Term project report
Natural Language Processing - T-61.5020
László Kozma <Lkozma@cis.hut.fi>, <77175U>

Text Autocomplete Systems

0. Abstract

In this paper I report on the experiments performed as part of the coursework in Natural Language Processing. First I briefly overview the problem domain and some existing implementations of the autocomplete feature, both from a user's and from an implementer's point of view. I present a small variation on the autocomplete idea, which consists of utilizing the cursor location information when suggesting a completion, thus completing in both directions in the same time. In the second part I do some experiments for measuring the autocomplete performance using different simple language models. Since a complete survey of the existing literature is outside the scope of this work, I can not claim that any of the ideas presented here are novel, however, at least the feature of bidirectional autocomplete is missing from current implementations I have seen, and it would readily improve most commercial systems, including web browsers or Google Suggest. Another possibly new approach described here is the measure used for assessing autocomplete performance using reading and typing cost.

1. Introduction

Word completion/autocomplete/autosuggest is a common user interface element in modern applications. Its goal is to anticipate what the user wants to type and automatically add parts of the text. The goal is to speed up typing, to assist those with typing difficulties, to correct/prevent spelling mistakes or to facilitate information retrieval by presenting available choices. Possibly the earliest occurence of the idea is in Witten and Daraghs' work on the Reactive Keyboard from 1983 [1]. Since then several other approaches have been described but the main idea remained the same.

The feature is widespread in all user interfaces, including word processors (MS Word, OpenOffice.org), programming editors (Emacs, Eclipse), desktop applications (web browsers, e-mail clients), HTML form elements on websites, web applications (Google Suggest, web-based e-mail clients), mobile phone interfaces, Unix terminals, etc. Depending on the particular scenario and implementation, autocomplete systems range from the highly useful to the wildly annoying. It is important to note that autocomplete usually works much better in restricted domains (source code editing, technical manuals, legal documents, etc.), and can be next to useless in arbitrary natural text entry.

On most mobile devices there is a system that facilitates fast typing, by using multiple functions of the keys and taking advantage of the redundancy in the language. The variants of this particular system (T9) are a different topic so I will ignore these in this discussion.

1.1. User interface

From an user interface point of view the two main approaches for implementation are the following:

- present the most probable candidate for completion, where the user has the choice of accepting or rejecting the suggestion
- present a list of candidates for autocomplete in some order (alphabetical, by relevance, etc.), allowing the user to pick one or continue typing

In both cases the autocomplete feature can be passive or active. In passive systems the user requests the suggestion (usually with a key-press), whereas active systems offer suggestions automatically, possibly if the expected probability of the acceptance is above a threshold. It is also possible to always suggest completions, in case one is available.

Another distinction from the user's point of view is whether the autocomplete system uses exact or fuzzy matching. In the latter case, misspelled text can also trigger the completion, which helps correcting spelling mistakes (and arguably worsens user's writing skills by inducing dependence on the tool) An example of this is the Google Suggest interface, where starting to type "**amrei**" triggers the suggestions "**American ...**". In web browsers if one starts typing the address omitting the beginning "**http://**", autocomplete still works correctly.

A distinction can be made in whether the system completes atomic elements (from a list) or stream of continuous text. While it's hard to imagine a system that could predict more sentences correctly, in the second case there isn't any hard boundary for the autocompletion, the longer the correctly suggested text, the better. Even if the autocompletion is done at the word level, the interface can be made such as to offer partial completion, if this is expected to be more useful. This is needed in case of agglutinative languages but it can be useful for English as well, for example to complete "**impor**" to "**importan**" which the user can further complete into either "**important**", "**importance**", "**importantly**", etc. A well known implementation of this partial autocomplete is the command line interface on Unix/Linux, where hitting the TAB key results in completing the command until the first point of ambiguity.

1.2. Backend

When implementing an autocomplete system one needs to have a mechanism for storage and fast retrieval of strings together with other data (for ex. frequency). The most commonly used data structures are prefix trees and variants thereof ([7]). For smaller sets of elements, simple list structures usually suffice. Trees can require only logarithmic time (in nr. of elements) for lookup.

