State-of-the-Art in Weighted Finite-State Spell-Checking

Tommi A Pirinen and Krister Lindén
University of Helsinki
Department of Modern Languages
tommi.pirinen@helsinki.fi and krister.linden@helsinki.fi

Last Modification: May 4, 2014

Abstract

The following claims can be made about finite-state methods for spell-checking: 1) Finite-state language models provide support for morphologically complex languages that word lists, affix stripping and similar approaches do not provide; 2) Weighted finite-state models have expressive power equal to other, state-of-the-art string algorithms used by contemporary spell-checkers; and 3) Finite-state models are at least as fast as other string algorithms for lookup and error correction. In this article, we use some contemporary non-finite-state spell-checking methods as a baseline and perform tests in light of the claims, to evaluate state-of-the-art finite-state spell-checking methods. We verify that finite-state spell-checking systems outperform the traditional approaches for English. We also show that the models for morphologically complex languages can be made to perform on par with English systems.

Keywords: spell-checking, weighted finite-state technology, error models

1 Introduction

[1] Spell-checking and correction is a traditional and well-researched part of computational linguistics. Finite-state methods for language models are widely recognized as a good way to handle languages which are morphologically more complex [1]. In this article, we evaluate weighted, fully finite-state spell-checking systems for morphologically complex languages. We use existing finite-state models and algorithms and describe some necessary additions to bridge the gaps and surpass state-of-the-art in non-finite-state spell-checking. For the set of languages, we have chosen to study North Sámi and Finnish from the complex, agglutinative group of languages, Green- landic from the complex poly-agglutinative group, and English to confirm that our finite-state formulations of traditional spelling correction applications are working as described in the literature.

As contemporary spell-checkers are increasingly using statistical approaches for the task, weighted finite-state models provide the equivalent expressive power, even for the

[1] This version is Tommi A Pirinen's post-print draft. The official publication may differ. This version uses my own documentclass instead of official publication's, if any. This version is optimised for screen reading. This version is fully reformatted from original which was made during serious time constraints due to my graduation.

morphologically more complex languages, by encoding the probabilities as weights in the automata. As the programmatic noisy channel models [2] can encode the error probabilities when making the corrections, so can the weighted finite-state automata encode these probabilities.

The task of spell-checking is split into two parts, error detection and error correction. Error detection by language model lookup is referred to as non-word or isolated error detection. The task of detecting isolated errors is often considered trivial or solved in many research papers dealing with spelling correction, e.g. [3]. More complex error detection systems may be used to detect words that are correctly spelled, but are unsuitable in the syntactic or semantic context. This is referred to as real-word error detection in context [4].

The task of error-correction is to generate the most likely correct word-forms given a misspelled word-form. This can also be split in two different tasks: generating sug- gestions and ranking them. Generating corrections is often referred to as error modeling. The main point of error modeling is to correct spelling errors accurately by ob- serving the causes of errors and making predictive models of them [5]. This effectively splits the error models into numerous sub-categories, each applicable to correcting specific types of spelling errors. The most used model accounts for typos, i.e. the slip of a finger on a keyboard. This model is nearly language agnostic, although it can be tuned to each local keyboard layout. The other set of errors is more language and user-specific—it stems from the lack of knowledge or language competence, e.g., in non-phonemic orthographies, such as English, learners and unskilled writers commonly make mistakes such as writing their instead of there, as they are pronounced alike; similarly competence errors will give rise to common confusable words in other languages, such as missing an accent, writing a digraph instead of its unigraph variant, or confusing one morph with another.

A common source of the probabilities for ranking suggestions related to competence errors are the neighboring words and word-forms captured in a language model. For morphologically complex languages, part-of-speech information is needed [3, 6], which can be compared with the studies on isolating languages [4, 7]. Context-based models like these are, however, considered to be out of scope for spell-checking, rather being part of grammar-checking.

Advanced language model training schemes, such as the use of morphological analyses as error detection evidence [4], require large manually verified morphologically analyzed and disambiguated corpora, which do not exist as open, freely usable resources, if at all. In addition, for polysynthetic languages like Greenlandic, even a gigaword corpus is usually not nearly as complete as an English corpus with a million word-forms.

