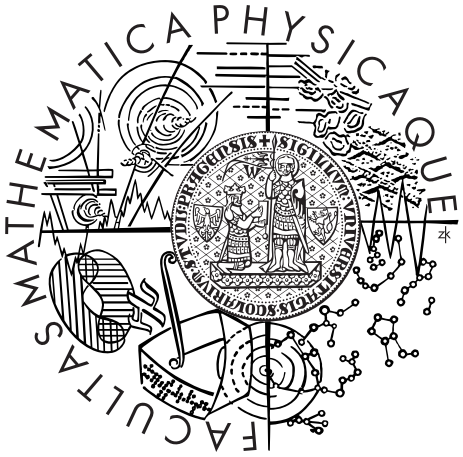
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Michal Koutný

Word prediction using language models

Institute of Formal and Applied Linguistics

Supervisor of the bachelor thesis: Mgr. Martin Popel
Study programme: Computer Science
Specialization: Programming

Prague 2012

I would like to thank my supervisor Mgr. Martin Popel for his advice, help and self-sacrifice during my work on the thesis.

I would also thank Mgr. Martin Majliš for his Web To Corpus project.

Last but not the least, I am grateful to my supportive parents.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Použití jazykových modelů k předvídání psaných slov

Autor: Michal Koutný

Katedra: Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: Mgr. Martin Popel, Ústav formální a aplikované lingvistiky

Abstrakt: Práce využívá ngramových jazykových modelů k usnadnění zadávání textů pomocí QWERTY klávesnice předvídáním psaných slov. Nejprve jsou představena existující obdobná řešení a položen teoretický základ práce. Následující analýza dělí problém do čtyř částí: trénování modelů, využití modelů k predikci, GUI komponenta a nástroje pro hodnocení. Byly použity jazyky Python a C++. Použité textové korpusy jsou z české a anglické Wikipedie (19 a 84 miliónů slov), k testům přizůsobení je též použit malý český korpus vzdělávacích textů. Pomocí definovaných metrik jsou ohodnocena různá nastavení. Nejlepší výsledek pro testovací data byl 0.44, resp. 0.55 úhozů na znak pro angličtinu, resp. češtinu.

Klíčová slova: předpovídání slov, jazykový model, doplňování textu, asistivní technologie

Title: Word prediction using language models

Author: Michal Koutný

Department: Institute of Formal and Applied Linguistics

Supervisor: Mgr. Martin Popel, Institute of Formal and Applied Linguistics

Abstract: The thesis utilizes ngram language models to improve text entry with QWERTY keyboard by the means of word prediction. Related solutions are briedly introduced. Then follows theoretical background for the work. The analysis in the next part divides problems into four tasks: language model training, incorporating model for word prediction, GUI component and evaluation framework. The realization combines Python and C++. The used corpora come from Czech (19 M words) and (84 M words) English Wikipedia articles. A small corpus of Czech educative texts was used to test domain adaptation. The quality metrics are defined and various configuration are measured. The best solutions reduced keystrokes per character to 0.44, resp. 0.55 for English, resp. Czech on testing data.

Keywords: word prediction, language model, autocompletion, assistive technology

Contents

1. Introduction

Word prediction is a way how to help users to insert words, phrases or text faster, more easily and with less errors. In situations when target domain is small (address book, search engine phrase completion) and/or the grammar of inserted text is known (source code completion, URL completion, bash completion), word prediction can be realized with relatively simple means (regarding domain size).

Common methods for general text completion are based on suggesting already typed words (KDE's Kate editor)[1] or parts of them (Open Office Writer).[2]

This project's objective is to provide word prediction for general texts where already typed words are combined with large dictionary for a given language. The central method to achieve quality of predictions is based on sorting suggestions by their probability in the context. The probability is estimated by large ngram language models combined with cached models and small models trained on user data for better adaptation to user's domain.

In order to allow direct testing and potential spread of word prediction, a reusable graphic user interface component was implemented. The component takes advantage of the language models as well as additional methods that were implemented to further simplify and accelerate input process, including automatic capitalization, partial words suggestions and space character completion.

The original idea stems from using word prediction to improve text input for disabled users or users of devices with reduced or unusual keyboards. However, during the project the aiming was modified to common computer users who could also benefit from word prediction. Though implemented for common computers only, the involved methods are general enough and they could be used with today's mobile devices as well.

As there is a need to evaluate and suggested methods for word prediction, script for automated training and testing were implemented to automatize testing phase of the development process.

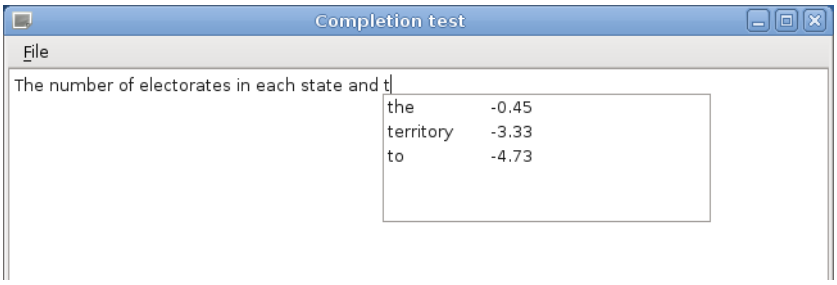


Figure 1.1: GUI component showing probable suggestions.

[1] `http://kate-editor.org/`
[2] `http://www.openoffice.org/`

1.1 Related work

1.1.1 Theoretical

The most relevant article for this work is the article [1] by Trnka. They ran an experiment testing the effect of word prediction in terms of: input rate (keystrokes per second), communication rate (words per minute) and prediction utilization (ratio between actual and potential keystroke savings). They used a basic model – suggestions consisted of recently used words followed by alphabetically sorted words from a dictionary; and an advance model – trigram model smoothed with simplified Katz's backoff and trained on 2.6 million words from Switchboard corpus. They report 0.738 resp. 0.443 potential keystrokes per character for basic resp. advanced model.

Al-Mubaid [3] describes use of language models to reduce number of keystrokes and considers increased cognitive load on users when there are too many suggestions. Their method to limit the suggested items incorporates language syntactic structure too, however, they don't present any results regarding keystroke savings.

Interesting method for increasing text input rate incorporating language models is discussed by Shieber [5]. It allows user type words in a compressed form (e.g. *mscmnctn* instead of *miscommunication*) and word trigram model is used to choose best matching decompressed form (for OOVs, they use up to 10-gram letter model). Although, they method isn't comparable with direct word prediction, they expected typing 0.717 keystrokes per character.

1.1.2 Implementations

WordQ[3] is a commercial software suite for general typing assistance and one of its capabilities is also word prediction. It apparently cooperates with Microsoft Office SW, browser and other desktop programs as well. Because of its commercial nature and non-GNU/Linux OS targeting, no more exact information about word prediction were found.

Swiftkey [14] is another commercial solution that incorporates language models. It's designed for use on mobile devices with special impact on learning from user texts (as implies published information). Again, as it's a commercial product, it was not further examined.

It is also worth mentioning the Dasher Project,[4] a software for text entry without keyboard, targeted mainly on disabled people. It uses letter language models and novel user interface. But because of the different entry method, it's performance cannot be compared with classic keyboard solutions.

[3] `http://www.goqsoftware.com/`

[4] `http://www.inference.phy.cam.ac.uk/dasher/`

2. Theory

Theory in this section is based on textbook [2], artilce [4] and LM toolkit manual [13].

2.1 Statistical language models

For our purposes it's convenient to look upon a text as a "random" sequence of words, in the first approach similar to rolling a dice. For a fair dice and independent throws, all sequences are equally probable, however, it is not the same for the language. Different sequences have various probabilities due to the grammar rules, theme or style of the text.

Formally, for the vocabulary V of all considered words, we define a probability space $(\Omega, \mathcal{F}, P)$ where

- $\Omega = V^L$ are all word sequences of length L,
- $\mathcal{F} \subseteq 2^\Omega$ is a set of possible events forming a σ-algebra,
- $P : \Omega \to [0, 1]$ is a probability function (for given language).

Remember the probability of an event A under conditions of an event B is given by

$$P(A|B) = \frac{P(A \cap B)}{P(B)}, \quad \text{when} \quad P(B) > 0\,. \tag{2.1}$$

Therefore, we can formally express the probability of a word w coming after a sequence $h = (w_1, \ldots, w_{L-1})$ like $P(W|H)$ where

$$\begin{aligned} W &= \{(v_1, \ldots, v_L) \in \Omega \mid v_L = w\} \\ H &= \{(v_1, \ldots, v_L) \in \Omega \mid v_i = w_i, i \in \{1, \ldots, L-1\}\} \\ \text{i.e. } W \cap H &= \{(w_1, \ldots, w_L)\}\,. \end{aligned}$$

This notation is quite lengthy hence we'll use the following notation common in NLP texts

$$\begin{aligned} P(w|h) = P(w|w_1, \ldots, w_{L-1}) &\stackrel{\text{def}}{=} P(W|H) \\ P(hw) = P(w_1, \ldots, w_{L-1}w) &\stackrel{\text{def}}{=} P(H \cap W)\,. \end{aligned}$$

2.1.1 Ngram models

Theoretically, it would be great to use the whole history of known words to get the probability of the next word. Informally said, the influence of the words from the history declines as the distance from the last word rises. Furthermore, there's a practical need to estimate (section 2.1.2) the probability function and thus we would have to somehow store all possible histories, which is unfeasible even for relatively short histories as the needed space grows exponentially with the length of the history.

The arguments above lead to an idea that instead of the complete history, we would consider only a sliding length-limited part of it. Thus, $(n-1)^{\text{th}}$ order

Markov assumption, states that we can approximate the probability with only $n-1$ last words.

$$P(w|w_1,\ldots,w_{L-1}) \approx P(w|w_{L-n+1},\ldots,w_{L-1})$$

Such a model is called the n-gram model.[1] Optimal value of n depends on many factors, including training data size. Not always the greater is the better. As shows up in Goodman's report [6], for $n > 3$ in their tests, the quality of the model increase only slightly or even degrades (quality of the models is defined in the section 2.2).

2.1.2 Estimation

In the previous sections, we supposed that we somehow know what the probabilities of the word sequences are but in the reality, we need actual numbers.

One of the possibilities, is to use the knowledge of the language grammar and base model upon it. This would, however, require specialized model for every language and wouldn't be robust in the case of common but non-grammatical utterances (furthermore supposing that we really had some usable grammar).

The applicable methods reckon on statistical processing of the training set of texts.

Following sections use this notation:

- V for the set all considered words (even those out of the training set),
- N for total number of words in the training text,
- $C(w)$ for number of occurrences of the word w in the training set,
- $C(w_1,\ldots,w_n)$ for number of occurrences of the word sequence $w_1,\ldots,w_n$.

Maximum likelihood estimate

This method is most straightforward and fully utilizes the training data. In general, it assumes the probability function $P(w|h)$ with some set of parameters $\vec{\theta}$ and finds such values of $\vec{\theta}$ that maximize likelihood of observed (training) data (which explains method's name).