The simplest systems perform a lookup of elements among those previously entered by the user. More complex systems use some larger corpus to build the background model and they use some kind of language model to approximate the probabilities of a given word appearing in some context.

In practical implementations the list (or tree) structure containing the suggestions can be stored on the user machine or remotely on a server. The first solution offers shorter response times, thus a more responsive user interface. Using a remote database for retrieving suggestions can make the interface more sluggish but offers more flexibility in dynamically updating the database, using larger databases, or merging information from various users. Google Suggest applies this approach, as do many web based autocomplete systems.

In my experiments it was not a goal to build an actual user interface, so responsiveness was not considered and I used simple list-based lookups with possibly suboptimal algorithms.

2. Bidirectional autocomplete

A common feature of all the existing autocomplete implementations I have used is that they use the text preceding the cursor as context. In some cases, where one atomic string is used for completion, the whole text is used as context, not taking the cursor location into account at all. This leads to suboptimal behavior in many cases when the user is correcting an entry, moving back with the cursor.

Some usage scenarios where traditional autocomplete does not behave optimally are the following:

- I enter "**German football team**" in Google Suggest, then I change my mind and go back to change the query to "**Italian football team**". Even when "**Italia| football team**" is typed, (the | sign shows the cursor location), autocomplete still doesn't help, as it does not use the cursor location. Otherwise, "**Italian football team**" is also a frequent query, if I type it normally from the beginning, as soon as I get to "**f**", Google Suggest makes the right completion.

- In the browser I typed "**mail.yahoo.com**". Then I want to go to "**mail.google.com**" next. Even though "**mail.google.com**" appears in the history, when I correct the middle part, such as to "**mail.g|.com**" or even "**mail.googl|.com**", autocomplete is clueless.

- The Eclipse editor makes smart autocompletions, for example completing a method name with its signature. Typing "**myMe|**" could autocomplete to "**myMethod(int i, int j)**". However, as it often happens, I want to change this to another method that has same signature, typing "**myOther|(int i, int j)**" and asking for autocomplete results in "**myOtherMethod(int i, int j)(int i, int j)**". The autocomplete did not notice that to the right of the cursor the method signature was already typed, it only looked to the left of the cursor for context.

- I typed "**jenny@fictivecompany.com**" in the e-mail To: field. Now I go back to change it to "**jennifer@fictivecompany.com**". Even though this address is in the address book, "**jen| @fictivecompany.com**" does not activate the autocomplete. This problem is similar to the previous ones, but in this case a whole email address can be considered an atomic element.

- I entered a command in the terminal, but realized the file path was wrong. I go back, delete part of the path, use TAB to ask for help, and manage to enter the correct address. In this case the autocomplete knows where the cursor is, but it still ignores the text to the right of the cursor. Suppose, I typed the path "**/home/myuser/Desktop/myStuff**", then I went back to change "**myuser**" to "**otheruser**". Autocomplete does help me with that, it inserts "**otheruser**", but it does not check that "**otheruser**" does not have the directory "**/Desktop/myStuff**" and this creates a wrong command. The natural behavior in this case would be not to do autocompletion, unless the string to the right of the cursor (until the first separator) is also part of the autocompleted text (the path exists).

2.1. Solution

One solution to the above problems is to perform bidirectional autocomplete. If the basic unit for autocompletion is a word, we just have to look at the text to the right from the cursor until the next word boundary and do a lookup of the reverse of this text in a similar data structure to the one we use for the regular autocomplete. Autocomplete is performed only if there is a suggestion that is valid for both the forward and the reverse part.

For example if we have entered "**United | America**", where " are the separators and | is the cursor, a string is a candidate for autocompletion if it matches "**United** " in the left search tree and matches " **America**" in the right search tree, so "**States of**" can be correctly inserted, whereas if only forward autocomplete were used, "**United Arab Emirates**" would have been a valid suggestion.

This method does not add complexity to the autocomplete method used, as whatever data structure we have used for the lookup, we can duplicate it for the reverse lookup.