As we compare existing finite-state technologies with contemporary non-finite-state string algorithm solutions, we use Hunspell[2] as setting the current de facto standard in open-source spell-checking and the baseline for the quality to achieve. Taken together this paper demonstrates for the first time that using weighted finite-state technology, spell-checking for morphologically complex languages can be made to perform on par with English systems and surpass the current de facto standard.

This article is structured as follows: In Subsection 1.1, we briefly describe the history of spell-checking up to the finite-state formulation of the problem. In Subsection 1.2, we revisit the notations behind the statistics we apply to our language and error models. In Section 2, we present existing methods for creating finite-state language

[2] `http://hunspell.sf.net`

and error models for spell-checkers. In Section 3, we present the actual data, the language models, the error models and the corpora we have used, and in Section 4, we show how different languages and error models affect the accuracy, precision, and speed of finite-state spell-checking. In Section 5, we discuss the results, and finally, in Section 6, we conclude our findings.

1.1 A Brief History of Automatic Spell-Checking and Correction

Automatic spelling correction by computer is in itself, an old invention, with the initial work done as early as in the 1960's. Beginning with the invention of the generic error model for typing mistakes, the Levenshtein-Damerau distance [8, 9] and the first applications of the noisy channel model [10] to spell-checking [11], the early solutions treated the dictionaries as simple word lists, or later, word-lists with up to a few affixes with simple stem mutations and finally some basic compounding process- es. The most recent and widely spread implementation with a word-list, stem muta- tions, affixes and some compounding is Hunspell, which is in common use in the open-source world of spell-checking and correction and must be regarded as the ref- erence implementation. The word-list approach, even with some affix stripping and stem mutations, has sometimes been found insufficient for morphologically complex languages. E.g. a recent attempt to utilize Hunspell for Finnish was unsuccessful [12]. In part, the popularity of the finite-state methods in computational linguistics seen in the 1980's was driven by a need for the morphologically more complex languages to get language models and morphological analyzers with recurring derivation and compounding processes [13]. They also provide an opportunity to use arbitrary finite-state automata as language models without modifying the runtime code, e.g. [14].

Given the finite-state representation of the dictionaries and the expressive power of the finite-state systems, the concept of a finite-state based implementation for spelling correction was an obvious development. The earliest approaches presented an algorithmic way to implement the finite-state network traversal with error-tolerance [15] in a fast and effective manner [16, 17]. Schulz and Mihov [18] presented the Levenshtein-Damerau distance in a finite-state form such that the finite-state spelling correction could be performed using standard finite-state algebraic operations with any existing finite-state library. Furthermore, e.g., Pirinen and Lindén [19] have shown that the weighted finite-state methods can be used to gain the same expressive power as the existing statistical spellchecking software algorithms.

1.2 Notations and some Statistics for Language and Error Models

In this article, where the formulas of finite-state algebra are concerned, we assume the standard notations from Aho et al. [20]: a finite-state automaton $\mathcal{M}$ is a system $(Q, \Sigma, \delta, Q_s, Q_f, W)$, where Q is the set of states, Σ the alphabet, δ the transition mapping of form $Q \times \Sigma \to Q$, and Q_s and Q_f the initial and final states of the automaton, respectively. For weighted automata, we extend the definition in the same way as Mohri [21] such that δ is extended to the transition mapping $Q{\times}\Sigma{\times}W \to Q$, where W is the weight, and the system additionally includes a final weight mapping $\rho : Q_f \to W$. The structure we use for weights is systematically the tropical semiring $(\mathbb{R}_+ \cup \infty, \min, +\infty, 0)$, i.e. weights are positive real numbers that are collected by addition. The tropical semiring models penalty weighting.

For the finite-state spell-checking, we use the following common notations: $\mathcal{M}_{\mathrm{D}}$ is a single tape weighted finite-state automaton used for detecting the spelling errors,