For the ngram model the parameters $\vec{\theta}$ would be the probabilities of individual ngrams.

Example Consider a unigram model with only two words w_1 and w_2, we only have a single parameter θ – probability of word w_1; for w_2, it's simply a complement $1-\theta$. The likelihood of training data is

$$P_\theta(v_1,\ldots,v_N) = \prod_{i=1}^{N} P_\theta(v_i) = \theta^{C(w_1)}(1-\theta)^{C(w_2)} = \tag{2.2}$$

$$\theta^{C(w_1)}(1-\theta)^{N-C(w_1)} \tag{2.3}$$

[1] When talking about concrete value of n, I use term n-gram. In other cases, when referring to consecutive sequence of word, I use ngram.

where we supposed independence[2] of single words in (2.2). The MLE value $\hat{\theta}$ can be easily found by taking a derivative of (2.3) with respect to θ, which yields

$$\hat{\theta} = \frac{C(w_1)}{N}\,.$$

Similarly, just with more complicated intermediate formulas, we would obtain the intuitive result that maximum likelihood estimate for a given event is ratio between occurrences of the event in the training data and all samples in the training data.

Due to MLE's strong orientation to training data, it has a shortcoming that it would assign zero probabilities to events unseen in training data. For unigram models this doesn't matter that much because there's only a few unseen words in large training data (relatively). However, for higher order ngram models, the space of possible ngrams grows exponentially and the model would suffer from training data sparseness. That's the reason why *smoothing* of the probabilities is used. Some probability mass is taken from the seen events (that is referred as *discounting* and is uniformly spread among those unseen).

Laplace smoothing

This method smooths the probability by increasing number of occurrences of all ngrams with a constant $0 < \lambda \leq 1$, thus unseen tokens also gain some probability (common value $\lambda = 1$ yields so called add-one smoothing). Generally, the smoothed probability is

$$P(w_1, \ldots, w_n) = \frac{C(w_1, \ldots, w_n) + \lambda}{N + \lambda |V|^n}\,.$$

Here it's important to point out that this smoothing assigns too much probability mass to unseen events, especially for $n > 1$, making it "more uniform". If we used this estimate for calculating conditional probability as in definition (2.1), we would get poor results, therefore it's used to smooth the conditional probability directly (fixing the conditioning history and smooth over V only).

$$P(w|h) = \frac{C(hw) + \lambda}{C(h) + |V|\lambda}$$

Still this method doesn't perform well (Chen's result [4], my result – table A.1).

Absolute discounting

To reduce probability mass of seen event, we subtract a small constant $0 < \delta <= 1$ from the count of the event. Conditional form of discounted probability is

$$P(w|h) = \frac{C(hw) - \delta}{C(h)} \qquad \text{for} \quad C(hw) > 0\,,$$

$$P(w|h) = \frac{\delta |W_h|}{C(h)|V \backslash W_h|} \qquad \text{for} \quad C(hw) = 0\,, \quad \text{where} \quad W_h = \{v \in V | C(hv) > 0\}\,.$$

[2]Training text of course doesn't consist of independent words but this is only unigram approximation.

Kneser-Ney discounting

Similarly to absolute discounting, counts of the highest order ngrams are discounted by a constant δ. Discount for lower order ngrams is different, their probability is estimated like

$$P(w|h) = \frac{N(\cdot hw)}{N(\cdot h\cdot)}$$

where $N(\cdot hw) = |\{v \in V | C(vhw) > 0\}|$, $N(\cdot h\cdot) = |\{(v_1 v_2) \in V^2 | C(v_1 h v_2) > 0\}|$, i.e. $N(\cdot hw)$ denotes number of words that are followed by given sequence and $N(\cdot h\cdot)$ are all word pairs that "adjoin" the sequence h.

Proper derivation of the lower order discount can be found in Chen's study [4].

There's also introduced a method referred to as the *modified Kneser-Ney discounting* that also stems from absolute discounting, however it uses three different values that are discounted, depending on the original count of the ngrams. It's considered to be the best, which was more or less confirmed (table A.1).

Good-Turing discounting

Good-Turing discounting method states, that for ngram with r occurrences in training data, we should consider it appeared $r^{\cdot}$ times with

$$r^{\cdot} = (r+1)\frac{n_{r+1}}{n_r}$$

where n_r denotes number of tokens that occurred r times during training.

This method requires $n_r > 0$ for all r, so in practice, counts greater than a certain constant k are considered reliably enough and aren't discounted with this method.

Other approaches of coping with $n_r > 0$ requirement and derivation of this method are mentioned in Chen's study [4].

2.1.3 Model combination

Presented models, so far, were based on ngrams of only one length. The higher order models provide better discrimination for known events, however, they would degrade to uniform distribution for unknown events (as implies the principle of discounted smoothing), which is obviously worse than lower order model that has more training data (relatively to the total size of the space).

Therefore, the techniques for model combinations are engaged. As we assign non-zero probability from lower order models to unknown events of higher order models, the model combination can be thought of as a form of smoothing.

Backing-off

When we use this way of combining models, we try to get the probability from the higher order model and when it has no data, we back off to the lower order model.

$$P_{\text{bo}}(w_n|w_1,\ldots,w_{n-1}) = \begin{cases} d_{hw_n}\dfrac{C(hw_n)}{C(h)} & \text{when} \quad C(hw_n) > 0 \\ \alpha_h P_{\text{bo}}(w_n|w_2,\ldots,w_{n-1}) & \text{when} \quad C(hw_n) = 0 \end{cases} \tag{2.4}$$

where h denotes the history $w_1, \ldots, w_{n-1}$, d_{hw_n} is the discounting coefficient for given ngram and α_h is back-off weight for history h.

It's important that we employ a discounting method, otherwise there wouldn't be any probability mass left for unknown words, enforcing $\alpha_h = 0$. Actually, the value of the back-off weight α_h has to be chosen such that resulting the distribution is normalized to one. That gives us

$$\alpha_h = \frac{1 - \sum\limits_{w \in W_h} d_{hw} \frac{C(hw)}{C(h)}}{\sum\limits_{w \in V \setminus W_h} P_{\text{bo}}(w|w_2, \ldots, w_{n-1})} = \frac{1 - \sum\limits_{w \in W_h} d_{hw} \frac{C(hw)}{C(h)}}{1 - \sum\limits_{w \in W_h} P_{\text{bo}}(w|w_2, \ldots, w_{n-1})}$$

where $W_h = \{w \in V | C(hw) > 0\}$.

Linear interpolation

In the greatest generality, linear interpolation of models $P_1, \ldots, P_n$ is

$$P_{\text{int}}(w|h) = \sum_{i=1}^{n} \lambda_i(h) P_i(w|h)$$

where $0 \leq \lambda_i(h) \leq 1$ are linear combination coefficient and $\sum_{i=1}^{n} \lambda_i(h) = 1$ so the combined function is valid probability distribution.

Here, it's useful to show that such a linear combination can be rewritten into recursive form

$$\begin{aligned} P_{\text{int}}(w|h) = P^n_{\text{int}}(w|h) &= \sum_{i=1}^{n} \lambda_i(h) P_i(w|h) \\ P^n_{\text{int}}(w|h) &= \lambda'_n(h) P_n(w|h) + (1 - \lambda'_n(h)) P^{n-1}_{\text{int}}(w|h). \end{aligned} \tag{2.5}$$

We used a new set of coefficients $\lambda'_1(h), \ldots, \lambda'_n(h)$ that are related to the original coefficients by formula

$$\lambda_i(h) = \lambda'_i(h) \prod_{j=1}^{n-i} (1 - \lambda'_{n-j+1}(h)) \,.$$

The expression (2.5) is formally similar to the backing-off expression (2.4), which suggests what the difference between interpolation and back-off is. When we use back-off models, the estimation from lower order models is used *only if* the higher order model has no data, while interpolated models use the lower order estimates every time.

The linear interpolation may not be used only for different order ngram models but also for various language models in general. A simpler variant of the linear interpolation uses constant weights for all models throughout the history.

$$P_{\text{int}}(w|h) = \sum_{i=1}^{n} \lambda_i P_i(w|h)$$

EM algorithm A specialized variant of expectation-maximization algorithm might be used to find optimal weights for combined language models. It's an iterative algorithm that searches for global optimum of linear coefficients with respect to the probability[3] of the training data according to the model.

Briefly said, training data should be different from those used for estimating the models, to avoid *overtraining*. Description of the general EM algorithm can be found in the textbook [2].

```
// initial weights
λ_1, ..., λ_n ← uniform;
// stop iterating if change is less than
ε = 1 · 10^−4;
δ ← −∞;
while −δ > ε do
    H ← −Σ_{i=1}^{N} log_2 (Σ_{k=1}^{n} λ_k P_k(w_i|h_i));        // original entropy
    for i ← 1 to N do
        c_j ← c_j + P_j(w_i|h_i) / Σ_{k=1}^{n} λ_k P_k(w_i|h_i)   ∀j ; // contribution of j-th model
    end
    c_j ← c_j/N   ∀j;
    H' ← −Σ_{i=1}^{N} log_2 (Σ_{k=1}^{n} c_k P_k(w_i|h_i));      // modified entropy
    δ = (H' − H)/H';                                              // relative change
    λ_j ← c_j   ∀j;                                               // set new weights
end
```

Algorithm 1: EM algorithm for linear coefficients optimization

2.1.4 Trigger based models

Trigger based models are designed to capture long distance dependency between words. Instead of the context of several preceding words like in the case of ngrams, they look further in the history for *triggering* word a and if present, they assign higher probability to *triggered* word b.

Potentially, there are $|V|^2$ trigger pairs. Tillman [7] describes few methods for choosing the significant pairs. Generally, they are based on the perplexity improvement of a baseline model. I experimentally implemented[4] a method referred as low-level triggers in Tillman's article [7], however, time-demanding training phase of my own implementation and not so persuasive perplexity improvement results reported in articles [7, 8] in comparison with cached model, led me to reject this idea and use cached models with better ratio of implementation effort and prospected results.

2.1.5 Cached models

Another way of long dependency estimation are cached language models. Moreover, they also provide means of text topic adaptation and better perplexity improvements for free word order languages as mentioned in the article [9].

[3] In the algorithm, the probability is decomposed into sum of their logarithms, which effectively minimizes cross entropy (defined in 2.2) of the model over training data.

[4] The mentioned code isn't a part of the final work.

The simplest cached model is basically an ngram model trained on several last words whose count is referred as the cache size. Due to the extreme sparseness of the training data, cached model are used interpolated with another model trained on larger data and thus smoothing the overall estimate.

Because of the combination with other models, the cached model that was implemented isn't smoothed and returns simply MLE values.

2.2 Language model evaluation

Entropy of random a variable

In the information theory, entropy of a (discrete) random variable X with distribution $p(x)$ is defined as

$$H(X) = \mathbb{E}(-\log p(X)) = \sum_x -p(x) \log p(x)$$

where the base of the logarithm is usually two in order to get the entropy in bits.

