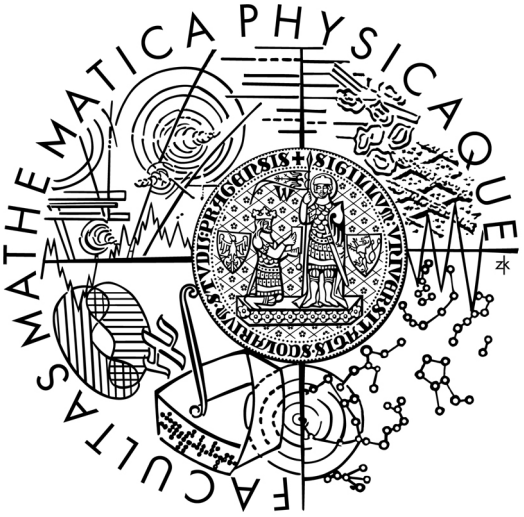
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Michal Richter

Pokročilý korektor češtiny

Ústav formální a aplikované lingvistiky

Vedoucí diplomové práce: *Mgr. Pavel Straňák*

Studijní program: *Informatika, Matematická lingvistika*

Chtěl bych poděkovat Mgr. Pavlu Straňákovi za vedení práce, poskytnutí velmi cenných rad a potřebného hardwaru. Dále pak Prof. Dietrichu Klakowovi za rady týkající se statistického modelování a kombinace jazykových modelů. Své ženě Olze děkuju za vytrvalou podporu, jazykovou korekturu podstatné části textu a za pomoc při trénování chybových modelů.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 5.8.2010 Michal Richter

Název práce: *Pokročilý korektor češtiny*

Autor: *Michal Richter*

Katedra (ústav): *Ústav formální a aplikované lingvistiky*

Vedoucí diplomové práce: *Mgr. Pavel Straňák*

e-mail vedoucího: *stranak@ufal.mff.cuni.cz*

Abstrakt: *Cílem práce je implementovat český spell-checker, který bude využívat jazykové modely a lexikální morfologické analýzy za účelem nabízení co nejkvalitnějšího seznamu možností oprav pro jednotlivé překlepy a za účelem odhalení překlepů, které jsou zároveň platnými českými slovy. Systém by měl zároveň poskytovat službu obnovy diakritiky v českém textu. Za cílovou platformu byl zvolen operační systém Mac OS X. Během implementace byl kladen důraz zejména na efektivní paměťovou reprezentaci statistických modelů.*

V práci je podán přehled o použitých metodách - HMMs, language models, Viterbi algorithm. *Dále je popsána vlastní implementace systému a trénování statistických modelů. Na závěr pak číselná evaluace úspěšnosti systému a diskuze dosažených výsledků.*

Klíčová slova: *opravování překlepů, obnovení diakritiky, statistické metody*

Title: *Advanced Czech Spellchecker*

Author: *Michal Richter*

Department: *Institute of Formal and Applied Linguistics*

Supervisor: Mgr. Pavel Straňák

Supervisor's e-mail address: stranak@ufal.mff.cuni.cz

Abstract: *The aim of this work is to implement a Czech spell-checker using several language models and a lexical morphological analyser in order to offer proper correction suggestions and also to find real-word spelling errors (spelling errors that happen to be in the lexicon). The system should also be able to complete diacritics to Czech text. Mac OS X was chosen as the target platform for the application. During the implementation, emphasis was put especially on memory-efficient representation of the above-mentioned statistical models.*

In the beginning, a gentle introduction to Hiden Markov Models, Language Models *and* Viterbi algorithm *is given. The actual system implementation and the statistical models training is discussed further. In the final part of the work, the achived results are evaluated and discussed in depth.*

Keywords: *spell-checking, diacritic completion, statistical methods*

Contents

1 Diploma Thesis Assignment

1.1 In English

The aim of this thesis is to design and implement a spell-checker, that will use the morphological lexicon and the thesaurus developed at UFAL (Institute of Formal and Applied Linguistics). A key feature of the application will be an ability to find real-word spelling errors (the spelling errors that accidentally form another lexical entry). The application will use the ability of morphological lexicon to setup restriction on style variants. The application should be also able to check a text with no diacritics or a text with diacritics encoded by TeX sequences. The important feature of the system will be an ordering of correction suggestions according to their probabilities. The application will also provide service for removing diacritics from a Czech text or completing diacritics to a text with no diacritics. The thesaurus will suggest properly inflected synonyms to a given word form.

The system will be implemented as a Spell Server for Mac OS X or as a web application.

1.2 In Czech

Cílem práce je navrhnout a implementovat spellchecker, který bude využívat morfologického slovníku a tezauru vyvinutého na UFALu. Zároveň bude klíčovou vlastností schopnost nalézt překlep, který tvoří správně zformované české slovo (a tedy je takové slovo ve slovníku).

Nástroj bude využívat možností nabízených morfologickým slovníkem pro omezení stylových variant, bude korigovat jak text s akcenty, tak bez nich i text s akcenty zapsanými TeXovými sekvencemi, to vše i v jednom dokumentu. Důležitou součástí systému bude také optimalizace nabízených náhrad podle jejich pravděpodobnosti. Dále nástroj nabídne možnost z textu diakritiku odstranit či ji přidat do textu, kde diakritika není. Tezaurus bude nabízet k danému slovnímu tvaru synonyma správně vyskloňovaná, časovaná, stupňovaná a negovaná.

Celý systém bude implementován Spell Server pro Mac OS X nebo jako webová aplikace.

2 Introduction

A spell-checker is a computer program whose task it is to find spelling errors in a written text and to suggest corrections to these spelling errors eventually. Spell-checkers usually run as a background process within a text editor such as Microsoft Word or OpenOffice and flags misspelled words by red underlining. A list of correction suggestions usually appears in a context menu, when the user right-clicks on a misspelled word or it is often possible to open a standalone spell-checking panel and carry out certain actions such as accepting or rejecting the spell-checker suggestions.

Most spellcheckers available are not particularly good at linguistics. They search for words typed by a user in their lexicons and flag those that fail to be recognized. Correction suggestions are usually sorted according to their similarity to a misspelled word or according to a more sophisticated model that assigns specific probabilities to distinct error types. However, the idea of using the information coming from the context of a misspelled word to improve the spell-checkers' performance is quite old (Mays, Damerau, & Mercer, 1991). In recent years, advanced spell-checkers that use context information and are sometimes even capable of recognizing real-word errors started appearing. For example, Microsoft Word 2007's spell-checker detects a certain number of real-word errors and underline them with green color. Google has recently released a feature called *Google Suggest* that checks search queries for spelling errors and offers quite reasonable corrections.

These methods are usually based on *Noisy-channel approach*, which will be described in Section 3.1. These spell-checkers use language models determining which word sequences are likely to appear in a written text. The system implemented in this work also belongs to the *Noisy-channel framework* and it also makes an extensive use of language models. In this case, however, an attempt was made to go one step further and to use language models that work with morphological features of words. These models prefer word sequences whose morphological categories form a likely sequence (for example, preposition *bez* followed by a noun in the 2nd case is likely to occur, but the occurence of *bez* followed by a 4th case noun is highly unlikely). Such models can to some extend capture simple grammatical phenomena such as subject-verb agreement in person and number or adjective-noun agreement in case and number. These models were combined with the standard models used in context-sensitive spell-checking in a *log-linear model*, which can be seen as a generalization of the *Noisy channel model*.

The purpose of this thesis, nevertheless, was not to come up just with a scientific prototype. The real aim was to develop a practical application that would help Czechs to write less erroneous texts. Because of this, special care was devoted to the efficiency of provided implementation, both in terms of memory and CPU consumption. ZipTBO method that uses about 4 bytes per n-gram was used for the representation of language models[1].

In the Czech language, there are certain special letters that do not belong to the standard Latin alphabet. These are all variations of standard Latin letters made by adding diacritic sign such as caron (háček) to Latin letters. When typing on computers, Czechs often write without diacritics (they substitute diacritic letters for their basic Latin-letter equivalent). The reasons for that were described in (Vrána, 2002) who wrote his Master Thesis on the topic of diacritic completion. To sum up, the omission of diacritics is caused by the fact that people use English keyboard layout and also by the fact that a text without diacritics does not suffer from problems caused by different diacritic encoding on different platforms[2].

Because of the nature of the implemented system, adding functions that complete diacritics was not that complicated a step. It was accomplished by simple substitution of an error model component that assigns costs to distinct edit operations such as a letter substitution, deletion, insertion and a letter swap. The error model for diacritic completion simply assigns zero costs to substitutions that add diacritic to a latin letter and infinite costs to any other edit operation. As a consequence, the only correction that system does, is the diacritic completion.

By providing custom error model implementation, it is also possible to achieve a functionality of spell-checking of text without diacritics (a word is considered to be correct, if a correct word can be formed by adding diacritics to it).

The target platform of the system is Mac OS X, yet the overwhelming part of the code was written in C

C++ and can be used on other platforms as well[3]. Only the classes providing Mac OS X Spelling Server interface are platform-dependent. The reason for chosing Mac OS X as the target platform was that it provides a unified spell-checking interface that almost every modern native Mac application respects. It opens up the opportunity to use the implemented spellchecker

[1]It is possible to go even further and reduce memory consumption to 3 bytes per n-gram (as proposed by (Church, Wa, Hart, & Gao, 2007)), but this representation is less CPU-efficient and ZipTBO seemed to be more suitable for the given task.

[2]This problem is less acute nowadays because of the widespread usage of UTF8 encoding.

[3]The current implementation, though, runs on low-endian machines only. For efficiency reasons, data structures such as lexicon, morphology lexicon and language models are stored in bit arrays and low-level bit manipulations are used. This is not a big limitation, since most personal computers are low-endian nowadays.

systemwide - when writting emails in Safari web browser, inside any text editor etc. However the drawback is that Apple computers are still quite rare among Czechs.

In Section 3, the statistical models used in this work will be introduced together with the description showing how they were utilized for the given tasks of spell-checking and diacritics completion. Section 4 describes the data that were used, the processing of these data and the process of the model training. Section 5 describes the details of the system implementation and Section 6 presents the results of the performance evaluation. Section 7 contains the conclusions of what was done and outlines possible future plans.

3 Background

The task of context-sensitive spelling correction and diacritics completion can be seen as a problem of sequence decoding which is often formulated in terms of noisy channel model or a more fine-grained log-linear model. These frameworks will be described in the following text together with statistical models and the decoding algorithm that was used. Sections on *language models* and *Viterbi Algorithm* are heavily based on the information provided in (Jurafsky & Martin, 2008), however the practical use of these concepts for the spelling-correction problem is described as well. Section 3.3.2 is based on the information that was found in (Yuret & Stolcke, n.d.).

3.1 Noisy Channel Model for Spelling Correction

Noisy-channel is widely used to model a wide area of NLP problems such as speech recognition, machine translation, question answering and also spelling correction. The noisy channel model expresses the following idea: A transmitter sends a sequence of symbols to a reciever. During the transfer, though, certain symbols of the transmitted sequence are confused due to the deficiencies of the transmission channel. The distorted sequence is then received by the receiver. The receiver's goal is to reconstruct the original sequence using the knowledge of the input data and transmission channel properties, the latter meaning e.g. the probabilities of specific errors' occurrence.

More formally, when the target sentence t is received, the goal is to find the most probable input sentence s according to

$$s = argmax_{s'} P(s'|t)$$

Using the Bayes theorem this can be rewritten as

$$s = argmax_{s'} P(s'|t) = argmax_{s'} P(t|s')P(s') \tag{1}$$

The term $P(s')$ expresses the a priori probability of the input sentence. It estimates the probability of s' being the transmitted sequence. The term $P(t|s')$ expresses the a posteriori probability of s' being transformed into t during the transfer. A probabilistic model that estimates $P(t|s')$ for all possible combinations of values for t and s' is usually denoted as an *error model*.

The problem of spelling correction can be easily expressed in the terms of the Noisy-channel

model. The transmitter is a person that intends to type a natural-language text. In his/her mind, an uncorrupted form of the text is present, but during its realization, errors are introduced, both due to the typing and spelling skills of that person. This process of text realization that introduces errors can be seen as a transfer phase. Finally, there is a receiver, a spell-checking application, that should recognize what was originally in the writer's mind and suggest possible corrections. The knowledge which words or sequences of words are common in the particular language is used to form the model of a priori probability. The knowledge of probabilities of distinct error types such as confusion of adjacent letters is used to form the error model.

More formally, the spelling-correction task can be defined as a problem of finding an optimal sequence of words $w_1^*...w_n^*$ given the input word sequence $w_1^{'}...w_n^{'}$, possibly containing spelling errors. The noisy channel formulation of spelling-correction problem conforming to Equation 1 can be written as

$$(w_1^*...w_n^*) = \underset{(\hat{w}_1...\hat{w}_n)}{\operatorname{argmax}} P\big(\hat{w}_1...\hat{w}_n|w_1^{'}...w_n^{'}\big) = \underset{(\hat{w}_1...\hat{w}_n)}{\operatorname{argmax}} P\big(w_1^{'}...w_n^{'}|\hat{w}_1...\hat{w}_n\big)P\big(\hat{w}_1...\hat{w}_n\big)$$

3.2 Log-Linear Model Incorporating Morphological Features

The purpose of a log-linear model is to provide a more general framework that allows to combine any number of feature functions and to provide a greater flexibility for parameter settings. A detailed description of log-linear models and their theoretical background is provided in (Smith, 2004). A general equation capturing a log-linear model for decoding of sequence $w_1^{'}...w_n^{'}$ can be written as follows

$$(w_1^*...w_n^*) = \underset{(\hat{w}_1...\hat{w}_n)}{\operatorname{argmax}} \sum_{j=1}^{N} \alpha_j \times f_j(\hat{w}_1...\hat{w}_n, w_1^{'}...w_n^{'})$$

where $f_{j=1}^N$ are feature functions and $\alpha_{j=1}^N$ are their weights.

Noisy-channel described in section 3.1 can be seen as a special instance of log-linear model as shown by the following derivation:

$$\begin{aligned}(w_1^*...w_n^*) &= \underset{(\hat{w}_1...\hat{w}_n)}{\operatorname{argmax}} P\big(w_1^{'}...w_n^{'}|\hat{w}_1...\hat{w}_n\big)P\big(\hat{w}_1...\hat{w}_n\big) \\ &= \underset{(\hat{w}_1...\hat{w}_n)}{\operatorname{argmax}} \big[1 \times \log\big(P(w_1^{'}...w_n^{'}|\hat{w}_1...\hat{w}_n)\big) + 1 \times \log\big(P(\hat{w}_1...\hat{w}_n)\big)\big]\end{aligned}$$

There are two feature functions $f_e = \log\big(P(w_1^{'}...w_n^{'}|\hat{w}_1...\hat{w}_n)\big)$ and $f_f = \log\big(P(\hat{w}_1...\hat{w}_n)\big)$ with weight parameters $\alpha_e = \alpha_f = 1$. In some situations, it may be reasonable to assign unequal

weights to the models (for example when there is plenty of data for a reliable estimation of the first model, but not enough data for the estimation of the second model).

However it may also be desirable to add other feature functions. Given the target language of the system (Czech), it might be desirable to utilize information coming from the rich morphology of the Czech language. For example, in the sentence

Úředníci vyhověli žádosti občanů.

the morphological suffix of the verb *vyhověli* shows that the subject of the sentence should be in plural and in masculine gender. This is consistent with morpholexical interpretation of the subject word *úředníci* that holds both properties required by the given verb. Such constraints can be defined by rules, but they can be also captured by statistics. Given that the morphological analysis for the given sentence is

```
Úředníci|NNMP1-----A---- vyhověli|VpMP---XR-AA---
žádosti|NNFP3-----A---- občana|NNMS2-----A----
```

a statistical system can induce from the given sentence that the word tagged as `VpMP---XR-AA---` is likely to appear after a word tagged as `NNMP1-----A----`. Of course, many training sentence instances are needed in order to estimate probabilities of distinct tag sequences correctly. The advantage of statistical models working with morphological tags over statistical models working with word forms is that they provide a basic generalization. Assuming that this sentence was included in training data, another sentence

Policisté naslouchali stížnosti nájemníka.

could be considered as highly probable according to a morphology-based statistical model, because the morphological interpretation is exactly the same as the morphological interpretation of the previous sentence which is included in the training data. On the other hand, a statistical model working with word forms would consider the sentence as highly unprobable if we assume that none of the word bigrams forming the sentence were contained in the training corpus.