$\mathcal{M}_{\mathrm{S}}$ is a single tape weighted finite-state automaton used as a language model when suggesting correct words, where the weight is used for ranking the suggestions. On many occasions, we consider the possibility that $\mathcal{M}_{\mathrm{D}} = \mathcal{M}_{\mathrm{S}}$. The error models are weighted two-tape automata commonly marked as $\mathcal{M}_{\mathrm{E}}$. A word automaton is generally marked as $\mathcal{M}_{\text{word}}$. A misspelling is detected by composing the word automaton with the detection automaton:

$$\mathcal{M}_{\text{word}} \circ \mathcal{M}_{\mathrm{D}}, \tag{1}$$

which results in an empty automaton on a misspelling and a non-empty automaton on a correct spelling. The weight of the result may represent the likelihood or the correctness of the word-form. Corrections for misspelled words can be obtained by composing a misspelled word, an error model and a model of correct words:

$$\mathcal{M}_{\text{word}} \circ \mathcal{M}_{\mathrm{E}} \circ \mathcal{M}_{\mathrm{S}}, \tag{2}$$

which results in a two-tape automaton consisting of the misspelled word-form mapped to the spelling corrections described by the error model $\mathcal{M}_{\mathrm{E}}$ and approved by the suggestion language model $\mathcal{M}_{\mathrm{S}}$. Both models may be weighted and the weight is collected by standard operations as defined by the effective semiring.

Where probabilities are used, the basic formula to estimate probabilities from discrete frequencies of events (word-forms, mistyping events, etc.) is as follows: $P(x) = \frac{c(x)}{\text{corpussize}}$, x where is the event, $c(x)$ is the count or frequency of the event, and corpussize is the sum of all event counts in the training corpus. The en coding of probability as tropical weights in a finite-state automaton is done by setting is the end weight of path $Q_{\pi_x} = -\log P(x)$, though in practice the $-\log P(x)$ weight may be distributed along the path depending on the specific implementation. As events not appearing in corpora should have a larger probability than zero, we use additive smoothing $P(\hat{x}) = \frac{c(x)+\alpha}{\text{corpussize}+\text{dictionarysize}\times\alpha}$, so for an unknown event , the probability will be counted as if it had α appearances. Another approach would be to set $P(\hat{x}) < \frac{1}{\text{corpussize}}$, which makes the probability distribution leak but may work under some conditions [22].

1.3 Morphologically Complex Resource-Poor Languages

One of the main reasons for going fully finite-state instead of relying on word-form lists and affix stripping is the claim that morphologically complex languages simply cannot be handled with sufficient coverage and quality using traditional methods. While Hunspell has virtually 100 % domination of the open-source spell-checking field, authors of language models for morphologically complex languages such asTurkish (cf. Zemberek[3]) and Finnish (cf. Voikko[4]) have still opted to write separate software, even though it makes the usage of their spell-checkers troublesome and the coverage of supported applications much smaller.

Another aspect of the problems with morphologically complex languages is that the amount of training data in terms of running word-forms is greater, as the amount of unique word-forms in an average text is much higher compared with morphologi- cally less complex languages. In addition, the majority of morphologically complex languages tend to have fewer resources to train the models. For training spelling checkers, the data needed is merely correctly written unannotated text, but even that is scarce

[3] http://code.google.com/p/zemberek

[4] http://voikko.sf.net

when it comes to languages like Greenlandic or North Sámi. Even a very simple probabilistic weighting using a small corpus of unverified texts will improve the quality of suggestions [19], so having a weighted language model is more effective.

2 Weighting Finite-State Language and Error Models

The task of spell-checking is divided into locating spelling errors, and suggesting the corrections for the spelling errors. In finite-state spell-checking, the former task requires a language model that can tell whether or not a given string is correct. The error correction requires two components: a language model and an error model.

The error model is a two-tape finite-state automaton that can encode the relation between misspellings and the correctly typed words. This relation can also be weighted with the probabilities of making a specific typo or error, or arbitrary hand-made penalties as with many of the traditional non-finite-state approaches, e.g. [23].

The rest of this section is organized as follows. In Subsection 2.1, we describe how finite-state language models are made. In Subsection 2.2, we describe how finite-state error models are made. In Subsection 2.3, we describe some methods for combining the weights in language models with the weights in error models.

2.1 Compiling Finite-State Language Models

The baseline for any language model as realized by numerous spell-checking systems and the literature is a word-list (or a word-form list). One of the most popular examples of this approach is given by Norvig [24], describing a toy spelling corrector being made during an intercontinental flight. The finite-state formulation of this idea is equally simple; given a list of word-forms, we compile each string as a path in an automaton [25]. In fact, even the classical optimized data structures used for efficiently encoding word lists, like tries and acyclic deterministic finite-state automata, are usable as finite-state automata for our purposes, without modifications. We have: $\mathcal{M}_{\mathrm{S}} = \mathcal{M}_{\mathrm{D}} = \bigcup_{\mathrm{wf} \in \mathrm{corpus}} \mathrm{wf}$, where wf is a word-form and corpus is a set of word-forms in a corpus. These are already valid language models for Formula 1 and 2, but in practice any finite-state lexicon [1] will suffice.

