>>> from emores import config
>>> from emores.text import Grabber, Text
>>> from emores.db import db
```

The `text` module is used for getting text and extracting new word forms (tokens) for `emores`. The `Grabber` class provides some `wget` functionality for fetching HTML and text pages:

```
>>> grabber = Grabber()
>>> text = grabber.get('http://google.com/robots.txt')
>>> text.find('Disallow:') > 0
True
```

The `Text` class itself inserts into the `text` database table: Its main entry point is the `new(corpus, name, url)` method which uses the `token` module to tokenize and store all newly found words. Here we use a random name for repeatedly inserting the same test text in the tests, as a second insert with the same name and url will not be performed:

```
>>> text = Text(config.LanguageConfig())
>>> name = str(int(random.uniform(0, 10**10)))
>>> text.new('test', name, 'http://google.com/robots.txt')
```

For overall test predictability, it is crucial that we deactivate this “test noise” text. Otherwise the output of `test_abduction.py` and `usecase_en_X` gets disturbed.

```
>>> db.execute_update("""
...     UPDATE text
...     SET active = FALSE
...     WHERE name = %s
...     """, parameters=(name,))
```

4.5.4 token.py

Tokenize corpus texts

Python imports and setup:

```
>>> import emores
>>> from emores import config
>>> from emores.db import db
>>> from emores.token import Tokenizer, Token
```

```
>>> conf = config.LanguageConfig()
>>> conf.set_language('en_X')
```

The `token` module is used by the `text` module. The `Tokenizer` class removes all non-language chars from an input text and breaks it up into single tokens, returning a dictionary with a word count for each found token:

```
>>> tokenizer = Tokenizer(conf)
>>> tokens = tokenizer.tokenize(u'My emores work costs people $0.-.')
>>> sorted(tokens.keys())
```

```
[u'My', u'costs', u'emores', u'people', u'work']
>>> tokens = tokenizer.tokenize(u'Uh oh oh oh')
>>> oh_count = tokens[u'oh']
>>> oh_count
3
```

The Token class inserts into the token and texttoken tables for the test text with textid=1. Inserting token already existing produces a test interdependence problem, as deleting them might not work no more due to foreign key constraints. Therefore I used french token which are unlikely to occur in an en_X text.

```
>>> token = Token(conf)
>>> txt = u'Ajoute nouveau terme...'
>>> token.accomodate(1, txt)
>>> inserted_token = db.execute_list("""
... SELECT t.token
... FROM token t
... INNER JOIN texttoken tt
...     ON tt.tokenid = t.tokenid
... WHERE tt.textid = 1
... """)
>>> sorted(inserted_token)
[u'Ajoute', u'nouveau', u'terme']
```

At last, cleanly clean up (there was a time I thought it could be done without):

```
>>> db.execute_update("DELETE FROM texttoken WHERE textid = 1")
>>> db.execute_update("""
... DELETE FROM token WHERE token IN
... ('Ajoute', 'nouveau', 'terme')
... """)
```

4.6 SFST-related modules

4.6.1 morphology.py

Compiling a morphology

Python imports and setup:

```
>>> import emores
>>> from emores import config
>>> from emores.morphology import Morphology
```

Only one version of a morphology for a particular language can be active at the same time. The morphology gets compiled and inserted into the database with the new (name, version) method, here for the default language en_X. The associated lexicon implicitly gets the same name/version pair.

```
>>> morph = Morphology(config.LanguageConfig())
>>> morph.new('doctest morphology', '1.0.0')
>>> morph = None
```

After storing that particular morphology, load it again.

```
>>> morph = Morphology(config.LanguageConfig())
>>> morph.load('doctest morphology')
```

If the lexicon name differs from the morphology name, it can be loaded separately:

```
>>> morph.load('en_X-scratch', 'empty lexicon')
```

4.6.2 inflectionclass.py

Extract the inflectional classes from an SFST lexicon

Python imports and setup:

```
>>> import emores
>>> from emores import config
>>> from emores.inflectionclass import InflectionClass
>>> from emores.transducer import GuesserTransducer
```

The `inflectionclass` module is used to extract the inflectional classes from a seed lexicon as part of the SFST morphology. The found classes don't necessarily mirror the morphological inflection classes found in books for a particular language, as the extraction method is just formal: After removing any non-inflectional letters from each lexical entry and replacing them with wildcards, the remaining distinct strings count as inflectional classes. The `InflectionClass` uses the `transducer` module to generate the `GuesserTransducer` instances.

```
>>> infl = InflectionClass(config.LanguageConfig())
>>> lemma = u'<Stem>mo:iu:<>s:ce<N><base><native><NounPl>'
>>> infl.accommodate_lemma(lemma)
>>> len(infl.classes)          # expect 1 inflectionclass from our lemma
1
>>> infl_class = infl.classes.keys()[0]
>>> infl_class
u'<Stem>.+o:iu:<>s:c.+<N><base><native><NounPl>'
```

To be able to use that inflectional class, we need to compile it. The transducer instance is accessible via the inflectional class as key and produces a (usually wrong) analysis for a particular wordform; for `emores` it's only important that the inflectional class actually can analyze that particular wordform.

The `compile_inflectionclasses` method needs a `GuesserTransducer` instance:

```
>>> guessertransducer = GuesserTransducer(config.LanguageConfig())
>>> infl.compile_inflectionclasses(guessertransducer)
>>> infl_transducer = infl.transducers[infl_class]
>>> repr(infl_transducer)[:16]
'<sfst.Transducer'
>>> sorted(infl_transducer.analyze(u'mice'))
[u'<Stem>Mouse<N><base><native><NounPl><pl>', u'<Stem>mouse<N><base><native><NounPl><pl>']
```

For guessing real lemmas (without the trailing analysis annotation as `<pl>` in the example), use the `guesser` module.

4.6.3 transducer.py

Compile specific transducers

Python imports and setup:

```
>>> import emores
>>> from emores import config
>>> from emores.transducer import \
... LanguageTransducer, LemmaTransducer, GuesserTransducer
```

These `*Transducer` classes are factories for generating specific SFST `Transducer` instances via their `compile()` method.

The `LanguageTransducer` class encapsulates the compilation of a language main source file (which probably will contain includes) with an explicit lexicon passed as a string argument to `compile`.

The main `.fst` file is the original one, therefore the `de_DE` transducers also analyze and generates non-inflected word forms like pronouns (`PRO.fst`) or numbers (`NUM.fst`). This broader acceptance range is needed for skipping these word forms for inductions.