An example implementation and a more detailed writeup (including the examples listed here) can be found at:
http://www.Lkozma.net/autocomplete.html#l2

3. Experiment

In this section I perform some simple experiments with autocomplete systems that suggest words or parts of a word in a natural language text entry scenario. The documents used as test data are randomly picked from the 20 Newsgroups (20NG) data set (rec.sport.hockey). A different set of documents are used as training data.

3.1. Measuring performance

Measuring the performance of an autocomple system is difficult to do in a precise, quantitative way, as users have individual typing/reading patterns and preferences, which makes it hard to estimate the time saved in typing. Furthermore, there is a highly subjective "level of annoyance", which is difficult to measure but is nevertheless important in assessing the usefulness of such a system.

Some of the research papers formulate the performance measure in the traditional precision-recall or ROC curve frameworks, counting accepted and rejected suggestions as well as suggestions that could have been made but were not. A more heuristic approach is to estimate the typing time saved, which, while easy to understand, makes several simplifying assumptions. In my experiments I use a simple variant of this heuristic measure.

I assume each character has a uniform typing cost, adding up to **t**. When reading and evaluating a suggestion, each character has a uniform reading cost, adding up to **r**. While t relates to the amount of time saved in typing, r measures the "annoyance" or the reading burden the user is facing in using the autocomplete system.

The two different user interfaces evaluated are the ones described before: we either insert the suggested text directly into the text (auto-selecting it), or display a list of possible suggestions. In both cases we assume that there is no typing cost of rejecting a wrong suggestion (in the first case the user can continue writing, thus overwriting the suggested text, in the second case the user can ignore the pop-up list altogether). In both cases however, there is a cost of reading the suggestion. I assume that the user always reads until the first point of disagreement in case of wrong suggestions or until the end in case of correct suggestions. For example, if the user wants to write "**automatic**" and the software suggests "**automobile**", the user needs to read until the second "**o**" to reject the suggestion. It is also assumed that the user does not read the prefix of words that is common with the already typed text.

Furthermore, I assume that accepting a suggestion requires one keystroke equivalent with typing a character. This can mean pressing ENTER in the first kind of interface or performing a mouse-click in the second. In either case with learning this can become an automatic reflex comparable to pressing a single key. In the case of a list of suggestions, I assume that the user reads all suggestions until the point of disagreement in each of them. If he finds the correct suggestion he accepts it, otherwise continues

typing. The penalization of offering too many suggestions or a correct suggestion very far in the list is implicit: the user needs to read the text to decide whether he can reject it or not. Since the user is assumed to read all suggestions, it is common sense to assume that if a suggestion is rejected, it won't be offered later. This improves the performance significantly.

Some of the rather strong implicit assumptions made are:

- reading/scanning text proceeds linearly
- the time needed to read/write text is a linear function of the text length
- the time needed to evaluate all elements in the list is linear (i.e. there is no ordering of the list that would allow binary search)
- reading proceeds optimally in the sense that a word is not read any further than a character that allows rejection (but not any more optimally, as in checking random characters or starting from the end, etc.)
- the user can not reject a correct suggestion (for example by not observing it or being too busy typing)
- user does not read the prefix that is typed already
- suggestions are either rejected or accepted, even if a suggestion differs only in one character from the correct text, the user won't accept it and manually correct it (this assumption is tacitly made in most research and is somewhat motivated experimentally).

While some of these assumptions might seem arbitrary or too constraining, hopefully the overall distortion they introduce is not significant.

In my opinion, using two separate measures for reading cost and typing cost is better than most other heuristics used in various papers that express cost as a single value. The **t** and **r** values can be combined into a single cost for example using a linear function $\mathbf{C = t + \alpha * r}$. The α coefficient expresses the ratio of the cost between typing and reading a character, or alternatively the amount of reading the user is willing to do in order to reduce typing. This coefficient can capture important differences between users. A person with impaired typing capabilities might prefer reading suggestions in order to avoid typing at all costs, whereas a proficient typist should be offered suggestions only when there is a clear time saving expected.

