þÿ H F S T a S y s t e m f o r C r e a t i n g N L P T o o l s

Linden, Krister

Springer-Verlag
2013-09

Linden , K , Axelson , E , Drobac , S , Hardwick , S , Kuokkala , J , Niemi , J , Pirinen , T & þÿ S i l f v e r b e r g , M 2 0 1 3 , H F S T a S y s t e m f o r C r e a t i n g N L P T o o l s . i n C M a (eds) , Systems and Frameworks for Computational Morphology : Communications in Computer and Information Science . Communications in Computer and Information Science , Springer-Verlag , pp. 53-71 . https://doi.org/10.1007/978-3-642-40486-3_4

http://hdl.handle.net/10138/42166
https://doi.org/10.1007/978-3-642-40486-3_4

HFST—a System for Creating NLP Tools

Krister Lindén, Erik Axelson, Senka Drobac, Sam Hardwick,
Juha Kuokkala, Jyrki Niemi, Tommi A Pirinen, and Miikka Silfverberg

University of Helsinki
Department of Modern Languages
Unioninkatu 40 A
FI-00014 University of Helsinki, Finland
{krister.linden, erik.axelson, senka.drobac, sam.hardwick,
juha.kuokkala, jyrki.niemi, tommi.pirinen,
miikka.silfverberg}@helsinki.fi

Abstract. The paper presents and evaluates various NLP tools that have been created using the open source library HFST–Helsinki Finite-State Technology and outlines the minimal extensions that this has required to a pure finite-state system. In particular, the paper describes an implementation and application of Pmatch presented by Karttunen at SFCM 2011.

Keywords: finite-state technology, language identification, morphological guessers, spell-checking, named-entity recognition, language generation, parsing, HFST, XFST, Pmatch

Introduction

In natural language processing, finite-state string transducer methods have been found useful for solving a number of practical problems ranging from language identification via morphological processing and generation to part-of-speech tagging and named-entity recognition, as long as the problems lend themselves to a formulation based on matching and transforming local context.

In this paper, we present and evaluate various tools that have been created using HFST–Helsinki Finite-State Technology[1] and outline the minimal extensions that this has required to a pure FST system. In particular, we describe an implementation of Pmatch presented by Karttunen at SFCM 2011 [7] and its application to a large-scale named-entity recognizer for Swedish.

The paper is structured as follows: Section 1 is on applications and their evaluation. In Section 2, we present examples of user environments supported by HFST. In Section 3, we present some of the solutions and extensions needed to implement the applications. This is followed by the Sections 4 and 5 with an outline of some future work and the discussion, respectively.

[1] http://hfst.sf.net

1 Applications and Tests

In this section, we describe and evaluate some applications implemented with HFST. When processing text corpora, it is useful to first identify the language of the text before analyzing its words morphologically. Words unknown to the morphological lexicon need a guesser derived from the lexicon. The reverse operation of morphological analysis is morphological generation. Generating inflections of words unknown to the morphological lexicon can be used for eliciting information from native speakers before adding the words to the lexicon. For information extraction, it is important to be able to identify multi-word expressions such as named entities, for which purpose HFST has a pattern matching tool. Finally, we also describe a traditional application area of finite-state morphology, i.e. spell-checking and spelling correction, which is now served by a uniform implementation using weighted finite-state technology.

1.1 Language identification

Language identification is the task of recognizing the language of a text or text fragment. It is useful in applications that need to process documents written in various languages where the language might not be explicitly marked in the document. For example, a translation application might need to identify the language of a document in order to apply the correct translation model. Another example is a speller for Finnish, which might need to identify paragraphs written in English, in order not to spell-check those paragraphs.

In this section we outline how to use HFST tagger tools and language identification tools for creating language identifiers. We also present an experiment on language identification for documents written in Dutch, English, Estonian, Finnish, German or Swedish. The experiment shows that HFST language identifiers are highly accurate (99.5% of the input sentences were correctly classified).

There are several methods for performing language identification. Highly accurate language identification can be accomplished by treating data as a letter sequence and training a Markov chain from training documents whose language is known [3]. One Markov chain is trained for each language that the system recognizes. Language identification consists of applying each Markov chain on input and choosing the language whose model gives the highest likelihood for the text.

HFST language identifiers adopt a Markov chain framework, which can be implemented with weighted finite-state technology. Using HFST tagger tools [19], we train Markov models for all languages. A program, `hfst-guess-language`, reads the models and input text and labels each sentence with the language whose model gave the highest likelihood for the sentence.

We present an experiment on applying HFST language identifiers for guessing the language of sentences written in six languages. For all languages except Swedish, we used training data from corpora containing newspaper text. For Swedish, we used more general text.

For Dutch we used the Alpino treebank [1], for English we used the Penn Treebank [12], for Estonian we used the Estonian National Corpus[2], for Finnish we used text from the largest Finnish newspaper Helsingin Sanomat year 1995[3], for German we used the TIGER Corpus [2] and for Swedish we used Talbanken [5].

Table 1: For each language, we used 2000 sentences for training and 200 sentences for testing. We give the sizes of the data sets in utf-8 characters.

Language	Train data (utf-8 chars)	Test data (utf-8 chars)
Dutch	245,000	24,000
English	265,000	26,000
Estonian	238,000	23,000
Finnish	155,000	14,000
German	280,000	28,000
Swedish	164,000	16,000

For each language, we chose 2200 sentences for training and testing. Of the sentences, every eleventh sentence was used for testing and the rest for training. This totals 2000 sentences for training and 200 sentences for testing for each language. The sizes of the data sets in utf-8 characters are described in Table 1. The average length of a sentence in characters was shorter for Finnish and Swedish than for the other languages.

Table 2: We give accuracy of the language guesser per language and for all languages.