The use of morphological features makes the data sparseness problem less acute, because the amount of possible morphological tag combinations is much smaller than the amount of possible combinations of word forms. Parameter space of the morphological model is thus significantly

smaller and the model can be estimated more accurately.

In this work, incorporation of morphological features was done by performing spelling correction, part-of-speech tagging and lemmatization simultaneously. Probability values given by a lemmatizer and POS tagger were used as extra features in the log-linear model.

POS tagging can be formalized as a search for the optimal sequence of morphological tags $t_1^*...t_n^*$ given a sequence of words $w_1^{'}...w_n^{'}$.

$$\begin{aligned}(t_1^*...t_n^*) &= \underset{(\hat{t}_1...\hat{t}_n)}{\operatorname{argmax}} P\big(\hat{t}_1...\hat{t}_n|w_1^{'}...w_n^{'}\big) = \underset{(\hat{t}_1...\hat{t}_n)}{\operatorname{argmax}} P\big(w_1^{'}...w_n^{'}|\hat{t}_1...\hat{t}_n\big)P\big(\hat{t}_1...\hat{t}_n\big) \\ &= \underset{(\hat{t}_1...\hat{t}_n)}{\operatorname{argmax}}\big[\log\big(P(w_1^{'}...w_n^{'}|\hat{t}_1...\hat{t}_n)\big) + \log\big(P\big(\hat{t}_1...\hat{t}_n\big)\big)\big] \\ &= \underset{(\hat{t}_1...\hat{t}_n)}{\operatorname{argmax}} f_t\big(\hat{t}_1...\hat{t}_n, w_1^{'}...w_n^{'}\big)\end{aligned} \tag{2}$$

Similarly, lemmatization can be formalized as a search for the optimal sequence $l_1^*...l_n^*$

$$\begin{aligned}(l_1^*...l_n^*) &= \underset{(\hat{l}_1...\hat{l}_n)}{\operatorname{argmax}} P\big(\hat{l}_1...\hat{l}_n|w_1^{'}...w_n^{'}\big) = \underset{(\hat{l}_1...\hat{l}_n)}{\operatorname{argmax}} P\big(w_1^{'}...w_n^{'}|\hat{l}_1...\hat{l}_n\big)P\big(\hat{l}_1...\hat{l}_n\big) \\ &= \underset{(\hat{l}_1...\hat{l}_n)}{\operatorname{argmax}}\big[\log\big(P(w_1...w_n|\hat{l}_1...\hat{l}_n)\big) + \log\big(P\big(\hat{l}_1...\hat{l}_n\big)\big)\big] \\ &= \underset{(\hat{l}_1...\hat{l}_n)}{\operatorname{argmax}} f_l\big(\hat{l}_1...\hat{l}_n, w_1^{'}...w_n^{'}\big)\end{aligned} \tag{3}$$

The functions f_t and f_l defined in equations 2 and 3 were used as features of the log-linear model which carries out a simultaneous search for the optimal word, tag and lemma sequence.

$$\begin{aligned}(w_1^*...w_n^*, l_1^*...l_n^*, t_1^*...t_n^*) = \underset{(\hat{w}_1...\hat{w}_n,\hat{t}_1...\hat{t}_n,\hat{l}_1...\hat{l}_n)}{\operatorname{argmax}} &\big[\alpha_e \times f_e(\hat{w}_1...\hat{w}_n, w_1^{'}...w_n^{'}) + \\ &\alpha_f \times f_f(\hat{w}_1...\hat{w}_n) + \alpha_t \times f_t(\hat{t}_1...\hat{t}_n, \hat{w}_1...\hat{w}_n) + \\ &\alpha_l \times f_l(\hat{l}_1...\hat{l}_n, \hat{w}_1...\hat{w}_n)\big]\end{aligned}$$

Instead of just looking for $(w_1^*...w_n^*)$ it is now requested to look for $(w_1^*...w_n^*, l_1^*...l_n^*, t_1^*...t_n^*)$, which may seem to be a much more complicated task. Nevertheless, the sequence $(w_1^*...w_n^*)$ remains the only important output parameter and the fact that we search for $(l_1^*...l_n^*, t_1^*...t_n^*)$ as well should only help us to obtain better $(w_1^*...w_n^*)$.

The idea of combining probabilistic models working with different morphological factors via a log-linear model was already used in machine translation (Koehn & Hoang, 2007).

3.3 N-Gram Language Models

The purpose of the language model is to provide a probabilistic distribution over the set S of all possible sentences of the given language, such that $\sum_{s \in S} P(s) = 1$. It models how likely a given sentence is to appear in a written text. This may seem absurd and it is arguable whether there really is such a thing as a probability of a given sentence. For the application purposes, the question of plausability of such probabilistic atitude is irrelevant. One can think of such probabilistic measure as a scoring function assigning the scores to syntanctically or semantically malformed sentences and high scores to well-formed sentences.

The probability $P(s)$ cannot be modelled directly, because the set of all possible sentences is tremendously large. It is posible, though, to split it into simpler probabilistic terms that can be modelled directly. By using the *chain rule of probability*, the sentence probability can be expressed as

$$P(s) = P(w_1, ..., w_n) = P(w_1) \times P(w_2|w_1) \times P(w_3|w_1, w_2) \times ... \times P(w_n|w_1, ..., w_{n-1})$$

Modelling of the conditional probabilities for longer word histories is not yet feasible. The intuition of the N-gram model suggests that it is possible to approximate the entire word history just by the few last words. The assumption that the probability of a word depends on the limited number of previous words only is called Markov assumption. For example, a trigram language model estimates the probability distribution for the next word on the basis of two previous words. For natural languages, the Markov assumption is always over-simplifying and leads to loss of important information. On the other hand, it makes the estimation of probability distribution feasible under the condition of limited data sources. For a trigram language model, the term expressing the probability of a sentence can be rewritten as

$$P(w_1, ..., w_n) = P(w_1) \times P(w_2|w_1) \times P(w_3|w_1, w_2) \times P(w_4|w_2, w_3) \times ... \times P(w_n|w_{n-2}, w_{n-1})$$

In a practical implementation, the start-of-sentence symbols <*s*> are being added to the beginning of the sentence and a sentence-end symbol </*s*> is being added to the end of the sentence. Sentence probability can then be expressed as (4)

$$P(w_1,...,w_n) = P(w_1|<\text{S}>,<\text{S}>)\times P(w_2|<\text{S}>,w_1)\times...\times P(w_n|w_{n-2},w_{n-1})\times P(</\text{S}>|w_n,w_{n-1}) \tag{4}$$

which is very convenient, because all of the subterms are conditional probabilities with the same length of history.

Conditional probabilities $P(w_i|w_{i-2},w_{i-1})$ are estimated on the basis of *N-gram counts* collected from a text corpus. There are many different ways of constructing probability distribution given the *N-gram counts*.

In order to focus on the methods used, Section 3.3.1 describes the most straightforward way of probability estimation, Section 3.3.2 describes the general form of most smoothing algorithms, Sections 3.3.3 and 3.3.4 describe particular methods of *N-gram* probability discounting.

3.3.1 Maximum - likelihood estimation

The Maximum-likelihood estimation (MLE) for *N-gram LM* is defined by the equation (5)

$$P(w_i|w_{i-N+1},...,w_{i-1}) = \frac{C(w_{i-N+1},...,w_i)}{C(w_{i-N+1},...,w_{i-1})} \tag{5}$$

where $C(w_1,...,w_n)$ denotes the number of occurrences of a sequence of words $w_1,...,w_n$ in a training corpus. However, there is one serious drawback to this method that makes it virtually useless. It assigns all probability mass to the events that occurred in the training corpus. Assuming that a trigram language model is used, this implies that every sentence which contains at least one trigram unseen in the training corpus will be assigned zero probability according to the MLE language model (there will be at least one zero element in the multiplication chain). More importantly, though, this simple method serves as a basis for other more elaborate estimation techniques.

3.3.2 General Form of LM Smoothing

Most of the discounting algorithms can be expressed by the equation 6

$$P(w_i|w_{i-n}...w_{i-1}) = \begin{cases} f(w_i|w_{i-n}...w_{i-1}) & \text{if } C(w_{i-n}...w_i) > 0 \\ \alpha(w_{i-n}...w_{i-1}) \times P(w_i|w_{i-n+1}...w_{i-1}) & \text{if } C(w_{i-n}...w_i) = 0 \end{cases} \tag{6}$$

where $f(w_i|w_{i-n}...w_{i-1})$ is a discounted probability estimation and $\alpha(w_{i-n}...w_{i-1})$ is a normalizing factor of *N-gram* $(w_{i-n}...w_{i-1})$.

If *N-gram* $(w_{i-n}...w_i)$ was seen in the training data, the result is equal to the discounted probability estimation f. f estimates lower values than the MLE, some probability mass is reserved for unseen events (when $C(w_{i-n}...w_i) = 0$).

If *N-gram* $(w_{i-n}...w_i)$ was not seen in the training data, the last word of *N-gram* is dropped and the estimation is based on the shorter history. This operation is called *backoff* and can be applied recursively. The shorter history estimation $P(w_i|w_{i-n+1}...w_{i-1})$ must be normalized, because there might be an unempty set $W_1 = \{w' : C(w_{i-n}...w_{i-1}, w') > 0\}$, which means that the probability mass that is to be distributed by the shorter history distribution comprises/accounts only for only $1 - \sum_{w' \in W_1} f(w'|w_{i-n}...w_{i-1})$. This normalization is ensured by the *back-of weight* term.

Let W denote the whole vocabulary, W_1 is defined as above and $W_0 = \{w' : C(w_{i-n}...w_{i-1}, w') = 0\}$. *back-of weight* is then derived as shown in (7)

$$
\begin{aligned}
1 &= \textstyle\sum_{w \in W} P(w|w_{i-n}...w_{i-1}) \\
1 &= \textstyle\sum_{w \in W_1} f(w|w_{i-n}...w_{i-1}) + \sum_{w \in W_0} (bow(w_{i-n}...w_{i-1}) \times P(w|w_{i-n+1}...w_{i-1})) \\
1 &= \textstyle\sum_{w \in W_1} f(w|w_{i-n}...w_{i-1}) + bow(w_{i-n}...w_{i-1}) \times \sum_{w \in W_0} P(w|w_{i-n+1}...w_{i-1}) \qquad (7)
\end{aligned}
$$

$$
bow(w_{i-n}...w_{i-1}) = \frac{1 - \sum_{w \in W_1} f(w|w_{i-n}...w_{i-1})}{\sum_{w \in W_0} P(w|w_{i-n+1}...w_{i-1})} = \frac{1 - \sum_{w \in W_1} f(w|w_{i-n}...w_{i-1})}{1 - \sum_{w \in W_1} P(w|w_{i-n+1}...w_{i-1})}
$$

In the case that f is estimated purely on the basis of the entire word history $(w_{i-n}...w_{i-1})$, the model is denoted as *backoff language model*.

In the case that f has the form of Equation 8,

$$
f(w_i|w_{i-n}...w_{i-1}) = g(w_i|w_{i-n}...w_{i-1}) + bow(w_{i-n}...w_{i-1}) \times P(w_i|w_{i-n+1}...w_{i-1}) \qquad (8)
$$

the model is denoted as *interpolated LM*, because the lower order models are always taken into consideration. In Equation 8, g is a probability estimation based solely on the basis of the entire word history $(w_{i-n}...w_{i-1})$.

3.3.3 Good-Turing Discounting Method

The intuitution of the Good-Turing algorithm is to estimate the probability of unseen events by using the probability of events that were seen once. The Good-Turing algorithm works with *frequency of frequencies* quantity N_c, the number of n-grams that occured c times. In Good-Turing Discounting counts of N-grams are discounted in such a way that an N-gram with the count c will be assigned a new count value c^* such that

$$c^* = (c+1)\frac{N_{c+1}}{N_c}$$

where N_0 is equal to the size of the training data.

Good-Turing n-gram probability estimation can be computed by using Equation 5, it is only needed to replace the original n-gram counts with modified c^* counts. However there is a practical problem. N_c values can be zero for greater values of c. This problem can be solved by defining

$$c^* = c \text{ FOR } c > k$$

where the suggested value of k is 5 (Katz, 1987).

The correct equation for c^* in this variant is

$$c^* = \frac{(c+1)\frac{N_{c+1}}{N_c} - c\frac{(k+1)N_{k+1}}{N_1}}{1 - \frac{(k+1)N_{k+1}}{N_1}}$$

as was proposed by (Katz, 1987).

3.3.4 Kneser-Ney Discounting Method

Kneser-Ney discounting (Kneser & Ney, 1995) method is based on the idea that backoff probability can be better estimated from counts of contexts in which the lower-order n-gram appears rather than from the standard n-gram counts. The Kneser-Ney estimation of a bigram probability can be expressed as

$$P_{KN}(w_i|w_{i-1}) = \begin{cases} \frac{C(w_{i-1},w_i)-D}{C(w_{i-1})} & \text{if } C(w_{i-1},w_i) > 0 \\ bow(w_{i-1})\frac{|\{w_{i-1}:C(w_{i-1},w_i)>0\}|}{\sum_{w_i}|\{w_{i-1}:C(w_{i-1},w_i)>0\}|} & \text{otherwise.} \end{cases}$$

for a backoff language model and as

$$P_{KN}(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i) - D}{C(w_{i-1})} + bow(w_{i-1}) \frac{|\{w_{i-1} : C(w_{i-1}, w_i) > 0\}|}{\sum_{w_i} |\{w_{i-1} : C(w_{i-1}, w_i) > 0\}|}$$

for an interpolated language model. D is a properly chosen discounting constant, *bow* denotes the backoff weight[4].

In the variant of Kneser-Ney called *modified Kneser-Ney* (Chen & Goodman, 1998), there are multiple discounting constants D_1, D_2, D_{3+}.

$$Y = \frac{n_1}{(n_1 + 2 \times n_2)}$$

$$D_1 = 1 - 2 \times Y \frac{n_2}{n_1}$$

$$D_2 = 2 - 3 \times Y \frac{n_3}{n_2}$$

$$D_{3+} = 3 - 4 \times Y \frac{n_4}{n_3}$$

where n_i denotes the number of N-grams with count i. D_1 is used for the discounting of unigrams, D_2 is used for the discounting of bigrams and D_{3+} is used for the discounting of the higher order N-grams.

There are empirical studies (Zhai & Lafferty, 2004) showing that modified Kneser-Ney outperforms other smoothing techniques such as Good-Turing or Witten-Bell (Witten & Bell, 1991).