2.2 Compiling Finite-State Versions of Error Models

The baseline error model for spell-checking is the Damerau-Levenshtein distance measure. As the finite-state formulations of error models are the most recent devel- opment in finite-state spell-checking, the earliest reference to a finite-state error mod- el in an actual spell-checking system is by Schulz and Mihov [18]. It also contains a very thorough description of building finite-state models for different edit distances. As error models, they can be applied in Formula 2.The edit distance type error models used in this article are all simple edit distance models.

One of the most popular modifications to speed up the edit distance algorithm is to disallow modifications of the first character of the word [26]. This modification provides a measurable speed-up at a low cost to recall. The finite-state implementation of it is simple; we concatenate one unmodifiable character in front of the error model.

Hunspell's implementation of the correction algorithm uses configurable alphabets for the error types in the edit distance model. The errors that do not come from regular typing mistakes are nearly always covered by specific string transformations,

i.e. confusion sets. Encoding a simple string transformation as a finite-state automaton can be done as follows: for any given transformation $S : U$, we have a path $\pi_{s:u} = S_1 : U_1 S_2 : U_2 \ldots S_n : U_n$, where $n = \max(|S|, |U|)$ and the missing characters of the shorter word substituted with epsilons. The path can be extended with arbitrary contexts L, R, by concatenating those contexts on the left and right, respectively. To apply these confusion sets on a word using a language model, we use the following formula: $\mathcal{M}_{\text{confusion}} = \bigcup_{\text{S:U}\in\text{CP}} S : U$, where CP is a set of confused string pairs. The error model can be applied in a standard manner in Formula 2. For a more detailed descrip- tion of a finite-state implementation of Hunspell error models, see Pirinen and Linden [19].

2.3 Combining Weights from Different Sources and Different Models

As both our language and error models are weighted automata, the weights need to be combined when applying the error and the language models to a misspelled string. Since the application performs what is basically a finite-state composition as defined in Formula 2, the default outcome is a weight semiring multiplication of the values; i.e., a real number addition in the tropical semiring. This is a reasonable way to combine the models, which can be used as a good baseline. In many cases, however, it is preferable to treat the probabilities or the weights drawn from different sources as unequal in strength. For example, in many of the existing spelling-checker systems, it is preferable to first suggest all the corrections that assume only one spelling error before the ones with two errors, regardless of the likelihood of the word forms in the language model. To accomplish this, we scale the weights in the error model to ensure that any weight in the error model is greater than or equal to any weight in the language model: $\hat{w_e} = w_e + maxw(\mathcal{M}_{\text{S}})$, where $\hat{w_e}$ is the scaled weight of error model weights, w_e the original error model weight and $maxw(\mathcal{M}_{\text{S}})$ the maximum weight found in the language model used for error corrections.

3 The Language and Error Model Data Used For Evaluation

To evaluate the weighting schemes and the language and the error models, we have selected two of the morphologically more complex languages with little to virtually no corpus resources available: North Sámi and Greenlandic. Furthermore, as a morphologically complex language with moderate resources, we have used Finnish. As a comparative baseline for a morphologically simple language with huge corpus resources, we use English. English is also used here to reproduce the results of the existing models to verify functionality of our selected approach.

This section briefly introduces the data and methods to compile the models; for the exact implementation, for any reproduction of results or for attempts to implement the same approaches for another language, the reader is advised to utilize the scripts, the programs and the makefiles available at our source code repository. [5]

In Table 1, we show the statistics of the data we have drawn from Wikipedia for training and testing purposes. In case of English and Finnish, the data is selected from

[5] `https://github.com/flammie/purplemonkeydishwasher/tree/master/2014-cicling`

Table 1: The extent of Wikipedia data per language

Data: Language	Train tokens	Train types	Test tokens	Test types
English	276,730,786	3,216,142	111,882,292	1,945,878
Finnish	9,779,826	1,065,631	4,116,896	538,407
North Sámi	183,643	38,893	33,722	8,239
Greenlandic	136,241	28,268	7,233	1,973

a subset of Wikipedia test tokens. With North Sámi and Greenlandic, we had no other choice but to use all Wikipedia test tokens.