In the section 2.1, we defined probability space for word sequences of fixed length, however, we can extend this definition to all finite length sequences and assign them natural numbers,[5] therefore defining a random variable for the language.

The entropy of the language L (with distribution $p(x)$), can be thought of as a mean entropy per word in a very long sequence of the language L. Formally, it's a limit

$$H(L) = -\lim_{n \to +\infty} \frac{1}{n} \log p(w_1, \ldots, w_n)\,. \tag{2.6}$$

Cross entropy

If we knew true distribution for the language L, we could calculate its entropy directly. However, that's not the real case and so we define the cross entropy between random variable X (with distribution $p(x)$) and another distribution $m(x)$.

$$H(X, m) = \sum_x -p(x) \log m(x)\,,$$

analogously to (2.6), we define cross entropy between a language L (again with distribution $p(x)$) and another distribution $m(x)$.

$$H(L, m) = -\lim_{n \to +\infty} \frac{1}{n} \log p(w_1, \ldots, w_n)\,. \tag{2.7}$$

As argued in the textbook [2], not only we can take a finite length sequence of language L to approximate (2.7) but also the cross entropy serves as an upper bound for the actual entropy of the language.

[5] For example, we can consider sequences as numbers written with "digits" V in $|V|$-ary numeral system.

Therefore, if we have a model that gives probabilities $m(w_i|h_i)$ to word w_i in the context h_i, we calculate its cross entropy as

$$\begin{aligned} H =& \frac{-1}{n} \log m(w_1, \ldots, w_n) = \frac{-1}{n} \log \prod_{i=1}^{n} m(w_i|h_i) = \\ =& \frac{-1}{n} \sum_{i=1}^{n} \log m(w_i|h_i) \,. \end{aligned} \tag{2.8}$$

(For practical convenience, if $\exists i : m(w_i|h_i) = 0$, we put $H = +\infty$.)

Thus, the lower the cross entropy is, the closer to the actual entropy of the language it is and the model is considered better.

Perplexity

Just for convenience and conventions in NLP, we define perplexity for the cross entropy H as

$$\text{ppl} = 2^H \,.$$

3. Implementation

3.1 Architecture of the system

The structure of the system is influenced mainly by these factors:

- usage of language models trained on large data,
- cooperation with graphics user interface (GUI) component,
- wide configuration options and automated evaluation of the system.

These requirements led to dividing the system into four functional parts as shown in figure 3.1.

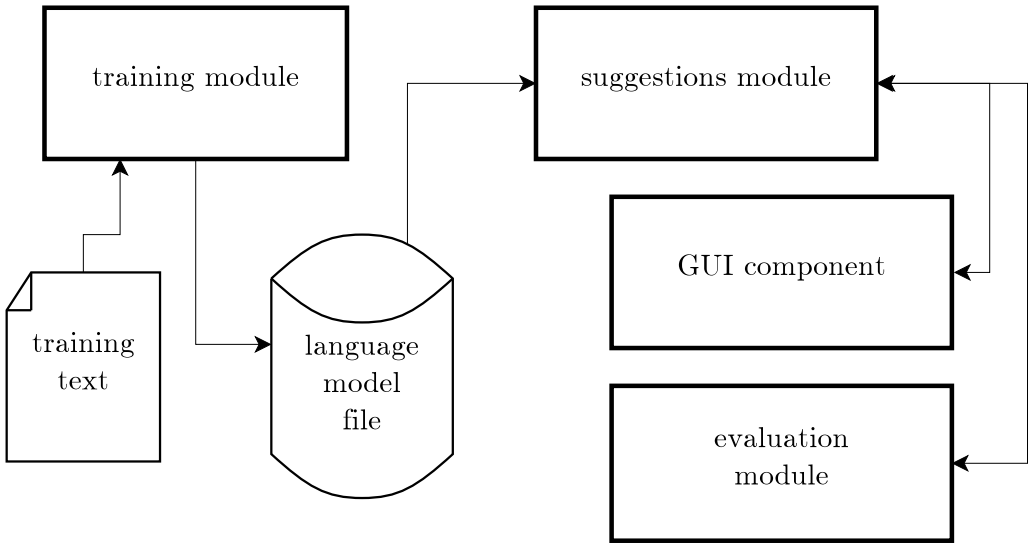


Figure 3.1: Main functional parts of the system

The heart of the system is referred to as the *suggestions module*. The main task of this part is to interpret data from language models, taking into account the user input and then search and filter potential suggestions that would match the user's intentions.

The part, that potential user would be in direct touch with, is the *GUI component*. Its goal is quite simple – accept input from the user, show him suggested continuations of the text and allow him unobtrusive acceptance of the suggestions.

The size of the language models and the fact that their training would take substantially longer time than usual waiting time of an user together with the possibility of storing once trained models – those were main reasons why the *training part* was made separate part of the system. The output of this part are trained models that are persistently stored on a disk, ready for later use by the suggestions module.

Finally, the *evaluation component* exists on its own in order to have the possibility objectively and automatically evaluate the system. It allows testing the suggestions under different conditions and formally, its interface is similar to that of the GUI component (therefore it could test what the real user would see).

3.1.1 Programming language considerations

Most (if not all) the system was intended to be written in a scripting language in order to allow quick testing of new ideas and allow easy extensibility with new features.

Finally, the I've chosen the Python programming language[1] with the exception of the parts where the time and specially memory were concern, there was used C++.

Mainly for its transparent support of Unicode strings and cleaner syntax, the Python used is Python 3.2.

Among minor reasons was also desire for usage of last, most developed version. Although, the first version of Python 3 was released in 2008, the third party libraries (PyQt4) for Python 3 are partly delayed and they need to be compiled from sources. In more detail, this is described in the installation section C.1.

Some parts of the code also rely on the UNIX utilities `make`, `head` and `ed`.

3.2 Training module

Briefly, the purpose of this module is to take the training text, train the language model from it (section 2.1.2) and store it persistently to the hard drive for later use.

3.2.1 Storing models

In the beginning, there was a decision what exactly should be stored in the trained model. One possibility was to store only the observed counts of ngrams. The advantage would be that only a single file would exist for various smoothing techniques. On the other hand, some concerned smoothing methods (Good-Turing, Kneser-Ney) require additional calculation based on the counts of ngrams that would have to be done every time model was loaded. The second possibility was to store directly smoothed probabilities. The arguments for this option are:

- model is fully estimated after load,
- the need for a single file for every smoothing method is bad for testing purposes, however, in the real life, one carefully chosen model that loads faster is better than one model that needs extra time during loading,
- there exist de facto standard ARPA files for storing language models.

In order to avoid reinventing the wheel, the trained model is therefore stored in the ARPA file format whose specification is in the SRI LM manual [12].

The ARPA files are designed to store back-off models (they contain probability and back-off weight for each ngram), however, as the relation (2.5) suggests, models interpolated from different ngram orders can be stored as well.

[1] `http://www.python.org/`

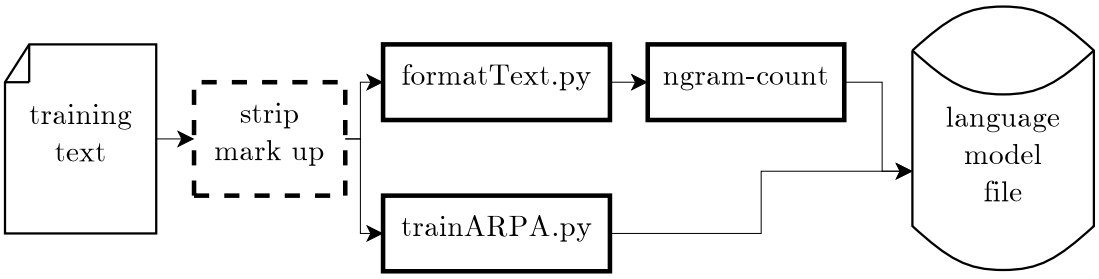


Figure 3.2: Tools in the training module

3.2.2 Input text

Expected format of the text for the model training is a UTF-8 encoded plain text file. The text should be free of any mark-up, white space character other than those necessary to separate words are ignored.

3.2.3 Tokenization

The first step of the text processing is dividing it into tokens, the tokenizer is simple, principle was loosely inspired by `flex`[2] where tokenization is realized by regular expression matching with given priority (when more patterns match, the one with higher priority is used).

In our case, token types are (with descending priority):

1. white space,
2. number,
3. word,
4. emoticon,
5. delimiter,
6. other,
7. (special tokens inserted during processing).

Example The following text (last sentence is intentionally incomplete):

`The Proto-Sinaitic script eventually developed into the Phoenician alphabet, which is conventionally called "Proto-Canaanite" before ca. 1050 BCE. The South Arabian alphabet, a sister script`

is split into tokens (without whites space tokens):

$\texttt{The}_3$, $\texttt{Proto}_3$, $\texttt{-}_5$, $\texttt{Sinaitic}_3$, $\texttt{script}_3$, $\texttt{eventually}_3$, $\texttt{developed}_3$, $\texttt{into}_3$, $\texttt{the}_3$, $\texttt{Phoenician}_3$, $\texttt{alphabet}_3$, $\texttt{,}_5$, $\texttt{which}_3$, $\texttt{is}_3$, $\texttt{conventionally}_3$, $\texttt{called}_3$, $\texttt{"}_5$, $\texttt{Proto}_3$, $\texttt{-}_5$, $\texttt{Canaanite}_3$, $\texttt{"}_5$, $\texttt{before}_3$, $\texttt{ca}_3$, $\texttt{.}_5$, 1050_2, $\texttt{BCE}_3$, $\texttt{.}_5$, $\texttt{The}_3$, $\texttt{South}_3$, $\texttt{Arabian}_3$, $\texttt{alphabet}_3$, $\texttt{,}_5$, $\texttt{a}_3$, $\texttt{sister}_3$, $\texttt{script}_3$

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

3.2.4 Sentence boundary

The sequence of tokens is further split into a sequence of sentences. The method is quite simple, fixed set of delimiter tokens[3] is considered an end sentence marker, so after those delimiters[4] sentence boundary is searched. Then it's set after the first non-delimiter token or directly after the marking token, when no delimiters follow.

Abbreviations

As the period symbol occurs often after an abbreviation in the middle of a sentence, a static list of abbreviations is used to discard such periods as end sentence markers.

The list of abbreviations is compiled from the training text and it takes into account values $\pi(w)$ and $\pi_{\mathrm{C}}(w)$ where $\pi(w)$ is a ratio of occurrences of word w that were followed by a period and $\pi_{\mathrm{C}}(w)$ is a ratio of occurrences of word w that were followed by a period and a capital letter (a potential end of sentence). The word w is considered an abbreviation when $\pi > t$ or $1 - \pi_{\mathrm{C}}/\pi_{\cdot} > t_{\mathrm{C}}$. The thresholds were chosen empirically as $t = 0.8$ and $t_{\mathrm{C}} = 0.5$.