3.3.5 Language Models as the Components of Log-Linear Model, Word Emission Models

Language models are essential for establishing many feature functions of the log-linear model that was proposed for spelling correction in Section 3.2. $P(w_1...w_n)$ in definition of f_f is estimated by language model based on word forms, $P(l_1...l_n)$ and $P(t_1...t_n)$ in the definitions of f_l and f_t are estimated by language models on lemmas and morphological tags. Using the definitions

$$\hat{f}_f\big((\hat{w}_{i-2}, \hat{w}_{i-1}) \to \hat{w}_i\big) = \log\big(P(\hat{w}_i|\hat{w}_{i-2}, \hat{w}_{i-1})\big)$$

$$\hat{f}_l\big((\hat{l}_{i-2}, \hat{l}_{i-1}) \to (\hat{l}_i, \hat{w}_i)\big) = \log\big(P(\hat{l}_i|\hat{l}_{i-2}, \hat{l}_{i-1})\big) + \log\big(P(\hat{w}_i|\hat{l}_i)\big)$$

[4]There is an error in (Jurafsky & Martin, 2008) in Equations 4.50 and 4.51. There should be $\alpha(w_{i-1})$ and $\beta(w_{i-1})$ instead of $\alpha(w_i)$ and $\beta(w_i)$, otherwise the equations do not conform to the general back-off/interpolated language model definitions.

$$\hat{f}_t\big((\hat{t}_{i-2},\hat{t}_{i-1})\rightarrow(\hat{t}_i,\hat{w}_i)\big)=\log\big(P(\hat{t}_i|\hat{t}_{i-2},\hat{t}_{i-1})\big)+\log\big(P(\hat{w}_i|\hat{t}_i)\big)$$

the feature functions f_f, f_l and f_t can be expressed as

$$\begin{aligned}
f_f(\hat{w}_1...\hat{w}_n) &= \log\big(P(\hat{w}_1...\hat{w}_n)\big)\\
&= \log\big(\prod_{i=1}^{n+1}P(\hat{w}_i|\hat{w}_{i-2},\hat{w}_{i-1})\big)\\
&= \sum_{i=1}^{n+1}\log\big(P(\hat{w}_i|\hat{w}_{i-2},\hat{w}_{i-1})\big)\\
&= \sum_{i=1}^{n+1}\hat{f}_f\big((\hat{w}_{i-2},\hat{w}_{i-1})\rightarrow\hat{w}_i\big)
\end{aligned}\tag{9}$$

$$\begin{aligned}
f_l(\hat{l}_1...\hat{l}_n,\hat{w}_1...\hat{w}_n) &= \log\big(P(\hat{l}_1...\hat{l}_n)\big)+\log\big(P(\hat{w}_1...\hat{w}_n|\hat{l}_1...\hat{l}_n)\big)\\
&= \log\big(\prod_{i=1}^{n+1}P(\hat{l}_i|\hat{l}_{i-2},\hat{l}_{i-1})\big)+\log\big(\prod_{i=1}^{n+1}P(\hat{w}_i|\hat{l}_i)\big)\\
&= \sum_{i=1}^{n+1}\log\big(P(\hat{l}_i|\hat{l}_{i-2},\hat{l}_{i-1})\big)+\log\big(P(\hat{w}_i|\hat{l}_i)\big)\\
&= \sum_{i=1}^{n+1}\hat{f}_l\big((\hat{l}_{i-2},\hat{l}_{i-1})\rightarrow(\hat{l}_i,\hat{w}_i)\big)
\end{aligned}\tag{10}$$

$$\begin{aligned}
f_t(\hat{t}_1...\hat{t}_n,\hat{w}_1...\hat{w}_n) &= \log\big(P(\hat{t}_1...\hat{t}_n)\big)+\log\big(P(\hat{w}_1...\hat{w}_n|\hat{t}_1...\hat{t}_n)\big)\\
&= \log\big(\prod_{i=1}^{n+1}P(\hat{t}_i|\hat{t}_{i-2},\hat{t}_{i-1})\big)+\log\big(\prod_{i=1}^{n+1}P(\hat{w}_i|\hat{t}_i)\big)\\
&= \sum_{i=1}^{n+1}\log\big(P(\hat{t}_i|\hat{t}_{i-2},\hat{t}_{i-1})\big)+\log\big(P(\hat{w}_i|\hat{t}_i)\big)\\
&= \sum_{i=1}^{n+1}\hat{f}_t\big((\hat{t}_{i-2},\hat{t}_{i-1})\rightarrow(\hat{t}_i,\hat{w}_i)\big)
\end{aligned}\tag{11}$$

In Equations 10 and 11, the following simplifying assumptions are made

$$P(w_1...w_n|l_1...l_n)=\prod_{i=1}^{n}P(w_i|l_i)\tag{12}$$

$$P(w_1...w_n|t_1...t_n)=\prod_{i=1}^{n}P(w_i|t_i)\tag{13}$$

The expressions $P(w_i|l_i)$ and $P(w_i|t_i)$ will be denoted as word emission probabilites and the

models that estimates these probabilities as word emission models. They are estimated as

$$P(w_i|l_i) = \frac{C(t_i, w_i)}{C(t_i)} \tag{14}$$

$$P(w_i|t_i) = \frac{C(l_i, w_i)}{C(l_i)} \tag{15}$$

where the counts $C(t_i, w_i)$ and $C(l_i, w_i)$ can be smoothed by adding a distinct small value to each word-lemma or word-tag pair with zero count.

3.4 Error Model

The purpose of an *error model* is to estimate the probability $P(W_t|W_s)$ that the person typed W_t having W_s in his/her mind. Obviously, $P(black|black)$ should be very high as most of the time, people type exactly the word they intend to. $P(back|black)$ should be much smaller, but relatively high, because *back* differs from *black* only by one letter. On the other hand, $P(democracy|black)$ should be extremely low.

As observed by (Damerau, 1964) and (Peterson, 1986) most of the spelling errors are caused by one of the following reasons:

1. Single letter insertion (black → bvlack)
2. Single letter deletion (black → back)
3. Swapping of two adjacent letters (black → balck)
4. Single letter replacement (black → nlack)

This fact was directly used by (Mays et al., 1991) in their error model implementation. When P(w—s) is requested, a confusion set CS which contains s together with all dictionary words that can be obtained from s by exactly one edit operation is constructed. The error model probability $P(s|w)$ is then defined as

$$P(s|w) = \begin{cases} 0 & w \notin CS \\ \alpha & w = s \\ \frac{1-\alpha}{|CS|-1} & w \in CS \ \& \ w \neq s \end{cases}$$

where α denotes the a priori probability that a word was typed correctly. This error model assigns the highest degree of probability to the word that was typed and distributes the rest along the confusion set uniformly.

In (Church & Gale, 1991), a more advanced error model is suggested. Similarly to (Mays et al., 1991), they consider only candidate words obtained by one edit operation. In their approach, however, each edit operation has its distinct probability, i.e. the probability of the letter substitution $s \rightarrow d$ may differ from the probability of $e \rightarrow a$. They also context-condition the probabilities of a letter insertion and deletion. These probabilities were trained on a large error corpus.

A more general model is proposed in (Brill & Moore, 2000). Their model allows an arbitrary edit operation of the form $\alpha \rightarrow \beta$, where α and β can be arbitrary letter sequences. As in the previous model, each edit operation has its probability score $P(\alpha \rightarrow \beta|PSN)$ assigned, where PSN is a positional feature with a following set of values $\{S = \textit{start of word, } M = \textit{middle of word, } E = \textit{end of word}\}$.

$P(s|w)$ is defined as

$$\begin{aligned} P(s|w) &= \max_{R \in Part(w), T \in Part(s)} P(R|w) \prod_{i=0}^{|R|} P(R_i \rightarrow T_i|PSN(i)) \\ &\simeq \max_{R \in Part(w), T \in Part(s)} \prod_{i=0}^{|R|} P(R_i \rightarrow T_i|PSN(i)) \end{aligned}$$

where $Part(w)$ denotes a set of all possible partitions of w into substrings.

In the above-cited paper, the following example is shown: $P(fisikle|physical)$ is to be evaluated. Considering that the optimal word partitions are $T = f - i - s - i - c - le$ and $R = ph - y - s - i - c - al$, the error model probability is computed as

$$\begin{aligned} P(fisikle|physical) &= P(ph \rightarrow f|S) \times P(y \rightarrow i|M) \times P(s \rightarrow s|M) \\ &\quad \times P(i \rightarrow i|M) \times P(c \rightarrow k|M) \times P(al \rightarrow le|E) \end{aligned}$$

An error model of such generalness can be particularly useful for languages with complicated relations between phonetic and orthographic representations of words. A typicle example of such language is English. In English it is often possible to come up with multiple letter sequences corresponding to a single sequence of phonemes such as *shell, shall; pea, pee.* Due to this fact, many spelling errors in English texts are caused by the writer's unfamiliarity with the correct

word form. On the other hand, there are also languages with relatively straightforward phonology ↔ orthography mapping, such as Czech language. Just by knowing a word phonetically, a Czech native speaker can figure out the correct written form with high confidence. The only problematic issues arise when choosing between i/y, these letters are pronounced identically in modern Czech. Furthermore, it is difficult to choose the right variant from a voiced/surd consonant pair and there is a limited number of other special phenomena like *bě* ↔ *bje*, *vě* ↔ *vje*, *mě* ↔ *mně* etc. that may lead to mistakes.

The error model that was used in this work resembles the error model of (Church & Gale, 1991). The four basic edit operations are distinguished - insertion, deletion, substitution and swap. Probabilities of letter deletions are conditioned on the context. The detailed classification of error types and their probabilities will be given in Section 4.4.

In the provided error model implementation, the $P(word_1|word_2)$ that $word_1$ was misspelled to $word_2$ is equal to the product of probabilities of all edit operations that are needed in order to obtain $word_1$ from $word_2$.

3.5 HMM model

So far, the components of statistical model have been introduced. However, in order to identify the optimal correction of a given sentence, the strategy of searching for the optimal hypothesis is needed. As stated above, spelling correction problem can be defined in terms of the noisy channel or log-linear model. A convenient way of implementation of these models is the *Hidden markov model (HMM)*.[5]

HMM provides a model of sequence generation.[6] It is a finite-state automaton consisting of a set of states S, set of transitions T and output alphabet $\sum$. The states are capable of emitting symbols from $\sum$ with a certain probability. The probabilities of all symbol emissions sum up to 1 for each state. The transitions are directed edges connecting the pairs of states, each transition has a probability value assigned and the sum of probabilities of all transitions leaving particular state is equal to 1. Sequence generation process of HMM proceeds as follows:

1. Start at the single start state.

[5] There are many variants of HMMs - HMMs with/without empty transition, HMMs emitting output symbols by states/transitions etc. Only the variant that is suitable for solving the spell checking problem will be described.

[6] But it is never directly used in this way. Common usage of HMM will be addressed later in this section after explaining the basic HMM concepts.

2. Choose randomly a transition leaving the current state. During choosing, respect the distribution of the transition probabilities.

3. Follow the chosen transition and traverse to a next state.

4. Emit randomly one output symbol with respect to the emission probabilities in the current state.

5. If the final state is reached finish the sequence generation process. Otherwise repeat from Step 2.

In the spelling correction scenario, an HMM capable of generating sentences possibly containing misspelled words can be utilized. Such a model directly incorporates the already described components - Language models, word emission models and error model.

In the remaining part of this section, the multi-factor second order HMM will be described. The states of the HMM correspond to the bigrams of triplets *wf = (w, l, t)*[7] and they can be denoted as (wf_{hist}, wf_{act}). Transitions are defined for all pair of states of the form $(wf_1, wf_2) \rightarrow (wf_2, wf_3)$ and the transition costs can be expressed, in conformity with the suggested log-linear model for the multi-factor spell-checking, as

$$\begin{aligned} f\big((wf_1, wf_2) \rightarrow (wf_2, wf_3)\big) &= \alpha_f \times \log P(w_3|w_1, w_2) \\ &\quad \alpha_l \times \log\big(P\big(l_3|l_1, l_2\big)P(w_3|l_3)\big) \\ &\quad \alpha_t \times \log\big(P(t_3|t_1, t_2)P(w_3|t_3)\big) \\ &= \alpha_f \times f_f\big((w_1, w_2) \rightarrow w_3\big) \\ &\quad \alpha_l \times f_l\big((l_1, l_2) \rightarrow (l_3, w_3)\big) \\ &\quad \alpha_t \times f_t\big((t_1, t_2) \rightarrow (t_3, w_3)\big) \end{aligned}$$

where t_1 stands for the tag coming from wf_1 triplet etc.

Emission costs are defined as

$$g\big(w_2^{'}|(wf_1, wf_2)\big) = \alpha_e \log\big(P(w_2^{'}|w_2)\big)$$

where the probability $P\big(w_2^{'}|form_2\big)$ is estimated by the error model and $w_i^{'}$ denotes the word at position i in the original sentence.

[7] wf is used as a abbreviation of *word factors*, w denotes word, l denotes lemma and t denotes morphological tag

The process of sentence decoding can be formulated in the following way: When given a sequence of words possibly containing typos, identify the most probable sequence of underlying *word factor triplets (wf)*. This can be formally expressed as

$$(wf_1^*...wf_n^*) = \underset{wf_1,...wf_n}{\operatorname{argmax}} \sum_{i=1}^{n+1} \Big(f\big((wf_{i-2}, wf_{i-1}) \rightarrow (wf_{i-1}, wf_i)\big) + g\big(w_i^{'}|(wf_{i-1}, wf_i)\big) \Big) \tag{16}$$

where $(wf_1^*...wf_n^*)$ is the optimal word factor triplet sequence, wf_0 = (<s>, <s>, <s>), wf_{-1} = (<s>, <s>, <s>) and $wf_{n+1} = (</s>, </s>, </s>)$.

Given that states correspond to word bigrams, it would be needed to examine $|D|^{2^n}$ possibilities, if the hypothesis space was searched exhaustively. Fortunately, there is an efficient dynamic algorithm that finds the optimal sequence of the underlying states.

3.6 Decoding - Viterbi algorithm

The Viterbi Algorithm will be described on the basis of description found in (Jurafsky & Martin, 2008)[8]. The formulas and notation will be adjusted in order to conform to the expressions introduced in Section 3.5, however for simplicity, the states will be denoted by a single letter (the inner structure of states - tuplets of triplets $wf = (w, l, t)$ will not be indicated anymore).

The sequence decoding task can be defined as the task of finding most probable sequence of states $Q = q_1, q_2, q_3, ...q_T$ of the given HMM when a sequence of observations $S = w_1^{'}, w_2^{'}, w_3^{'}, ..., w_T^{'}$ is given.

During the decoding, *Viterbi trellis* matrix is filled from left from to right as the input sequence is being processed. Each cell of the trellis $v_t(j)$, corresponds to the probability[9] that HMM is in state j after seeing the first t observations and passing through the most probable state sequence $q_0, q_1, ..., q_{t-1}$.

$$v_t(j) = \max_{q_0,q_1,...,q_{t-1}} \log P(q_0...q_{t-1}, w_1^{'}...w_t^{'}, q_t = j)$$

At the initialization step, the values v_0 are set to

$$v_0(0) = 0$$
$$v_0(j) = +inf \quad j = 1...N$$

[8] Few formulations are even written exactly as they were found in (Jurafsky & Martin, 2008)

[9] In order to conform with the definitions introduced in Section 3.5, $v_t(j)$ expresses a cost function, not a probability function as in (Jurafsky & Martin, 2008).

where 0 denotes the sentence start state and the rest of HMM states is denoted by numbers $1...N$.

Viterbi fills the cells recursively, at the time of computation of $v_t(j)$ all cells of the form $v_{t-1}(i)$ have been already computed and the values $v_t(j)$ for $j = 1...N$ are computed as

$$v_t(j) = \max_{t=1}^{N}\big(v_{t-1}(i) + f(i \to j) + g(w_t'|j)\big)$$

In order to reconstruct the best state sequence from the trellis matrix, back pointers bt are needed. They are computed as

$$bt_t(j) = \operatorname*{argmax}_{t=1}^{N}\big(v_{t-1}(i) + f(i \to j) + g(w_t'|j)\big)$$

The reconstruction of best state sequence starts with identifying the state *final* such that $v_{T+1}(\textit{final})$ is maximal[10]. The last state can be obtained as

$$q_T = bt_{T+1}(\textit{final})$$

The states $q_1...q_{T-1}$ are obtained recursively.

$$q_t = bt_{t+1}(q_{t+1})$$

[10]There are $T + 1$ columns in the trellis matrix and the output symbol at time $T + 1$ is the end of sentence symbol `</s>`

4 Data sources and Models Training

4.1 Lexicon

A high coverage lexicon is the key component of each spelling correction system. Open Source Aspell lexicon of Czech language was used as the basis for lexicon creation. Aspell contains around 3 000 000 word forms. Out of this number, 170 582 word forms are names. The number of distinct lemmas found in the dictionary is much smaller, it is only about 250 000. The Aspell lexicon provides quite a good coverage on common words, but its coverage on names could be further extended by providing a name lexicon. In order to further improve the lexicon coverage, words that appeared more than 5 times in the training corpus[11] were added into the lexicon.

Morphological analysis of all words in the lexicon was made using the morphology analyser created at ÚFAL (Hajič, 2004). The result of lexical morphological analysis of a word is a list of all `(lemma, morphological tag)` pairs that represent possible morphological interpretations of a given word. As a result of these analyses, a morphological lexicon was created[12]. Words that were unknown to the morphological analyser were filtered out. This prevents common spelling errors from becoming part of the spell-checker lexicon. After having added frequent words from the corpus and having filtered out the words unknown to the morphological analyser, the final size of the lexicon became about 2 800 000 words.