For the English language model, we use the data from Norvig [24] and Pirinen and Hardwick [25], which is a basic language model based on a frequency weighted wordlist extracted from freely available Internet corpora such as Wikipedia and project Gutenberg. The language models for North Sámi, Finnish and Greenlandic are drawn from the free/libre open-source repository of finite-state language models managed by the University of Tromsø.[6] The language models are all based on the morphological analyzers built in the finite-state morphology [1] fashion. The repository also includes the basic versions of finite-state spell-checking under the same framework that we use in this article for testing. To compile our dictionaries, we have used the makefiles available in the repository. The exact methods for this are also detailed in the source code of the repository.

The error models for English are combined from a basic edit distance with English alphabet a-z and the confusion set from Hunspell's English dictionary containing 96 confusion pairs[7]. The error models for North Sámi, Finnish and Greenlandic are the edit distances of English with addition of åääöšžđ and ŧ and for North Sámi and åääöšž for Finnish. For North Sámi we also use the actual Hunspell parts from the divvun speller[8]; for Greenlandic, we have no confusion sets or character likelihoods for Hunspell-style data, so only the ordering of the Hunspell correction mechanisms is retained. For English, the Hunspell phonemic folding scheme was not used. This makes the English results easier to compare with those of other languages, which do not even have any phonemic error sources.

The keyboard adjacency weighting and optimization for the English error models is based on a basic qwerty keyboard. The keyboard adjacency values are taken from the CLDR Version 22[9], modified to the standard 101—104 key PC keyboard layout.

The training corpora for each of the languages are based on Wikipedia. To estimate the weights in the models, we have used the correct word-forms of the first 90 % of Wikipedia for the language model and the non-words for the error model. We used the remaining 10 % for extracting non-words for testing. The error corpus was extracted with a script very similar to the one described by Max and Wisniewski [27]. The script that performs fetching and cleaning can be found in our repository. [10] We have selected the spelling corrections found in Wikipedia by only taking those, where the incorrect version does not belong to the language model (i.e. is a non-word error), and the corrected word-form does.

[6] `http://giellatekno.uit.no`
[7] as found in en-US.aff from Ubuntu Linux LTS 12.04
[8] `http://divvun.no`
[9] http://cldr.unicode.org
[10] `https://github.com/flammie/purplemonkeydishwasher/tree/master/2014-cicling`

Table 2: The word-form coverage of the language models on test data (in %).

English aspell	22.7
English full automaton	80.1
Finnish full automaton	64.8
North Sámi Hunspell	34.4
North Sámi full automaton	48.5
Greenlandic full automaton	25.3

3.1 The Models Used For Evaluation

The finite-state language and error models described in this article have a number of adjustable settings. For weighting our language models, we have picked a subset of corpus strings for estimating word form probabilities. As both North Sámi and Greenlandic Wikipedia were quite limited in size, we used all strings except those that appear only once (hapax legomena) whereas for Finnish, we set the frequency threshold to 5, and for English, we set it to 20. For English, we also used all word-forms in the material from Norvig's corpora, as we believe that they are already hand-selected to some extent.

As error models, we have selected the following combinations of basic models: the basic edit distance consisting of homogeneously weighted errors of the Levenshtein-Damerau type, the same model limited to the non-first positions of the word, and the Hunspell version of the edit distance errors (i.e. swaps only apply to adjacent keys, and deletions and additions are only tried for a selected alphabet).

4 The Speed and Quality of Different Finite-State Models and Weighting Schemes

To evaluate the systems, we have used a modified version of the HFST spell-checking tool `hfst-ospell-survey` 0.2.4 otherwise using the default options, but for the speed measurements we have used the `--profile` argument. The evaluation of speed and memory usage has been performed by averaging over five test runs on a dedicated test server: an Intel Xeon E5450 at 3 GHz, with 64 GB of RAM memory. The rest of the section is organized as follows: in Subsection 4.1, we show naïve coverage baselines. In Subsection 4.2, we measure the quality of spell-checking with real-world spelling corrections found in Wikipedia logs. Finally in Subsections 4.3 and 4.4, we provide the speed and memory efficiency figures for these experiments, respectively.