Note on case sensitivity

The system is designed to be case sensitive in order to suggest proper names with capitalized first letter. This is referred as the system is working with *truecased* words. We employ a simple truecasing method when all the first words in a sentence are lower cased on the first letter. Therefore, we silently assume there are more proper names in the middle of a sentence than on a beginning.

Example Previous text split into sentences (note: incompleteness of the last sentence indicated by missing `</s>` token, truecasing and abbreviation `ca`):

($\texttt{the}_3$, $\texttt{Proto}_3$, $\texttt{-}_5$, $\texttt{Sinaitic}_3$, $\texttt{script}_3$, $\texttt{eventually}_3$, $\texttt{developed}_3$, $\texttt{into}_3$, $\texttt{the}_3$, $\texttt{Phoenician}_3$, $\texttt{alphabet}_3$, $\texttt{,}_5$, $\texttt{which}_3$, $\texttt{is}_3$, $\texttt{conventionally}_3$, $\texttt{called}_3$, $\texttt{"}_5$, $\texttt{Proto}_3$, $\texttt{-}_5$, $\texttt{Canaanite}_3$, $\texttt{"}_5$, $\texttt{before}_3$, $\texttt{ca}_3$, $\texttt{.}_5$, 1050_2, $\texttt{BCE}_3$, $\texttt{.}_5$, $\texttt{</s>}_7$) ($\texttt{the}_3$, $\texttt{South}_3$, $\texttt{Arabian}_3$, $\texttt{alphabet}_3$, $\texttt{,}_5$, $\texttt{a}_3$, $\texttt{sister}_3$, $\texttt{script}_3$)

3.2.5 Token normalization

The last step of the tokenization process is token normalization, when the sequence of sentences is converted to simple sequence of words inserting special words for beginning and end of a sentence.

During this process tokens can be also mapped to special words. Only mapping of numbers to a special word was engaged.

Example Normalized form of the previous sentence sequence (note the special `<num>` word):

`<s>`, `the`, `Proto`, `-`, `Sinaitic`, `script`, `eventually`, `developed`, `into`, `the`, `Phoenician`, `alphabet`, `,`, `which`, `is`, `conventionally`, `called`, `"`, `Proto`, `-`,

[3] In this project the set was always: `.`, `!`, `?`, `:` and `...`.

[4] With an exception of abbreviations, discussed below.

`Canaanite`, `"`, `before`, `ca`, `.`, `<num>`, `BCE`, `.`, `</s>`, `<s>`, `the`, `South`, `Arabian`, `alphabet`, `,`, `a`, `sister`, `script`

3.2.6 Model estimation

SRI LM toolkit

The language model estimation is performed by a utility `ngram-count` from the SRI language modeling toolkit [12]. It supports various ngram orders and smoothing methods [13],among them those discussed in section 2.1.2.

The utility expects input as a text file, with one sentence per row and words in a sentence are space separated. This file is created from the original text file after processing by own tokenization chain. The output of `ngram-count` is the estimated language model in the form of an ARPA file.

Own implementation

Own training utility was also implemented, it's intended primarily for the users that don't have SRI LM toolkit installed. Due to its Python implementation, it cannot fully substitute the SRI LM that performs much better on large data. Therefore, it's meant rather for training smaller models from user data.

It supports absolute and Good-Turing discounting for back-off models.

3.3 Suggestions module

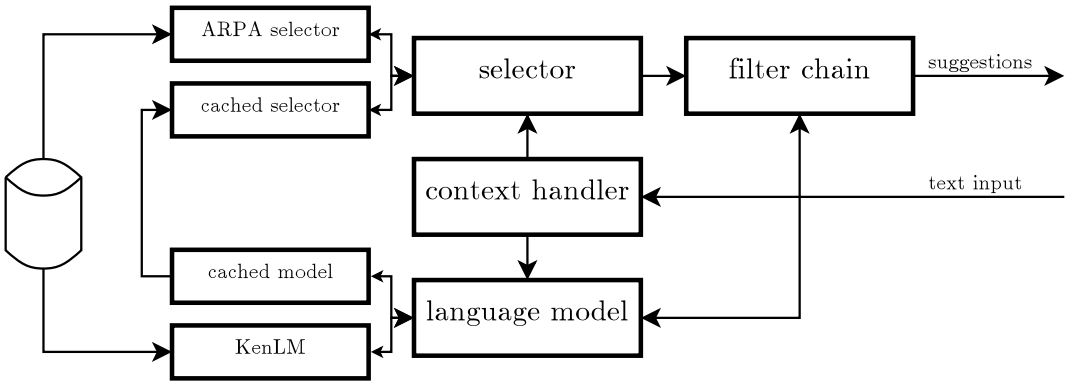


Figure 3.3: Components comprising suggestions module (in ngram + cached model configuration)

At the input of this part is the typed text and the output is the list of current suggestions.

Simply explained, context handler processes input text, selector chooses candidates for suggestions that go through the filter chain where only the best candidates are selected with the help of the language model and these suggestions are returned to the user.

3.3.1 Context handler

When the suggestions should be displayed, they are expected to match the current typing context. That context of course affects language model as well as the

selector results, however for the same context they results are the same (here, we mean by context all the text typed from beginning of the document). Suppose such objects have a state for the null context and each subsequent word of a longer context can change their state. Let's call them state-sensitive objects, as their answer depends on their state.

Basically, there are two opposite methods how to keep state-sensitive objects synchronized with the typing context:

- let them change their state automatically after each word,
- with every word provide them also the full context to derive their state from it (although deriving state from the whole context could be demanding).

The first method would be fully effective if the user inserted text word by word and never moved back. The second method would find its great use if the user only moved back. However, in reality user is supposed to do both the actions and that's why the context handler comes into play.

The context handler watches what the user types and if he is just adding new words (or moving forward over already typed text), it exploits state-sensitive objects' ability to change their state with a single word and when user moves back or does some editing operations, the state of the state-sensitive objects is derived from the full context.

3.3.2 Language model

Language model is the key part of the suggestions module. It assigns probabilities conditioned by the current context to the candidate suggestions in order to show the most probable suggestions on the top places of the list and thus reducing amount of characters user has to type. There are three kinds of language models that can be used separately or together depending on the chosen configuration.

(Large) ngram model

N-gram model accesses the data previously stored in the ARPA file and returns probabilities based on the training data. As the intention here is to use models trained on large data, this part was not written in Python, mainly due to the memory consumption. Third party C++ library KenLM[5] was chosen for the following reasons:

- it supports models stored in ARPA file format,
- despite brief documentation, its API is simple to use,
- it's optimized both for the time and the memory efficiency,
- it has permissive licence.

Only a Python module wrapping the library functionality was written.[6]

[5] `http://kheafield.com/code/kenlm/`

[6] The library is in active development and so in the last version, it includes its own Python wrapper. However, as it didn't exist at the time we needed it, we used the own implementation.

Cached model

As mentioned in section 2.1.5, the cached model is a simple way how to adapt to current topic. Because of the low length of the input texts (typically it's expected to be a single document), we used a unigram cache with sliding history "window" of fixed length.

Linear interpolated model

This model provides a manner how to aggregate data from more models, taking a weighted mean of probabilities from combined models (section 2.1.3). Weights are constant with respect to the context and they can be set manually or (if some training data exists) estimated via EM algorithm (algorithm 1).

3.3.3 Selector

Firstly, let's define two notions related to this topic.

Word completion Word completion is a process when we search a word with given prefix that would best match the current context.

Word prediction Word prediction is a process when we search a word that would best match given context. The prediction can use some hints like keystrokes on a reduced keyboard, simplified movements on a touch screen keyboard, word prefix etc. Therefore, by these definitions, word prediction is generalized word completion.

It would appear that without any hints, everything we need to predict a word is just to take all words in the dictionary, score them for the current context with the language model and top rated words would be the predictions. This approach would work for small dictionaries, however, usage of large training data leads to dictionaries with hundreds of thousands words, which is impossible to rate in user comfortable time.[7]

Thus, described word prediction requires pruning of the dictionary prior "consulting" the language model. Prediction hints mentioned above can serve as the pruning criterion.

The implementation in this work, that is oriented towards common computer keyboards, employs a word prefix as the pruning condition. Still, the goal of minimizing number of keystrokes needs the prediction to be based on no or very short prefix. But such a prefix isn't very effective for pruning hence only remaining way of limiting the search is using the context.

We would like to test only such words that occur in the context and best way to do this is to query the language model but again we would have to test each word in the dictionary.

Note on random text generators

The problem of choosing best matching word for the current context is solved also in ngram random text generators. I've examined solutions in SRI LM[8] and

[7] This is based on an experimental Python implementation.

[8] `ngram -lm <ARPA file> -gen 10` generates 10 random sentences.

Python NLTK.[9]

SRI LM calculates probability for each word of the dictionary for given context and then randomly chooses one (distribution is based on the probabilities).

Python NLTK uses an ngram *trie* where each node is a hashing table which maps a word to node. This implies that only the words that were seen in the given context are being evaluated in the LM.

The solution from Python NLTK appears to break the vicious circle problem we've run into. Unfortunately, it's unusable for large language models due to the memory overhead of Python data structures.

The KenLM library internally use a trie structure to evaluate the probabilities,[10] however ngrams are stored there in reverse order that's more appropriate for back-off model evaluation. (Example: for the query $P(w_3|w_1w_2)$ the stored ngram is w_3, w_2, w_1.)

3.3.4 ARPA selector

Final solution incorporates a trie to prune the dictionary with the context. Reusing the trie structure code from the KenLM library was considered, however because of it's optimization for back-off models and lack of documentation, it was rejected in favor of an own C++ implementation.

Trie structure

word index	word	offset
0	`God`	0
1	`home`	1
2	`my`	1
3	`save`	2
	—	2

offset	word index
0	3
1	0
2	1

Figure 3.4: Simple one-level trie with bigrams: `God save`, `my God` and `my home`. (Note that `home` and `save` have no continuations and they point to an empty node.)

As pruning with a context of single word was shown sufficient (section 4.3.1), only simple one-level trie was implemented (example in figure 3.4). Instead of storing whole strings, only word indices are used and auxiliary table of words is kept with string representation. Word indices within a node are sorted lexicographically with respect to their referred word[11] so binary search with $\mathcal{O}(\log n)$ time can be performed, n being number of words in the node.

[9] `http://nltk.org/`, method `nltk.model.ngram.NgramModel.generate`

[10] To be more specific, it's one of the possibilities. KenLM can also use a hashing table where the keys are whole ngrams. Because of the better memory efficiency [10], this project use the trie variant.

[11] Because the word indices are from the sorted auxiliary table, the lexicographical ordering is the same as ordering of the indices.

Creating a selector

Data for the selector are loaded from an ARPA file (some of the loading routines are reused from KenLM). In the case the ARPA file is built only for a unigram model, the selector is loaded successfully (only auxiliary table) but cannot perform context-pruned searches.

When loading from an ARPA file, it can be specified that only bigrams with conditional probability higher than certain value should be loaded.