3.2. Set-up

In this experiment I use a few randomly selected documents from the 20 NG data set, containing a total of approximately 7000 words and 42K characters (incl. spaces). No actual interface is built, the **t** and **r** costs are calculated by a perl script that parses the documents and simulates the suggestions. Using the two approaches described above (single suggestion, list of suggestions) and the performance metric given above (with various values of the α parameter between 0 and 1) I compare different methods.

Parameter **α** is smaller than 1, partly because it is intuitive that reading is faster than typing, partly because if it weren't so, no autocomplete would be possible, as it would not be worth reading any of the suggestions.

The total cost for a document **D** without autocomplete is **C = length(d)**. (no reading, just typing each character). Depending on the language model used, the system tries to approximate the cost incurred by offering a suggestion, and taylor the suggestion such as to have an expected reduction in total cost.

For simplicity all punctuation marks and non-text characters were removed and the text was transformed to lower case only. In this way the text is transformed to a sequence of words separated by spaces (in case of an accepted suggestion the space is counted also as saved typing). The From: and Subject: lines were removed, as well as words of a single character.

4. Autocomplete methods

4.1. Naïve unigram model

This method is similar to the one implemented in most word processors. Autocomplete is offered only for words that appeared previously in the same document. A simple unigram model is built on the go from the current document (the system estimates the probability of a new word from the frequencies of words starting with the same prefix). For a given value of **α** the system estimates for each word whether there is an expected saving in cost by suggesting the word. If there is, it suggests the word. If we want only one suggestion, the one with highest expected saving is offered, in the case of a list of suggestions all words with positive expected cost saving are suggested.

Both in this and other models a difficult question is how to treat unseen words. While there are principled methods of doing this, in my experiments I simply estimate the probability of unseen words from the text typed so far. For all kinds of background models I count how many times it occurred in the past that the next word was an unseen one. It is assumed then that the probability of the next word being unseen is equal to this frequency (this is in fact the number of distinct words divided by the total number of words) . This way the probability of each known word is reduced significantly (together with the optimism of the autocomplete).

4.2. Unigram background model

In this version the unigram model is initialized from a corpus of text. For comparison both a corpus of similar topic (same newsgroup but different texts) and a generic corpus (all 20 newsgroups) are used. The cost function is used in the same way as in the previous tests but the probability of the occurrence of a word is estimated from the loaded unigram model. The model is updated continuously with the words typed in the current text but these have little influence, as the corpus is much larger.

4.3. Partial autocomplete

The test is performed in the same way as earlier but instead of suggesting full words, parts of the word are suggested to the extent that is expected to yield reduction in cost. Only a single suggestion is made in this case, as a list of multiple choices would not be natural here. As the unigram model slows down the running times, this experiment was run with the model built on-the-fly only, such as in the first, "naive" model. Using an external corpus is expected to have similar effect as previously.

4.4. Part-of-speech bigram model

A bigram model for all words requires too many parameters and many word pairs will never occur in the training corpus. Therefore a bigram model on part-of-speech classes is proposed, also using the probability of appearance of a word within its class. The bigram part-of speech model should be trained on the whole newsgroup corpus and tagged using standard part-of-speech tagging. Unfortunately due to lack of time I had to skip performing this experiment.

4.5. Models on equivalence classes

Building an n-gram model on all words is troublesome, so using the part-of speech labels seems useful. However the part-of-speech classes are too few and it is not obvious that these classes are optimal for predicting next word in a text. Therefore it would be useful to cluster words into equivalence classes, and if (in case of the bigram model) $\mathbf{P(w_n|w_{n-1})}$ is not known, we can approximate it with $\mathbf{P(w_n|G(w_n)) * P(G(w_n)|G(w_{n-1}))}$.

The goal is to find an optimal clustering. Even more useful would be to have components instead of strict clusters, where a word could be expressed as a mixture of components.

We could then express W**xy** (probability of word x followed by word y) as

$$W_{xy} = \sum_i \sum_j \mu_{xi} \mu_{yj} C_{ij}$$

Here Cij is the probability of element from Ci followed by element from Cj and μ are the mixture coefficients (the amount of a word belonging to a certain cluster).
Alternatively the relation could be expressed as: $C = \sigma' W \sigma$
W is approximated from the observed frequencies, C is calculated and σ should be learned.
Possible criteria for optimality would be that rows in C have low perplexity, which leads to a formulation similar to ICA [10]. The matrix C can be imagined as the adjacency matrix of a transition graph.