4.2 Text Corpus

A large scale text corpus was needed in order to produce well-estimated language models and word emmision models. This need was met by *Czech Web Document Collection*[13] (WebColl) by Pavel Pecina, which is a collection of 111 milion words contained in 223 000 articles downloaded from news servers and on-line archives of Czech newspapers.

4.2.1 Corpus Cleaning

When a text corpus is being built out of the documents found on the web, it is important to filter out irrelevant parts of the web pages' content - HTML tags and small stubs of text that are not suitable for further linguistic processing. From this aspect, the data provided by WebColl were of a high quality. However there was still a content that needed to be filtered out, such

[11] The text corpus will be described later in this chapter
[12] The implementation of the morphology lexicon is described in Section 5.3
[13] This corpus is not available for public, it is designated for the research purposes at UFAL only.

as chunks of text written in forreign languages (mostly English and Slovak) and Czech texts written without diacritics. The following procedure was used in order to filter out these bad data: A score was computed for each sentence of the corpus. Each sentence received +1 point for each word that was found in the Czech lexicon, -2 points for each word not found in Czech lexicon, which was found in a Slovak, English or German lexicon and -1 point for each word that was not found in any lexicon. The sentences with a negative score were filtered out. By this procedure, 146 000 sentences out of 7 002 000 were filtered out. Most of them contained either erroneous or foreign-language text.

There were also a lot of duplicities in the data. By a thorough analysis of the corpus, it was revealed that out of all 7 million sentences, there were only about 3.5 million unique sentences. This means that on average, each sentence was included twice in the corpus. Of course some duplicities are natural, because certain very simple sentences are likely to occur in different articles independently or one article may be quoted in another one. However, a large amount of duplicities in the corpus was unnatural, it was most likely caused by multiple downloads of various documents that happenned during web crawling[14]. This corpus feature had not been revealed until the testing of the diacritics completion feature. The results achieved on the testing data coming from the corpus were to good to be true (around 99,3%). Then, an extensive overlap between the training and the testing data was shown.

4.2.2 Corpus Splitting

The cleaned corpus was splitted into three subparts: training data, heldout data and testing data. The training data were created out of 98% of all corpus data, heldout data and testing data were both created out of 1% of the data. In NLP, it is always important not to evaluate system performance on the data that were used during the training of a system. The reason is that results obtained on training data are usually significantly better than results obtained on data that was not used during the training.

The training data were used for the training of the language models and emmision models and also as a source for lexicon enrichment (the most frequent words from the corpus were included into the dictionary if not yet present there). Heldout data were used for finding of optimal weights of distinct features of the log-linear model. The testing data were used for evaluation

[14]Web crawling is a process of automatic web content downloading carried out by a computer program

```
když|když|J, syna|syn|NNMS4 s|s|RR7 vypětím|vypětí|NNNS7 všech|všechen|PL-P2
sil|síla|NNFP2 vypravila|vypravit|VpQW-RA ,|,|, úlevně|úlevně|Dg si|se|P7--3
oddychla|oddychnout|VpQW-RA a|a|J večer|večer|Db spokojeně|spokojeně|Dg
usnula|usnout|VpQW-RA .|.|.
```

Figure 1: Result of lemmatization and morphological tagging of sentence *Když syna s vypětím všech sil vypravila, úlevně si oddychla a večer spokojeně usnula.*

of system performance.[15]

4.2.3 Training Data Processing

Lemmatization and morphological tagging were the first steps in the training data processing. In this way, the most probable lemma and morphological tag was assigned to every token in the training corpus. The result of this lemmatization and tagging process was stored in a format shown in Figure 1. There is a morphological triplet `form|lemma|tag` for each word. As a next step, the morphological triplets of words that were not found in the dictionary were substituted by `<name>|<name>|<name>` if the word was capitalized or by `<unk>|<unk>|<unk>` in other cases[16]. Each number expression was substituted by `<number>|<number>|<number>`.

4.3 Language Models and Emmision Models Training

The data format that was used for the language model and error model training is shown in Figure 1. By using the training corpus, n-gram counts for each morphological factor and counts of `form-lemma` and `form-tag` combinations were collected. For the word forms and lemmas, n-grams up to order 3 were collected. For morphological tags, 4-grams were collected as well. These n-grams were used for estimation of the language models as described in Section 3.3. It has been proved empirically that the modified Kneser-Ney smoothing provides better results than other smoothing methods such as Witten-Bell or Good Turing (Zhai & Lafferty, 2004). Tables 1, 2 and 2 show the perplexities achived on held-out data for distinct smoothing techniques. As expected, the interpolated Kneser-Ney smoothing achieved the best results on word forms and lemmas. However, the other smoothing techniques achieved better results on

[15] However results of the evaluation made on the same corpus that was used for training (but training and testing data were not overlapping) can be significantly better than results obtained during the real usage of the system. The reason is that the language of the training and the testing data is very similar, as they both come from the same corpus. Because of this, evaluation was made on other data sets containing data from different domains as well.

[16] In the data format used in this step, first words of a sentence do not start with capital letters unless it is a name

Smoothing Method	Perplexity
Good-Turing, interpolated	348.5
Good-Turing, Katz backoff	348.5
Witten-Bell, interpolated	355.7
Witten-Bell, Katz backoff	350
modified Kneser-Ney, interpolated	322.7
modified Kneser-Ney, Katz backoff	332.3

Table 1: Perplexities of the language models on word forms

morphological tags. This may have been caused by the fact that there are only about 1000 distinct morphological tags in the training corpus, the language model vocabulary thus being much smaller in this case. A possible explanation is that in the case of an extremely small language model vocabulary, it is better to estimate lower order probabilities on the basis of regular counts. To examine this hypothesis, it would be helpful to compare the performance of language models on tags on testing data coming from different domains. It is possible that Kneser-New could prove a better solutione when the language used in the training and testing data differs significantly.

Experiments on testing data coming from a different domain were made for language models on word forms. The results are outlined in Table 4. The results show a clear superiority of Kneser-Ney in such a case, while on the heldout data coming from the same corpus all methods performed almost equally well.

As already mentioned in Section 4.2.1, the web corpus used for the language model training contained a very high number of duplicities. This fact went unnoticed for a long time and a lot of experiments were made on the held-out and testing data that overlapped with training data significantly. In this case, Kneser Ney performed even much worse than Good Turing and Witten-Bell (Witten & Bell, 1991).

To estimate the language models on word forms and lemmas, an interpolated version of modified Kneser-Ney smoothing was used, to estimate the language model on morphological tags, an interpolated Witten-Bell was chosen.

The emission models were estimated on the basis of counts of the *form-lemma* and *form-tag* combination found in the training data. In order to smooth these models, the count of each *form-lemma* and *form-tag* that was valid according to the morphological lexicon was increased by +1. Emission models were estimated according to Equations 14 and 15 in Section 3.3.

Smoothing Method	Perplexity
Good-Turing, interpolated	192.2
Good-Turing, Katz backoff	192.2
Witten-Bell, interpolated	195.3
Witten-Bell, Katz backoff	192.7
modified Kneser-Ney, interpolated	182.4
modified Kneser-Ney, Katz backoff	187.4

Table 2: Perplexities of the language models on lemmas

Smoothing Method	Perplexity
Good-Turing, interpolated	24.2
Good-Turing, Katz backoff	24.2
Witten-Bell, interpolated	20.6
Witten-Bell, Katz backoff	20.8
modified Kneser-Ney, interpolated	20.6
modified Kneser-Ney, Katz backoff	21.1

Table 3: Perplexities of the language models on morphological tags

Smoothing Method	Perplexity
Good-Turing, interpolated	1256
Good-Turing, Katz backoff	1256
Witten-Bell, interpolated	1350
Witten-Bell, Katz backoff	1223
modified Kneser-Ney, interpolated	1165
modified Kneser-Ney, Katz backoff	1142

Table 4: Perplexities of the language models on word forms - testing on data from different domain (Lion Feuchtwanger: Foxes in the Vineyard)

4.4 Error Corpus and Error Models

The probabilities of distinct error types can be best estimated from the spelling error corpus. For the Czech language, such corpus is available (Pala, Rychlý, & Smrž, 2003). This corpus contains about 2000 spelling errors; other types of errors such as morpho-syntactical errors, stylistical errors and punctuation errors are anotated as well. Nevertheless, the existence of this corpus was not known until the later stage of the system evaluation and this error corpus has not been utilized so far.

Another option is to obtain spelling error instances automatically from a big text corpus. Such method was proposed by (Church & Gale, 1991). They considered each word that does not appear in the dictionary and is not further than one edit operation from a word included in the dictionary as spelling error. They built their error corpus out of such words. First, they set probabilities of all edit operations uniformly. Later on, they iteratively spell-checked their error corpus, found the best correction for each word and updated edit probabilities according to the proposed *error* → *suggestion* pairs.

This method of finding spelling errors was tried out on the WebColl corpus. However this method turned out to be useless. The reason was that the vast majority of words identified as spelling errors were correct words or they were colloquial word forms.

The next idea was to build an error corpus manually. A small scale error corpus was created during this work. It contains 9500 words, 570 of them are spelling errors. The corpus was constructed by making transcriptions of an audio version of a Czech novel by Jaroslav Hašek: *Osudy dobrého vojáka Švejka*.[17] There was no postcorrection made on the transcribed text and the spelling error rate in the resulting text is relatively high. Nevertheless, this error corpus has two drawbacks: Its size is limited and all data were typed by a single person. Because of this fact, the error model constructed on the basis of these data can be biased (the error distribution function of the training person is more or less different from the error distribution function of the "average" person). On the basis of the errors found in this error corpus, a classification of spelling error types was made. The error classification and the counts of each error type are shown in Table 5. Four basic error types can be distinguished (insertion, deletion, substitution and swap) and the insertions depend on context. This is in conformity with the error model definition used by (Church & Gale, 1991), although the error model described here

[17] The audio extracts can be downloaded for free from the web-sites of Český rozhlas: http://www.rozhlas.cz/ctenarskydenik

Error Type	Occurence Count
Substitution - horizontally adjacent letters	142
Substitution - vertically adjacent letters	2
Substitution - $z \rightarrow s$	6
Substitution - $s \rightarrow z$	1
Substitution - $y \rightarrow i$	10
Substitution - $i \rightarrow y$	10
Substitution - non-adjacent vocals	13
Substitution - diacritic confusion	21
Substitution - other cases	19
Insertion - horizontally adjacent letter	162
Insertion - vertically adjacent letter	13
Insertion - same letter as previous	14
Insertion - other cases	46
Deletion - other cases	58
Swap letters	34

Table 5: Error Counts - Manually created corpus

is much less detailed. However, given the characteristics of Czech language, an error model of such generalness as described in (Church & Gale, 1991) would not necessarily result in a significant performance gain.[18] On the basis of error counts, error probabilities $P(a \rightarrow b)$ that letter sequence b was intended when letter sequence a was written were estimated. The way these values were obtained will be demonstrated on the error type $ABC \rightarrow AC \mid$ *other cases*. In the error corpus there were 46 errors caused by insertion of letter non-adjacent to any of its neighbours. The error corpus contained 49547 letters that were non-adjacent to their neighbouring letters. So the probability that the letter was inserted accidentally was computed as $P = 46/49547$. Probability estimations of each error type are shown in Table 6.

After the spellchecker had been completed, the idea how to modify the method of authomatic error corpus creation (Church & Gale, 1991) was devised. The modified version builds an error corpus out of the words recognized by the spellchecker as spelling errors, however there must be a significant evidence that the proposed correction is right, otherwise the spelling error is not added to the error corpus. To be more specific, both bigrams (w_{i-1}, s) and (w_{i+1}, s), where w_{i-1} is the predecessor of misspelled word e, w_{i+1} is the successive word and s is the correction suggestion, must be present in the language model, otherwise the error-correction pair $e \rightarrow s$ is not included in the error corpus. Recall of this method is rather small, but the precission is quite satisfactory and most of the recognized error-correction pairs were correct. This method

[18] This issue is discussed in the end of Section 3.4.

Correction	$-\log_{10}$(Probability
$\lambda \rightarrow A$	4.435
$ABC \rightarrow AC \mid \text{hadj}(B, A) \vee \text{hadj}(B, C)$	1.28
$AA \rightarrow A$	1.081
$ABC \rightarrow AC \mid \text{vadj}(B, A) \vee \text{vadj}(B, C)$	2.375
$ABC \rightarrow AC \mid$ *other cases*	3.032
$A \rightarrow$ Ă, $A \rightarrow$ Á	2.891
Ă $\rightarrow A$, Á $\rightarrow A$	3.308
$A \rightarrow B \mid \text{hadj}(A, B)$	2.8
$A \rightarrow B \mid \text{vadj}(A, B)$	4.655
$A \rightarrow B \mid \text{vocal}(A) \wedge \text{vocal}(B)$	3.734
$i \rightarrow y$	2.454
$y \rightarrow i$	2.097
$s \rightarrow z$	3.291
$z \rightarrow s$	2.129
$A \rightarrow B \mid$ *other cases*	4.898
$AB \rightarrow BA$	3.123

Table 6: Error Model estimated on the basis of manually created error corpus - capital letters denote variables - they can be substituted for any letter, lower case letters stand for the letter they represent (and also for its uppercase variant)

identified 12761 words out of 111 000 000 words in the cleaned WebColl as spelling errors, the classification of these errors is shown in Table 7 and the corresponding probabilities in Table 8. Both error models (the first one obtained from a manually created error corpus, the second one obtained by automatic processing of WebColl) were used during the evaluation.

.

Error Type	Occurence Count
Substitution - horizontally adjacent letters	630
Substitution - vertically adjacent letters	103
Substitution - $z \to s$	30
Substitution - $s \to z$	86
Substitution - $y \to i$	57
Substitution - $i \to y$	54
Substitution - non-adjacent vocals	325
Substitution - diacritic confusion	4473
Substitution - other cases	1845
Insertion - horizontally adjacent letter	380
Insertion - vertically adjacent letter	155
Insertion - same letter as previous	212
Insertion - other cases	1226
Deletion - other cases	2637
Swap letters	548

Table 7: Error Counts - Automatically obtained error instances

Correction	$-\log_{10}(\textsc{Probability}$
$\lambda \to A$	4.14
$ABC \to AC \mid \text{hadj}(B, A) \vee \text{hadj}(B, C)$	2.29
$AA \to A$	1.227
$ABC \to AC \mid \text{vadj}(B, A) \vee \text{vadj}(B, C)$	2.661
$ABC \to AC \mid$ *other cases*	2.975
$A \to \check{A}, A \to \acute{A}$	2.250
$\check{A} \to A, \acute{A} \to A$	2.235
$A \to B \mid \text{hadj}(A, B)$	3.519
$A \to B \mid \text{vadj}(A, B)$	4.305
$A \to B \mid \text{vocal}(A) \wedge \text{vocal}(B)$	3.706
$i \to y$	3.167
$y \to i$	2.679
$s \to z$	2.747
$z \to s$	2.854
$A \to B \mid$ *other cases*	4.285
$AB \to BA$	3.278

Table 8: Error Model estimated on the basis of automatically obtained error instances - capital letters denote variables - they can be substituted for any letter, lower case letters stand for the letter they represent (and also for its uppercase variant)

5 Implementation

Spell-checking usually runs as a background process inside a text editor. It shouldn't make computer run slowly and it should also consume only a little amouth of memory, because from the user's point of view it is only a low priority task. In other words, spellchecker should be invisible and only correct typos when it's appropriate. Consequently, a good spellchecker should be implemented maximally efficiently in terms of both memory and CPU time consumption. When context sensitive spell-checking feature was implemented for Microsoft Office 2007, the momory consumption question was stressed out and the sophisticated method of loosy language models representation (HashTBO) was applied (Church et al., 2007). Following sections contain a describtion of individual parts of the system and ideas behind their implementation.

5.1 Auxiliary Data Structures

During the implementation, the need for certain non-standard data structures arose and, followingly, these data structures were implemented. These data structures are based on my own ideas, nevertheless, they are all quite simple and were certainly already discovered[19]

5.1.1 Memory Efficient Static Array

During the implementation, one particular data structure was needed in many contexts - a static array of non-negative integer values. Obviously, there are standard data structures already available for such case. In the standard implementations, one can easily store 16 bit, 32 bit or 64 bit integer types. However, Since low memory consumption is viewed as a priority in the current task, it might be desirable to store for example 10 bit or 34 bit integer value.