```
>>> conf_de = config.LanguageConfig()
>>> conf_de.set_language('de_DE')
>>> lat = LanguageTransducer(conf_de)
>>> t1 = lat.compile(u'<Base_Stems>Dompfaff<NN><base><nativ><
    NMasc_es_en>')
>>> type(t1)
<class 'sfst.Transducer'>
>>> t1.analyze(u'Dompfaffs') # uses the lemma just compiled
[u'Dompfaff<+NN><Masc><Gen><Sg>']
>>> t1.analyze(u'ich')
[u'ich<+PPRO><pers><l><Sg><NoGend><Nom>']
>>> t1.analyze(u'zwo')
[u'zwo<+CARD>']
```

The `LemmaTransducer` subclass derived from `LanguageTransducer` already uses the disjunction-free (and therefore algebraically solvable) `main_guesser` file and is optimized for compilation speed by using `solve_precompile` from the `pysfst` package. It is used for generating all word forms for a particular Lemma. Otherwise, it behaves like its parent class:

```
>>> let = LemmaTransducer(config.LanguageConfig())
>>> t2 = let.compile(u'<Stem>easy<ADJ><base><native><AdjReg>')
>>> type(t2)
<class 'sfst.Transducer'>
>>> t2.analyze(u'easier')
[u'easy<ADJ><comparative>']
>>> sorted(t2.generate(u'easy<ADJ><comparative>'))
[u'easier', u'easiier', u'easiyer', u'easyer', u'easier', u'easier']
```

Obviously the shipped `en_X` morphology massively overgenerates (and therefore recognizes wrong word forms). It therefore is unlikely that all of these forms will be found in corpus data and thus the lemma technically never can be saturated.

In contrast, the `GuesserTransducer` subclass is used to compile an inflectional class lexicon generated by `InflectionClass`. It also uses `solve_precompile` like `LemmaTransducer`, but actually solves the `main_guesser` for the lemma. Therefore the generated transducers can be used as a guesser for the lemma for a word form:

```
>>> get = GuesserTransducer(config.LanguageConfig())
>>> t3 = get.compile(u'<Stem>.+<ADJ><base><native><AdjReg>')
>>> sorted(t3.analyze(u'easier'))
[u'<Stem>Easie<ADJ><base><native><AdjReg><comparative>', u'<Stem>Easier
<ADJ><base><native><AdjReg><positive>', u'<Stem>Easy<ADJ><base><
native><AdjReg><comparative>', u'<Stem>Easye<ADJ><base><native><
AdjReg><comparative>', u'<Stem>Easier<ADJ><base><native><AdjReg><
positive>', u'<Stem>easie<ADJ><base><native><AdjReg><comparative>',
u'<Stem>easier<ADJ><base><native><AdjReg><positive>', u'<Stem>easy
<ADJ><base><native><AdjReg><comparative>', u'<Stem>easye<ADJ><base
><native><AdjReg><comparative>', u'<Stem>easier<ADJ><base><native><
AdjReg><positive>']
```

In this case (input word in comparative form), the guesser also massively overgenerates. For the non-comparative case, the result is much better, there is no obvious overgeneration except the upper case for the first letter which is needed to recognize word forms occurring at the beginning of a sentence:

```
>>> sorted(t3.analyze(u'easy'))
[u'<Stem>Easy<ADJ><base><native><AdjReg><positive>', u'<Stem>easy<ADJ><
base><native><AdjReg><positive>']
```

4.6.4 guesser.py

Guess lemmas using inflectional classes

Python imports and setup:

```
>>> import emores
>>> from emores import config
>>> from emores.transducer import GuesserTransducer
>>> from emores.guesser import Guesser
```

The `guesser` module provides a core functionality for `emores`: to guess a lemma for a wordform using a particular inflectional class:

```
>>> guesser = Guesser(config.LanguageConfig())
>>> mice_guesser = GuesserTransducer(config.LanguageConfig())
>>> mice_inflectionclass = u'<Stem>.+o:iu:<>s:c.+<N><base><native><
NounPl>'
>>> mice_guesser_transducer = mice_guesser.compile(mice_inflectionclass
)
>>> guessed_lemmas = guesser.guess_lemmas(
... mice_inflectionclass, mice_guesser_transducer, u'mice')
>>> sorted(guessed_lemmas) # matches our mice_inflectionclass
[u'<Stem>Mo:iu:<>s:ce<N><base><native><NounPl>', u'<Stem>mo:iu:<>s:ce<N
><base><native><NounPl>']
```

The lower cased guessed lemma is identical to the lemma found in the SFST `en_X` example lexicon: `<Stem>mo:iu:<>s:ce<N><base><native><NounPl>`. The upper cased guess is an artefact produced by the morphology to recognize corresponding token at the beginning of a sentence.

4.7 Python DB ORM classes

4.7.1 Common characteristics

There is a module in the `db` package for each database table. They use the ORM (object relational mapping) functionality provided by `SQLAlchemy`. All of the table classes inherit from the `db.DBTable` base class. There are at least two module functions and one base class method for accessing the database. The `identifier` is in the majority of cases symbolic (for intuitive interactive use). It often restricts the select to the respective name column of the underlying tables.

- `[tablemodule].select(identifier)` Factory method for exactly one database object instance. Raises an exception if there is none or more than one row.

- `[tablemodule].[table]id(identifier)` Returns the primary key id of the selected item as an integer. Raises an exception if there is none or more than one row.
- `[tableclassinstance].insert([identifier])` Inserts a row for the current object instance into the database. The SQLAlchemy session is immediately flushed to the database only if there's a primary key id to retrieve (see below). Raises an exception if the row corresponding to the identifier already exists due to DB uniqueness constraints.

4.7.2 The insert methods

The class insert method takes its argument in the same order as the module select method. If the table contains foreign keys, they often must be symbolically specified, e.g.

```
corpus.insert(language, name)
```

is used as

```
corpus.insert('de_DE', 'goethe').
```

Optional columns (nullable or with a default value) can always be added using keyword arguments, e.g.

```
corpus.insert('de_DE', 'arno.schmidt', active=False).
```

An insert call returns the primary key id of the row just inserted for tables having children which should be inserted, too. To retrieve this id at any time, user the `[table]id(name)` function.

Per default, the insert is immediately flushed. If the new row violates some uniqueness constraints of the database table, an exception is raised. This is always the case for tables that are not automatically filled, e.g. the language table, which is intended to be populated by hand.

On the other hand, there are tables which usually are filled by the emores engine, e.g. the token table, in the sense of accommodation. Its insert method just returns the tokenid if the token already exists.

4.7.3 Delayed flushing to the database

The basic idea was to hide the concrete ORM technique and also the library from the emores main source and to produce a distinct OO interface using above class-specific tailored insert methods for each ORM class.

SQLAlchemy performance problems partially caused by immediately flushing each object insert to the database complicated the machinery a bit, because obtaining a primary key id inevitably requires actually writing the corresponding row to the database: The OO caller therefore needs to “know” about the underlying DB for delaying flushing for the sake of performance optimization. This contradicts the usual N-tier design. The other way round, SQLAlchemy cannot “know” whether there is a primary key id that is actually needed as return value of an insert method.

The derived ORM classes thus need a special flush argument to “tell” its base class, DBTable, whether an immediate call to flush() is required. And the behavioural classes need to manually flush the session after accomplishing an user-driven “unit of work”, especially when emores is used interactively. This is the single point where the leak in the N-tier abstraction hurts most: The most abstract classes need to dive down and directly call flush() on the low level part of the database tier.

For some enlightenment about the theoretical inevitability of such leaks, you might want to take a look at Spolsky: “The Law of Leaky Abstractions”.

4.7.4 db/db.py

Database connectivity

Python imports and setup:

```
>>> import emores
>>> from emores.db import db
```

The `db.db` module defines the database connectivity, both for the higher level SQLAlchemy API and the lower level Python DB 2 API using `psycopg2` (which just uses the connection pool facility of SQLAlchemy).

psycopg2 usage

Here’s a simple example for directly using SQL to retrieve one value (here the first token in the database) with `execute_scalar`. The second function parameter is a list of SQL parameters to be interpolated into the query:

```
>>> sql = """
... SELECT token
... FROM token
... WHERE tokenid = %s
... """
>>> token = db.execute_scalar(sql, (1,))
>>> token
u'mouse'
```

If several values are to be retrieved, use `execute_list`:

```
>>> sql = """
... SELECT token
... FROM token
... """
>>> tokens = db.execute_list(sql)
>>> type(tokens)
<type 'list'>
```

To change the database, use `execute_update`. For clean execution, insert and delete in two statements. The optional second parameter passes any arguments:

```
>>> sql = """INSERT INTO text (corpusid, name, text)
... VALUES(1, 'test db.txt', '')
... """
>>> db.execute_update(sql, (1,))
>>> sql = """
... DELETE FROM text
... WHERE name = %s
... """
>>> db.execute_update(sql, ('test db.txt',))
```

4.7.5 db/language.py

ORM class for the language table

Python imports and setup:

```
>>> import emores
>>> from emores.db import language
```

As `language` is not dependent on any other table, the identifier is just the code column. It is usually used by other table modules referencing the `language` table for performing inserts with symbolical foreign key specification.

```
>>> lang = language.select('en_X')
>>> lang.languageid
1
```

4.7.6 db/corpus.py

ORM class for the corpus table

Python imports and setup:

```
>>> import emores
>>> from emores.db import corpus
```

Identical corpora for several languages can coexist, therefore the foreign key identifier is `language, name`.

```
>>> corpusid = corpus.corpusid('en_X', 'Kafka')
>>> corpusid
2
```

4.7.7 db/text.py

ORM class for the text table

Python imports and setup:

```
>>> import emores
>>> from emores.db import text
```

Each text belongs to a corpus, therefore the identifier is `language, corpus, name`.

```
>>> textid = text.textid('en_X', 'Kafka', 'emptiness')
>>> textid
2
```

4.7.8 db/tokenype.py

ORM class for the tokentype table

Python imports and setup:

```
>>> import emores
>>> from emores.db import tokentype
```

Token type are predefined and thus not intended to be inserted with this module. The foreign key identifier is its name:

```
>>> tokentypeid = tokentype.tokenypeid('wordform')
>>> tokentypeid
1
```

4.7.9 db/token_.py

ORM class for the token table

Python imports and setup:

```
>>> import emores
>>> from emores.db import token_
```

The trailing underscore is required because `token` is a built-in Python module. Tokens are not dependent on any particular language (they can belong to several languages, dependent on the text/corpus they're found in), therefore the selector is a single unicode string:

```
>>> tokenid = token_.tokenid(u'mice')
>>> tokenid
2
```

For inserting a token, the `tokentype` must be explicitly stated. Here we take an existing token (which therefore will actually not be written to the database):

```
>>> token = token_.Token()
>>> tokenid = token.insert('wordform', u'mice')
>>> tokenid
2
```

4.7.10 db/texttoken.py

ORM class for the texttoken table

Python imports and setup:

```
>>> import emores
>>> from emores.db import db, texttoken
```

Because it is usually used only internally during the acquisition of new texts, the foreign key identification for inserts is not symbolical. Only where the id's are known, the identifier for the insert method is `textid`, `tokenid`, `count`.

```
>>> token_count = texttoken.TextToken()
>>> token_count.insert(1, 1, 17)
>>> db.flush() # explicitly needed due to composite primary key
>>> sess = db.Session()
>>> query = sess.query(texttoken.TextToken)
>>> db_token = query.filter_by(textid = 1,
...                           tokenid = 1).one()
>>> db_token.count
17
```

At last, clean up:

```
>>> db.execute_update("""DELETE FROM texttoken
...                       WHERE textid = 1 AND tokenid = 1""")
```

4.7.11 db/morphology.py

ORM class for the morphology table

Python imports and setup:

```
>>> import emores
>>> from emores.db import morphology
>>> import sqlalchemy
```

The identifier is language, name, optionally version='...' and only retrieves active morphologies:

```
>>> morphologyid = morphology.morphologyid('de_DE', 'de_DE-scratch',
... version='0.0.0')
>>> morphologyid          # "active" in the db
2
```

For initialisation purposes, there is an exists method, checking whether a given morphology already exists in the database marked as active:

```
>>> morphology.exists('de_DE', 'de_DE-scratch')
True
>>> morphology.exists('de_DE', 'de_CH not yet considered')
False
```

However, the language must exist, otherwise an exception gets thrown:

```
>>> try:
...     morphology.exists('de_CH', 'Swiss German not yet considered')
... except sqlalchemy.exceptions.InvalidRequestError, error:
...     print error
No rows returned for one()
```

4.7.12 db/lexicon.py

ORM class for the lexicon table

Python imports and setup:

```
>>> import emores
>>> from emores.db import lexicon
```

Its identifier is symbolic: language, morphology_name, lexicon_name. It can be used to retrieve any active language transducer using that particular lexicon. Lexicon uses the transducer storing mechanism to instantiate it.

```
>>> db_lexicon = lexicon.select('de_DE',
... 'de_DE-scratch', 'empty lexicon')
>>> type(db_lexicon.transducer)
<class 'sfst.Transducer'>
```

4.7.13 db/inflectionclass.py

ORM class for the inflectionclass table

Python imports and setup:

```
>>> import emores
>>> from emores.db import inflectionclass
```

Its identifier is (non-symbolically) morphologyid, pattern, but it's usually retrieved by the selectall method to analyse a new token. Inflectionclass uses the transducer storing mechanism.

```

>>> existing_infl = '<Stem>.+<ADJ><base><native><AdjReg>'
>>> morphologyid = 1
>>> infl = inflectionclass.select(morphologyid, existing_infl)
>>> infl.inflectionclassid
1

```

An exact length test is not predictable within the test framework, as e.g. `test_inflectionclass.py` uses the scratch morphology, too and inserts data which it cannot retract easily.