I'm not sure if there is an algorithm for efficiently finding such a mapping, or if this is sensible (or correct) at all. For splitting the words in equivalence classes, there are several approaches in the literature similar to k-means or EM - style algorithms as well as other methods [11, 12, 13]. A full study of these is outside the scope of this report.

5. Experimental results

5.1. Sample output

For debugging and monitoring purposes the script prints out the progress in the following format ([] are suggestions, * is accepted):

j[just]**o**[jokerit]**e**[joel] **k**[know]**o**[kovalev]**cur** **p**[play]**l**[players]**a**[player]**y**[playoffs]**i**[playing]*
e[edu]**v**[even]**e**[every]* **n**[not]**i**[night]* **h**[have]**e** **i**[in]**s** **n**[not]*

5.2. Results

In the following the total costs are presented as charts for all methods. In all cases I plot the cost as a function of the α parameter. Tests were carried out for values between 0 and 1 with resolution of 0.05, and in the [0,0.1] range with resolution 0.01. As expected, the cost always grows with alpha, reaching for $\alpha = 1$ that of the cost of typing the full text without autocomplete. I should note that in all experiments the same α value was used both for making the suggestions and for later evaluating the performance. This makes intuitive sense but different values could be used to make the autocomplete more active/passive than the user's expectations.

The horizontal lines are the scores without autocomplete (not dependent on α). On **Figures 1 and 2** the reading and typing scores are shown separately (number of characters), as can be seen, the smaller the α the more typing is traded off for reading. If more candidates are suggested, the behavior of the cost is still similar. This kind of interface has an advantage for smaller α only, later both methods converge to the same cost. Again, this makes sense: there is an advantage in seeing many suggestions at once only if the user is willing to do a lot of reading at the expense of typing.

In **Figure 3.** I compare all unigram models. As can be seen the relation between the single and multiple suggestions is similar across methods (continuous and dashed lines). The best result (red line) can be obtained with the model trained on the same newsgroup, in this case the autocomplete system knows from the start the “lingo” of the field, in this case hockey. Training on a generic corpus helps compared to not having a model at all, but the result is worse than for the specific corpus, the model is “diluted” in this case.

In **Figure 4.** the partial autocomplete is compared with the full autocomplete with the on-the fly unigram model (the partial one uses the same model). The results are rather surprising, partial

autocomplete yields worse results, even though it can make more fine-grained choices. This is probably because too often partial autocomplete prefers “safe” choices, making several short completions, rather than guessing a full word at once. This result (unless it is due to some bug) together with the awkwardness of a partial autocomplete interface make this solution less useful for natural language entry. Also, surprisingly for the partial autocomplete method the cost was observed to even decrease in short portions of the graph, while all other plots were strictly increasing.