As parsing the ARPA file is time consuming, selector can be stored in a binary representation and repeatedly loaded faster from that.

Quering a selector

ARPA selector is meant to provide prefix searches in a whole dictionary as well as in the context-pruned dictionary.

Prefix search exploits the ordering within the nodes and it's basically a binary search; the key for each word are only the first k characters of the word where k is a length of the prefix. So the last occurrence of the key found and as all the words with the same prefix are stored consecutively, they are easily returned.

Limits on the result In the cases when the context and/or prefix don't prune the dictionary sufficiently, selector returns an empty set. The decision is based on the number of suggestions that *would* be returned if their count didn't exceed the preset limit.

This method reduces the number of queries to the language model in situations when there are many possible suggestions and supposedly none of them would have high probability.

3.3.5 Combining selector

Output from more selectors can be taken into account, usually, there's one selector per language model. Combining selector exists for that purpose and its output is set union of combined selectors.

3.3.6 Filter chain

Before the words suggested by the selector get to the user, they go through the *filter chain*. The filter can:

- leave out some suggestions,
- reorder the suggestions,
- map suggestion to different one or append some metadata.

Such notion of a filter is general enough and still it provides enough space for configuration. Bellow are listed implemented filters.

Probability filter

This filter just appends information of conditional probability for each suggested word.

Sorting filter

This filter sorts items in descending probability order. The secondary criterion for equally probable words is length of the suggestion.

Suggestions count limiter

The purpose of this filter is to reduce the number of shown suggestions in order to eliminate additional cognitive load on the user. The list can be cropped either by its size or by the minimal probability of the last word or by both.

Prefix conditioning When cropping with minimal probability, the probability value obtained from the language model is valid only for en empty prefix. As the user provides longer prefix, the conditional probability rises.

$$P(w|h \cap p) = \frac{P(w \cap h \cap p)}{P(h \cap p)} = \frac{P(w \cap h \cap p)}{P(h)P(p|h)} = \frac{P(w \cap p|h)}{P(p|h)} \geq P(w|h) \qquad (3.1)$$

In the inequality (3.1), the p denotes an event of word having a prefix p, thus $P(w \cap p|h) = P(w|h)$ if p is prefix of w and $P(w \cap p|h) = 0$ otherwise.

Proper functioning of minimal probability cropping requires scaling the probabilities from the language model by factor $1/P(p|h)$ which is estimated like

$$P(p|h) = \sum_{w \in W_{hp}} P(w|h)$$

where W_{hp} should be all words with prefix p. In practice, W_{hp} are all words that enter suggestions count filter.

Suffix aggregation filter

Motivation for this filter are inflected languages where many words share the same stem and differ only with their suffix. Typical use case should be when a user intents to write one of the suffixed variant but it has low probability and isn't offered. Therefore, the suggestions with the same stem are grouped into one with sum of the probabilities for the variants. This aggregated suggestion has higher chance to be shown to the user and consequently stemmed suggestion is handled differently by GUI component (section 3.4.3).

As the system is designed to be language independent, only very simple method for stemming was implemented – last k characters of a word are considered a suffix and the remaining part a stem.

However, it can happen that language model assigns substantially higher probability to some of the suffixed variants, such variant are detected and aren't included in the group. The detection method checks the variance of the log probabilities within a group and if it exceed certain thresholds t, the first e most probable variants are excluded from the group.

Another event that can occur is when the stem is also valid word. Such a word is of course part of the group but decision has to be made whether such a group is passed to the GUI as a stem or a word. Grouped suggestion is considered a word iff the probability assigned to that word is greater than half the probability of the whole group.

Capitalization filter

As the system internally works with truecased words, reverse truecasing must be done when returning suggestions. Due to the simple truecasing method, all suggestions following a beginning of a sentence are capitalized. Note that this doesn't imply that all suggestions following a period are capitalized (see section 3.2.4).

Punctuation filter

By default punctuation signs are considered a word by the system, however, because most of them are only in character long and have quite high unigram probability, they are omitted from the suggestion list to reduce distraction of the user.

Added characters filter

This filter is also motivated by reduction of the user cognitive load. It accepts only such suggestions whose utilization by the user would be more that certain number of characters (e.g. it won't suggest word `probability` when `probabilit` is already typed).

Advantage of this filter is that it doesn't use word probabilities hence it can be put before the probability filter in the chain and decrease the number of queries to the language model.

3.4 GUI component

Most of GUI components for assisted typing can be generalized into a form where three tasks have to be done

- display and navigate in the already typed text,
- provide user a way to "express" his intention,
- show the suggestions and allow to choose any of them.

Display the text and navigate in it

The displayed text can be variously modified, for example:

- typographic formatting (WYSIWYG editors),
- improved readability (syntax highlighting, word wrapping),
- highlighted spelling mistakes etc.

However, any of those methods neither reduce the number of keystrokes needed during typing nor increase typist speed (at least not directly).

Navigation in the text using cursor keys, various keyboard shortcuts, mouse or touchscreen can help to faster editing but for continuously typed text, they don't matter that much too.

Reading user intention

The most simple way is just an ordinary (QWERTY) keyboard where user types the text character-wise (effectively expressing his intentions with a word prefix). On the other hand, the input interfaces of mobile devices don't provide such space and various methods of compressed input are employed, thus inherently needing a language model.

Usually, a reduced numeric keyboard is used where a single key is mapped to more characters. Another methods include: simplified moves over touch screen,[12] handwritten text,[13] voice input[14] or insertion of compressed text [5].

Displaying and choosing suggestions

Most common variants include:

- inline displaying the most probable suggestion, alternatively skipping to another suggestion with a single keypress (DOS command line tab completion),
- displaying whole list of suggestions (e.g. NetBeans IDE, Google search suggestions) and cursor keys selection,
- display suggestions as buttons on a touchscreen accepting suggestion with a single "tap" [14].

3.4.1 Requirements

As it was not a main goal of this project to implement a GUI component that would provide all aforementioned features, the following was kept in mind.

- Experimental nature, thus direct cooperation with the implemented suggestions module.
- Targeted on common computer users, potentially users with disabilities (however, still able to use classic QWERTY keyboard).
  - Though, because of "more comfortable" development, target platform is GNU/Linux OS.
  - No special methods for reading user intentions but prefix.
- Encapsulation of the component, allowing its further integration.
- Editing of plain text only.
- Displaying debug/testing information.
- Inherit behavior of standard text editing component.
- Minimize number of keystrokes.

[12] `http://www.swype.com/`

[13] `http://www.phatware.com/index.php?q=product/details/writepad`

[14] `http://support.google.com/mobile/bin/answer.py?hl=en&topic=27213&answer=168450#1187415`

3.4.2 Technology

As the base for the GUI component was chosen multiplatform C++ library Qt[15] which provides a great variety of classes for GUI development.

QCompleter

The Qt library itself contains a class `QCompleter` that performs word completion.[16] Usage of this class, however, is for "single word" input fields or comboboxes and completion is realized with rather static dictionary.

Inheriting this class was due to its different purpose rejected.

KDE KatePart

KDE is a desktop environment for X Window System built on Qt comprising comprehensive set of programs and libraries.[17] One of them is a text editor component KatePart that itself has simple word completion abilities (mentioned in 1) and allows their customization by inheritance of specified classes. Key classes for completion are:

- `KTextEditor::CodeCompletionModel` – class that provides actual suggestions,
- `KTextEditor::CodeCompletionModelControllerInterface` – class that controls when suggestion list is displayed, what context is used,
- `KTextEditor::CodeCompletionInterface` – extension class that adds completion behavior to the editor component.

Building the GUI component as a KatePart plugin seemed a promising variant. On the other hand, its complex (although general) API and existing Python code for suggestions module led that also this solution was rejected in favor of the following variant.

PyQt4

PyQt4 is a binding of the Qt library for Python.[18] It enables transparent use of the Qt classes from Python code. Because of the reasons mentioned above, GUI component was implemented in Python as a descendant of `QtGui.QPlainTextEdit` class.

In spite of the arguments favoring PyQt4, there are also some negatives.

One of them are limited opportunities for integration of the component as only applications written in Python with PyQt4 GUI can take advantage of it. Unfortunately, there isn't much (widely used) software that would use PyQt4.[19]

Moreover, combination of Python 3.2 and PyQt4 often requires a manual installation as discussed in section C.1.

[15] `http://qt.nokia.com/`
[16] `http://doc.qt.nokia.com/4.7-snapshot/qcompleter.html`
[17] `http://www.kde.org/`
[18] `http://www.riverbankcomputing.co.uk/software/pyqt/download`
[19] `http://wiki.python.org/moin/PyQt`

3.4.3 Own implementation

GUI component is based on multiline plain text edit component and as such it should behave in the case of ordinary typing, i.e. when using no suggestions. The typed characters are directly displayed because using the spacebar or the return key for accepting suggestions would be too obtrusive when the user wants to type a word that is a prefix of a suggestion. Also the suggestion list is displayed only when there are any suggestions available and their amount depends on the settings of the filter chain (section 3.3.6).

On the other hand, when user decides to take advantage of the offered suggestions, the component tries to minimize the number of keystrokes as possible. (The user's intention to use the suggestions is detected by pressing keys that are not used for standard typing – function keys and cursor keys.) Therefore, space characters are automatically appended after each confirmed suggestion,[20] however, punctuation characters that shouldn't follow the space are transparently inserted before it. Still, the user's intention was favored, so when user actually types space followed by a punctuation mark, it's kept as is.

3.5 Evaluation

As one of the goals of this project was to test various methods of word prediction, there was also need to employ some means how to objectively compare performance of chosen approaches.

3.5.1 Evaluation system

A Python script was written to evaluate the system performance with various metrics over testing text. This script can test multiple text files with multiple metrics at one time. However, to keep the script simple, single run is needed for each tested configuration. Therefore, `make` utility was used to manage multiple tests and preparing testing data. In more detail this is described in section C.4.2.

Actual results of the evaluation are described in section 4.3.

3.5.2 Metrics

Perplexity

Common method (e.g. article [4] mentions it) how to test performance of language models is use of perplexity that's derived from cross entropy (section 2.2).

But as implies definition (2.8) in section 2.2, we could easily an infinite obtain entropy.[21]

There are two methods how to cope this issue. Firstly, smoothed models assign a non-zero probability even to out of vocabulary (OOV) words and so avoiding high entropy. Secondly, OOV events can be detected and such events are completely ignored in the result (this approach is used also in SRI LM).[22]

[20] If it's not a partial suggestion (section 3.3.6), otherwise continuation is supposed and space doesn't follow.

[21] Infinite values occurs as a consequence of logarithming a zero probability. Technically, $\log_2 0$ was substituted -100 thus entropy/perplexity is always a real (digitally represented) number.

[22] `http://www.speech.sri.com/projects/srilm/manpages/srilm-faq.7.html`

Keyboard metrics

Because of the project's purpose to test the influence on the typing rate and because language models are only a part of the chain, two metrics that would measure typing improvements were designed.