Language	Accuracy
Dutch	99.0%
English	99.5%
Estonian	99.5%
Finnish	99.5%
German	100.0%
Swedish	99.5%
ALL	99.5%

We ran the language identifier for test sentences from all six languages (1200 sentences in total) and computed the accuracy of the language identification system as $corr/all$, where $corr$ is the number of sentences whose language was correctly guessed and all is the number of all sentences. In Table 2, we show results for each individual language and all the languages combined.

[2] `http://www.cl.ut.ee/korpused/segakorpus/`

[3] `http://www.csc.fi/kielipankki/`

Of all sentences, 99.5% were correctly classified, which demonstrates that the language identification system is accurate. This is encouraging because Finnish and Estonian have similar orthographies. This applies to German, Swedish and Dutch as well.

Currently identification is limited to identifying the closest language corresponding to a sentence. There is no option to label a sentence as belonging to an unknown language.

1.2 Morphologies and Guessers

Language technology applications for agglutinating languages such as Finnish and Hungarian benefit greatly from high-coverage morphological analyzers, which supply word forms with their morphological analyses. This makes applications dependent on the coverage of the morphological analyzer. Building a high-coverage morphological analyzer (with recall over 95%) is a substantial task, and even with a high-coverage analyzer, domain-specific vocabulary presents a challenge. Therefore, accurate methods for dealing with out-of-vocabulary words are needed.

With HFST tools it is possible to use an existing morphological analyzer to construct a morphological guesser based on word suffixes. Suffix-based guessing is sufficient for many agglutinating languages such as Finnish [10], where most inflection and derivation is marked using suffixes. Even if a word is not recognized by the morphological analyzer, the analyzer is likely to recognize some words which inflect similarly to the unknown word. These can be used for guessing the inflection of the unknown word.

The guessing of an unknown word like “twiitin” (the genitive form of “twiitti”, tweet in Finnish) is based on finding recognized word forms like “sviitin” (genitive form of “sviitti”, hotel suite in Finnish) that have long suffixes, such as “-iitin”, which match the suffixes of the unrecognized word. The longer the common suffix, the more likely it is that the unrecognized word has the same inflection as the known word. The guesser will output morphological analyses for “twiitin” in order of likelihood.

Besides the length of the matching suffix, guesses can also be ordered based on the probability that a suffix matches a given analysis. This can be estimated using a labeled training corpus. In addition, any existing weighting scheme in the original morphological analyzer can be utilized.

If the morphological analyzer marks declension class, the guesser can also be used for guessing the declension class. If the declension class is marked, the guesser can be used for the generation of word forms as well as analysis. This is described in Section 1.3.

Constructing a morphological guesser from OMorFi[4]–The open-source Finnish morphology [15], the three top guesses for “twiitin” are (the markup is slightly simplified):

[4] `http://code.google.com/p/omorfi/`

```
twiit  [POS=NOUN] [GUESS_CATEGORY=5]  [NUM=SG][CASE=GEN]
twiiti [POS=NOUN] [GUESS_CATEGORY=33] [NUM=SG][CASE=NOM]
twiit  [POS=VERB] [GUESS_CATEGORY=53] [VOICE=ACT][MOOD=INDV] ...
```

The first field corresponds to the stem of the word, the second field to its main part of speech and the third to its declension class. The fourth field shows the inflectional and derivational information of the guess. In this case, the first guess is correct. It is modeled after declension class number 5, which is a class of nouns containing among others the noun "sviitti".

1.3 Language Generation for Out-of-Vocabulary Words

Natural-language user interfaces, such as dialogue systems, need a language generation component for generating messages for the user. The aim is to supply the user with information about the internal state of some database containing information such as airline connections or weather phenomena.

Language generation systems for agglutinating languages will benefit from morphological analyzers, because generating syntactically correct sentences requires inflecting words according to syntactic context. Depending on the domain and coverage of the morphological analyzer, it might also be necessary to inflect words that are not recognized by the morphological analyzer.

HFST morphological guessers presented in Section 1.2 can be used for generation as well as morphological analysis. For example, using the OMorFi morphology for Finnish, the best morphological guess for the unknown word "twiitin" is

```
twiit  [POS=NOUN] [GUESS_CATEGORY=5]  [NUM=SG][CASE=GEN]
```

Replacing the inflectional information `[NUM=SG][CASE=GEN]` (singular genitive case) by `[NUM=PL][CASE=PAR]` (plural partitive case) gives the analysis

```
twiit  [POS=NOUN] [GUESS_CATEGORY=5]  [NUM=PL][CASE=PAR]
```

which can be fed back to the guesser to generate the surface forms "twiitteja" and "twiittejä". The latter one is correct, though the first one would also be possible in theory, since the variation between "-ja" and "-jä" is governed by Finnish vowel harmony and the stem "twiit" is neutral with respect to vowel harmony.

1.4 Extending a lexicon with the help of a guesser

The morphological guesser has proven to be a useful tool when adding large bulks of new vocabulary to a lexicon. We tested this on the Finnish Open Source lexicon OMorFi. According to our experience with handling ca. 260.000 proper nouns, the guesser achieved roughly 90% accuracy in assigning the correct inflection class to new lexicon entries on the first guess, so the manual work needed was reduced to only checking the guesser's results and correcting ca. 10% of the suggested entries.

The names to be added to the lexicon were given in their base form, so we could benefit from accepting only suggestions by the guesser in the nominative case `[CASE=NOM]`. The data was presented to native speakers with key word forms generated for each entry, which could be used to distinguish between different inflection classes, so that it was not necessary to understand the linguistic encoding scheme:

```
Aura     9 Aura : Auraa : Aurat : Aurain, Aurojen : Auroja : Auroihin
Oura    10 Oura : Ouraa : Ourat : Ourain, Ourien : Ouria : Ouriin
Pura    10 Pura : Puraa : Purat : Purain, Purien : Puria : Puriin
Saura    9 Saura : Sauraa : Saurat : Saurain, Saurojen : Sauroja : Sauroihin
Peura    9 Peura : Peuraa : Peurat : Peurain, Peurojen : Peuroja : Peuroihin
Tiura   10 Tiura : Tiuraa : Tiurat : Tiurain, Tiurien : Tiuria : Tiuriin
Heikura 13 Heikura : Heikuraa : Heikurat : Heikurojen, Heikuroiden,
           Heikuroitten : Heikuroja, Heikuroita : Heikuroihin
```

We did a preliminary test to assess the accuracy of the guesser when only some basic proper nouns were included in OMorFi's lexicon. A sample of 100 words from each proper noun list to be added (place names, companies, organizations, given names, family names) showed that the guesser's success rate for finding the correct inflection class within first five guesses ranged from 68% (companies) to 93% (place names). The differences between the groups are readily explained by the facts that the place names most often have endings corresponding to regular nouns, whereas the organization and company names often contain foreign and acronym components not recognized by the guesser.