```

>>> classes = inflectionclass.selectall(morphologyid)
>>> len(classes) >= 2
True

```

4.7.14 db/lemma.py

ORM class for the lemma table

Python imports and setup:

```

>>> import emores
>>> from emores.db import lemma

```

Its identifier is `lemma`. Inserting the same lemma twice just returns the existing `lemmaid`. `Lemma` uses the transducer storing mechanism to instantiate the corresponding transducer:

```

>>> existing_lemma = '<Stem>easy<ADJ><base><native><AdjReg>'
>>> lemm = lemma.select(existing_lemma)
>>> lemm.lemmaid
1
>>> type(lemm.transducer)
<class 'sfst.Transducer'>

```

Chapter 5

The Emores Database

5.1 Database model

The database model is drawn with Dia and its XML source is converted to SQL using `tedia2sql`, therefore the diagram is physically the actual source code in the sense of “the preferred form of the work for making modifications to it” (GNU general public license Version 2, chap. 3).

A spontaneous drawing written with paper and pencil was the very beginning of the emores project. The database model is the main conceptual model of emores. Any logical decision taken during emores’ operation should be traceable by analyzing the data in the database, without the need for instantiating any SFST transducers or even stepping through debugging sessions.

An important feature of the normalised relational model is the dependency of any logical reasoning step in the tables `induction` and `deduction` from a particular morphology, the former through `inflectionclass`, the latter indirectly through `lemma` and `induction` or through `lexiconlemma` and `lexicon`.

The current database ERM can be found on in the appendix A on pg. 55

5.2 Special columns

Any columns usually are mapped to corresponding object attributes by SQLAlchemy, but there are exceptions (those columns are prefixed with an `s` for Python strings): Within emores, the application uses `sfst.Transducer` instances, but in the database, they need to be stored as binary transducer `byteareas`. An `stransducer` column contains these. When loaded via the `db` module, the object instances are reconstructed as an additional `transducer` attribute. Another exception are the `slemmatransducersrc` and `sguessertransducersrc` columns which hold Python pickles of the pre-compiled algebraical solutions for quickly generating `LemmaTransducer` and `GuesserTransducer` instances.

5.2.1 Extending the database

To add another table and make it accessible within emores, perform these steps:

1. Draw it with Dia using the UML sheet in the diagram `emores/doc/src/dia/emores_db.dia` like the other tables.
2. Run `make` in the `emores` root directory to generate the `create_emores_tables.sql` DDL script. (see pg. 22)
3. Clone a similar ORM class/tests module pair with the `cp_tablemodule` script in the `emores/scripts` directory, (see pg. 24) and adapt the `select/insert` methods for your needs.
4. Add a `doctest` in the `emores/src/db/doctest` directory.
5. Add a `run` statement for the unit test and the `doctest` in the `emores/src/test.py` test driver script.
6. Optionally add some `INSERT` statements in the `emores/sql/populate_emores_tables.sql` script.
7. Run the `db_create_tables_test` script in the `emores/scripts` directory (see pg. 24) to actually create the new table in the database.
8. Run the tests and fix the code until the tests succeed.

Erm, these are lots of steps, and I'm tempted to admit that I might not have shown enough consideration for the advice from Neward about ORM mapping: "Developers must be willing to take the "wins" where they can get them, and not fall into the trap of the Slippery Slope by looking to create solutions that increasingly cost more and yield less".

At least the division between using SQLAlchemy mainly for inserting data (and delegating the task of typecasting or retrieving the generated primary key to it) and directly using SQL via `psycopg2` for more complex data mining purposes (including data export to R) could generously be interpreted as a conscious "Acceptance of O/R-M limitations" (Neward).

5.3 Database Programming

To ease performing data mining tasks within the database itself, some data retrieving functions have been refactored from the `db` modules to the `postgresql` database using SQL directly and new functions were added on top of them using PL/pgSQL. The functions are implemented in the file `./sql/create_emores_functions.sql` and "doctested" with the file `./src/db/doctests/sql_functions.txt` below.

5.3.1 SQL functions accessed by `db/db.py`

Test SQL functions implemented in the DB itself

Python imports and setup:

```
>>> from emores.db import db
```

Remark about testability

Because the primary intent of these SQL functions is to summarize the the database content, the strict test interdependence prohibition doctrine makes it almost always impossible to check the actual results within this test, as especially the usecase doctests enter useful data, and it's quite their intent to leave that data in the database for further investigation after the tests have been run.

On the other hand, providing verifiable data for the summarisation functions would require to run a significant amount of code just for these tests.

Simple scalar SQL functions

These functions (currently only one) are mirroring corresponding functions in the db modules:

```
>>> sql = """
... SELECT morphologyid('de_DE', 'de_DE-scratch')
... """
>>> morphologyid = db.execute_scalar(sql)
>>> morphologyid
2
```

Table functions

Table functions are returning a restricted rowset from a particular table. If SQL was used directly, several joins would be needed (as can be seen in the function definitions).

The token function is used to retrieve unique token from the active corpus texts in a particular language.

```
>>> sql = """
... SELECT *
... FROM token('en_X')
... ORDER BY token
... LIMIT 1
... """
>>> token_rows = db.execute_rows(sql)
```

The induction function retrieves the valid induction rows for a particular language and morphology and is a building block for performing data mining:

```
>>> sql = """
... SELECT tokenid, lemmaid, inflectionclassid
... FROM induction('en_X', 'usecase_en_X')
... ORDER BY tokenid, lemmaid
... LIMIT 1
... """
>>> induction_rows = db.execute_rows(sql)
```

The deduction function looks almost the same:

```
>>> sql = """
... SELECT lemmaid, tokenid
... FROM deduction('en_X', 'usecase_en_X')
... ORDER BY lemmaid, tokenid
... LIMIT 1
... """
>>> deduction_rows = db.execute_rows(sql)
```

Dynamic functions

It's not the point of doctests to show all auxiliary functions involved in dynamically creating a (weakly typed) crosstab view within the strongly typed language SQL. For demonstration purposes, we here create a crosstab for morphologies and languages. The first argument names the view to create, the second argument gives the source SQL for the data containing 3 columns (rowid, category, and values) and the third argument the SQL retrieving only the category (here the language).

```
>>> view_sql = """
... SELECT binary_crosstab(
...   'v_lang_morph',
...   'SELECT l.code, m.name, 1
...   FROM language l
...   INNER JOIN morphology m
...     ON m.languageid = l.languageid
...   AND m.active = TRUE
...   ORDER BY l.code, m.name'
... ,
...   'SELECT m.name
...   FROM morphology m
...   WHERE m.active=true
...   ORDER BY m.name'
... )
... """
>>> db.execute_update(view_sql)
>>> query_sql = """
... SELECT *
... FROM v_lang_morph -- above view
... ORDER BY 1
... """
```

The `binary_crosstab` function is intended to be used in the SQL environment to return a textual report to be imported in the R statistical environment. For doctest purposes, we need to manually extract the important column names from an explicit cursor, which is rather complicated. And the result can only be checked cursorily:

```
>>> conn = db.connection()
>>> curs = conn.cursor()
>>> curs.execute(query_sql)
>>> columns = []
>>> for column_desc in curs.description:
...     columns.append(column_desc[0])
>>> rows = curs.fetchall()
>>> curs.close()
>>> conn.close()
```

Now the columns contain all morphology names, we can check only for those entered with the initial database population:

```
>>> columns.count('en_X-scratch')
1
>>> columns.count('de_DE-scratch')
1
```

Finally, we should clean up by deleting the view.