Figure 1: Naïve autocomplete, single suggestion

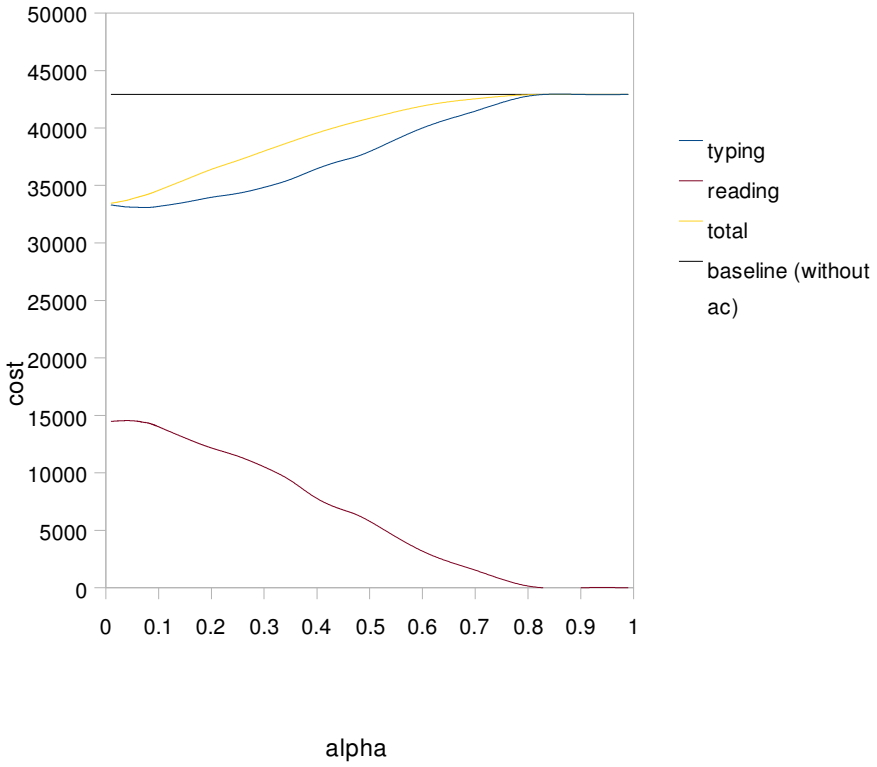


Figure 2.: Naïve autocomplete, multiple suggestions