When doing bulk additions of large numbers of lexical entries based on their suffixes, it is practical to sort the entries alphabetically according to the end of the word. As the first guess was very often correct, only one guess was provided. If the first guess is marked as incorrect by a native speaker, several words needing the same correction are likely to follow, so it is quick to apply the same correction.

After two lists of person names (ca. 12,000 family names and ca. 4,000 given names) had been manually corrected, they were included in OMorFi's lexicon in order to improve the guesser's performance when handling further proper noun data.

The guesser indeed performed well with the Finnish geographical names (ca. 230,000): 91% of the first inflection class codes generated were correct without any editing. A smaller collection of foreign geographical names – states, provinces and cities (ca. 12,000) – also yielded quite good results, considering the tiny amount of foreign lexical data previously known by OMorFi: 73% of the guesses were correct as such, and 6% with some added information.

The last batch of our proper names consisted of ca. 6,600 organization names. These included mostly Finnish but to some extent also international companies, societies and other organizations. Before handling the organizations, the geographical names were incorporated into OMorFi and the guesser was rebuilt. With this guesser, we got 86% of the organization names correctly assigned and 3% correctly with some additions. This was a significant improvement over the initial guesser.

1.5 Named-Entity Recognition

Named entities are among the most important elements of interest in information retrieval. In addition, names indicate agents and objects which are important in information extraction. Often named entities are denoted by multi-word expressions. In HFST, a pattern-matching tool, `hfst-pmatch`, has been implemented for identifying multi-word expressions and recognizing named entities.

Background. In his keynote speech at SFCM 2011, Karttunen presented toy examples of named-entity recognition (NER) with his FST pattern matching tool (`pmatch`) [7]. The HFST Pmatch tool has been modelled after Karttunen's, but it is an independent implementation with some differences in features. We have converted a full-scale named-entity recognizer for Swedish to use HFST Pmatch, and we are in the process of developing one for Finnish.

A named-entity recognizer marks names in a text, typically with information on the type (class) of the name [14]. Major types of names include persons, locations, organizations, events and works of art. NER tools often also recognize temporal and numeric expressions. Names and their types can be recognized based on internal evidence, i.e. the structure of the name itself (e.g., *ACME Inc.* probably denotes a company), or based on external evidence, i.e. the context of the name (e.g., *she works for ACME*; *ACME hired a new CEO*) [13]. In addition, NER tools typically use gazetteers, lists of known names, to ensure that high-frequency names are recognized with the correct type.

Named-Entity Recognition with Pmatch. A key feature of Pmatch that makes it well-suited for NER is the ability to efficiently add XML-style tags around substrings matching a regular expression, as in [7]. Such regular expressions are specified by suffixing the expression with `EndTag(`*`TagName`*`)`. For example, the following expressions mark company names ending in a company designator:

```
Define NSTag [? - [Whitespace|"<"|">"]] ;
Define CorpSuffix [UppercaseAlpha NSTag+ " "]+ ["Corp" | "Inc"]
    EndTag(EnamexOrgCrp) ;
Define TOP CorpSuffix ;
```

The built-in set `Whitespace` denotes any whitespace character and `UppercaseAlpha` any uppercase letter. String literals are enclosed in double quotation marks where Karttunen's FST uses curly braces [7]. For matching, Pmatch considers the regular expression with the special name `TOP`. Thus, to be able to tag the company names with the expression `CorpSuffix`, `TOP` must refer to it. In general, a Pmatch expression set (file) contains a list of named regular expression definitions of the form `Define` *`name regex`* `;`.

The above expressions mark the company names in the following input:

```
Computer Systems Corp announced a merger with Home Computers Inc .
```

The output is:

```
<EnamexOrgCrp>Computer Systems Corp</EnamexOrgCrp> announced a
merger with <EnamexOrgCrp>Home Computers Inc</EnamexOrgCrp> .
```

Pmatch considers leftmost longest matches of `TOP` in the input and adds the tags specified in `TOP` or the expressions to which `TOP` refers. If several subexpressions have the same leftmost longest match in the input, it is unspecified (but deterministic) which one Pmatch chooses. To disambiguate between matches, context conditions can be added to the matching regular expressions. If a part of the input does not match `TOP` or only matches a subexpression without an `EndTag` or any transductions, Pmatch outputs it unaltered.

HFST Pmatch regular expressions may also contain transductions that can add extra output or discard specified parts of the input. Even though they are not in general used in tagging named entities, they can be used in correction expressions that modify tags added by previous sets of expressions. (Pmatch makes a single pass over its input, so a transduction cannot modify tags added by the same set of expressions.) If several different expressions have the same leftmost longest match but different transductions, Pmatch deterministically chooses one of them and issues a warning that there were other possible matches.

Context Conditions. An expression may be accompanied with a context condition specifying that a match should be considered only if the left or right context of the match matches the context condition. For example, the following expressions mark the capitalized words following *rörelseresultatet för* ('operating profit of') with `EnamexOrgCrp`:

```
Define CapWord2 UppercaseAlpha NSTag+ ;
Define OrgCrpOpProfit CapWord2 [" " CapWord2]*
    EndTag(EnamexOrgCrp) LC("Rörelseresultatet för ") ;
Define TOP OrgCrpOpProfit ;
```

For example:

```
Rörelseresultatet för <EnamexOrgCrp>Comp Systems</EnamexOrgCrp> ...
```

As in [7], the regular expression in `LC()` specifies a left context that must precede the actual match. Similarly, `RC()` specifies a right context that must follow the match. `NLC()` and `NRC()` specify negative left and right context, respectively, that may not precede or follow the match. Context conditions may be combined with conjunction and disjunction.