```
>>> drop_view = """
... DROP VIEW v_lang_morph
... """
>>> db.execute_update(drop_view)
```

In the `./stats/Makefile`, the SQL output for the generated view is generated using a `psql` invocation like this:

```
psql -d emores_test -U emores -q -P format=unaligned -P
border=0 -P fieldsep=''-P null=0 -P footer -f view.sql >
view.dat
```

The tabular output (with the output of the function call removed) could look like this (NULL values converted to 0):

```
row.name de_DE-scratch en_X-scratch usecase_en_X
de_DE 1 0 0
en_X 0 1 1
```

This table is a valid R data file constituting an asymmetric binary observation telling which morphologies are present for which language. It can be imported using this simple R statement:

```
b <- read.table("view.dat", header=T, row.names="row.names
")
```

Chapter 6

Empirical Results

6.1 The data

The empirical data used for generating the statistics was a text from Kafka: “Die Verwandlung” (the original, used for `de_DE`) respective “Metamorphosis” (for `en_X`, translated by David Wyllie). Of course, this is a small sample, but the overall processing turned out to be excessively slow, so this must suffice for now.

But most of the “empirical data” used to generate the results presented in that chapter is not data about the linguistic behaviour of humans in general, but rather data about its implementation in an SFST morphology, which, of course, reflects the former, too. While induction and deduction (pg. 49) rather describe the morphology itself, abduction (pg. 52) describes the behaviour of the morphology with respect to natural language.

6.2 Computational performance

The calculations where performed on a laptop with a 1.83GHz Intel Core Duo processor and 2GB of 333MHz RAM. All C/C++ code was compiled with `gcc 4.1.2` for `-march=prescott`.

The per-process CPU usage for the induction and the deduction steps are completely different, as this table (based on manual observation and estimation) for the full size `de_DE` morphology shows:

Task	python	postmaster	Avg minutes per token
Induction	95%	5%	16:24
Deduction	10%	90%	02:41

The induction step is obviously the slowest one. For an ordinary 500'000 tokens corpus, a processing time of 15 years is to be expected :’-(

6.2.1 Possible optimisations

Both the induction and the deduction step consist of a massive amount of logically independent computing, which in theory would make possible equally massive parallelisation. But the database is built on the concept of referential integrity across single induction/deduction steps and therefore cannot easily be partitioned (such capabilities

are not built into the community postgresql engine and therefore would require something like the Mammoth PostgreSQL Replicator).

However, the good news within the disaster is that the induction step in fact could be parallelised: By far the most time is spent within the SFST compiler, and the expansions into separately compiled lemma transducers are completely independent of each other.

As the compiler is implemented as a non-pure flex/bison parser in SFST, it is guarded against being accessed by concurrent threads. But the yet-to-expand token list is generated by querying the database, therefore refactoring the corresponding SELECT statement into a transactional iterator should suffice.

6.3 Induction

The induction step generates lemmas from the token in the text. As this is highly ambiguous, a single token is expected to generate many different lemmas. This simple histogram shows the distribution of the number of lemmas generated by one token. It does not distinct between lemmas generated by distinct inflectional classes and ambiguous analyses of one inflectional class leading to similar lemmas.

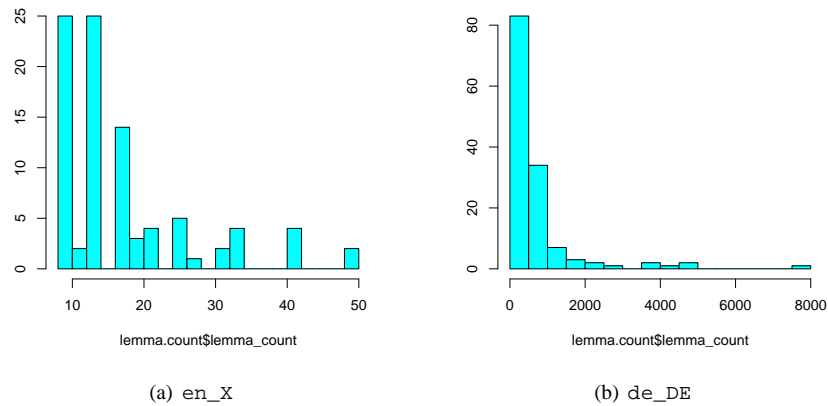


Figure 6.1: Induction: lemmas per token

The generative power of the induction for en_X and de_DE differs vastly by about two orders of magnitude. This reflects mainly the bigger seed lexicon leading to more technical inflectional classes for the de_DE morphology (5 vs. 440).

6.4 Deduction

The subsequent deduction step generates any token generateable by each lemma for the given morphology. The simple histogram shows the distribution of the number of token generated by one lemma.

Although the de_DE distribution seems to be less skewed right than the one for en_X, the difference is rather small. The reason may be that the the German mor-

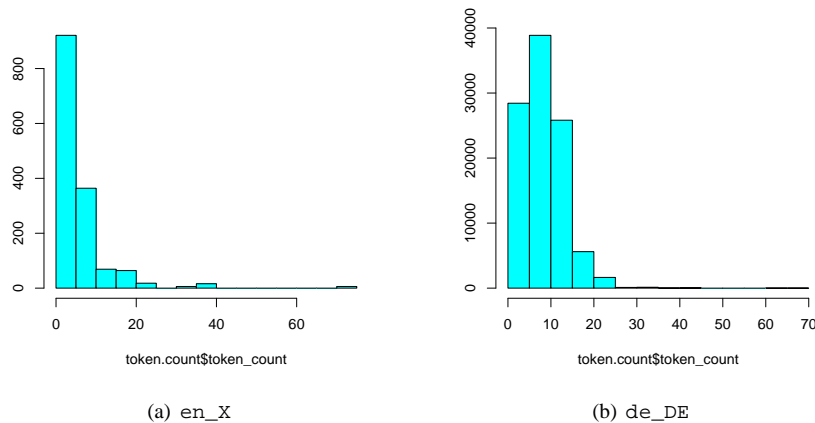


Figure 6.2: Deduction: tokens per lemma

phology coming with SFST is very elaborated and doesn't overgenerate as much as the English one, which more serves as an example to create new morphologies.

6.4.1 Clustering lemmas by their generative power