Such static array class was implemented in this work and it was called PackedArray. Instances of PackedArray can be initialized from the standard integer arrays or by loading from a binary file. In the prior case, the number of bits needed per element is computed as $bits = \lceil \log_2(maximal\ value + 1) \rceil$. The elements of the original integer array are stored in a raw bit array using *bits* bits per each value. For each static array, this conversion operation is done only once during the data preparation phase. The static arrays are always instantiated from the binary file in the release version of the spell-checker. The initialization from the binary

[19]The so called "Reinventing the wheel approach" is generally considered a bad programming practice. The personal view of the author of this thesis is, though, that a certain degree of reinventing the basics makes programming much more delightful and should be tolerated.

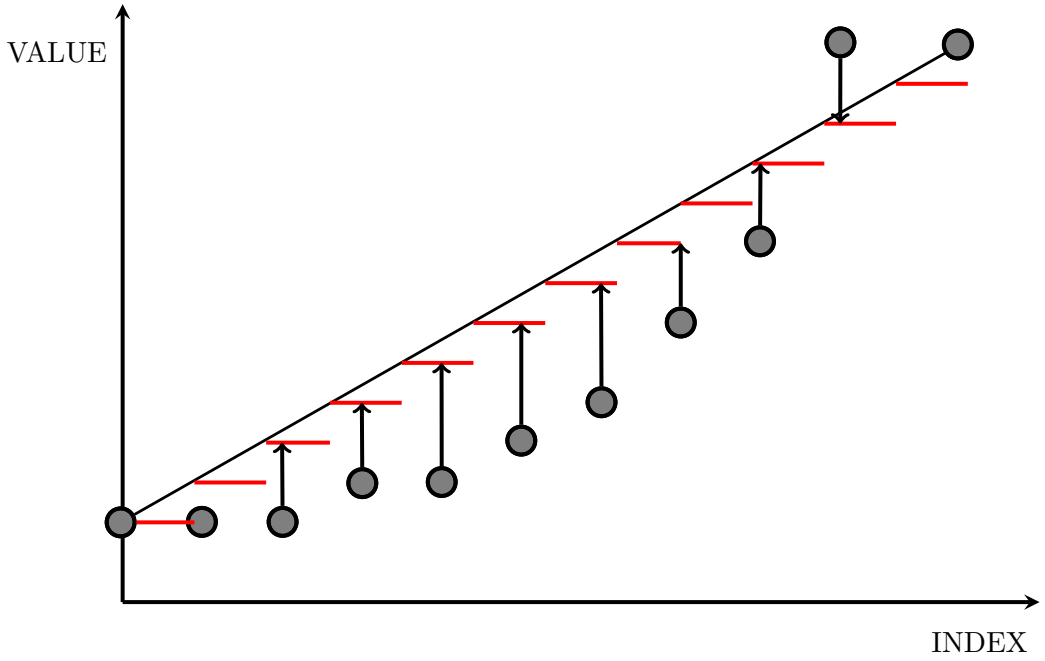


Figure 2: Non-decreasing array illustration. Only differences between array values and floored linear function values are stored.

file is extremely efficient, because it is done just by a single call of C routine for copying of continuous memory segments.

5.1.2 Memory Efficient Static Array of Non-Decreasing Values

A special implementation of non-decreasing integral arrays such as $(0, 3, 3, 6, 8, 8, 15)$ was provided for memory-efficient representation of such arrays. The implementation of this data structure is based on the idea that the stored values can be approximated by a linear function. Let $V(i)$ denote the i-th value that should be stored, $f(i)$ is the value of the approximating linear function f in point i. For each value $V(i)$, only the offset value $\mathit{offset}(i) = V(i) - \lfloor f(i) \rfloor$ will be stored in the data structure. The values $V(i)$ can be reconstructed using the formula $V(i) = \lfloor f(i) \rfloor + \mathit{offset}(i)$. For efficiency reason $\lfloor f(i) \rfloor$ is implemented only by using integer multiplication and divison. Coefficients *multiplier* and *denominator* are found, such that function $f'(i) = i \times \mathit{multiplier}/\mathit{denominator}$ provides best possible approximation of $\lfloor f(i) \rfloor$. The idea is ilustrated by figure 2 on page 36. Memory consumption per single value is defined by

$$\mathit{bits\ per\ value} = \lceil \log_2(\max_{i=1..N} \mathit{offset}(i) - \min_{i=1..N} \mathit{offset}(i)) \rceil \tag{17}$$

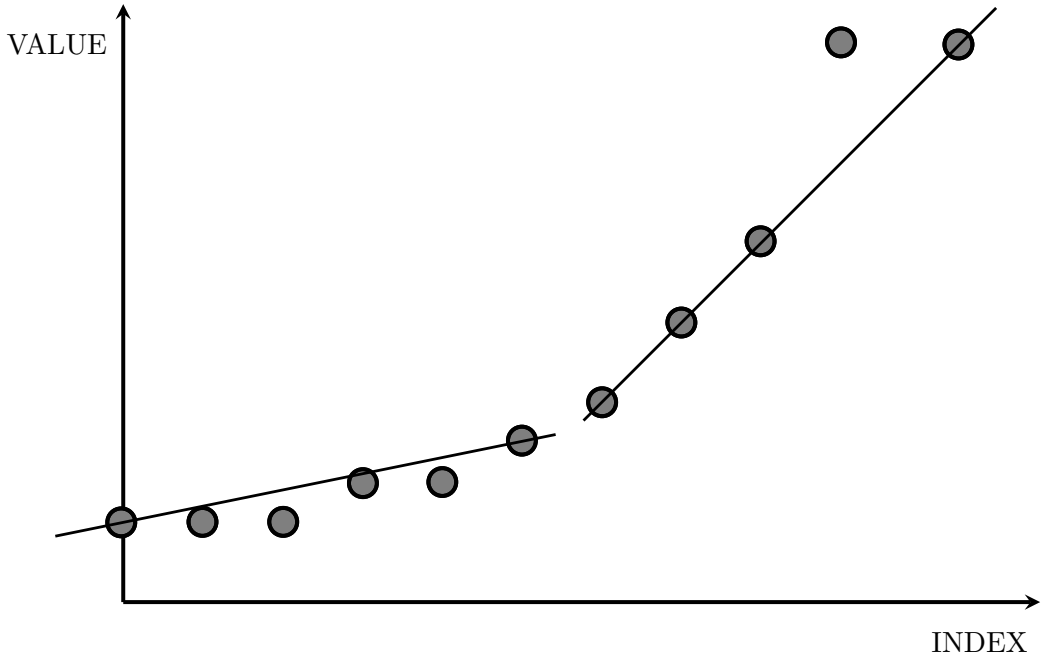


Figure 3: Non-decreasing array that uses two linear functions for different segments aproximation.

Sometimes the values to be stored cannot be conviniently approximated by a linear function. There can be different growth factors for different subsections of the data. In such a case, it is convenient to use several linear functions, each of them approximating its own data segment. This approach is illustrated by Figure 3. By using this technique, the memory consumption of the non-decreasing value array can be reduced rapidly in comparison to the *Packed Array* representation. For example, the number of bits needed per a single value of the offset array in the lexicon representation was reduced from 19 to only 5.

5.1.3 Forgetful Hash Map

The implemented spell-checker uses caching of data extensively in order to boost the application speed. Many important data such as the lexicon, morphology lexicon and language models were stored in bit arrays in order to decrease the memory consumption. Nevertheless, the drawback of such memory representation is the reduction of speed. To reduce this drawback, caching was used extensively. In order to make caching convenient, a generic data structure for caching of key-value pairs was implemented - the so-called forgetful hash map. It operates similarly to the standard hash map, but there is a limit on the number of elements that can be stored. When the limit has been reached and a new key-value pair is to be added, the key-value pair that was

not needed for the longest time is removed.

The forgetful hash map consists of a hash map, a dually-linked list and a counter of the stored values. There is a correspondence between the hash map and the linked list, each element in the linked list contains the key of a certain element stored in the hash map. For each key k, the hash map stores its value v and a pointer to the element inside the linked list that contains k. Whenever the value of a certain key k is required, the corresponding element of the linked list is moved to the front of the list[20]

Whenever a new key-value pair is being added, a new element containing the new key is aded to the front of the linked list, the key-value pair is stored in the hash map together with a pointer to the new linked-list element and the counter value is increased. If the counter value exceeds the predefined maximum, one element is erased from the back of the linked list and the hash map key that is stored in the erased element of the linked list is erased from the hash map as well. This ensures that there will not be more than the predefined maximum number of elements after each insertion.

It is possible to define a forgetful hash map for an arbitrary type of keys and values. Nevertheless, an implementation of hashing function[21] and equality comparator must be provided for type of key if there is no default implementation of these functions for the given data type in standard libraries[22].

5.2 Dictionary

Dictionaries are widely implemented as TRIE data structures. TRIE is a tree containing one letter in each node. Each node in the tree corresponds to a single word prefix which can be reconstructed by concatening all letters on the path from the root to the given node. For convenience, nodes that correspond to the dictionary entries will be denoted as *word nodes*, the remaining nodes will be denoted as *prefix nodes*[23] Building of the dictionary TRIE is a straightforward process. It starts with TRIE containing only a *root node* that has no letter inside. In each iteration of the TRIE building process, one word is picked from the dictionary and processed using the following procedure.

[20]The corresponding element can be found in constant time, because the hash map stores also the addresses of the linked list elements for all keys.
[21]This can be done easily with the use of boost::hash function
[22]In C++, default implementation is provided for integral types, string, pair and many others.
[23]The *word nodes* does not necesarrily need to be leaves. The *word node* can have outcoming edges as well if it is a prefix of another word).

1. Set *current node* to *root node*.

2. Pick the next letter of the word being added.

3. If *current node* does not have a child node marked with the picked letter, create such child node.

4. Set *current node* to the child that is marked with the picked letter.

5. If the word being processed contains more letters, repeat from Step 2

6. Mark *current node* as a *word node* (as a dictionary item).

The memory representation of TRIE that was implemented in this work was inspired by the representation of Finite State Transducers[24] in OpenFST toolkit (Allauzen, Riley, Schalkwyk, Skut, & Mohri, 2007).

Nodes were assigned IDs in such way that word nodes received greater IDs than prefix nodes. Edges were sorted according to their parent node IDs. The dictionary TRIE was represented using arrays *letter*, *offset* and *edge*. $letter[ID]$ stores the letter of the given node *ID*, *offset* stores pointers into the *edge* array. The edges leaving a particular node can be found at positions $offset[ID], ..., offset[ID+1]-1$ in the *edge* array. The *edge* array stores the only missing information about the edges - the IDs of their child nodes. The idea is illustrated in Figure 4. The *offset* array was implemented as described in Section 5.1.2, The *letter* and *edge* arrays were implemented as described in Section 5.1.1.

Due to the fact that *word nodes* have greater IDs it is straightforward to distinguish them from *prefix nodes*. If $nodeID \geq |\#prefix\ nodes|$, then the node is prefix. The unique integral identifier of word *wordID* is defined as $wordID = nodeID - |\#prefix\ nodes|$. These word identifiers represent the words in the other parts of the system.

5.3 Morphology Lexicon

The morphology lexicon stores all possible morphological analysis of words in the dictionary. Figure 5 provides an example of morphological anaysis of certain Czech words. Most of the words have several morphological interpretation, for example, *case* is often ambiguous by nouns. As it is shown in the figure, sometimes even *lemma* is ambiguous. In the morphological lexicon,

[24] To be more specific, it was inspired by the implementation of ConstFST class

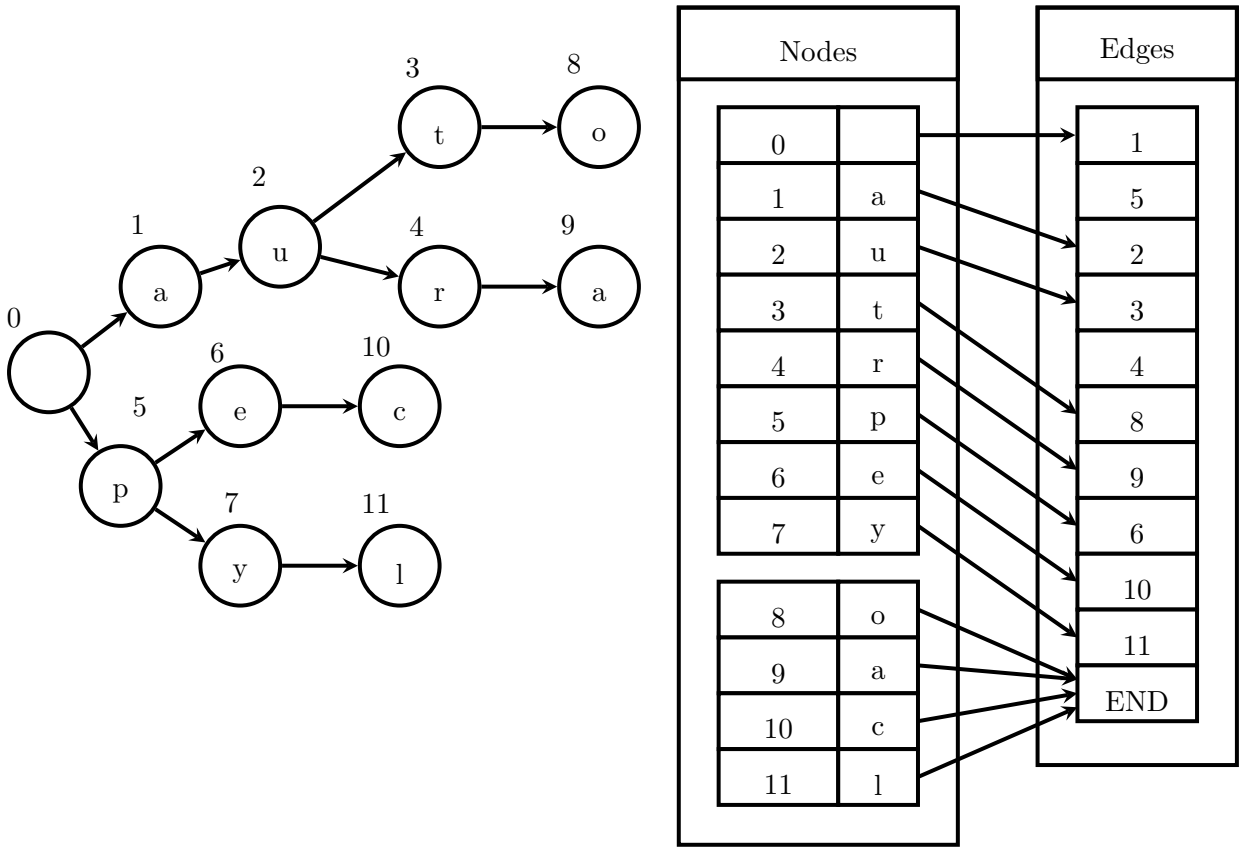


Figure 4: The simple dictionary storing words *auto*, *aura*, *pec* and *pyl*. IDs of prefix nodes are smaller than IDs of word nodes. Both the three structure and its internal representation are shown.