Conjunctive context conditions can also be specified at several stages in the expressions. For example, a name is marked as a sports event by the following expressions only if it is followed by a space and the word *spelades* ('was played') (right context expression from `EvnAtlIntl`) and preceded by a space or sentence boundary (`#`) (left context expression from `TOP`):

```
Define EvnAtlIntl [CapWord2 " "]+ "International "
    EndTag(EnamexEvnAtl) RC(" spelades") ;
Define TOP EvnAtlIntl LC(Whitespace | #) ;
```

In this case, the left context condition in `TOP` is considered for all the `EndTag` expressions contained or included in `TOP`. Karttunen [7] does not mention if his system can combine multiple context conditions in a similar way.

Converting a Swedish Named-Entity Recognizer to Use Pmatch. We have converted a Swedish named-entity recognizer [8] developed at the University of Gothenburg to use Pmatch. The Swedish NER tool works on tokenized running text input: punctuation marks are separated from words by spaces but the words are not annotated in any way. In contrast, the forthcoming Finnish NER tool will work on annotated text, which makes it easier to write more general rules, in particular for a morphologically rich language such as Finnish.

The original implementation of the Swedish NER tool [8] contained 24 different recognizers running in a pipeline and a correction filter run after each stage. 21 of the recognizers and the correction filter had been written using Flex[5] rules; the remaining three were Perl scripts recognizing names in gazetteers. The Flex rules recognize regular expression matches in the input, corresponding to names and their possible context, and the actions of the rules mark the name parts of the matches with XML tags in the output. The correction filter modifies, removes and adds new tags based on existing ones.

Motivations for reimplementing the Swedish recognizer in Pmatch included the slow compilation of some of the Flex rule sets, which hindered testing changes to the rules, and a desire to be able to use a single tool or formalism for all the components of the recognizer.

Since both Flex and Pmatch are based on regular expressions and recognizing the leftmost longest match, we were able to automate a large part of the conversion from Flex rules to Pmatch rules. The conversion script analysed the Flex actions to split the recognized match into a name and its context. The correction filter was converted by hand, since its rules were more varied than those in the recognizers.

However, because of differences between the semantics of Flex NER rules and Pmatch, some Pmatch expressions generated by the automatic conversion had to be edited by hand to work correctly. Firstly, the Flex rules were written so that the matched regular expressions covered the contexts in addition to the name to be recognized, whereas Pmatch excludes contexts from its leftmost longest match. Consequently, the leftmost longest match at a certain point in text may be found by different patterns in Flex and Pmatch.

Secondly, Flex rules are ordered whereas Pmatch expressions are not. Flex patterns can thus be ordered from the most specific to the most general, so the most specific pattern is chosen even if also a more general one would have the same leftmost longest match. In contrast, Pmatch cannot guarantee any

[5] `http://flex.sourceforge.net/`

specific order, so the ordering has to be replaced with more detailed context conditions or with regular language subtraction or both. For example, to prevent capitalized *järnväg* ('railway') from matching a more general expression marking street names, it is subtracted from the more general pattern:

```
Define LocStrSwe
    [Capword2 "väg" ("en")] - "Järnväg" EndTag(EnamexLocStr) ;
```

With some modifications to account for the lack of ordering, the Pmatch rules were able to recognize and classify the same names as the original Flex rules. However, many rules would be more natural if written from scratch to utilize the features of Pmatch, such as more powerful context conditions. A "native" Pmatch implementation could probably have been written without a correction filter.

The Pmatch implementation of the gazetteer lookup uses the construct `@txt"`*`filename`*`"` that treats the named file as a disjunction of strings, each line as one disjunct. The gazetteer has been divided into files by the type of the name:

```
Define LocStr @txt"LocStr.txt" EndTag(EnamexLocStr) ;
Define PrsHum @txt"PrsHum.txt" EndTag(EnamexPrsHum) ;
...
Define Boundary [" " | #] ;
Define TOP [LocStr | PrsHum | ...] LC(Boundary) RC(Boundary) ;
```

The context conditions in `TOP` allow a name to be recognized only at word boundaries. The name lists could be replaced with full-fledged morphological analyzers allowing the recognition of inflected words or names.

The original Swedish NER system marks named entities with XML elements encoding the precise type in attributes. The tags used by Pmatch can be converted to this format with Pmatch transductions or with a simple script. For example, the Pmatch-tagged text

```
<EnamexOrgCrp>Computer Systems Corp</EnamexOrgCrp>
```

is converted to

```
<ENAMEX TYPE="ORG" SBT="CRP">Computer Systems Corp</ENAMEX>
```

Performance. Compiling the Pmatch version of the Swedish NER was about ten times faster on the average than the Flex version, which we consider as a significant improvement. On our test machine[6], the average compilation time of a single recognizer was reduced from 53 minutes to 5.5 minutes, and the slowest one from 288 minutes (almost five hours) to 54 minutes. We will also investigate further ways to improve compilation speed. In contrast, at run time the Pmatch

[6] The test machine had Intel Xeon X7560 processors running at 2.27 GHz.

NER recognizers were about three times slower on the average than the Flex ones.

The total size of the current Pmatch FSTs for the Swedish NER is over three gigabytes, which is about eight times as large as the executables compiled from the Flex files. However, the FST sizes will be reduced as soon as expression caching is implemented in Pmatch. Using a recursive transition network feature similar to Karttunen's [7] `Ins()` will further reduce the FSTs and their compile times.