As one token generates many lemmas in the induction step, and each of these lemmas generate many token, the generated token sets do overlap. It is therefore useful to see how closely the generated lemmas for a token resemble to each other with respect to the token space they generate.

The data mining process for displaying the generative power of one specific token is somewhat complex, as it involves a statistical cluster analysis:

1. SQL: A first view extracts the lemmas and each token for that lemma row-wise.
2. SQL: A second, dynamically generated view uses the crosstab function to generate a matrix with the lemmas as rows and the generated token as columns. The columns constitute an asymmetric binary variable: Each element of the matrix states whether the lemma generates that particular token (1) or not (0). See the table on pg. 52 for an example.

Because the token which generated all the lemmas appears in all lemma rows, that column would render the variable unusable: In that case, R prints a warning that "at least one binary variable has not 2 different levels" and the cluster analysis cannot not be performed. Therefore an additional dummy row containing only "-" is added which is guaranteed to be different from all other rows.

3. Python: The `result.py` script discards any unwanted `psql` output before the real data.
4. R: The binary crosstab is loaded as a data frame into R.
5. R: A dissimilarity matrix is calculated using the `daisy` algorithm with asymmetric binary variables.

6. R: The clustering is performed with `agnes` for “agglomerative nesting”, a hierarchical clustering algorithm described in (Kaufman and Rousseeuw, 1990)
7. R: Finally, a dendrogram for the cluster analysis is plotted.

Choosing a clustering method

There are several clustering methods one can choose among. As the most important fact to visualise are the groups of identical observations with a dissimilarity of 0.0, we can't know in advance how many clusters there should be. Thus partitioning methods which construct k clusters are rather inappropriate. This lead to the hierarchical bottom-up algorithm `agnes`.

Within `agnes`, there again are several clustering methods to choose from which affect the intermediate groups with a dissimilarity above 0.0. The main difference between these methods is how the dissimilarity between groups is computed: With `average`, “the dissimilarity $d(R, Q)$ between the clusters R and Q is defined as the average of all dissimilarities $d(i, j)$ where i is any object of R and j is any object of Q ” (Kaufman and Rousseeuw, 1990, pg. 203). With `single` (single linkage), the smallest dissimilarity is taken and with `complete` (complete linkage), the largest one (Kaufman and Rousseeuw, 1990, pg. 226). The `weighted` (weighted average linkage) should be the most appropriate because it accounts for differently sized groups, which is obviously (based on the 0.0 level) a feature of our data: “Imagine a large cluster A being merged with a small cluster B . In this case the *objects* of B will carry much a larger weight than those of A in the dissimilarity”(Kaufman and Rousseeuw, 1990, pg. 235)

Linguistically, there seems to be little evidence for the accuracy of any algorithm to plausibly reconstruct word classes. At the very least, obvious contradictions between the clusters and linguistic word classes could indicate problems with the morphology.

6.4.2 English (en_X)

This is the binary data frame for the verb token “consider”. Even for the morphologically simple English language, the table gets too big to fit on one page (also due to overgeneration):

The dendrogram generated for “consider” evidently generates pairs of lemmas which differ only by the case of the first letter of the analysis string. The dissimilarity in these groups is always 0, therefore they generate exactly the same token.

The second grouping level (around 0.5) reflects the English $y \leftrightarrow i$ infix inflection which seems not to be linguistically restricted enough in the `en_X` example morphology. However, the (linguistically agnostic) statistical clustering method is sometimes unable to distinct between the dissimilarity in the productive power of that inflection and different word classes (e.g. `<Stem>consider<N><base><native><NounReg>` and `<Stem>consider<V><base><native><VerbReg>`).

The higher levels of the dendrogram mainly reflect the different word classes, e.g. the `AdjReg` clusters get split off around 0.8. The top level “-” is the artificial dummy row which made the cluster plot computable.

6.4.3 German (de_DE)

The example word “Schritt” was more or less randomly chosen, one criterion was the fact that it got a saturation level of 1 for some inflectional lemmas with the example