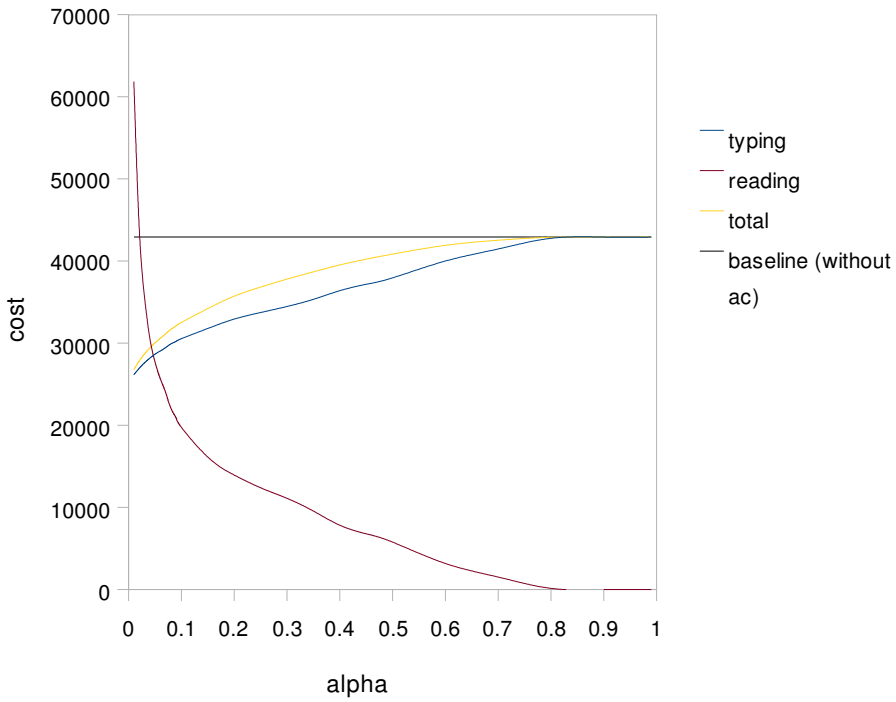


Figure 3.: Total costs, all unigram methods

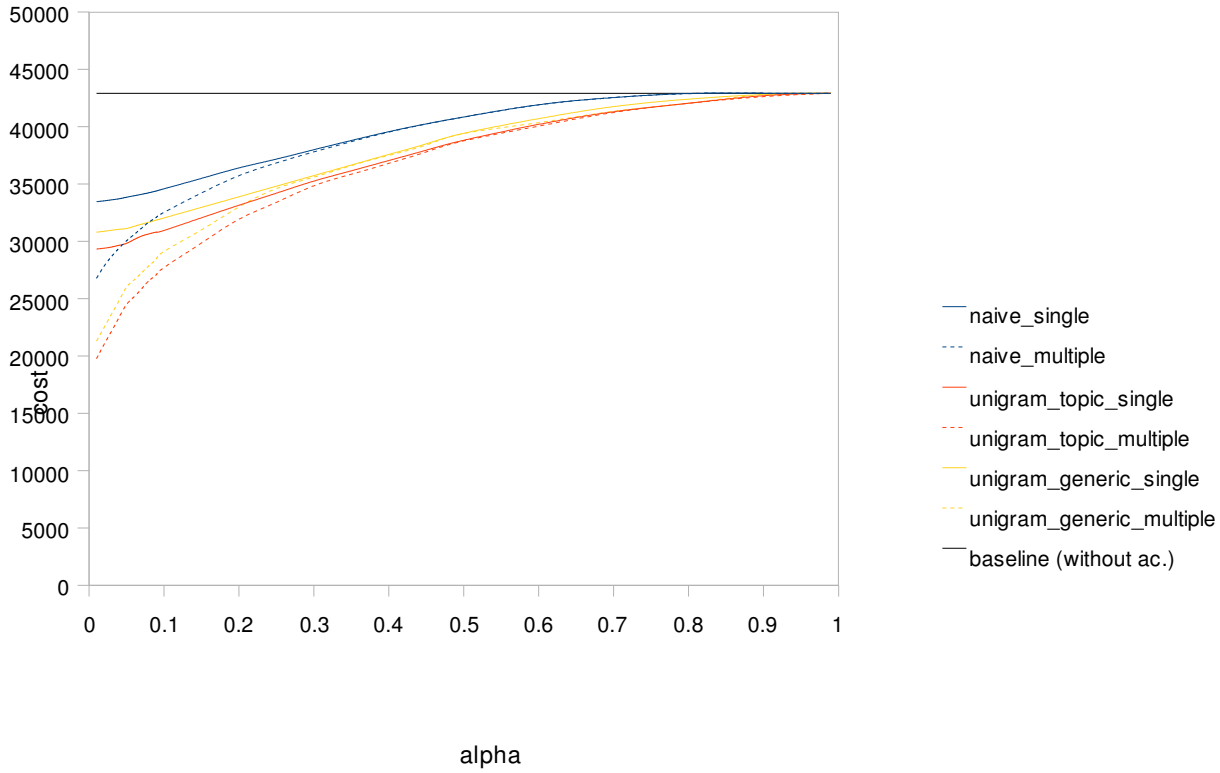
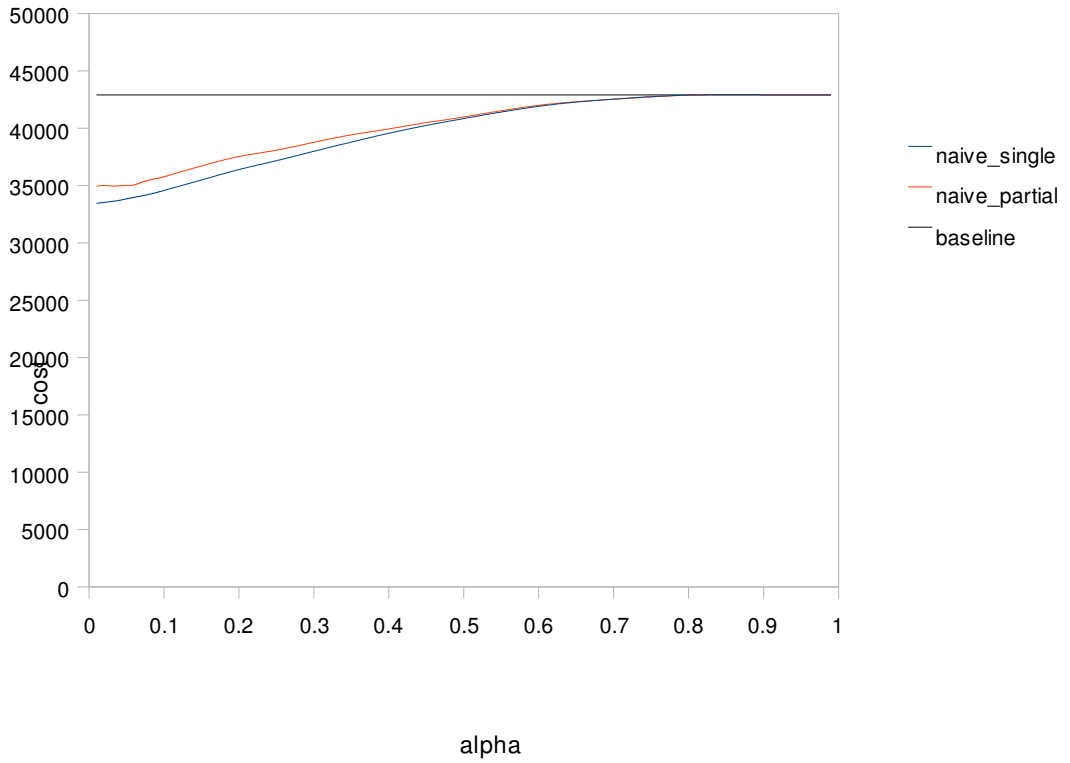