1.6 Spell-checking

Using weighted finite-state methods for spell-checking and correction is a relatively recent branch of study in spell-checking research. The concept is simple: finite-state morphological analyzers can easily be transformed into spell-checking dictionaries providing a language model for the correctly spelled words in the spell-checking system. A baseline finite-state model for correcting spelling errors can be inferred from the language model by creating a Levenshtein-Damerau automaton based on the alphabetic characters present in the language. The language model can be trained to prefer more common words when the Levenshtein-Damerau distance between two suggestions is the same. This is done with a unigram language model that maximizes the frequency of the suggested word. In our experience, even relatively moderate amounts of training material will improve the quality, as the statistical training improves the discriminative power of the model due to the observation that the likelihood of random typing errors is greater in more frequent words.

The practical process of creating a finite-state spell-checker and corrector is simple: given an analyzer capable of recognizing correctly spelled word-forms of a language, make a projection to the surface forms to create a single-tape automaton. The automaton is trained with a corpus word-form list, for which the final state weight of each word-form is, e.g., $-\log \frac{c(wf)}{CS}$, where $c(wf)$ is the word-form count and CS is the corpus size. Words not found in the corpus are given a maximal weight $w_{max} > -\log \frac{1}{CS}$ to push them to the end of the suggestion list; this weighting can be done, e.g., in finite-state algebra by composition with a weighted $\Sigma^\star$ language.

The error model can be improved from the baseline Levenshtein-Damerau distance metric as well. For this purpose we need an error corpus, i.e., a set of errors with their frequencies. This can be semi-automatically extracted from weakly annotated sources, such as Wikipedia. From Wikipedia we get, among other things, word-to-word corrections by inspecting the commit messages from Wikipedia's logs. It is possible to use the specific word-to-word corrections to create an extension of common confusables to the error model. Another way is to re-align the corrections using the Damerau-Levenshtein algorithm and train the original character distance measure with frequencies of the character corrections in the same manner as we did for word-forms above.

The application of the language and error model to spell-checking is a traversal or composition with a finite-state transducer. The checking of the correct

spelling is a composition $w \circ L$, where w is a single path automaton containing the word-form and L is a single-tape automaton recognizing the correct word-forms of a language. The spelling correction is $(w \circ E \circ L)_1$, where E is a two-tape automaton containing the error model, and $_1$ is a projection to the surface language.

As an example of the simplicity of this process, we obtained an open-source German morphological analyzer morphisto[7] to generate a spell-checker, trained it with word-forms extracted from the German Wikipedia[8] and applied it to Wikipedia data to find spelling errors and correct them. The whole script for this can be found in our version control[9], and it took us no more than one work day by one researcher to implement this application. The resulting system does spell-checking and correction with a baseline finite-state edit distance algorithm [17] applying up to 2 errors per word-form at a speed of 77,500 word-forms per second. For further evaluations on other language and error models, refer to [16].

2 Examples for User Environments

In this section, we provide some examples of how to implement applications on top of the HFST library using Python. The HFST library and its Python bindings are readily installable in all major operating systems.

2.1 An Interface in Python

In addition to an API library and command line tools, the HFST library can also be used through SWIG-generated Python bindings. The bindings are offered for the Python programming language versions 2 and 3. All HFST functionalities are available via both versions, but the Python interpreters themselves have some differences. For example, Python 2 allows HFST exceptions to be caught directly, but Python 3 requires the use of a special wrapper function written as a part of the bindings. On the other hand, Python 3 has better support for unicode characters, so it is probably a better choice for most linguistic applications.

Below is an example of iterating through the states and transitions of an HFST transducer using Python bindings:

```
# Go through all states in fsm
for state in fsm.states():
    # Go through all transitions
    for transition in fsm.transitions(state):
        # do something
```

And the same using the HFST API directly:

[7] http://code.google.com/p/morphisto/
[8] http://de.wikipedia.org
[9] svn://svn.code.sf.net/p/hfst/code/trunk/articles/sfcm-2013-article

```
// Go through all states in fsm
for (HfstBasicTransducer::const_iterator it = fsm.begin();
       it != fsm.end(); it++ )
    {
     // Go through all transitions
   for (HfstBasicTransducer::HfstTransitions::const_iterator tr_it
           = it->begin(); tr_it != it->end(); tr_it++)
        {
        // do something
        }
    }
```

The Python bindings in particular make it easy to use language models developed for HFST in rapid prototyping of advanced tools. For example, a chunker for Finnish was developed by simply bracketing adjacent agreeing cases and a few other similar expressions with a few lines of code on top of an existing morphological analyzer. E.g. given the Finnish sentence "miljoona kärpästä voi olla väärässä paikassa", we get a bracketing of all three phrases as illustrated in the following sentence with a gloss:

(1) Miljoona$_1$ kärpästä$_1$ voi$_2$ olla$_2$ väärässä$_3$ paikassa$_3$
million-NUM fly-PAR can-AUXV be-INFV wrong-INE place-INE

'A million flies can be in the wrong place'

In this case the rules governing chunking are all about pairs of words: a measurement phrase is a numeral followed by a partitive nominal, a verbal phrase is an auxiliary followed by a lexical verb and a noun phrase is an adjective and a noun in an agreeing case. The three pairs of words can be identified as common chunks in Finnish and having specific rules for these pairs will give a reasonable baseline surface syntax for applications where a more elaborate syntactic structure is not required.

A Chatroom Morphology Tool. One example of rapid development and leverage of language resources is an IRC bot performing morphological analysis and synthesis on command. Originally written as a source of entertainment for linguistics students, it is usable as a learning resource and discussion facilitator for language learners. It also proved useful as a testing environment; requested analyses that were not found in the transducers can be written to a log file.