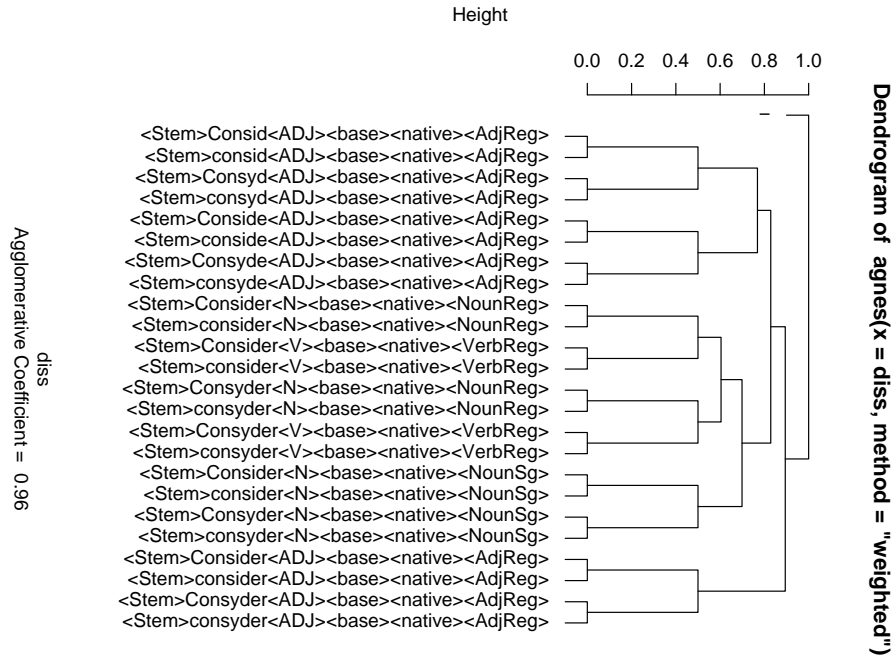


Figure 6.4: Lemmas for the token “consider” (en_X)

and adjectival forms. (...) This initial assumption was proved to be right later on by the comparison between our lexicon and Colfis, a frequency lexicon of the Italian language”.

For achieving more realistic saturation levels, it would help to get an inflected word list from a dictionary, e.g. from `aspell` with `aspell dump master de_DE > aspell.de_DE.txt`. Of course it would take many years to process it fully, but for the saturation level it suffices that it is referenced in the `texttoken` table.

6.5.1 Inherent abduction limitations

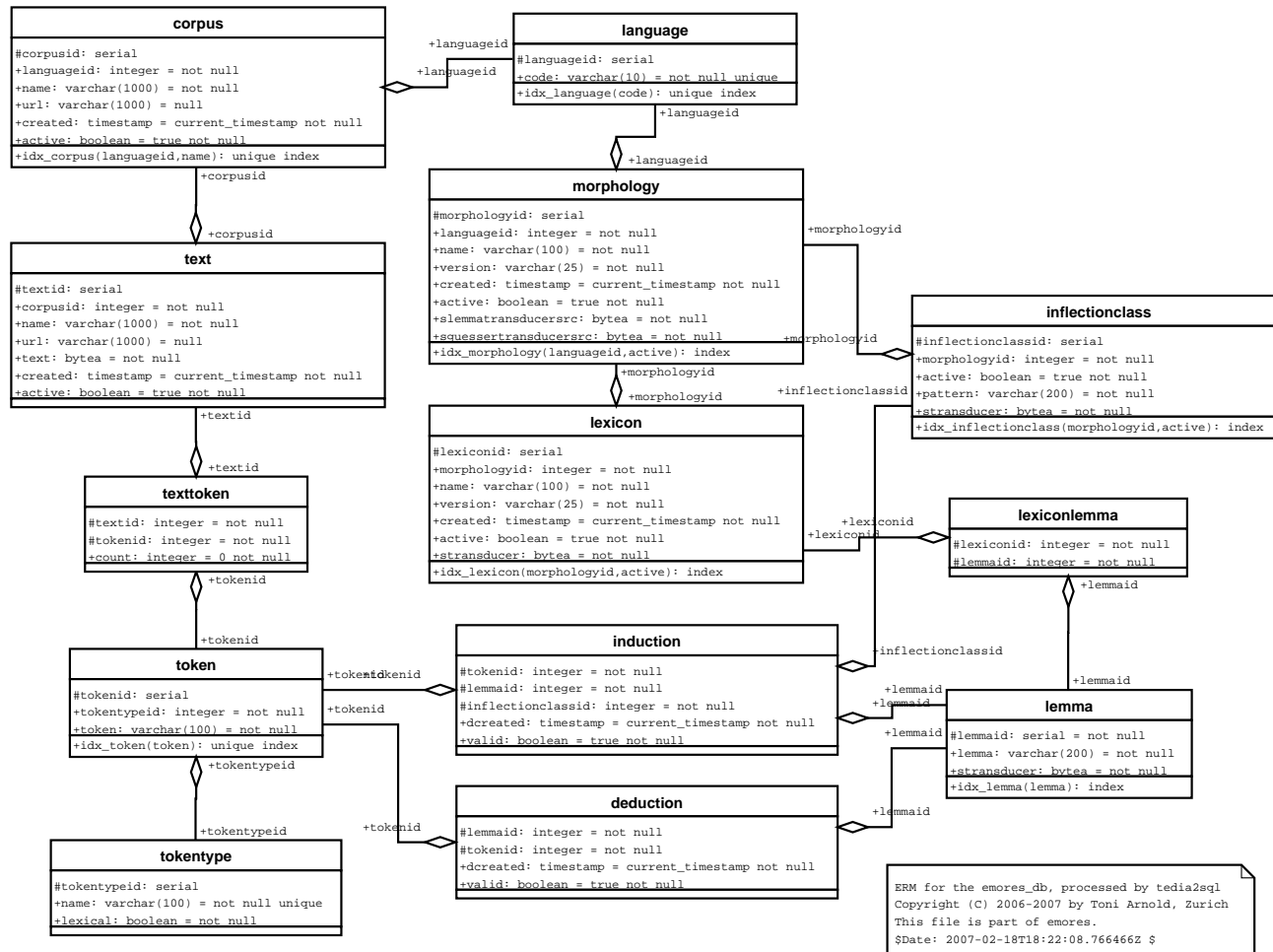
The abductive lemma ranking is inherently limited by the productive difference between lemma deductions. The results for the “Schritt” token agree with the observations made with the cluster dendrogram. The remaining ambiguity can be classified in terms of “supplementary information encoded in the lexicon” and “linguistically inherent”. The distinction between `<nativ>` and `<fremnd>` can be considered as supplementary, while for nouns the gender ambiguity (“NMasc” or “NNeut” within the declination type “s/sse”) is linguistically inherent in German, as can be seen on the “declination tables” on the documentation of my localisation of the `inform` library (Nelson, Arnold): In German, most declination types are shared in masculinum and neuter and thus principally cannot be distinguished by a formal reasoning engine like `emores`. In contrast, the femininum has its own declination types and therefore should be identifiable by `emores`.

Appendix A

Big diagrams

All big diagrams that tend to disturb the text flow have been moved to the appendix. Titles have been omitted to save space (and because e.g. the database ERM is rotated anyway).

Figure A.1: emores database model



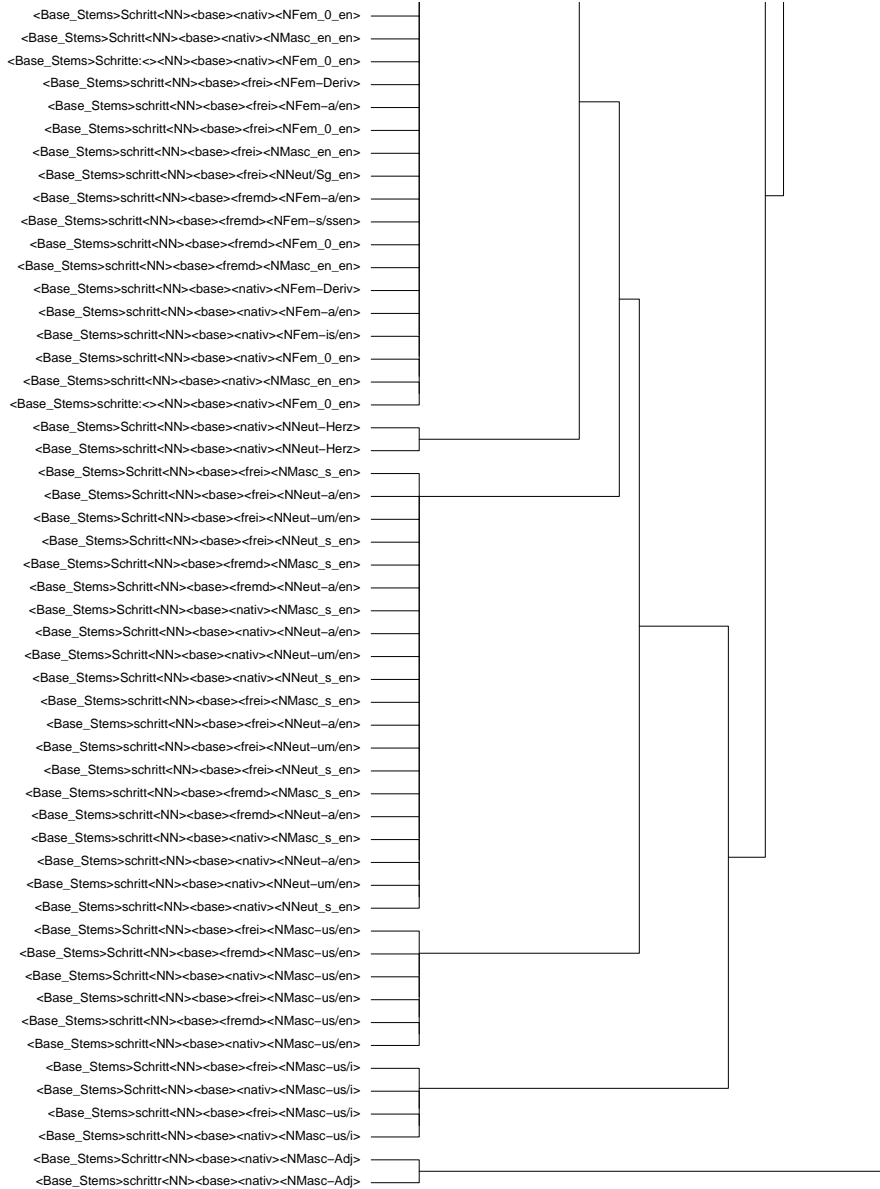


Figure A.6: (e) Lemmas for the token “Schritt”

Glossary

abduction Inference to the best explanation. In emores, the set of accepted facts are the token found in corpora. The hypotheses are the lemmas generated by induction. The abductive reasoning tries to minimise the number of lemmas to generate all empirically found token., 1, 18, 52

analysis The string that SFST returns as the result of the analysis of an inflected word form., 5, 6, 9, 28

analysis annotation The annotations at the end of an analysis, surrounded by less-than and greater-than signs each., 5, 6, 9

deduction Conclude possible observation data from a rule. In emores, deduction is bound to the morphology, and therefore overgeneration for a particular rule (lemma) is a problem of the morphology implementation., 1, 18, 49

induction Conclude a rule from a real observation. In emores, induction is performed unrestricted and therefore each token generates many (mostly wrong) inflection rules (lemmas), as most inflectional class patterns match a specific token., 1, 18, 49

inflectional class An unique combination of lexical annotations and additional character mapping rules, see the section “Inflectional classes” on pg. 5., 5, 6, 9, 28, 34

lemma A single entry in a SFST lexicon file., 5, 6, 9

lexeme The lexeme stored a SFST lexicon file: the string (including character mappings) between the lexical annotations., 5, 6, 9, 28

lexical annotation All annotations surrounded by less-than and greater-than signs each in a SFST lexicon file., 5, 6

morphology A concrete implementation for a particular language, either as a set of SFST source files or as a compiled transducer., 4

ORM Object-Relational mapping. Converts rows of SQL data tables into Python object instances. Data columns become class attributes., 11, 13, 28, 36, 44

seed lexicon Initial lexicon used to develop and test an SFST morphology for a particular language., 5

stem The constant base word form that SFST returns as the result of analyses of arbitrary inflected word forms. Stripping the analysis annotation from the analysis gives the stem., 5, 6

token Any, not necessarily linguistically correct surface realisation of a possibly incorrect lemma. See also “word form”, 49

word form A linguistically correct surface realisation of a word. See also “token”., 5, 6

List of Figures

1.1	Emores component diagram	12
2.1	Work on the SFST morphology	16
2.2	Extend the lexicon with emores	17
2.3	A fully instantiated morphology	17
6.1	Induction: lemmas per token	49
6.2	Deduction: tokens per lemma	50
6.3	Deduction binary data frame for “consider”	52
6.4	Lemmas for the token “consider” (en_X)	53
A.1	emores database model	55
A.2	(a) Lemmas for the token “Schritt”	56