Analysis of word *ženu*:

- *lemma = žena*
  - `tag = NNFS4-----A----`
- *lemma = hnát*
  - `tag = VB-S---1P-AA---`

Analysis of word *stroje*:

- *lemma = stroj*
  - `tag = NNIP1-----A----`
  - `tag = NNIP4-----A----`
  - `tag = NNIP5-----A----`
  - `tag = NNIS2-----A----`
- *lemma = strojit*
  - `tag = VeYS------A----`

Analysis of word *jarní*

- *lemma = jarní*
  - `tag = AAFP1----1A----`
  - `tag = AAFP4----1A----`
  - `tag = AAFP5----1A----`
  - `tag = AAFS1----1A----`
  - `tag = AAFS2----1A----`
  - `tag = AAFS3----1A----`
  - ... 22 possibilites in total

Figure 5: Morphological analysis example

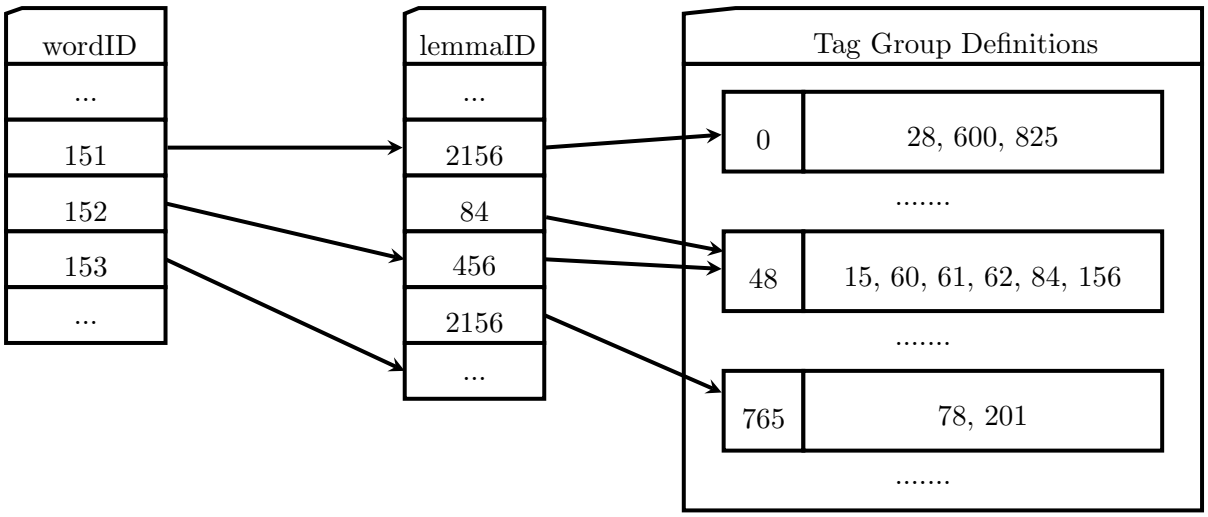


Figure 6: Representation of *multi-lemma* words' morphological analysis.

word forms, *lemmas* and tags are represented by IDs (Each category has its own numbering). Given string representation of a word, *wordID* can be obtained by traversing dictionary TRIE as described in Section 5.2).

There are several thousands of theoreticaly possible tag symbol. However there were only about 1000 distinct tag groups as results of the analysis of all *form-lemma* pairs found in the dictionary. This fact allows much more compact memory representation. Instead of storing array of tags for each *form-lemma* pair, it's possible just to store *groupID* pointing to the tag group definition. The memory representation of the morphology lexicon is shown in Figure 6. For each word, offset into the array storing *lemmaIDs* and *groupIDs* are stored.

5.4 Language Model Implementation

Section 3.3 describes how *n-gram* probabilities and *backoff* weights can be computed given the *n-gram* counts. General form of LM *smoothing* was expressed by equation (6). For performance reasons, functions f and *bow* used in this formula are usually precomputed. In this case, language model defined by enumeration of f and *box* values for set of *n-grams*. If $P(w_3|w_1, w_2)$ is requested and $f(w_3|w_1, w_2)$ is not stored, then *backoff* operation will be performed. In such case, if $box(w_1, w_2)$ is not stored, then its value is zero.[25]

Language model size reduction is non-trivial problem and there was a lot of research done on

[25] $[w_1, w_2]$ is not part of the model, which also implies that there is no *n-gram* of form $[w_1, w_2, w]$ in the model. This means that none of probability mass was consumed by higher level *n-grams*, thus *backoff* weight is zero).

methods that reduce number of stored n-grams rapidly without significant impact on language model performence. Some of these methods were mentioned in section 3.3. But appart from number of stored n-grams, memory efficient n-gram storing also plays an important role.

5.4.1 ZipTBO

ZipTBO representation is based on notion of tree structure of N-gram LM as it is shown in figure 7 on page 44. Detailed description of this representation can be found in (Whittaker & Ray, 2001). In LM tree, nodes in depth 1 correspond to *unigrams*, nodes in depth 2 to *bigrams* etc. Branching of the tree can be expressed by the equations (18) and (19).

$$children([w_1]) = \{[w_1, w]; w \in Dictionary \& [w_1, w] \in LM\} \tag{18}$$

$$children([w_1, w_2]) = \{[w_1, w_2, w]; w \in Dictionary \& [w_1, w_2, w] \in LM\} \tag{19}$$

Trigram consists of three distinct words, however there is only one *wordID* stored in *trigram* node. The missing *wordIDs* comes from parent and grand-parent nodes.[26] Sibling nodes are sorted according to their *wordIDs*. This feature allows using of *binary-search* during n-gram lookup.

As shown in figure 8 on page 45, LM tree can be efficiently represented by using separate arrays for storing nodes at distinct levels.[27] *Unigram array* contains pointers into *bigram array* and bigram array contains pointers into *trigram array*. There is no *backoff* weight for trigrams, since *backoff* weight is defined only for *n-grams* that can appear as word history of higher order *n-grams*.

5.5 Decoding Algorithm Implementation

The task is to find the optimal $S^* = (w_1^*...w_N^*)$ for the input sentence $S = (w_1^{'}...w_N^{'})$. As the underlying statistical model, the HMM is used. In Section 3.6, the Viterbi algorithm was introduced as an efficient algorithm of the sequence decoding for HMMs.

In the provided description of the algorithm, trellis values were computed for each state at each trellis stage. However, the number of states of the HMM for the multi-factor spell-checking as described in Section 3.5 is enormous. The set of HMM states consists of all (wf_i, wf_j) pairs[28],

[26] ancestor unigram node gives the first nodeID, ancestor bigram node gives the second nodeID.
[27] These levels corresponds to *n-grams* of different order.
[28] For convenience, multi-factor second order HMM is described here.

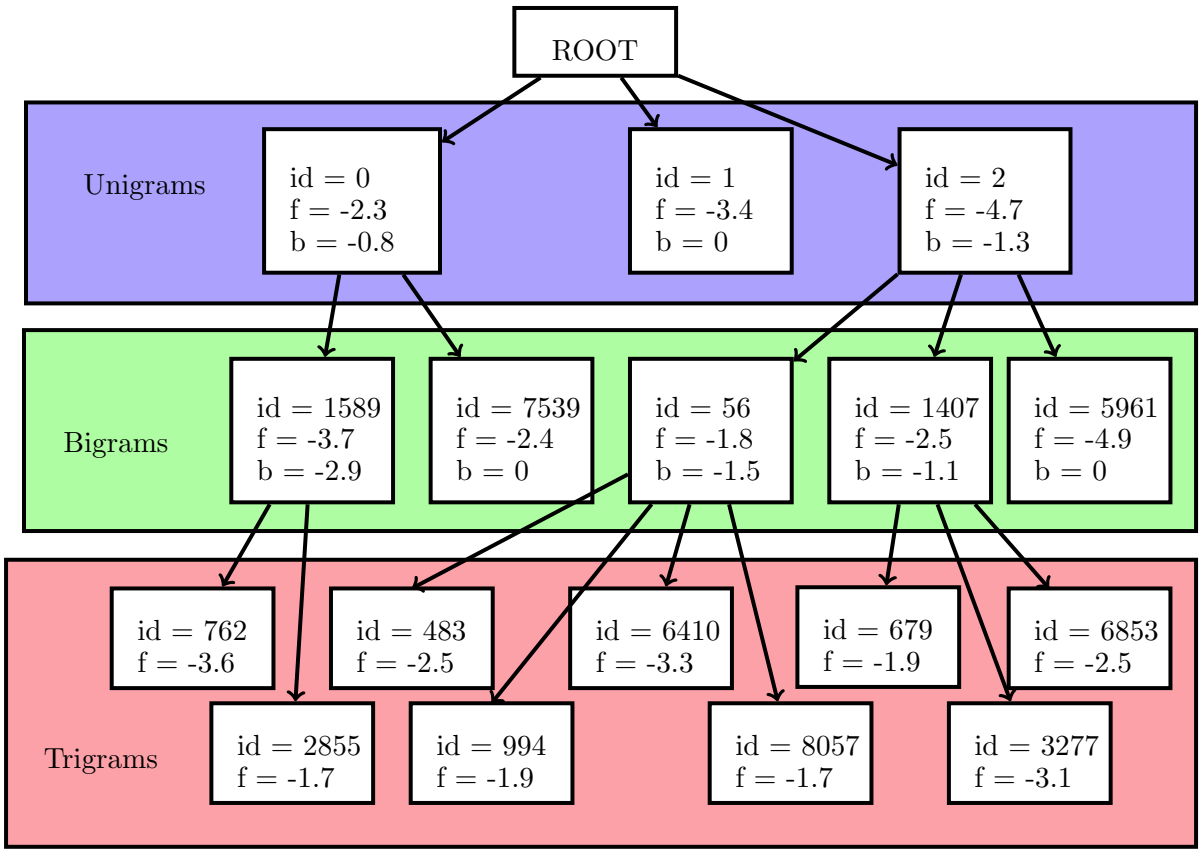


Figure 7: *Tri-gram* LM tree structure. (*b* stands for *backoff weight*)

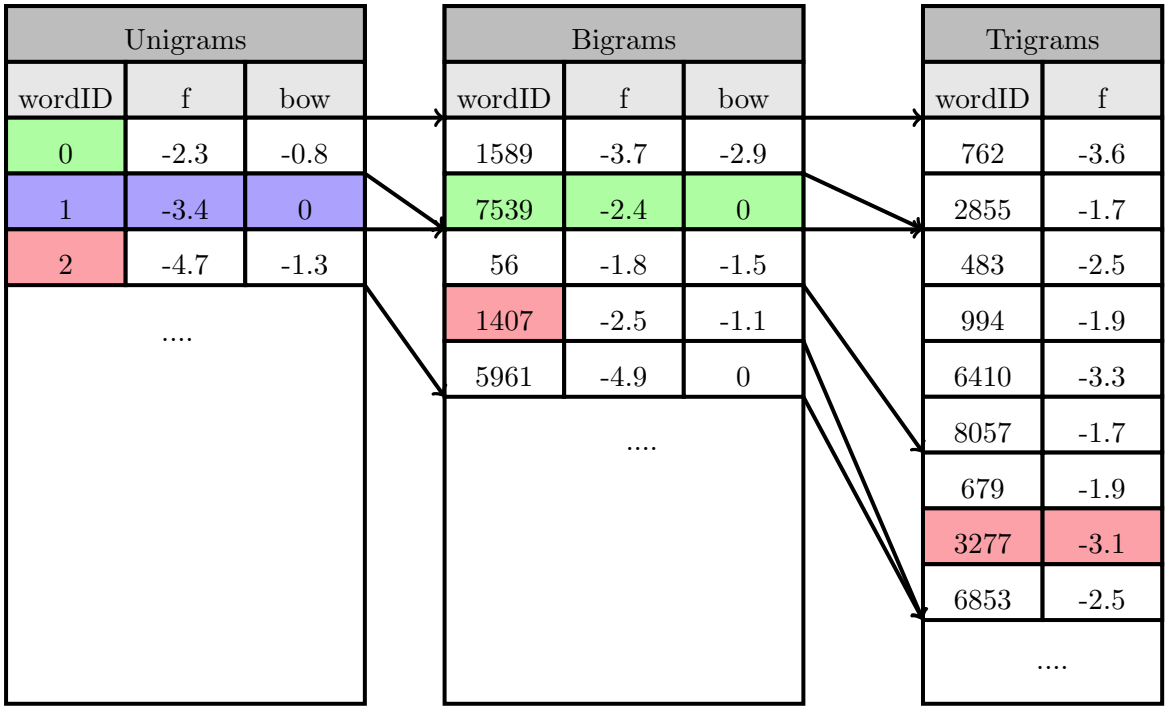


Figure 8: ZipTBO representation of LM tree shown in figure 7. Cells that defines *unigram* (1) are marked blue, cells that defines *bigram* $(7539, 0)$ are marked green, cells defining *trigram* $(3277, 1407, 2)$ are marked pink.

where wf_i and wf_j can be any *(form, lemma, tag)* triplet found in the morphology lexicon. Given that the dictionary contains about 3 million words, there are certainly more than 10^{12} possible states. Nevertheless, it is not necessary to build the memory representation of the given HMM, because all the transition and emission probabilities can be computed on demand.

Also, it is not necessary to examine each HMM state at each trellis stage. If there is the word *stromk* in the input sentence, then it makes no sense to consider the word *potopa* as a possible correction (the intended word was most likely *strom* or *stromek*). A list of possible corrections for each word of the input sentence is precomputed before the actual decoding starts. The list of possible corrections for word $w_i^{'}$ of the input sesntence is denoted as SP_i (Stage Posibilities). Such list of possible corrections for the word *stromk* is shown in Figure 9. Each correction forms a triplet $wf = (w, l, t)$ (word, lemma and morphological tag).

SP_i is computed for each input word $w_i^{'}$ in the input sentence, $SP_{N+1} = \{wf_{\texttt{</s>}} = (\texttt{</s>}, \texttt{</s>}, \texttt{</s>})\}$.

Further, the Viterbi decoding can start. However instead of filling the trellis matrix, trellis sets TS are to be constructed for each trellis stage. Trellis sets store the same information as the trellis matrix in the viterbi algorithm description in Section 3.6 - trellis probabilities v and back pointers bt.

Initialization step:

$$TS_0 = \{wf_{<s>} = (\texttt{<s>}, \texttt{<s>}, \texttt{<s>})\}$$

Recursive step:

```
For i = 0 to N
    For each (wf_x, wf_y) ∈ TS_i
        For each wf_z ∈ SP_{i+1}
            //summing logs of probabilities
            prob = f((wf_x, wf_y) → wf_z) + g(w'_{i+1} | (wf_y, wf_z))
            if (wf_y, wf_z) ∉ TS_{i+1}
                add (wf_y, wf_z) into TS_{i+1}
                v_{i+1}((wf_y, wf_z)) = prob
                bt_{i+1}((wf_y, wf_z)) = (wf_x, wf_y)
            else if prob > v_{i+1}((wf_y, wf_z))
                v_{i+1}((wf_y, wf_z)) = prob
                bt_{i+1}((wf_y, wf_z)) = (wf_x, wf_y)
    bestProb = max_{state ∈ TS_{i+1}} v_{i+1}(state)
    For each state ∈ TS_{i+1}
        if v_{i+1}(state) < bestProb − prunningConstant
            remove state from TS_{i+1}
```

Construction of the optimal solution:

```
state = argmax_state v_{N+1}(state)
For i = N downto 1
    state = bt_{i+1}(state)
    wf*_i = state.second   //(wf1, wf2).second = wf2
    w*_i = wf*_i.w         //(wi, li, ti).w = wi
```

The functions f and g are the transition and emission probability functions introduced in Section 3.5.

If $w_i^* \neq w_i^{'}$, an error is indicated at i-th word[29]. In such case, a list of correction suggestions sorted according to their probabilities is constructed using the following procedure:

[29] If $w_i^{'}$ is a valid dictionary entry, then a grammar error is indicated, a standard spelling error is indicated otherwise.

```
strom|strom|NNIS4
strom|strom|NNIS1
stromů|strom|NNIP2
stromy|strom|NNIP1
stromy|strom|NNIP7
stromy|strom|NNIP4
stromu|strom|NNIS2
stromu|strom|NNIS3
stromky|stromek|NNIP1
stromky|stromek|NNIP7
stromků|stromek|NNIP2
stromky|stromek|NNIP4
strome|strom|NNIS5
stromek|stromek|NNIS4
stromek|stromek|NNIS1
stromě|strom|NNIS6
stromky|stromek|NNIP5
stromy|strom|NNIP5
stromku|stromek|NNIS5
stromku|stromek|NNIS2
stromku|stromek|NNIS3
stroma|stroma|NNNS5
stromu|strom|NNIS6
stromku|stromek|NNIS6
strok|strok|NNMS1
stroma|stroma|NNNS1
stroma|stroma|NNNS4
```

Figure 9: Posible corrections of word *stromk*.

For each $wf_s \in SP_i$

$$suggProb = f\big((wf^*_{i-2}, wf^*_{i-1}) \rightarrow (wf^*_{i-1}, wf_s)\big) + f\big((wf^*_{i-1}, wf_s) \rightarrow (wf_s, wf^*_{i+1})\big)$$
$$+ f\big((wf_s, wf^*_{i+1}) \rightarrow (wf^*_{i+1}, wf^*_{i+2})\big) + g\big(w'_i|(wf^*_{i-1}, wf_s)\big)$$

$word = wf_s.w$

If $word \notin suggMap$ or $suggMap[word] < suggProb$

$suggMap[word] = suggProb$

$sorted\_suggestions =$ keys of $suggMap$ sorted in decreasing order according to the corresponding probabilities

By using this approach, it is possible to obtain a list of correction suggestions without performing the search for n-best Viterbi paths which would decrease the decoding speed significantly.