The pertinent Python code for performing lookup on an appropriate transducer is as simple as:

```
with libhfst
    transducer = HfstTransducer(HfstInputStream("transducer.hfst"))
     results = transducer.lookup(message)
     for result in vectorize(results):
         irc_message(result)
```

For more than just providing analyses of words, either the underlying transducer or the bot can be customized to allow specific queries:

```
<user> hfstbot: kintereellä
<hfstbot> user: kinner<N><Sg><Ade>
<user> hfstbot: gen kinner<N><Pl><Nom>
<hfstbot> user: kintereet
```

In this case, the user wants to see the analysis for "kintereellä", which translates to "on the hock". Being informed that it is a singular noun in the adessive case, the base or nominative form of which is "kinner", the user asks for the plural nominative, which is "kintereet".

2.2 HFST on Unix, Mac and Windows

Portability has been one of the design goals of the HFST system. The current versions are available or compilable on all POSIX-supporting systems, including Cygwin under Windows, Mac OS X and Linux. Compilation is also possible on MinGW under Windows.

Fresh versions of HFST source code can be fetched from our Subversion repository at Sourceforge[10]. We also offer, approximately twice a month, new release packages that include a source tarball (compilable on all the aforementioned platforms), Debian binaries (for Linux), a MacPort distribution (for MacOS X) and an NSIS installer (for Windows).

2.3 Other Usability Improvements

Four new command line tools have been added since 2011. The most important are the native XFST parser `hfst-xfst` and the tagging tool `hfst-tagger`. Also two functions that were earlier available only through the API can now be used as command-line tools: `hfst-shuffle` and `hfst-prune-alphabet`. The former is a special operation that freely interleaves the symbols of any two strings recognized by the respective transducers. The latter removes symbols that do not occur in the transitions of the transducer from its alphabet. Two existing tools that perform transducer-to-text conversion also have new features: `hfst-fst2txt` can write to dot/graphviz and PCKIMMO format and `hfst-fst2strings` has a new parameter that controls its output to achieve better interoperability with other command-line tools.

There is some additional control over special symbols as we have added a parameter for binary operators controlling whether unknown and identity transitions are expanded, the default being true. We also have a new special symbol, the `default` symbol matching any symbol if no other transition in a given state matches.

We have kept the number of dependencies in HFST as low as possible. All back-ends (SFST, OpenFst and foma) are now bundled with HFST. There is no

[10] `http://hfst.sf.net`

longer a need to install them separately or worry about having the right version. We have also made modifications to the back-end libraries; for instance, some of the compile-time warnings are now fixed or suppressed. GNU- and Bash-specific commands were also removed from the scripts to make them more portable.

3 Under the Hood

In the following section, we describe some of the technical choices made to implement the HFST library and the applications addressed in the previous sections, as well as some minor design differences with regard to XFST.

3.1 An independent XFST module

The HFST command-line tools include an XFST parser tool that can be used in interactive mode or to compile script files. The tool implements the same functionalities as the original XFST (Xerox Finite-State Tool) which is a general-purpose utility for computing with finite-state networks. There are over 100 commands in `hfst-xfst`, the same as those documented in the Xerox tool. In addition, there is an option to independently use the regular expression parser which the `hfst-xfst` module was built on through the `hfst-regexp2fst` tool for those who wish to parse regular expressions in Bash scripts.

Below is an example of using `hfst-xfst` in interactive mode where we define two transducer variables, use them in a regular expression and print random words recognized by the expression.

```
$ hfst-xfst2fst
hfst[0]: define Foo foo;
hfst[0]: define Bar bar;
hfst[0]: regex [[Foo|0] baz [Bar|0]];
424 bytes. 4 states, 4 arcs, 4 paths
hfst[1]: print random-words
baz
bazbar
foobaz
foobazbar
hfst[1]:
```

To test `hfst-xfst2fst`, we have compiled 17 out of the 22 XFST exercises that are found on the homepage of Beesley and Karttunen's book Finite State Morphology[11]. We have omitted the exercises that do not include an answer. We have compiled the exercises using both Xerox's XFST and HFST's `hfst-xfst` and compared the results for equivalence. We have also tested the functionality of the `hfst-regexp2fst` tool by rewriting the original exercises using HFST command line tools (other than `hfst-xfst`).

[11] `http://www.fsmbook.com`

Although we are aiming at complete backward compatibility with XFST, we have noticed that in some borderline cases the results may differ when using replace rules in regular expressions. One example in which XFST and `hfst-xfst2fst` may give different results is the longest match.

By definition, in left-to-right longest match:

```
A @-> B || L _ R
```

where `A`, `B`, `L` and `R` denote languages, the expression `A` matches input left-to-right and replaces only the longest match at each step.

Therefore, a left-to-right longest match is supposed to give exactly one output string for each input string. However, when compiled using XFST, both of the following left-to-right longest match rules result in transducers which for input `aabbaax` give two outputs: `aaxx` and `xxx`.

```
xfst[0]: regex a+ b+ | b+ a+ @-> x \\ _ x ;
3.1 Kb. 8 states, 31 arcs, Circular.
xfst[1]: down aabbaax
aaxx
xxx
xfst[1]: regex a+ b+ | b+ a+ @-> x \/ _ x ;
3.1 Kb. 8 states, 31 arcs, Circular.
xfst[2]: down aabbaax
aaxx
xxx
```

In the examples, the `\\` sign denotes that the left context `L` is to be matched on the input side of the relation and the right context `R` is to be matched on the output side of the relation. The `\/` sign denotes that both contexts are to be matched on the output side of the relation.

The same regular expressions compiled with `hfst-xfst2fst` will for the same input give only one output `xxx`, which we consider to be the only correct result.

It is likely that, in this case, the difference is caused by different compilation approaches. In `hfst-xfst2fst`, replace rules are compiled using the preference operator [4], which in this case successfully finds that the output string `aaxx` is less preferable in comparison with the output string `xxx` and is therefore excluded from the final result.