Figure 4.: Total cost, single suggestion, naïve unigram full/partial



6. Conclusions

As there are many different ways of implementing autocomplete systems and there has been a significant amount of research done in this field, I did not attempt to try out every possible method. Instead I experimented with a small set of rather arbitrarily picked ideas. More rigorous and more extensive testing would be needed to accurately assess the quality of the methods used. Also some of the assumptions made in these tests should be more thoroughly examined, especially the assumption that the user reads all suggestions and always selects the good one if it is available. This does obviously not hold in real usage, but it should be seen if there is any systematic error introduced this way or not.

The results show that using a language model yields a certain amount of improvement but the evidence is not conclusive that it is worthwile to use complex models instead of simple ones or just heuristic rules. The language model helps especially when it is built on text from similar domain with the actually typed text, therefore it is probably worthwile to try inferring the text topic. It seems probable that if autocomplete works at all, it does when it can exploit the peculiarities of a certain domain, rather than when it has strong generic knowledge of language.

In this work I just covered the surface of autocomplete systems, many aspects have not been discussed: prediction of phrases, rather than words, detecting the context (automatically choosing the background model based on the typed text), using different language models, using larger corpuses for training, etc. An overview of such methods can be found for example in [8]. It should be noted that as more complex language models are used, there is also a need for larger training corpus, and with the growing complexity of the lookup the responsiveness of the user interface could suffer.

7. References

[1] Witten, I.H., Cleary, J.G., and Darragh, J.J. (1983) "The reactive keyboard: a new technology for text entry" Converging Technologies: Proc Canadian Information Processing Society Conference, 151-156, Ottawa, ON, May.

[2] Adaptive Predictive Text Generation and The Reactive Keyboard
John J. Darragh and Ian H. Witten
May 1989

[3] Wikipedia :: Autocomplete
http://en.wikipedia.org/wiki/Autocomplete

[4] Wikipedia :: Word Completion
http://en.wikipedia.org/wiki/Word_completion

[5] Bidirectional autocomplete implementation and write-up (personal website)
http://www.Lkozma.net/autocomplete.html

[6] Google Suggest
http://www.google.com/webhp?complete=1&hl=en

[7] Wikipedia :: Radix Tree, Prefix Tree
http://en.wikipedia.org/wiki/Radix_tree
http://en.wikipedia.org/wiki/Trie

[8] Afsaneh Fazly: The Use of Syntax in Word Completion Utilities (M.Sc. Thesis)
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.310

[9] 20 Newsgroups data set.
http://people.csail.mit.edu/jrennie/20Newsgroups/

[10] T. Honkela and A. Hyvärinen.
Linguistic feature extraction using independent component analysis

[11] P. Brown, P. de Souza, et al
Class-Based n-gram Models of Natural Language (1992)

[12] Sami Virpioja
New methods for statistical natural language modeling (2006) - M.Sc. thesis

[13] Sven Martin, Hermann Ney, et al
Algorithms for bigram and trigram word clustering

Code

The scripts used in the experiments as well as the raw data files can be found at:
http://www.cis.hut.fi/lkozma/nlp