4.1 Coverage Evaluation

To show the starting point for spell-checking, we measure the coverage of the lan- guage models. That is, we measure how much of the test data can be recognized using only the language models, and how many of the word-forms are beyond the reach of the models. The measurements in Table 2 are measured over word-forms in running text that can be measured in reasonable time, i.e. no more than the first 1,000,000 word-forms of each test corpus. As can be seen in Table 2, the task is very different for languages like English compared with morphologically more complex languages.

Table 3: The effect of different language and error models on correction quality (precision in % at a given suggestion position)

Rank:	1st	2nd	3rd	4th	5th	rest
Language and error models						
English aspell	55.7	5.7	8.0	2.2	0.0	0.0
English Hunspell	59.3	5.8	3.5	2.3	0.0	0.0
English w/ 1 error	66.7	7.0	5.2	1.8	1.8	1.8
English w/ 1 non-first error	66.7	8.8	7.0	0.0	0.0	1.8
Finnish aspell	21.1	5.8	3.8	1.9	0.0	0.0
Finnish w/ 1 error	54.8	19.0	7.1	0.0	0.0	0.0
Finnish w/ 1 non-first error	54.8	21.4	4.8	0.0	0.0	0.0
North Sámi Hunspell	9.4	3.1	0.0	3.1	0.0	0.0
North Sámi w/ 1 error	3.5	3.5	0.0	6.9	0.0	0.0
North Sámi w/ 1 non-first error	3.5	3.5	0.0	6.9	0.0	0.0
Greenlandic w/ 1 error	13.3	2.2	6.7	2.2	0.0	8.9
Greenlandic w/ 1 non-first error	13.3	2.2	6.7	2.2	0.0	8.9

4.2 Quality Evaluation

To measure the quality of spell-checking, we have run the list of misspelled words through the language and error models of our spelling correctors, extracting all the suggestions. The quality, in Table 3, is measured by the proportion of correct sugges- tions appearing at a given position 1-5 and finally the proportion appearing in any remaining positions.

On the rows indicated with error, in Table 3, we present the baselines for using language and error models allowing one edit in any position of the word. The rows with "non-first error" show the same error models with the restriction that the first letter of the word may not be changed.

Finally, in Table 3, we also compare the results of our spell-checkers with the actual systems in everyday use, i.e. the Hunspell and aspell in practice. When looking at this comparison, we can see that for English data, we actually provide an overall im- provement already by allowing only one edit per word. This is mainly due to the weighted language model which works very nicely for languages like English. The data on North Sámi on the other hand shows no meaningful improvement neither with the change from Hunspell to our weighted language models nor with the restriction of the error models.

Some of the trade-offs are efficiency versus quality. In Table 3, we measure among other things the quality effect of limiting the search space in the error model. It is important to contrast these results with the speed or memory gains shown in the corresponding Tables 4 and 5. As we can see, the optimizations that limit the search space will generally not have a big effect on the results. Only the results that get cut out of the search space are moved. A few of the results disappear or move to worse positions.

4.3 Speed Evaluation

For practical spell-checking systems, there are multiple levels of speed requirements, so we measure the effects of our different models on speed to see if the optimal models can actually be used in interactive systems, off-line corrections, or just batch processing. In Table 4, we show the speed of different model combinations for spell-

Table 4: The effect of different language and error models on speed of spelling correction (startup time in seconds, correction rate in words per second)

Input: 1st word	all words	non-words	
Language and error models			
English Hunspell	0.5	174	40
English w/ 1 error	0.06	5,721	6,559
English w/ 1 non-first error	0.20	16,474	17,911
Finnish aspell	<0.1	781	686
Finnish w/ 1 error	1.0	166	357
Finnish w/ 1 non-first error	1.0	303	1,886
North Sámi Hunspell	4.51	3	2
North Sámi w/ 1 error	0.28	2,304	2,839
North Sámi w/ 1 non-first error	0.27	5,025	7,898
Greenlandic w/ 1 error	1.27	49	142
Greenlandic w/ 1 non-first error	1.25	85	416

checking—for a more thorough evaluation of the speed of the finite-state language and the error models we refer to Pirinen et al. [25]. We perform three different test sets: startup time tests to see how much time is spent on startup alone; a running cor- pus processing test to see how well the system fares when processing running text;and a non-word correcting test, to see how fast the system is when producing correc- tions for words. For each test, the results are averaged over at least 5 runs.