A.3	(b) Lemmas for the token “Schritt”	57
A.4	(c) Lemmas for the token “Schritt”	58
A.5	(d) Lemmas for the token “Schritt”	59
A.6	(e) Lemmas for the token “Schritt”	60

Bibliography

- Toni Arnold. A german translation of inform, 1998. URL <http://www.copyriot.com/tarnold/edinf.html#Deklinationstabellen>.
- aspell. Gnu aspell is a free and open source spell checker. URL <http://aspell.net/>.
- Josiah Carlson. [python-ideas] regular expression algorithms. URL <http://mail.python.org/pipermail/python-ideas/2007-April/000407.html>.
- Russ Cox. Regular expression matching can be simple and fast. URL <http://swtch.com/~rsc/regexp/regexp1.html>.
- Jan Daciuk. Finite state utilities. URL <http://www.eti.pg.gda.pl/katedry/kiw/pracownicy/Jan.Daciuk/personal/fsa.html>.
- ddd. Gnu data display debugger. URL <http://www.gnu.org/software/ddd>.
- Dia. A diagram creation tool that supports uml modeling. URL <http://www.gnome.org/projects/dia/>.
- Docutils. Set of python tools for processing plaintext docs into html, xml, etc... URL <http://docutils.sourceforge.net/>.
- Duden. *Grammatik der deutschen Gegenwartssprache*. Dudenverlag, 1995.
- Gentoo. A special flavor of linux that can be automatically optimized and customized for just about any application or need. URL <http://www.gentoo.org>.
- glossary. Latex package for assisting generating a glossary. URL <http://theoval.cmp.uea.ac.uk/~nlct/>.
- GNU. Gnu general public license. URL <http://www.fsf.org/licensing/licenses/gpl.html>.
- Ralf Hermann. German inform library, 2000. URL <http://www.niflheim.de/inform/ginform.html>.
- Hunmorph. Hunmorph: open source word analysis. URL www.metacarta.com/docs/Kornai_acl05software.pdf.
- Kafka. Die verwandlung. URL <http://www.gutenberg.org/files/22367/22367-8.zip>.
- L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data*. 1990.

Kile. An integrated latex environment. URL <http://kile.sourceforge.net>.

LaTeX. A document preparation system. URL <http://www.latex-project.org>.

libxslt. Xslt libraries and tools. URL <http://www.xmlsoft.org/>.

LINQ. Language integrated query. URL <http://msdn.microsoft.com/netframework/future/linq/>.

Morph-it! A free corpus-based morphological resource for the italian language. URL <http://dev.sslmit.unibo.it/linguistics/morph-it.php>.

Graham Nelson. Inform version 6. URL <http://www.inform-fiction.org/inform6.html>.

Ted Neward. Object/relational mapping is the vietnam of computer science, 2004. URL <http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx>.

Alex Papadimoulis. De-compilin' back in '71. URL http://worsesthanfailure.com/Articles/De-Compilin_0x27__Back_In__0x27_71.aspx.

postgresql. Postgresql is a sophisticated object-relational dbms. URL <http://www.postgresql.org>.

psycopg2. Postgresql database adapter for python. URL <http://www.initd.org/projects/psycopg2>.

pylint. Pylint analyzes python source code looking for bugs and signs of poor quality. URL <http://www.logilab.org/projects/pylint>.

pysfst. Python bindings for the sfst library. URL <http://home.gna.org/pysfst/>.

Python. The python programming language. URL <http://www.python.org>.

R. R is gnu s - a language and environment for statistical computing and graphics. URL <http://www.r-project.org/>.

Mammoth PostgreSQL Replicator. Mammoth postgresql replicator is an asynchronous replication system designed to be wan and faulty connection tolerant. URL <http://www.commandprompt.com/products/mammothreplicator/>.

Helmut Schmid, Arne Fitschen, and Ulrich Heid. Smor: A german computational morphology covering derivation, composition, and inflection. 2004. URL www.ims.uni-stuttgart.de/projekte/gramotron/PAPERS/LREC04/smor.ps.

SFST. Stuttgart finite state transducer tools. URL <http://www.ims.uni-stuttgart.de/projekte/gramotron/SOFTWARE/SFST.html>.

Joel Spolsky. The law of leaky abstractions, 2002. URL <http://www.joelonsoftware.com/articles/LeakyAbstractions.html>.

SQLAlchemy. The python sql toolkit and object relational mapper. URL <http://www.sqlalchemy.org>.

tedia2sql. Convert database erd designed in dia into sql ddl scripts. URL <http://tedia2sql.tigris.org>.

Viktor Trón, László Németh, Péter Halácsy, András Kornai, György Gyepesi, and Dániel Varga. Hunmorph: open source word analysis. 2005. paper presented at the ACL05 Software Workshop.

winpdb. Graphical python debugger. URL
<http://www.digitalpeers.com/pythondebugger/>.

Eros Zanchetta and Marco Baroni. Morph-it! a free corpus-based morphological resource for the italian language. *Corpus Linguistics 2005*, 1(1), 2005. ISSN 1747-9398. URL <http://home.sslmit.unibo.it/~eros/downloads/Morph-it.pdf>.

Zope3. Version 3 of the zope application server. URL
<http://www.zope.org/Products/Zope3>.