Keystrokes per character This metric measures potential keystrokes savings on QWERTY keyboard. It emulates a user who searches the suggestions list before hitting any key and if there exists valid suggestion, he uses it (effectively with a single keystroke according to section 3.4.3) otherwise he types the next character of the word and repeats the aforementioned decision.

The numeric result of this metric is the average number of keystrokes per character,

$$\frac{\text{overall keystroke count}}{\text{overall character count}}.$$

Mean suggestion position Despite the possibility of choosing a suggestion with a single keystroke, the suggestions are displayed in a list which the user is supposed search linearly. This metric measures the average position of a valid suggestion when it appears for the first time during typing a word.

Time performance

The system is required to react promptly in order to to be usable during typing. On the nowadays machines, this is satisfied (mainly thanks to appropriate dictionary pruning considered in section 3.3.4). However, during automated evaluation was observed that time performance can surprisingly vary (table A.9) and so this metric was also incorporated.

3.5.3 User study

Evaluating the system with the real users under real conditions can provide invaluable results for further development (as presents Trnka [1]). The keystrokes savings metric can only approximate real improvement in the input rate as using word suggestions increases cognitive load on users. The used metrics also cannot capture potentially decreased error rate while typing with built-in dictionary.

Unfortunately due to the lack of time and "human resources", the system wasn't tested with real users.

4. Experiments

4.1 Experimental corpora

For training of the large models were used corpora of Wikipedia texts coming from Web to Corpus project [11]. These texts consist of the Wikipedia articles and any mark up is stripped off.[1] Specifically, two languages were examined – Czech and English.

Texts from the 25$^{\text{th}}$ year of FYKOS (physical competition[2]) were used to test adaptation to user supplied data. The texts are popular physics problems and their solutions with minor parts written in Slovak. The texts were marked up with LaTeX that was completely removed (including mathematical formulas, section C.5).

For the purposes of the evaluation, data were split into four (disjoint) parts:

- training,
- held-out part for EM algorithm (in section 2.1.3),
- testing,
- debug.

Size of the testing data was chosen regarding the small size of the potential real data (single messages, short documents). And as it later during automated tests appeared, greater sizes would slow down the tests.

corpus	training	held-out	testing	debug
English wiki	82.7 M	1.6 M	35 k	14 k
Czech wiki	18.5 M	0.9 M	30 k	2.7 k
FYKOS	47 k	8.5 k	7.4 k	800

Table 4.1: Sizes of data for evaluation (word sizes include end sentence token `</s>`)

4.2 Experimental setup

The experiments were run in a form of *tests*. A single *test* could be identified with: the used language model, the testing data and the configuration (parameters). Therefore, the *tests* correspond to the rows of tables in section 4.3.

The testing data for each test were split into ten parts of approximately equal sizes (due to the granularity as the split boundary corresponded either to a paragraph or an article boundary). The chosen metrics were measured on each of the

[1] The corpus was a continuous text and double blank lines were recognized as article boundaries. Because those are only artefacts, there's not a "strict bijection" between double blank lines and article boundaries.

[2] `http://fykos.cz`

created partitions and the result of the test was a weighted mean of the values with number of tokens being the weight.

Correlation between metrics were also calculated, each of the ten measurements providing a data point. The correlation also took token counts into account.[3] When tests were run in a group, correlations for the whole group were calculated too.

4.2.1 Default settings

If not stated differently, the following settings were used.

- model: interpolated trigram model with modified Kneser-Ney smoothing, trained on Czech Wikipedia texts[4]
- test text: test texts from Czech Wikipedia
- selector: limit 10 000, unigram threshold: 2 characters (details in section 3.3.4)
- filter chain: (details in section 3.3.6)
  - added characters filter: 0 (no filtering)
  - punctuation filter: enabled
  - suffix aggregation: disabled
  - count limiter: enabled, minimal probability: 2^{-5}, maximal count: 5, allowed for prefix conditioning

When more models were combined, the linear combination coefficients were found by EM algorithm (section 2.1.3), trained on held-out data for corpus which the testing data were from. The optimal weights are in a auxiliary table bellow main table (for tests that include combined modes).

Default cache size for cached models used in combinations was 100 words.

4.3 Results

Following values were measured or calculated for each test:

- OOV ratio (column *OOV*) – ratio of words that weren't in the model's vocabulary,
- perplexity on all words (column *PPL*),

[3] With weighted correlation defined as

$$\mathrm{corr}_w(X,Y) = \frac{\sum_{i=1}^{N} w_i(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{N} w_i(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{N} w_i(y_i - \bar{y})^2}}$$

where $\vec{w}$ are weights, $\vec{x}$ and $\vec{y}$ are observations of X resp. Y and $\bar{x}, \bar{y}$ are weighted means of the observations.

[4] This smoothing method was chosen as default because it was expected to perform best of all considered methods [13] and informal experiments confirmed this conjecture.

- perplexity on known words (column *PPL'*) – OOV events are ignored,
- time performance (column *tok/s*) – average number of words from test text that were processed in a second,[5]
- keystrokes per character (column *KS*) – potential minimal value, in more detail in section 3.5.2,
- mean suggestion position (column *pos*) – average position for the first appearance of valid suggestion, in more detail in section 3.5.2,
- keystrokes per character and perplexity correlation (column r_{KS}) – perplexity is without OOVs,
- mean suggestion position and perplexity correlation (column r_{pos}) – ditto.

4.3.1 Comments

Smoothing method

The default setting were used and only smoothing method varied. The best results[6] in terms of keystrokes savings recorded modified Kneser-Ney smoothing with ngram interpolation (results in table A.1).

Laplace smoothing (with $\lambda = 1$) that was evaluated only for completeness, performs even worse than no smoothing in terms of keystrokes per character.

Model order

Model of order 0 was tested – all suggestions equally probable, results in table A.2. Recorded 100 % OOV rate is caused by the fact that only mock language model without vocabulary was used. However, the suggestions were provided by a unigram selector.

Cache size

Tested were two variants – only cached model and cached model combined with default trigram model. Results in the table A.3 shows that single cached model performs best when unlimited cache size incorporated (likely because of the greatest vocabulary). On the other hand, using unlimited cache model in combination with trigram model leads to worse keystrokes per character score (56.7 % vs. 54.8 %, the uncached model result is the trigram model row in the table A.2).

Dependency on training set size

Trigram models combined with cache were used for both languages. For English were used test texts for English corpus. In both cases, limited training corpora were created as subsets of the whole training corpora for each language.

Note that keystroke score for English is "quite good" even for "small" models (table A.5) and doesn't improve that much as does Czech for comparable training sizes (table A.4).

[5]The tests were on a common laptop with Intel T7300 processor @ 2GHz and 2 GB RAM and no other demanding tasks were running concurrently.

[6]Note these are only mean values, significance tests weren't done.

Suffix aggregation

Suffix aggregation filter (section 3.3.6) was used with following settings:

- variance threshold $t = 1$,
- suffix length $k = 1$,
- excluded count $e = 1$.

Tests were run with default settings as well as on a small model trained from user data only combined with cache.

Against expectations, the suffix aggregation didn't improve keystrokes per character ratio (table A.6). Possible cause is that when there was high probability of a variant, it was not grouped and oppositely when all variants were equally probable they would probably be shown all, thus suitable to accepted with single key. When those equal probable suggestions were grouped, more keystrokes, would be needed.

Opportunities are in the better suffix detection and/or testing with real users where other factors could became important.

Prefix conditioning

Prefix conditioning showed up to be effective in reduction of keystrokes (table A.7). On the other hand, its disabling caused that valid suggestions were listed on better positions in the offer. However, better method for increasing the position score is the one mentioned next.

Probability cropping

Limiting the list of suggestions with the minimal (prefix conditioned) probability, seems to be more effective method to increase the mean position score (table A.8) than disabling prefix conditioning as it does not decrease keystrokes per character score so heavily as the latter.

Also note non-monotonous dependency of keystrokes score on the limiting probability.

Selector limit

Limiting the selector output has similar effect to probability cropping regarding the list position metric, though weaker. (Apparently when there many suggestions, they tend to have lower probability per one). Whatsoever, limiting the selector greatly influences the time performance of the system (table A.9) and, thus, could be potentially used on less powerful devices where trade-off between speed and keystrokes savings is acceptable.

User domain adaptation

Three ways of adaptation were tested:

- no – using only trigram model trained on large corpus (Czech Wikipedia),
- cache – model mentioned above was combined with cached model,

- user data – large trigram model is combined with small trigram model trained on part of the user data and cached model is used too.

Two user data models were tested – the first (*user data*) with modified interpolated KN smoothing and the second (*user data'*) with own implementation of Good-Turing smoothing. This should cover the use case, when user doesn't have any LM toolkit, thus uses large language models in ARPA files and runs the mentioned script to train small model for his purposes.

Results in table A.10 indicate that for the testing data using general large trigram model with cached model achieves comparable results to model trained only on user data (last two rows of table A.6).

Combination with models from user data (both variants) under tested conditions further improves the keystrokes metric, however, it has relatively greater effect on the mean position score.

4.3.2 Correlations

The observed correlations between perplexity of the underlying language models and keyboard metrics suggest that improving language models in terms of perplexity would also positively affect the word prediction for typing.

On the other hand, some methods (mainly probability cropping) can lead to better scores of the mean position metric independently on the perplexity of the supportive language model.

5. Conclusion

In the beginning of the work, there were discussed existing solutions for word prediction task with the focus on ngram language models.

Consequently, theoretical background needed for the word prediction was introduced. That included methods for smoothing probability estimates, in order to cope with sparse training data of the language models and approaches to combine more models together.

During the analysis of the problem, four individual parts were identified.

The training module ensures processing of large text corpora yielding backoff model files later used by the suggestions module. A way of splitting the text into words and sentences is described.

The part about the suggestions module is concentrated on the problem of pruning a dictionary of all possible suggestions and subsequent processing of the resulting suggestions in order to provide appropriate word predictions.

Later, there is described the implemented GUI component that utilizes the provided suggestions and attempt to minimize keystrokes. The chosen technology accelerated development of the component but limited its potential use. On the other hand, a viable alternative is found available.

The next part deals with evaluation of the system. Traditional language modeling evaluation is used together with metric specialized for the minimization of keystrokes task. The missing user study is pointed out as it would provide directing information for further optimizations.

The final part presents the evaluation results for various configurations. The observation was done that limiting the size of the pruned suggestions dictionary doesn't negatively affect the keyboard metrics. On the other, method designed to cope with inflected languages didn't perform very well.

The work also found that using large ngram models for word prediction during typing effectively reduces keystrokes per character. There was also shown that word completion for Czech performs worse than for English, stemming from the worse perplexity of the Czech models.

The GUI component can be used separately and the whole project can later serve as a platform for improving the prediction abilities for the Czech or for the unrealized user study.