In Table 4, we already notice an important aspect of finite-state spelling correction: the speed is very predictable, and in the same ballpark regardless of input data. Furthermore, we can readily see that the speed of a finite-state system in general outperforms Hunspell with both of the language models we compare. Furthermore, we show the speed gains achieved by cutting the search space to disallow errors in the first character of a word. This is the speed-equivalent of Table 3 of the previous section, which clearly shows the trade-off between speed and quality.

4.4 Memory Usage Evaluation

Depending on the use case of the spell-checker, memory usage may also be a limiting factor. To give an idea of the memory-speed trade-offs that different finite-state models entail, in Table 5, we provide the memory usage values when performing the evaluation tasks above. The measurements are performed with the Valgrind utility and represent the peak memory usage. It needs to be emphasized that this method, like all of the methods of measuring memory usage of a program, has its flaws, and the figures can at best be considered rough estimates.
[11]

5 Discussion

The improvement of quality by using simple probabilistic features for spell-checking is well-studied, e.g. by Church and Gale [29]. In our work, we describe introducing

[11] Drobac & al. [28] report on how to hyper-minimize finite-state lexicons keeping the Greenlandic lexicon at less than 20 MB at runtime which gives a considerable speed-up of loading with only a small reduction of runtime speed.

Table 5: The peak memory usage of processes checking and correcting word-forms with various language and error model combinations.

Measurement:	Peak memory usage
Language and error models	
English Hunspell	7.5 MB
English w/ 1 error	7.0 MB
English w/ 1 non-first error	7.0 MB
Finnish aspell	186 kB
Finnish w/ 1 error	79.3 MB
Finnish w/ 1 non-first error	79.3 MB
North Sámi Hunspell	151.0 MB
North Sámi w/ 1 error	31.4 MB
North Sámi w/ 1 non-first error	31.4 MB
Greenlandic w/ 1 error	300.0 MB
Greenlandic w/ 1 non-first error	300.7 MB

probabilistic features into a finite-state spell-checking system giving a similar increase in the quality of the spell-checking suggestions as seen in previous approaches. The methods are usable for a morphologically varied set of languages.

The speed to quality trade-off is a well-known feature in spell-checking systems, and several aspects of it have been investigated in previous research. The concept of cutting away string initial modifications from the search space has often been suggested [26, 30], but only rarely quantified extensively. In this paper we have investigated its effects on finite-state systems and complex languages. We noted that it gives speed improvements in line with previous solutions, and we also verified that the quality deterioration on real-world data is minimal.

In this paper, we have reviewed basic finite-state language and error models for spelling correction. The obvious future improvements that need to be researched are extensions, e.g. to errors at edit distance 2 or more, as well as more elaborate models for both model types. The combination of adaptive technologies based on user feed-back at runtime and finite-state models has not been researched in spelling correction, but it has shown good results in practical spelling correction applications.

6 Conclusion

We have demonstrated that finite-state spell-checking is a feasible alternative to traditional string algorithm-driven versions by verifying three claims. The language support has been demonstrated by the fact that there is a working implementation of Greenlandic that could not have been successfully implemented without finite-state models, and by giving finite-state versions of the North Sámi and English language models that cover more Wikipedia word forms than the non-finite-state equivalents. In addition, the suggestion mechanism using a weighted finite-state implementation isable to provide better quality suggestions for English and Finnish than corresponding non-finite-state implementations. The efficiency of the finite-state approach is verified by showing a reasonable or greater speed when compared with Hunspell.

Acknowledgements

We thank our fellow researchers in the HFST research group at the University of Helsinki for ideas and discussions.

References

[1] Kenneth R Beesley and Lauri Karttunen. *Finite State Morphology*. CSLI publications, 2003.

[2] Eric Brill and Robert C. Moore. An improved error model for noisy channel spelling correction. In *ACL '00: Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, pages 286–293, Morristown, NJ, USA, 2000. Association for Computational Linguistics.

[3] J. Otero, J. Grana, and M. Vilares. Contextual spelling correction. *Computer Aided Systems Theory–EUROCAST 2007*, pages 290–296, 2007.

[4] Eric Mays, Fred J. Damerau, and Robert L. Mercer. Context based spelling correction. *Inf. Process. Manage.*, 27(5):517–522, 1991.

[5] S. Deorowicz and M.G. Ciura. Correcting spelling errors by modelling their causes. *International journal of applied mathematics and computer science*, 15(2):275, 2005.