5.6 Diacritic Completion

In order to provide the implementation of the diacritics completion feature, the error model component was substituted. The other components of the system could remain unchanged. for a diacritics completion error model that assigns zero costs to the substitutions that add diacritics to a Latin letter. All the other edit operations are assigned infinite costs. As a consequence, all the candidate words for words in the input sentence are possible diacritics completions of these words.

For example, candidate words for the word *radu* according to the diacritics completion error model are *rádu, řadu, řádu, řádů, radů.*

According to the standard spell-checking error model, the candidate words are *radu, rádu, řadu, rady, rada, rad, radě, rady, hradu, rodu, řádu, radí, rado, vadu, radů, zradu, sadu ...* and many others.

After the candidate words are generated, both the spell-checking and diacritics completion work exactly the same.

Nevertheless, the lists of possible corrections are not generated for the diacritics completion, the result of this task is just the text with diacritics added.

5.6.1 Letter Language Model For Diacritics Completion on Unknown Words

It may possibly happen that for a given word of the input sentence, no candidate word is found. The example of such word is the word *nemeckofrancouzsky.* In such case, word remains untouched and no diacritics is added. However there is a high probability of error in such case, in the provided example the diacritics should rather be completed as *německofrancouzský* or *německofrancouzsky.*

In order to decrease the number of errors made on unknown words, a custom implementation of Viterbi decoder was provided. The states on the underlying HMM are tuplets of letters and the transition probabilities are given by a letter n-gram language model (it estimates the probability of next letter on the basis of previous letters). The aim of this Viterbi decoder is to find the most probable letter sequence given the input letter sequence. The only substitutions allowed are the substitutions that add diacritics.

Using this approach, diacritics can be added correctly even to the unknown words.

Given that the vocabulary of letter n-gram language model is extremely small (size of the alphabet), it is possible to train letter LMs of very high order. In this work, letter LMs of order

up to 7 were trained. The letter LMs were trained on the training part of WebColl. The data were preprocessed in such way that there was only one word per each line and the letters were separated by spaces. From the perspective of SRILM toolkit, each word was a sentence and each letter was a word.

5.7 Spell-checking of Text with No Diacritics

Spellchecking of a text that does not contain diacritics - diacritics insensitive spell-checking (DI spell-checking), should not consider the words such as *priusnice*, *skolnik*, *sislam* as spelling errors, because by adding of diacritics, the proper Czech words *příušnice*, *školník* and *šišlám* can be formed out of them.

For the word that is spell-checked, the DI spell-checking tries to find a diacritics completion that is contained in the dictionary. If such diacritics completion is found, the word is not considered as a spelling error.

Similarly to the implementation of the diacritics completion feature, the diacritics insensitive spell-checking functionality is achieved by providing of a custom implementation of an error model. In the DI spell-checking error model, edit operations that add diacritics to latin letters do not increase the edit distance and have assigned the zero cost. The cost of all the other edit operations remains the same as in the standard spell-checking error model. Figure 10 shows the list of the correction candidates for the word *stribo* according to DI spell-checking error model. According to the standard error model, the edit distance of words *stribo* and *stříbro* is 3 and the word *stříbro* is not considered as a correction candidate. According to the DI spell-checking error model, the edit distance of these words is only 1, because the operation of adding of diacritics does not increase the edit distance.

After the decoding is finished, the diacritics insensitive equality comparison is made for each pair (w_i^*, w_i')[30]. If w_i^* is not equal to w_i', then the spelling error is indicated. For example, if the word on input is *jablicko* and the decoded word is *jablíčko*, then the spelling error is not indicated.

5.8 MacOS X - Spell Checking Interface

On the Mac OS X operation system, there is a unified spellchecking interface that allows developers to create their own spellchecker implementations that can be used by any native Mac OS

[30]The pair of the decoded and input word.

```
stříbro|stříbro|NNNS4
stříbro|stříbro|NNNS1
Stříbro|Stříbro|NNNS1
Stříbro|stříbro|NNNS1
stříbro|stříbro|NNNS5
Strabo|Strabo|NNMS5
střído|střída|NNFS5
Stříbro|Stříbro|NNNS5
Stříbro|stříbro|NNNS5
Stříbro|Stříbro|NNNS4
Stříbro|stříbro|NNNS4
Strabo|Strabo|NNMS1
```

Figure 10: Correction candidates for word *stribo* in diacritics insensitive spell-checking

X application. On Linux or Windows there is no such easy possibility to create a system-wide applicable spell-checker and that is the main reason why Mac OS X was chosen as a target platform of this work.[31] The major programming language of the Mac platform is Objective C and the classes that provide the spellchecking interface are also written in it. The core spellchecking class is called NSSpellServer. It processes the spellchecking tasks sent by the applications through the instances of NSSpellChecker class. Those tasks are sent in traditional Objective-C manner - in the form of messages. It is a simplification, yet not an extremely big one, to say that a message in Objective-C is the equivalent of function call in C++. A spelling server forwards the messages to an instance of spellserver delegate and this is the class whose implementation is provided by the spellchecker developer and that does the important work - finding misspelled words, suggesting corrections etc. There are many tasks that a spellserver delegate may be able to perform, it only needs to provide an implementation of the corresponding methods.

Following is the list of methods that a spellserver delegate may implement:

5.8.1 findMisspelledWordInString

```
- (NSRange)spellServer:(NSSpellServer *)sender
  findMisspelledWordInString:(NSString *)stringToCheck
  language:(NSString *)language
  wordCount:(NSInteger *)wordCount
  countOnly:(BOOL)countOnly
```

[31] Although functionality could be easily transported into any other platform, because all backend functions were written in C++ in a platform independent way.

This method should check for misspelled words in *stringToCheck* and return a range of the first misspelled word found. For example, if the method is called on text

> *"The French defence ministyry said last week that it had profided logistical and technical support to Mauritanian forces carrying out the raid in northern Mali."*

it should return the range of *ministyry* which is a typo and should have been *ministry* instead. After NSSpellChecker received this result from the spellserver, it usually sends a new message to the spellserver demanding the spellchecking of the rest of the text. In this case, consequent spellchecking would be asked for text

> *" said last week that it had profided logistical and technical support to Mauritanian forces carrying out the raid in northern Mali."*

which should return the range of word *profided* which is a typo and should have been *provided* instead. *findMisspelledWordInString* method is not capable of returning any other useful information such as suggestion of possible corrections.

In this work, this method was simply implemented by consecutive dictionary lookup on each token formed by letters and by returning the range of the first such token that was not found in the dictionary. The Viterbi decoder cannot be utilized inside this method.

5.8.2 checkGrammarInString

```
- (NSRange)spellServer:(NSSpellServer *)sender
  checkGrammarInString:(NSString *)string
  language:(NSString *)language
  details:(NSArray **)outDetails
```

This method checks for grammar errors in *string*. These errors are usually highlighted by green underscoring by an application that uses a spell-checker with grammar checking turned on (spelling errors are highlighted by red underscoring). Grammar checking can be used for the purpose of checking of subject - verb agreement, proper sentence start capitalization and many other things (for example, native Mac OS X spellserver highlight phrase *We is* as a grammar error). In this work, grammar checking interface was used for identifying spelling errors that

accidentally form words contained in the dictionary.[32]

Function *checkGrammarInString* returns a range of text segment containing grammar errors. Further details about errors found in this text segment can be specified by setting *outDetails* output parameter. *outDetails* should contain details of every grammatical error found in the checked text segment. A detailed description of each error consists of specifying a subrange of this error in the context of the checked text segment, an array of correction suggestions and an error description (which is intended to be presented to the user).

A text checking server stores error details for sentences that have already been checked in a *checked sentences* hash map. This feature is utilized by the provided implementation of *checkGrammarInString* method which works as follows: *string* is first splitted into sentences. The first sentence is looked for in a hash map of checked sentences. If it is found, then the error details stored in the hash map are returned. If it is not found, then the ViterbiDecoder is run on the sentence in order to find errors. The result is then stored in the *checked sentences* hash map. Both standard and grammar errors are looked for. However, only grammar error details are stored in *outDetails* output variable.

However, the interface method *suggestGuessesForWord* which should provide a list of suggestions is context in-sensitive, the only parameter than can be specified is the word itself. Because of this limitation,

Due to the limitations of *suggestGuessesForWord* function that will be discussed later, suggestions for standard spelling errors are stored into *suggestions* hash map (keys of this hash map are misspelled word forms and values are arrays of suggestions). When there is a request for the list of suggestions for a given misspelled word, the list of suggestions found in hash map is used.

5.8.3 Installation of Spell-checker On Mac

All the information regarding the installation and using of spell-checker developed in this thesis can be found in `README` file in the root directory of the accompanying CD.

[32] This is a different purpose from what people usually mean by *grammar checking*. However it is practical to highlight such spelling errors in a way different from highlighting the standard spelling errors, because there is a greater possibility that the spell-checker's suggestion is wrong (the word that was typed might in fact be correct), so the less alarming green colour is a good choice here.

6 Evaluation

In order to evaluate the system's performance for both tasks (context sensitive spell-checking and diacritics completion), dirty data and golden standard data are needed. Dirty data contain problems that need to be solved (spelling errors or missing diacritics), golden standard data contain the right solutions. The more the output of the system matches the golden standard data, the better.

The system evaluation was made in order to judge the contribution of the proposed method of multifactor spelling correction and diacritics completion. As the baseline for the comparison, a single-factor approach utilizing language models based on word forms solely can be used.

6.1 Diacritics Completion Evaluation

Obtaining of testing data for diacritics completion is an easy task. Any text containing diacritics can be taken as golden standard data, dirty data can be obtained easily by removing the diacritics. The diacritics can be easily evaluated by counting the rate of words that were completed correctly (i.e. the ratio of output words that are identical to the golden standard words, the punctuation tokens and numbers are not considered, because there is nothing to be done for these tokens).

The diacritics Completion was evaluated on four different datasets, part of the WebColl corpus devoted for testing and three different books[33]: *Martin Gilbert: A History of the Twentieth Century* (non-fiction), *Lion Feuchtwanger: Foxes in the Vineyard* (fiction) and *August Sedláček: Sbírka pověstí historických lidu českého v Čechách , na Moravě i ve Slezsku* (archaic).

The main parameters are the weights α_f, α_l and α_t of features f_f (word form feature), f_l (lemma feature) and f_t (tag feature).

First, the contribution f_l and f_t was examined separately. In these experiments α_f was ranging from 0 to 1 and the weight $(1 - \alpha_f)$ was given either f_l or f_t, all the language models used were trigrams. The results of such experiments for each data set are plotted in Figures 11, 12. It is clear from the plots that both features f_l, f_t improve the system performance. However the contribution of f_t is more significant. Surprisingly, it seems to better to give all the weight to f_t than to give all the weight to f_f.

The performance boost achieved by using f_t is most visible on a comparison of results achieved

[33]Obviously, Czech translation were used for books that were written in a forreign language originally

on history domain and fiction domain data. For baseline setup ($\alpha_f = 1, \alpha_t = 0$), the accuracy is 97,39% on non-fiction data and 96,74% on fiction data, which means that the error rate is 25% bigger on fiction data. Nevertheless, by increasing the weight of f_t the difference in performence was becoming less significant and for the best parameter settings ($\alpha_f = 0.4, \alpha_t = 0.6$), the error rate on fiction data was only 7% bigger (97.72% accuracy on fiction data and 97.89 on non-fiction data).

The performance on the domain of archaic text follows the same pattern, but the achieved results were much worse. The maximal accuracy of 94.61% was achieved for $\alpha_f = 0.4, \alpha_t = 0.6$. The chosen for out data represents a really old form of Czech and both the vocabulary and writing style are different from the modern Czech:

Na poděkování, Že jej Pán Bůh od jisté záhuby zázračně vysvobodil, umínil si postaviti při Lnářích klášter, když panství to jako díl otcovský a bratrský obdržel. Již se začaly základy vybírati u bažantnice, když se Černín tam, kde nyní klášter stojí, procházel. Ohlížel se po všech předmětech známých, ale najednou uzřel tu strom o třech korunách, na nějž se nepamatoval, že by jej tu byl viděl. Zdálo se mu, že v tom vidí znamení nejsvětější Trojice, a proto kázal začaté základy zaházeti a nově dal kopati na tom místě, kde ten strom viděl. Tak povstal klášter s kostelem, na jehož hlavní oltář dal Černín takový obraz, jako mu byl zanechal archanděl.

Next, the estimation of the best parameter setting for each data set was done using a simple hill-climbing algorithm[34] (description can be found in (Russell & Norvig, 2003)) As the starting point, all the weights were set equally. The resulting parameters and the accuracy values are shown in the Figure 10. The results of experiments with the letter LM feature turned on were made as well for the particular settings.

It can be seen that the use of letter LM for the completion of the unknown words improves the results significantly. Table 9 provides a detailed view on the performance of letter LM on each dataset. For the data that was translated from a forreign language (fiction and non-fiction) there was a lot of errors made on names. When foreign proper names are translated to the Czech language, the diacritics is added only to the word's suffix if grammar demands that. This result suggest that it might be plausible to provide a custum behaviour of this feature for both

[34]Start in a random point in the parameter space, make update steps iteratively in the direction of gradient of the given fitness function until the stopping criterion is met.

dataset	success - general	fail - general	success - proper	fail - proper
non-fiction	505	52	279	202
fiction	326	69	174	237
archaic	237	36	134	39
heldout	823	66	437	136

Table 9: The detailed statistics of Letter LM feature performance. *success - general* denotes the number of succesfull completions on general words, *fail - general* denotes the number of errors newly introduced on general words. The same statistics are given for the proper names in the next two columns.

dataset	α_f	α_l	α_t	accuracy	accuracy with letter LM
non-fiction	0.309	0.283	0.407	97.91	98.31%
fiction	0.312	0.141	0.547	97.73	97.88%
archaic	0.313	0.236	0.451	94.53	95.7%
WebColl	0.340	0.330	0.330	98.57	99.11%

Table 10: The best accuracy values achieved on each testing set (no letter language model, all the LMs are trigrams).

word types (For example, separate letter LMs for proper names and general words words or there could be a possibility to disable the feature for proper names).

6.2 Spell-Checking Evaluation

In contrast to diacritics completion, obtaining of the testing data for spell-checking is much more complicated task. Such data must be produced manually. The error corpus *Chyby* (Pala et al., 2003) which is being built in Brno could provide an ideal data for evaluation of the given task, since the error such as *i/y s/z* or morphological errors that could not be found by non-context sensitive spell-checker are anotated as well. Nevertheless, the evaluation on this corpus is left for the future work.

However two smale scale test sets were created in order to provide a limited evaluation of the spell-checking.

One of these testing sets was created by a volunteer person by a transcription of the spoken word. This testing set contains 218 spelling errors (out of this number, 12 errors are real-word errors) and there were 1371 words in total (this testing data set will be denoted as *olga*).