Bibliography

[1] Trnka, Keith, John McCaw, Debra Yarrington, Kathleen F. McCoy, and Christopher Pennington. *User Interaction with Word Prediction: The Effects of Prediction Quality*. ACM Trans. Access. Comput. 2009, vol. 1, no. 3, Article 17. ISSN 1936-7228. Online at `http://doi.acm.org/10.1145/1497302.1497307`.

[2] Manning, Christopher D., Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. Cambridge, Mass.: MIT Press, 1999. ISBN 978-0-262-13360-9.

[3] Al-Mubaid, Hisham, Ping Chen. *Application of word prediction and disambiguation to improve text entry for people with physical disabilities (assistive technology)*. International Journal of Social and Humanistic Computing. 2008, vol. 1, no. 1, p. 10–27. Online at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.169.6371&rep=rep1&type=pdf`.

[4] Chen, Stanley F., Joshua Goodman. *An empirical study of smoothing techniques for language modeling*. Proceedings of the 34th annual meeting on Association for Computational Linguistics. Santa Cruz, California: Association for Computational Linguistics, 1996, p. 310–318. Online at `http://dl.acm.org/citation.cfm?id=981904`.

[5] Shieber, Stuart, Rani Nelken. *Abbreviated text input using language modeling*. Natural Language Engineering. 2007, vol. 13, no. 2, p. 165–183. Online at `http://dash.harvard.edu/bitstream/handle/1/2027204/shieber-nelken-abbreviated-text-input.pdf?sequence=4`.

[6] Goodman, Joshua T. *A bit of progress in language modeling*. Computer Speech & Language. 2001, vol. 15, no. 4, p. 403–434. Online at `http://www.sciencedirect.com/science/article/pii/S0885230801901743`.

[7] Tillmann, Christoph. Hermann Ney. *Selection Criteria for Word Trigger Pairs in Language Modeling*. ICGI. Springer, 1996, p. 95–106. Online at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.593`.

[8] Tillmann, Christoph. Hermann Ney. *Word Triggers And The EM Algorithm*. Proceedings of the Workshop Computational Natural Language Learning (CoNLL 97). Association for Computational Linguistics, 1997, p. 117–124. Online at `http://acl.ldc.upenn.edu/W/W97/W97-1014.pdf`.

[9] Vaičiūnas, Airenas, Gailius Raškinis. *Cache-based Statistical Language Models of English and Highly Inflected Lithuanian* Informatica. 2006, vol. 17, no. 1, p. 111–124. Online at `http://dl.acm.org/citation.cfm?id=1413838`.

[10] Heafield, Kenneth. *KenLM: Faster and Smaller Language Model Queries*. Proceedings of the Sixth Workshop on Statistical Machine Translation. Association for Computational Linguistics, 2011, p. 187–197. Online at `http://kheafield.com/professional/avenue/kenlm.pdf`.

[11] Majliš, Martin, Žabokrtský Zdeněk. *W2C - Web To Corpus*. Data/software, Institute of Formal and Applied Linguistics, MFF UK, Praha, Czechia, http://ufal.mff.cuni.cz/~majlis/w2c/, 2011.

[12] http://www.speech.sri.com/projects/srilm/ as of July 2012.

[13] http://www.speech.sri.com/projects/srilm/manpages/ngram-discount.7.html

[14] http://www.swiftkey.net/ as of July 2012.

List of Abbreviations

ARPA file	text file representation of back-off language model
GUI	graphics user interface
EM algorithm	expectation-maximization algorithm
KDE	KDE Desktop Environment
KN smoothing	Kneser-Ney smoothing
LM	language model or language modeling
MLE	maximum likelihood estimate
NLP	natural language processing
OOV	out of vocabulary (word)
Python NLTK	Python natural language toolkit
SRI LM toolkit	Stanford Research Institute language modeling toolkit
WYSIWYG editor	"what you see is what you get" editor

Attachments

Attached CD contains:

- source codes of the implemented software,
- sample ARPA file with trained models,
- text files with corpora to train other models,
- PDF file with this thesis.

A. Experiment results

method	tokens	OOV	PPL	PPL'	tok/s	KS	pos	r_{KS}	r_{pos}
Mod KN int.	30344	2.71 %	585.2	636.8	52.8	**0.548**	2.12	0.641	0.791
KN int.	30344	2.71 %	589.2	632.5	54.1	0.549	2.10	0.623	0.768
Mod KN back.	30344	2.71 %	595	655.9	54.6	0.550	2.14	0.643	0.688
Good-Turing	30344	2.71 %	823.2	678.3	64.4	0.551	2.15	0.626	0.722
KN back.	30344	2.71 %	586.9	637.9	61.4	0.552	2.13	0.629	0.722
no	30344	2.71 %	1.7+07	5.5e+06	63.3	0.581	2.10	0.741	0.739
Laplace	30344	2.71 %	3.8+04	3.6e+04	117.8	0.646	2.39	-0.038	0.199
all	–	–	–	–	–	–	–	0.197	-0.070

Table A.1: Effect of smoothing method

order	tokens	OOV	PPL	PPL'	tok/s	KS	pos	r_{KS}	r_{pos}
0	30344	100.00 %	1.3e+30	nan	21.2	0.907	2.69	0.000	0.000
1	30344	2.71 %	3393	3877	344.2	0.714	2.26	-0.176	0.785
3	30344	2.71 %	585.2	636.8	53.2	0.548	2.12	0.641	0.791
all	–	–	–	–	–	–	–	0.825	0.992

Table A.2: Dependency on ngram model order

cache size, mod.	tokens	OOV	PPL	PPL'	tok/s	KS	pos	r_{KS}	r_{pos}
50	30344	69.80 %	5.2e+21	29.5	1529.9	0.930	2.12	-0.614	0.433
100	30344	63.32 %	8.0e+19	39.66	1223.4	0.906	2.12	-0.646	0.463
200	30344	58.12 %	2.6e+18	53.27	779.3	0.888	1.99	-0.652	0.298
unlimited	30344	45.41 %	4.0+16	148	153.2	0.837	2.03	-0.941	0.861
50, trigram	30344	2.58 %	522.1	563.4	24.5	0.546	2.11	0.671	0.744
100, trigram	30344	2.55 %	503.2	541.8	26.0	0.547	2.09	0.656	0.796
200, trigram	30344	2.51 %	494.1	530.8	25.5	0.549	2.08	0.652	0.784
unlim., trigram	30344	2.44 %	504.7	542	20.5	0.567	2.09	0.611	0.631
all	–	–	–	–	–	–	–	-0.934	0.234

cache size	main	user	cached
50	0.953	–	0.047
100	0.936	–	0.064
200	0.922	–	0.078
unlimited	0.944	–	0.056

Table A.3: Effect of cache size

training size	tokens	OOV	PPL	PPL'	tok/s	KS	pos	r_{KS}	r_{pos}
18 M	30344	2.55 %	503.2	541.8	26.0	0.547	2.09	0.656	0.796
6 M	30344	4.48 %	527.9	614.3	34.6	0.584	2.11	0.420	0.561
1.8 M	30344	7.75 %	479.9	646.6	45.0	0.630	2.11	0.245	0.610
all	–	–	–	–	–	–	–	0.518	0.666

training words	main	user	cached
18 M	0.936	–	0.064
6 M	0.929	–	0.071
1.8 M	0.923	–	0.077

Table A.4: Dependency on training set size, Czech

training words	tokens	OOV	PPL	PPL'	tok/s	KS	pos	r_{KS}	r_{pos}
83 M	34855	0.57 %	143.1	142.9	28.3	0.436	1.95	0.846	0.761
24 M	34855	0.92 %	158.4	158.9	28.5	0.449	1.96	0.874	0.666
8 M	34855	1.60 %	177.7	180.5	32.5	0.467	1.99	0.800	0.682
2.4 M	34855	2.56 %	205.4	214.6	39.2	0.495	1.98	0.762	0.616
all	–	–	–	–	–	–	–	0.879	0.635

training words	main	user	cached
83 M	0.905	–	0.095
24 M	0.897	–	0.103
8 M	0.892	–	0.108
2.4 M	0.887	–	0.113

Table A.5: Dependency on training set size, English

configuration	tokens	OOV	PPL	PPL'	tok/s	KS	pos	r_{KS}	r_{pos}
no agg., no cache	30344	2.71 %	585.2	636.8	51.6	0.548	2.12	0.641	0.791
agg., no cache	30344	2.71 %	585.2	636.8	23.2	0.560	2.02	0.661	0.820
no agg., cache, 47 k	7454	14.49 %	606	268.4	212.2	0.633	2.18	0.736	0.252
agg., cache, 47 k	7454	14.49 %	606	268.4	130.8	0.637	2.11	0.778	-0.114
all	–	–	–	–	–	–	–	-0.370	-0.040

training words	main	user	cached
47 k	0.893	–	0.107

Table A.6: Effect of suffix aggregation

prefix conditioning	tokens	OOV	PPL	PPL'	tok/s	KS	pos	r_{KS}	r_{pos}
no	30344	2.71 %	585.2	636.8	53.1	0.888	1.46	0.951	0.375
yes	30344	2.71 %	585.2	636.8	51.3	0.548	2.12	0.641	0.791
all	–	–	–	–	–	–	–	0.075	0.076

Table A.7: Effect of prefix conditioning

min. probability	tokens	OOV	PPL	PPL'	tok/s	KS	pos	r_{KS}	r_{pos}
2^{-2}	30344	2.71 %	585.2	636.8	45.7	0.683	1.14	0.805	0.349
2^{-3}	30344	2.71 %	585.2	636.8	43.3	0.621	1.41	0.725	0.526
2^{-4}	30344	2.71 %	585.2	636.8	51.6	0.575	1.87	0.663	0.712
2^{-5}	30344	2.71 %	585.2	636.8	50.5	0.580	2.62	0.663	0.711
2^{-6}	30344	2.71 %	585.2	636.8	53.0	0.625	3.76	0.764	0.736
2^{-7}	30344	2.71 %	585.2	636.8	53.0	0.667	4.98	0.698	0.601
all	–	–	–	–	–	–	–	0.284	0.038

Table A.8: Effect of cropping suggestions with probability

limit	tokens	OOV	PPL	PPL'	tok/s	KS	pos	r_{KS}	r_{pos}
10000	30344	2.71 %	585.2	636.8	52.8	0.548	2.12	0.641	0.791
5000	30344	2.71 %	585.2	636.8	172.6	0.555	2.11	0.575	0.767
2500	30344	2.71 %	585.2	636.8	268.6	0.558	2.09	0.564	0.772
1000	30344	2.71 %	585.2	636.8	399.6	0.563	2.07	0.552	0.769
500	30344	2.71 %	585.2	636.8	551.0	0.572	2.04	0.517	0.728
250	30344	2.71 %	585.2	636.8	802.5	0.589	1.97	0.490	0.685
all	–	–	–	–	–	–	–	0.453	0.548

Table A.9: Dependency on selector limit

adaptation	tokens	OOV	PPL	PPL'	tok/s	KS	pos	r_{KS}	r_{pos}
no	7454	7.30 %	891.6	1247	56.2	0.645	2.19	0.952	-0.548
cache	7454	6.21 %	626.2	787.8	28.7	0.624	2.14	0.790	-0.217
user data'	7454	5.02 %	315	345.3	23.4	0.609	1.99	0.720	0.155
user data	7454	5.02 %	306	334.3	23.4	0.609	1.98	0.720	0.246
all	–	–	–	–	–	–	–	0.576	0.661

method	main	user	cached
main + cache	0.881	–	0.119
main + user + cache	0.372	0.540	0.088
main + user' + cache	0.383	0.528	0.089

Table A.10: Adaptation to user domain

B. Programmer's reference

Most of the code is documented in source code files – the C++ code with the JavaDoc style comments and Python code with docstrings.