Furthermore, we have noticed that there are some differences in pruning the alphabet after performing certain operations. These two examples will give the same transition graphs, but different alphabet sets if compiled with XFST:

```
regex [a | b | c ] & $[a | b] ;
resulting alphabet: a, b

regex [a | b | c ] & [a | b] ;
resulting alphabet: a, b, c
```

HFST's `hfst-xfst2fst` always prunes the alphabet after the following operations: **replace rules** (contexts are also pruned before being compiled with the rule), **complement**, **containments**, **intersection**, **minus**. However, it seems that XFST prunes the alphabet only if at least one of the operands contains the `unknown` symbol and if the result does not contain the `any` symbol. Therefore, if the above commands were run in the `hfst-xfst2fst` environment, the resulting alphabet is different from that of XFST, being `a`, `b` in both cases.

In HFST, the alphabet pruning only removes symbols from the alphabet if the pruning has no effect on the function of the transducer. Therefore, we have not managed to find an example in which the above difference influences the correctness of the result, but pruning the alphabet results in a slightly smaller transducer.

3.2 Pmatch with Applications for NER

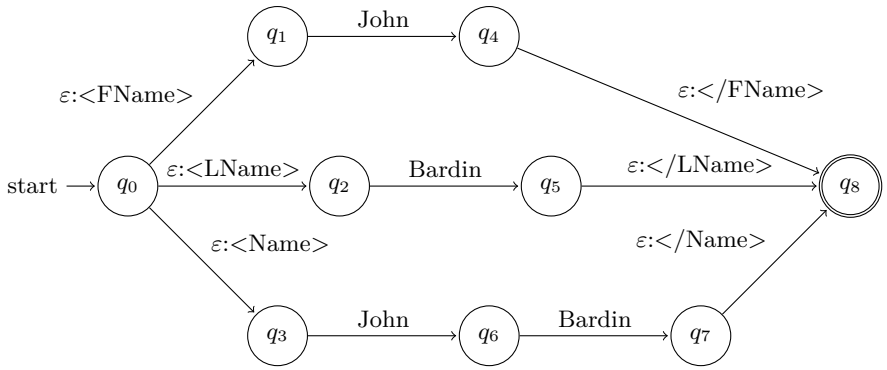
At the 2011 SFCM conference, it was remarked that the Pmatch system presented in [7], while of obvious practical interest, lacked a free implementation and certain useful features, such as flag diacritics. The idea of implementing something similar for an existing FST library became apparent, and ultimately the HFST team became motivated to design a rule-based named-entity recogniser (NER) by first implementing a subset of Pmatch deemed necessary for that purpose. Beyond the tagging concept, runtime contexts, named subnetworks and various utilities were most crucial and were implemented as need arose.

An overview of the relevant features is presented in Section 1.5. Building on an existing Xerox-oriented regex parser API (in `libhfst`) and a runtime-oriented transducer format with support for flag diacritics (`hfst-optimized-lookup`, see [18] and [9]), the remaining requisites were:

1. A mechanism for naming and retrieving transducers during compilation.
2. A scheme of control symbols to direct the runtime operation of the matching.
3. Logic for compiling new features.
4. A runtime tool that particularly needs to deal with the non-FST or state-preserving aspects of Pmatch.

We will first overview the details of some features.

Named-Entity Tagging. A straightforward way to accomplish tagging at the beginning and end of matches of first, last and complete names might look like this:

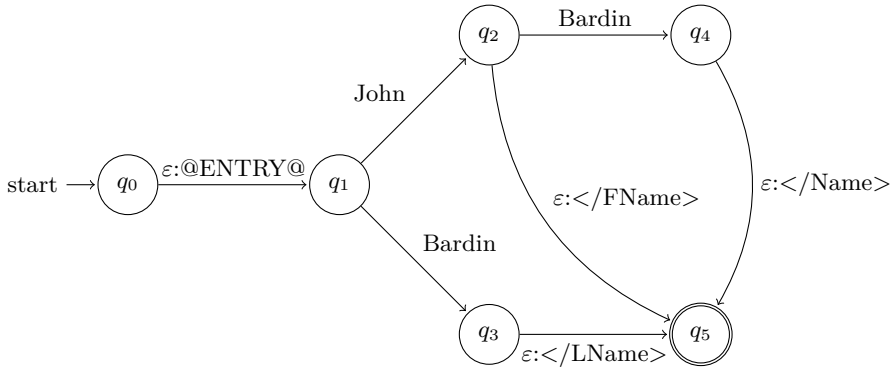


In this scenario, every new name is repeated in two places in the network. With large lists and multiple sources of this type of ambiguity, size inefficiencies can become serious.

One idea of Pmatch was to recognise the shared prefix in the first name "John" and the entire name "John Bardin" and to defer tag-writing until the entity has become unambiguous. In HFST, this is accomplished by detecting the tag directive during compilation and prefixing the subnetwork in question with an entry marker. After matching is complete, the entry marker is resolved in linear time with a simple position stack (in pseudocode):

```
for each symbol in result:
    if symbol == entry marker:
         push position into stack
    if symbol is end tag:
         insert corresponding start tag into position at stack top
         pop stack
         append end tag
    else:
         append symbol
```

With just the entry and end tags, the network simplifies to:



Contexts and States during Matching. Context markers trigger special runtime behavior and restrict progress during matching, very similarly to flag diacritics. There are two special considerations:

1. Left contexts are compiled to the left side of the network, in reverse (so that the first symbol to the left is at the end of the context).
2. Processing direction and position must be preserved during matching in a state stack.

Additionally, a stack for preserving the input tape position and the output tape content during each RTN (recursive transition network) invocation must be kept separately from the runtime context-checking stack. Otherwise, transition data is not duplicated, and these stacks are the only arbitrary amounts of memory reserved for accomplishing non-finite-state extensions.

Transduction. Each matching rule is by default compiled as an identity transduction. In many applications, however, it is useful to operate on input with some additional information, but give the output without such information. Matching is therefore not performed with automata, but with arbitrary transducers.

4 Future Work

The idea of combining linguistic rules and statistical models is intriguing but nontrivial. However, a pure finite-state left-to-right system is likely to be less efficient for syntactic parsing than a chart-based system, so the solution is probably to add linguistic constraints in the form of weighted finite-state constraints to a statistical parser before estimating the weights.