The other set was created from the part of WebColl devoted for testing (this testing data

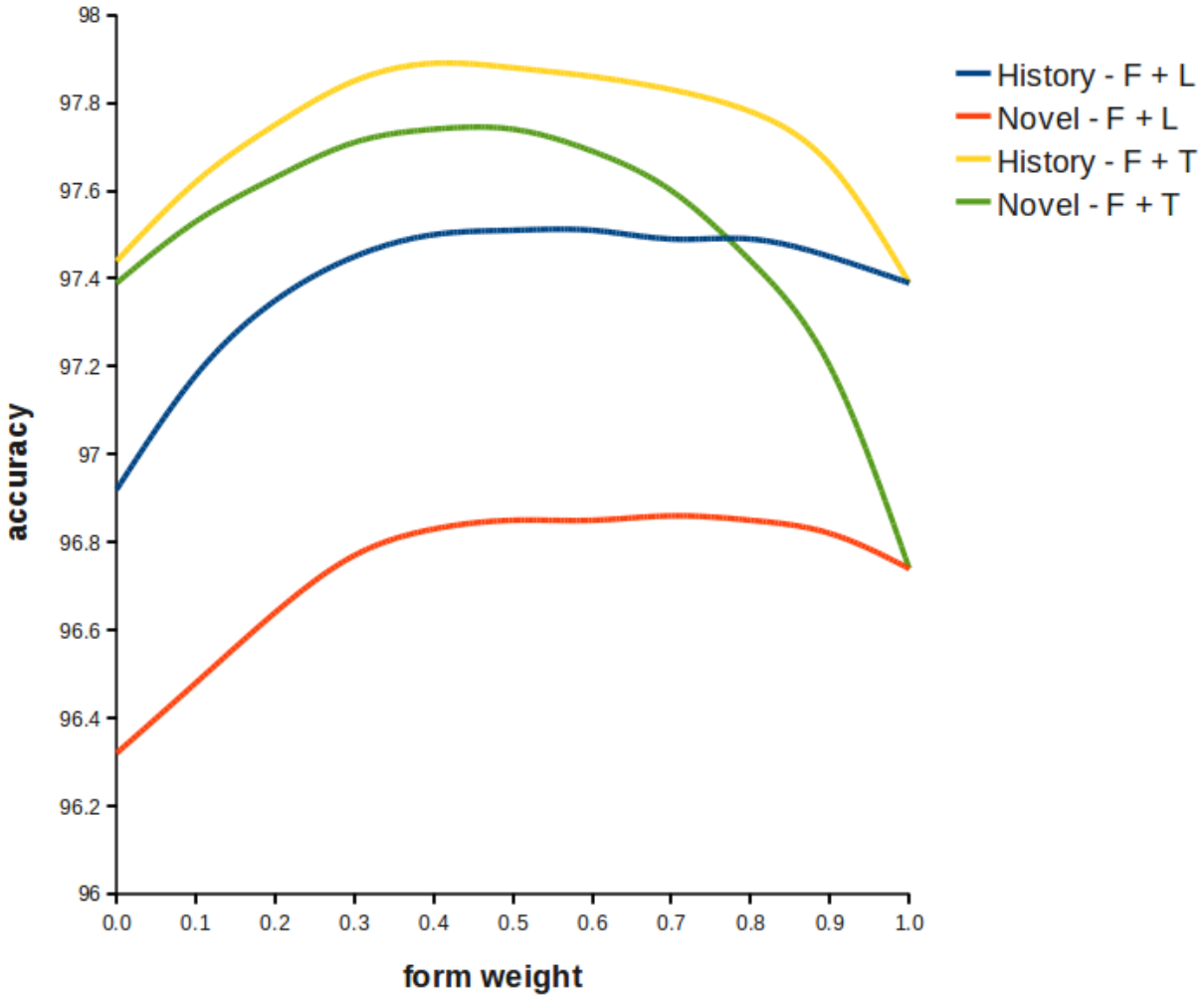


Figure 11: Results of *form* - *lemma*, *form* - *tag* experiments. The values of α_f are on the x-axis. The other feature gets the remaining weight $(1 - \alpha_f)$. The blue line: *form* - *lemma* combination on non-fiction data, the yellow line: *form* - *tag* on non-fiction, the red line: *form* - *lemma* on fiction, the green line: *form* - *tag* on fiction.

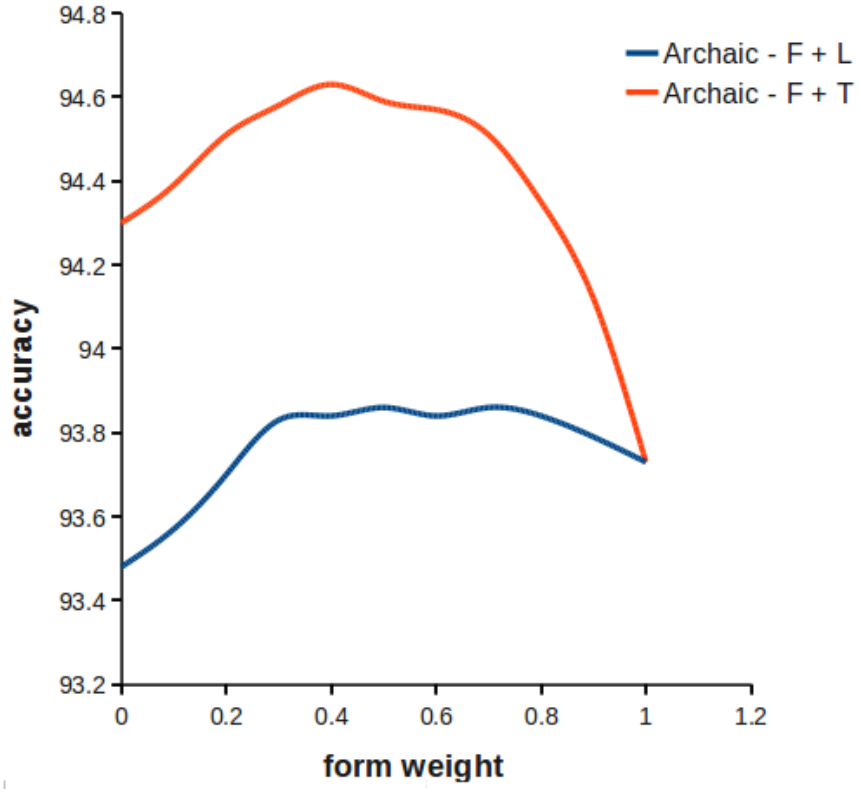


Figure 12: Results of *form* - *lemma*, *form* - *tag* experiments on archaic texts. The values of α_f are on the x-axis. The other feature gets the remaining weight $(1 - \alpha_f)$. The blue line depicts the results for *form* - *lemma* combination, the red line for *form* - *tag* combination.

set will be denoted as *test*). The words identified by the spell-checker[35] as spelling errors were examined manually and the words that were flagged as spelling errors by mistake were filtered out. The result of this process was the set of sentences containing spelling errors authorized by a human. The golden standard data were created manually in the next step. This approach made the collecting of errors in the WebColl testing data feasible, however all the real-word errors were missed (they were overlooked, because they were not flagged as spelling errors by spell-checker in the first step).

The quality of spell-checkers is usually measured by the spelling correction error rate (i.e. what is the probability that the first given suggestion is correct or that the correct suggestion is included in the list of first three suggestions etc.) If the context sensitive spell-checker is considered and the ability of recognizing the real-word errors is to be tested, *F-measure* based on *Precission* and *Recall* can be used.

$$P = \frac{TP}{TP + FP}$$

$$R = \frac{TP}{TP + FN}$$

$$F = \frac{2 \times P \times R}{P + R}$$

where TP (true positives) denotes the number of succesfully recognized real-word errors, FP denotes the number of words that were incorrectly marked as real-word errors and FN denotes the number of real-word errors that were not recognized. *F-measure* provides the harmonic mean of *Precission* and *Recall* and it is a good indicator of a quality of a classifier.

Small scale experiments were made for the combinations of two testing data sets and two estimated error models (the error model creation process was described in section 4.4). The optimal setting of parameters α_f, α_l, α_t, α_e of features f_f, f_l, f_t, f_e is looked for using the hill-climbing algorithm. During the lookup, the fitness function of the hill-climbing algorithm was constructed by the interpolation of the accuracy on the first suggestion, the accuracy on first two suggestion and F-measure on real word errors. The results achieved for distinct combinations of testing corpus and error model are shown in Table 11.

[35]The spell-checker made look-up for the out of vocabulary words easier. The correction suggestions given by spellchecker weren't taken into the consideration during the creation of golden standard data, so the fact that the spell-checker that is to be tested participated in the creation of the testing set does not invalidate the testing set.

evaluation set	error model	1 sugg	1st 2 sugg	5 sugg	precission	recall	f-measure
olga	auto	91.63	97.2	98.6	1	0.77	0.87%
olga	manual	91.63	97.2	98.6	1	0.77	0.87%
test	auto	91.39	95.08	96.31	-	-	-
test	manual	90.57	95.08	96.31	-	-	-

Table 11: Multi-factor spell-checking: The best results achieved for each combination of test set and error model (no letter language model, all the LMs are trigrams). *1 sugg* stands for the accuracy on the first suggestion, *2 sugg* stands for the accuracy on the first two suggestions etc.

evaluation set	error model	1 sugg	1st 2 sugg	5 sugg	precission	recall	f-measure
olga	auto	90.09	96.7	97.64	0.875	0.54	0.66%
olga	manual	90.09	96.7	97.64	0.875	0.54	0.66%
test	auto	90.16	95.08	96.31	-	-	-
test	manual	90.16	95.08	96.31	-	-	-

Table 12: Single-factor spell-checking: The best results achieved for each combination of test set and error model (no letter language model, all the LMs are trigrams). *1 sugg* stands for the accuracy on the first suggestion, *2 sugg* stands for the accuracy on the first two suggestions etc.

The results achieved for baseline setup - f_l and f_t are not used - are shown in Table 12.

Although better results were achieved for the multifactor spell-checking, the differences are insignificant. In both testing sets, there are around 200 errors, so the 0.5% difference in spelling correction error rate means the difference on one single words. Given this fact, on the basis of the current evaluation of context sensitive spell-checking, the contribution of the multi-factor decoding cannot be measured. However the results obtained on real-word spelling errors (f-measure = 0.87) for multi-factor decoding are quite promissing (although there were only 12 real-word errors).

Both error models performed equally good although there were big differences in the estimated error probabilities for distinct edit operations. For the evaluation of the error models, more testing data would be needed as well.

For the comparison, (Brill & Moore, 2000) achieved the spelling correction error rate of 97.6% for the first suggestion when trigram language model was used. This promissing result was achieved by the use of very powerful error model. However the results obtained for the different languages are not easily comparable.

7 Conclusions

A context-sensitive method of spell-checking and diacritics completion was designed and implemented in this work. Both features can be helpful in producing better-quality Czech texts with less effort. Regarding the spell-checking task, emphasis was put on the ability of the system to recognize real-word spelling errors and also on the ability to suggest the right corrections for spelling errors.

The overall accuracy of diacritics completion was about 98% with training and testing data coming from different domains[36]. The accuracy on testing data coming from the same corpus was over 99%. This seems to be a significantly better result than the one achieved by (Vrána, 2002), because the best accuracy he published was 97.4% and his training and testing data came from the same corpus[37].

However, only a small scale evaluation of context sensitive spell-checking was made and the contribution of multi-factor decoding for spelling-correction cannot be determined from the obtained evaluation results.

The proposed solution of context-sensitive spell-checking and diacritics completion is based on statistical methods and uses language modelling extensively. The following hypothesis was addressed, too: Can the use of statistical models that work on morphological features such as lemma and morphological tag lead to better results? The evaluation showed that the use of these features can improve the system's performance significantly. For diacritics completion, it led to 20% - 30% decrease of error rate (depending on the testing dataset) when compared to the baseline setup that uses language models based on word forms solely. In the spelling correction task, the use of morphological features improved f-measure for the real-word spelling error identification problem from 0.66 to 0.87, however a bigger testing data would be needed for the serious evaluation.

A method for diacritics completion on unknown words was proposed. This method is based on a language model that works on letter sequences. By using this approach for handling unknown words, error rate decreased significantly. However, the application of this approach often led to ridiculous errors such as completion of *Picasso* to *Pičasso*. It is evident that adding diacritics incorrectly can be more harmfull than failing to recognize a word that diacritics should be added

[36] Statistical models were trained on a corpus of news articles, the testing was carried out on a novel by Lion Feuchtwanger and a book on the history of 20th century

[37] He also published a 98% accuracy when his training and testing data were identical, but this figure does not say much about the real-life system performance

to. This is the aspect of automatic diacritics completion that could be further looked into.

The spell-checker application developed in this work implements the standard Spelling Server interface of Mac OS X Snow Leopard. Leopard operation system and was released as a standard Mac OX X application bundle - *michalisekSpell*. Because of this, the users of any modern Mac OS X application enabling spell-checking can decide to use *michalisekSpell* for the correction of Czech texts. The application *michalisekSpell* provides diacritics completion feature as a standard Mac OS X system service which means that it can be used system-wide. A user just selects a text s/he wants to add diacritics to, chooses the *Add diacritics!* service in the context menu (or uses a keyboard shortcut) and diacritics is added instantly.

A minor point of the thesis assignment was to integrate certain linguistical resources developed at UFAL, such as morphological lexicon and thesaurus, into the spell-checker. This point of the assignment was fullfilled only partially. The morphological lexicon and its feature of style signs of lemmas was integrated, which allows the application of style restrictions on the lexicon (meaning that vulgar, expressive or archaic words may or may not be accepted). The integration of the thesaurus was left for future work, as it has no direct connection with the main focus of this work and it is only a matter of implementation without a significant scientific challenge.

References

Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., & Mohri, M. (2007). Openfst: a general and efficient weighted finite-state transducer library. In *Ciaa'07: Proceedings of the 12th international conference on implementation and application of automata* (pp. 11–23). Berlin, Heidelberg: Springer-Verlag.

Brill, E., & Moore, R. C. (2000). An improved error model for noisy channel spelling correction. In *Acl '00: Proceedings of the 38th annual meeting on association for computational linguistics* (pp. 286–293). Morristown, NJ, USA: Association for Computational Linguistics.

Chen, S. F., & Goodman, J. (1998). *An empirical study of smoothing techniques for language modeling* (Tech. Rep.). Cambridge, Massachusetts: Computer Science Group, Harvard University.

Church, K., & Gale, W. (1991). Probability scoring for spelling correction. *Statistics and Computing*, *1*(7), 93-103.

Church, K., Wa, R., Hart, T., & Gao, J. (2007). Compressing trigram language models with golomb coding. In *In proceedings of emnlp-conll 2007.* Prague, Czech Republic.

Damerau, F. J. (1964). A technique for computer detection and correction of spelling errors. *Commun. ACM*, *7*(3), 171–176.

Hajič, J. (2004). *Disambiguation of rich inflection (computational morphology of czech)*. Nakladatelství Karolinum.

Jurafsky, D., & Martin, J. H. (2008). *Speech and language processing: An introduction to natural language processing, computational linguistics and speech recognition* (Second ed.). Prentice Hall. Paperback. Available from `http://www.worldcat.org/isbn/013122798X`

Katz, S. M. (1987). Estimation of probabilities from sparse data for the language model component of a speech recognizer. In *Ieee transactions on acoustics, speech and signal processing* (pp. 400–401).

Kneser, R., & Ney, H. (1995). *Improved backing-off for m-gram language modeling* (Vol. 1). Available from `http://dx.doi.org/10.1109/ICASSP.1995.479394`

Koehn, P., & Hoang, H. (2007). Factored translation models. In *In proceedings of emnlp-conll 2007.* Prague, Czech Republic.

Mays, E., Damerau, F. J., & Mercer, R. L. (1991). Context based spelling correction. *Information Processing & Management*, *27*(5), 517 - 522. Avail-

able from http://www.sciencedirect.com/science/article/B6VC8-469WV1X-10/2/93f4211e2b7779cae43d8b1dc2db6585

Pala, K., Rychlý, P., & Smrž, P. (2003). Text corpus with errors. In *Text, speech and dialogue* (pp. 90–97). Springer Verlag.

Peterson, J. L. (1986). A note on undetected typing errors. *Commun. ACM*, *29*(7), 633–637.

Russell, S., & Norvig, P. (2003). *Artificial intelligence: A modern approach* (2nd edition ed.). Prentice-Hall, Englewood Cliffs, NJ.

Smith, N. A. (2004). *Log-linear models.*

Vrána, J. (2002). *Obnovení diakritiky v českém textu.* Diploma Thesis. (Version 5.10.0)

Whittaker, E., & Ray, B. (2001). Quantization-based language model compression. In *In proceedings of eurospeech.*

Witten, I., & Bell, T. (1991). The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. In *Ieee transactions on information theory* (p. 1085-1094).

Yuret, D., & Stolcke, A. (n.d.). *ngram-discount - notes on the n-gram smoothing implementations in srilm.* Available from http://www-speech.sri.com/projects/srilm/manpages/ngram-discount.7.html. (SRILM man pages)

Zhai, C., & Lafferty, J. (2004, April). A study of smoothing methods for language models applied to information retrieval. *ACM Trans. Inf. Syst.*, *22*(2), 179–214. Available from http://dx.doi.org/10.1145/984321.984322