The following text describes concepts that would a programmer find useful when using or extending the system.

B.1 Integrating GUI component

First of all, make sure you have PyQt4 installed, that the project's directory with project's packages is in your `PYTHONPATH` and C++ bindings are correctly compiled. Now, you should be ready, to run a minimal working example.

```
import sys
from PyQt4 import QtGui
import ui.completion
import lm.selection
import ui

class Window(QtGui.QMainWindow):
    def __init__(self):
        super(Window, self).__init__()
        handler = ui.ContextHandler()
        selector = lm.selection.BigramSelector("path/to/arpa", handler)
        handler.addListener(selector)

        txtMain                = ui.completion.TextEdit(self)
        txtMain.selector       = selector
        txtMain.contextHandler = handler
        #txtMain.addFilter(myFilter)
        self.setCentralWidget(txtMain)
        self.show()

app = QtGui.QApplication(sys.argv)
win = Window()
sys.exit(app.exec_())
```

In the example, the component uses suggestions from a bigram ARPA selector (section 3.3.4) and they are displayed in alphabetical order as no filter chain (section 3.3.6) was configured.

Filter chain is implemented as a sequence of callables with only constraint that the first callable must take as an argument a sequence of suggested strings from the selector and the last must return a sequence of tuples with the structure `(suggestion, type, probability)`. For more details look into `ui.filter` module.

B.2 Implementing an own metric

During the automatic evaluation text is processed token by token each of them being send to a metric. Metric is an object that keeps its state according to the processed tokens. Metric's result is a tuple of numbers.

If you want to create an own metric, just inherit from the class `testing.metric.Metric` and reimplement at least methods in the example below. Through the parent object, every metric has access to the configuration object (section B.3).

```
class MyMetric(testing.metric.Metric):
    def reset(self):
        super().reset() # parent call needed
        # set the initial state

    def measure(self, token):
        super().measure(token) # parent call needed
        # code that evaluates the given token in the stream
        # self._config is a reference to the configuration object

    def result(self):
        # return tuple with numeric results

    def resultHeader(self):
        # return tuple with
```

B.3 Configuration object

As there is need for many service objects in the system (selector, filters in the chain, context handler), a simple variant of the *service locator*[1] was implemented and is referred to as the configuration object. There is an "abstract"[2] configuration class (`common.configuration.Configuration`) that only takes care of lazy creation of the demanded service object and keeps some static parameters (configuration object can be directly parametrized from INI files and/or command line interface).

Descendant classes implement factory methods for the service objects, thus defining object graph structure.

```
class MyConfiguration(common.configuration.Configuration):
    def _createBar(self):
        return Bar()

    def _createFoo(self):
        # factory method can access other objects
        # from the configuration object
        return Foo(self.bar)

conf = MyConfiguration(INIparams, CLIparams)
foo = conf.foo # Bar and Foo instances are lazily created here
```

In the project, various descendants of the `common.configuration.Configuration` class were used to construct object graphs for different language models combinations.

B.4 SIP bindings

SIP bindings provide a way how to use C++ libraries from Python. There is a need to write a SIP definitions file from which wrapping code is generated by the

[1] `http://martinfowler.com/articles/injection.html#UsingAServiceLocator`

[2] In Python it is not actually abstract.

`sip` utility.[3] The binaries of the wrapping code are consequently linked with the binaries of the library into a shared object file that can be loaded as a Python module (see figure B.1).

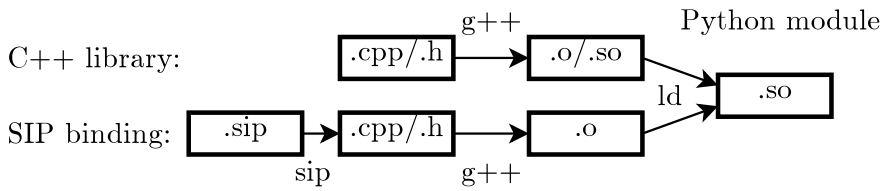


Figure B.1: Files and tools involved in the creation of bindings from Python to C++ code

Because the KenLM library didn't have an API that could be easily wrapped with SIP, an intermediate own C++ wrapper was implemented and used for SIP bindings.

The example below shows how to use KenLM in Python.

```
import kenlm

model = kenlm.Model("path/to/arpa")
model.reset() # begin sentence state
log2prob = model.probability("word")
words = model.vocabulary() # all known words
```

B.5 File formats

The crucial data are stored in the ARPA file format, more information provides section 3.2.1.

However, several other formats are used for auxiliary files.

.bin Those are files with substantially same information as the ARPA files. They purpose is to accelerate loading of the models into the memory in comparison with textual ARPA files. Because the ARPA file format ensures portability of trained models, the binary files are supposed to be site-specific and for their creation serves utility `bin/kenlm/build_binary`[4] provided by KenLM library.

.sel.bin The existence of this format is analogously motivated as for .bin files mentioned above. These files store the selector's trie (section 3.3.4). They are built with `bin/selector/build_binary`[5]

.txt.sentences This format is specified by SRI LM[6] and is useful to keep information about dividing text into tokens and sentences.

.results Those are files generated by the evaluation script. They contain results of running a test (definition in section 4.2), each row corresponds to

[3] `http://www.riverbankcomputing.co.uk/static/Docs/sip4/index.html`
[4] Relatively to the project root.
[5] Relatively to the project root.
[6] `http://www.speech.sri.com/projects/srilm/manpages/ngram-count.1.html`

measurements on a single file and columns (separated with whitespace characters) are determined by the used metrics. The lines beginning with `#` are ignored and they store test's metadata (used corpus, configuration, time).

.ini Configuration is stored in common key-value way in the INI files.

Note on compression In situations where KenLM or ARPA selector API expects a path to an ARPA file, a path to a "gzipped" ARPA file may be used or a path to a .bin, resp. .sel.bin file.

C. User documentation

Because this projects consists of the GUI component and evaluation scripts of the word prediction system, its potential users are programmers integrating the GUI component or people testing the quality of suggestions. So there is an expectation that they will be advanced computer users.

C.1 Installation

The software should theoretically work with more or less obstructions on any GNU/Linux operating system. However, only Ubuntu 12.04, 11.10 and Debian Squeeze were tested.

C.1.1 Requirements

- Python 3.2 and its header files,
- SIP 4.13 or higher,
- PyQt4 for Python 3,
- Qt library of version required by PyQt4 (currently ≥ 4.7),
- GNU make.

Three installation scenarios were tested:

- distribution packages
  - Ubuntu 12.04 contains all required software in its repositories
  - needed packages are: `python3.2-dev`, (`python3`), `python3-pyqt4`, `python3-sip-dev`, (`python3-sip`)
- own compilation of PyQt4
  - Ubuntu 11.04 and 11.10 repositories contain Python 3.2, however PyQt4 for Python 3 is missing
  - needed packages are as above, excluding `python3-pyqt4`
  - PyQt4 has to be build from the attached files
    1. working SIP is required
    2. this additionally requires installation of `qt4-qmake` and `qt4-dev-tools` packages
    3. in `libs/pyqt4`[1] consequently run: `python3 configure.py`, `make`, `make install`
- complete compilation (tested on Debian Squeeze)
  1. on systems without Python 3.2 or Qt ≥ 4.7
  2. sources for Python 3.2 and Qt ≥ 4.7 are downloaded and compiled
  3. compilation of SIP and PyQt4 from attached files

[1] Relatively to the project root.

C.2 Running demonstration UI

GUI component was integrated into a very simple editor-like application in order to directly test the component. Despite the spartan GUI, the application allows great configuration through INI file(s) and command line arguments.

At least two arguments are required:

- path to INI file(s) (with more INI files loaded, precedence is given by specified order – last file overrides previous),
- chosen configuration.

abbreviation	description
`00`	Uniform selector only.
`0`	No suggestions at all.
`0c`	Suggestions based on cached model only.
`u`	Unigram model for suggestions.
`n`	Ngram (≥ 1) model for suggestions. Bigram selector.
`uc`	Same as `u` but combined with cached model.
`nc`	Same as `n` but combined with cached model.
`uuc`	Combination of unigram model, user data model and cached model.
`nuc`	Combination of ngram model, user data model and cached model.

Table C.1: Available configurations of language models

Detailed descriptions of INI file options and sections are in the sample file `src/python/lmconfig.ini`[2]

Some of the INI file values can be overridden by direct input from command line. Exact options are listed in the editor's usage.

C.3 Evaluation

Evaluation script provides similar command line interface as the simple editor, i.e. it expects paths to INI files and a chosen configuration. But instead of running a GUI, it measures metrics on provided testing text files. Metrics can be changed only in the source code, definition of new metrics is described in section XX B.2.

C.4 Training own model

C.4.1 Small models

In the case only small model is needed, utility `trainARPA.py` can be used.

C.4.2 Large models

Large models are trained with the `ngram-count` tool from SRI LM [12]. Because the process involves more tools, `Makefile` is generated to coordinate them. Available smoothing methods and model orders are defined in `configure.py` file

[2]Relative to project root.

– each of the configuration has its identifier expressing order of the model and used smoothing method.

When user wants to train a model (or even create binary files for it, section B.5), the only requirement is to copy the file with training texts into the `large-data` directory and run `make` with the name of desired target (explanation in table C.2).

data.txt	input data
data.txt.sentences	data split into tokens and sentences (section 3.2.3)
data.<conf>	ARPA file with trained model
data.<conf>.gz	"gzipped" ARPA file
data.<conf>.bin	binary representation of the model for KenLM
data.<conf>.sel.bin	binary representation of the selector

Table C.2: Sequence of filenames for training models using `Makefile`, <conf> denotes chosen identifier of smoothing method and order, for file format details see section B.5.

C.5 Utility scripts

Utility script are located in `src/python/learning` directory and they provide documentation as part of their usage message.

- `findAbbr.py` finds possible abbreviations in the text, see section 3.2.4
- `formatText.py` splits text into tokens and sentences and stores
- `stripText.py` removes LaTeX mark up from the text
- `textPartition.py` splits one text file into more
- `trainARPA.py` trains small language models
- `trainLinterEM.py`
  - script is located in the `src/python`
  - it has same command line interface like `ui-demo.py` or `self-eval.py`
  - it's used to optimize weights of the models in the chosen configuration with the EM algorithm (section 2.1.3)