While existing statistical models like HMMs and PCFGs can incorporate a great deal of useful information for tasks like part-of-speech tagging and syntactic parsing, there are phenomena like non-local congruence which are too complex to estimate for these models. Probably because of this, there has been a growing interest in combining rule-based and statistical methods in core NLP tasks, such as part-of-speech tagging and syntactic parsing [11]. Such a combination presents challenges both for statistical estimation and inference methods and for the representation of linguistic information in a way which is compatible with a statistical system.

Finite-state transducers and automata can be used for expressing linguistically relevant phenomena for tagging and parsing as regular string sets. The validity of this approach is demonstrated by the success of parsing systems like Constraint Grammar [6], which utilizes finite-state constraints. Weighted machines offer the added benefit of expressing phenomena as fuzzy sets in a compact way. This makes them an excellent candidate for adding linguistic knowledge to statistical models.

5 Conclusion

The paper presented various NLP tools implemented with HFST and the minimal extensions they required to a pure finite-state system. In particular, the paper described an implementation of a full-scale named-entity recognizer for Swedish using Pmatch achieving a 10-fold compile-time speed-up compared with the original Flex implementation.

Acknowledgments The research leading to these results has received funding from FIN-CLARIN, Langnet and the European Commission's 7th Framework Program under grant agreement n° 238405 (CLARA).

References

1. Bouma, G., Noord, G.V., Malouf, R.: Alpino: Wide-coverage computational analysis of Dutch. In: CLIN 2000. vol. 8, pp. 45–59. Rodopi (2000)
2. Brants, S., Dipper, S., Hansen, S., Lezius, W., Smith, G.: The TIGER treebank. In: Proceedings of the Workshop on Treebanks and Linguistic Theories. Sozopol (2002)
3. Cavnar, W.B., Trenkle, J.M.: N-gram-based text categorization. In: Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval. pp. 161–175 (1994)
4. Drobac, S., Silfverberg, M., Yli-Jyrä, A.: Implementation of replace rules using preference operator. In: Proceedings of the 10th International Workshop on Finite State Methods and Natural Language Processing. pp. 55–59. Association for Computational Linguistics, Donostia–San Sebastián (July 2012), http://www.aclweb.org/anthology/W12-6210
5. Einarsson, J.: Talbankens skriftspråkskonkordans (1976)
6. Karlsson, F.: Constraint grammar as a framework for parsing running text. In: Karlgren, H. (ed.) Proceedings of the 13th conference on Computational linguistics – Volume 3. pp. 168–173. COLING '90, Association for Computational Linguistics, Stroudsburg, PA, USA (1990), http://dx.doi.org/10.3115/991146.991176
7. Karttunen, L.: Beyond morphology: Pattern matching with FST. In: Mahlow, C., Piotrowski, M. (eds.) Systems and Frameworks for Computational Morphology. Communications in Computer and Information Science, vol. 100, pp. 1–13. Springer, Berlin Heidelberg (2011)
8. Kokkinakis, D.: Swedish NER in the Nomen Nescio project. In: Holmboe, H. (ed.) Nordisk Sprogteknologi – Nordic Language Technology 2002, pp. 379–398. Museum Tusculanums Forlag, Copenhagen (2003)
9. Lindén, K., Axelson, E., Hardwick, S., Pirinen, T.A., Silfverberg, M.: HFST—framework for compiling and applying morphologies. In: Mahlow, C., Piotrowski, M. (eds.) Systems and Frameworks for Computational Morphology. Communications in Computer and Information Science, vol. 100, pp. 67–85. Springer, Berlin Heidelberg (2011)
10. Lindén, K., Pirinen, T.: Weighted finite-state morphological analysis of Finnish compounds. In: Jokinen, K., Bick, E. (eds.) Nodalida 2009. NEALT Proceedings, vol. 4 (2009), http://www.ling.helsinki.fi/~klinden/pubs/linden09dnodalida.pdf

11. Manning, C.D.: Part-of-speech tagging from 97% to 100%: Is it time for some linguistics? In: CICLing (1). LNCS, vol. 6608, pp. 171–189. Springer (2011)
12. Marcus, M.P., Santorini, B., Marcinkiewicz, M.A.: Building a large annotated corpus of English: The Penn Treebank. Computational Linguistics 19(2), 313–330 (1993)
13. McDonald, D.D.: Internal and external evidence in the identification and semantic categorization of proper names. In: Boguraev, B., Pustejovsky, J. (eds.) Corpus Processing for Lexical Acquisition, pp. 21–39. MIT Press, Cambridge, MA (1996)
14. Nadeau, D., Sekine, S.: A survey of named entity recognition and classification. Lingvisticae Investigationes 30(1), 3–26 (2007)
15. Pirinen, T.: Suomen kielen äärellistilainen automaattinen morfologinen analyysi avoimen lähdekoodin menetelmin. Master's thesis, Helsingin yliopisto (2008), http://www.helsinki.fi/~tapirine/gradu/
16. Pirinen, T., Silfverberg, M., Lindén, K.: Improving finite-state spell-checker suggestions with part of speech n-grams. In: IJCLA (2012)
17. Pirinen, T.A., Lindén, K.: Finite-state spell-checking with weighted language and error models. In: Proceedings of the Seventh SaLTMiL workshop on creation and use of basic lexical resources for less-resourced languagages. pp. 13–18. Valletta, Malta (2010), http://siuc01.si.ehu.es/%7Ejipsagak/SALTMIL2010_Proceedings.pdf
18. Silfverberg, M., Lindén, K.: HFST runtime format—a compacted transducer format allowing for fast lookup. In: Watson, B., Courie, D., Cleophas, L., Rautenbach, P. (eds.) FSMNLP 2009 (13 July 2009), http://www.ling.helsinki.fi/~klinden/pubs/fsmnlp2009runtime.pdf
19. Silfverberg, M., Lindén, K.: Combining statistical models for POS tagging using finite-state calculus. In: Nodalida 2011. Riga, Latvia (2011)