[6] Tommi A Pirinen, Miikka Silfverberg, and Krister Lindén. Improving finite-state spell-checker suggestions with part of speech n-grams. In *CICLING 2012*, 2012.

[7] L. Amber Wilcox-O'Hearn, Graeme Hirst, and Alexander Budanitsky. Real-word spelling correction with trigrams: A reconsideration of the Mays, Damerau, and Mercer model. In Alexander F. Gelbukh, editor, *CICLing*, volume 4919 of *Lecture Notes in Computer Science*, pages 605–616. Springer, 2008.

[8] Fred J Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

[9] В.И. Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. *Доклады Академий Наук СССР*, 163(4):845–8, 1965.

[10] C.E. Shannon. A mathematical theory of communications, i and ii. *Bell Syst. Tech. J*, 27:379–423, 1948.

[11] J. Raviv. Decision making in Markov chains applied to the problem of pattern recognition. *Information Theory, IEEE Transactions on*, 13(4):536–551, 1967.

[12] Harri Pitkänen. Hunspell-in kesäkoodi 2006: Final report, 2006. Technical report; refered on 16th of September, available: `http://www.puimula.org/htp/archive/kesakoodi2006-report.pdf`.

[13] Kenneth R. Beesley. Morphological analysis and generation: A first step in natural language processing. In *First Steps in Language Documentation for Minority Languages: Computational Linguistic Tools for Morphology, Lexicon and Corpus Compilation, Proceedings of the SALTMIL Workshop at LREC*, pages 1–8, 2004.

[14] Tommi A Pirinen and Krister Lindén. 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*, pages 13–18, Valletta, Malta, 2010.

[15] Kemal Oflazer. Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. *Comput. Linguist.*, 22(1):73–89, 1996.

[16] Måns Huldén. Fast approximate string matching with finite automata. *Procesamiento del Lenguaje Natural*, 43:57–64, 2009.

[17] Agata Savary. Typographical nearest-neighbor search in a finite-state lexicon and its application to spelling correction. In *CIAA '01: Revised Papers from the 6th International Conference on Implementation and Application of Automata*, pages 251–260, London, UK, 2002. Springer-Verlag.

[18] Klaus Schulz and Stoyan Mihov. Fast string correction with levenshtein-automata. *International Journal of Document Analysis and Recognition*, 5:67–85, 2002.

[19] Tommi A Pirinen and Krister Lindén. Creating and weighting Hunspell dictionaries as finite-state automata. *Investigationes Linguisticae*, 21, 2010.

[20] Alfred V. Aho, Monica S. Lam, Rajiv Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*, volume 1009. Pearson/Addison Wesley, 2007.

[21] Mehryar Mohri. Weighted automata algorithms. *Handbook of weighted automata*, pages 213–254, 2009.

[22] T. Brants, A.C. Popat, P. Xu, F.J. Och, and J. Dean. Large language models in machine translation. In *EMNLP*, 2007.

[23] László Németh. Hunspell manual. Electronic Software Manual (manpage), 2011.

[24] Peter Norvig. How to write a spelling corrector. referred 2011-01-11, available `http://norvig.com/spell-correct.html`, 2010.

[25] Tommi A Pirinen and Sam Hardwick. Effects of weighted finite-state language and error models on speed and efficiency of finite-state spell-checking. In *Preproceedings of the 10th International Workshop on Finite State Methods and Natural Language Processing FSMNLP*, pages 6–14. University of the Basque Country, July 23–25 2012.

[26] Meenu Bhagat. Spelling error pattern analysis of punjabi typed text. Master's thesis, Thapar University, 2007.

[27] A. Max and G. Wisniewski. Mining naturally-occurring corrections and paraphrases from wikipedia's revision history. In *Proceedings of LREC*, 2010.

[28] Senka Drobac, Krister Lindén, Tommi Pirinen, and Miikka Silfverberg. Heuristic hyperminimization of finite-state lexicons. In *In the Proceedings of LREC 2014*, Reykavik, Iceland, 2014.

[29] Kenneth W. Church and William A. Gale. Probability scoring for spelling correction. *Statistics and Computing*, 1:93–103, 1991.

[30] Karen Kukich. Techniques for automatically correcting words in text. *ACM Comput. Surv.*, 24(4):377–439, 1992.