Master’s Thesis

Word prediction in an Internet chat

Patrik Sjöberg
patrik@stp.ling.uu.se

Language Engineering Programme
Department of Linguistics
Uppsala University
February 2001

Supervisors
Mats Dahllöf, Uppsala University
Åsa Ericsson, DoBeDo AB, Stockholm
Abstract

This thesis deals with the development of an Internet chat word predictor. A word predictor guesses what word a user is typing, based on previously written words and prefix letters of the current word. The predictor can then either insert the word with the highest probability into the text, or let the user choose an alternative from some kind of list interface, which is then inserted. Ideally, this can speed up and ease the user's typing of words.

The method used here for developing the predictor is an interpolated trigram formula and a bigram cache. The formula combines unigram, bigram and trigram information from a training corpora of 11 million words and builds a database that is used in the prediction lookup process. The cache is built during one chat session, from the submissions a user makes, and prediction suggestions are also taken from the cache.

The thesis further looks into the usability of the word predictor developed, and linked to this discussion there is also a small test where we count the number of keystrokes saved by using the word predictor. The test is handled by a program that tries to mimic the behaviour of a user of the word predictor, and the results show that the predictor is not as effective as one could wish for. Finally, there are some suggestions on how the word predictor could be improved.
Contents

1 Introduction .................................................. 1
   1.1 Presentation ........................................... 1
   1.2 Purpose ................................................ 1
   1.3 Plan .................................................... 1

2 Word prediction ............................................. 3
   2.1 Word predictors ........................................ 3
       2.1.1 Users .............................................. 4
       2.1.2 Environment ....................................... 5
       2.1.3 Invocation of predictions and presentation of suggestions . 5
       2.1.4 Learning .......................................... 6
   2.2 Language modeling ...................................... 7
       2.2.1 Training of a LM .................................. 8
       2.2.2 N-gram word-based partitioning of the history ........ 9
       2.2.3 N-gram class-based partitioning of the history ...... 10
       2.2.4 Smoothing ....................................... 11
       2.2.5 Long-distance N-grams ............................ 12
       2.2.6 Dynamic LMs - caches, triggers and topics .......... 13
   2.3 Evaluating word predictors ............................ 14

3 DoBeDo and the word predictor .......................... 15
   3.1 The DoBeDo environment ................................ 15
       3.1.1 DoBeDo language .................................. 16
   3.2 Word predictor interface and functionality ............ 17
3.2.1 Language model - cache and static lexicon ................................ 17
3.2.2 Cache and static lexicon interaction ....................................... 18
3.2.3 Learning in the DoBeDo environment ...................................... 18
3.2.4 Invocation of the predictor ..................................................... 18
3.2.5 Presentation and choice of suggestions .................................... 19
3.2.6 Language independence ........................................................ 20

4 Implementation ............................................................................. 21
4.1 Architecture .................................................................................. 21
4.2 Implementation of the LM ............................................................. 22
    4.2.1 Training data ........................................................................... 22
    4.2.2 Tokenisation ........................................................................... 23
    4.2.3 Static main lexicon ................................................................. 24
    4.2.4 Modifications of the main lexicon ........................................... 26
    4.2.5 Chat cache .............................................................................. 27
4.3 Implementation of the functionality of the Word Predictor ........... 27
    4.3.1 Invocation of the word predictor .............................................. 29
    4.3.2 History requirements and the predictions ................................ 29
    4.3.3 Presentation of suggestions and choice ................................... 30

5 Evaluation of the word predictor .................................................. 32
    5.1 The prediction strategy of the program ...................................... 32
    5.2 Results from the tests ................................................................. 34

6 Discussion ....................................................................................... 36
    6.1 The results .................................................................................. 36
    6.2 Improvements ............................................................................. 37
    6.3 Summary remarks ...................................................................... 37

Bibliography ...................................................................................... 37

Appendices ......................................................................................... 39

A Omitted ............................................................................................. 40
B  Database operations
   B.1  Data definition commands .............................. 41
   B.2  Stored functions ...................................... 42
      B.2.1  wordpredict ...................................... 42
      B.2.2  wordpredict2 ...................................... 43
   B.3  Table optimization ..................................... 44

C  Program files
   C.1  logExtract.java ....................................... 47
   C.2  ChatStatistics.java .................................... 48
   C.3  UpdatePolwithTest.java ................................. 62
   C.4  ChatCache.java ......................................... 74
   C.5  Bigram.java ............................................. 76
   C.6  Avatar.java ............................................. 77
   C.7  Text.java ................................................ 91
   C.8  WordPredict.java .................................... 93
   C.9  DbWordPredictHandler.java ............................. 94

D  Selected test of the LM ................................ 97
   D.1  A test with wordpredict ............................... 97
   D.2  A test with wordpredict2 ............................. 102
List of Figures

2.1 Example of word predictor suggestion list ......................... 3

3.1 The DoBeDo environment. ........................................ 16
3.2 Chat language ...................................................... 16
3.3 History requirements of the word predictor ....................... 19
3.4 DoBeDo word predictor suggestion list. ......................... 19

4.1 Architecture. ...................................................... 21
4.2 Training data ....................................................... 23
4.3 Example chat language taken from file testdata.txt ............. 23
4.4 Tokenisation ......................................................... 24
4.5 Bigram relation ..................................................... 25
4.6 Prediction flow schedule. ......................................... 28
List of Tables

5.1 Calculation of number of keystrokes .......................... 33
5.2 Suggestions top 10 .............................................. 34
5.3 Average position in suggestion list ............................ 34
5.4 Keystroke savings ................................................... 34
5.5 OOV ................................................................. 34
Chapter 1

Introduction

1.1 Presentation

Anyone that has used an Internet chat, and is not the world champion in speed typing, knows that there are times when you wish that there was some way to type faster. One way of achieving this could be to use a device that assist you in spelling out the words as you write them.

The subject matter of this thesis is word prediction, which can be used in exactly this way. Word prediction is about guessing what word a user is producing, either in a spoken or written discourse, with the assistance of what the user has said so far. The theoretical backbone to word prediction, which will also be covered in this thesis, is language modeling (often abbreviated LM hereafter).

1.2 Purpose

The aim of this thesis is to develop a word predictor based on a well-known language model, for an Internet chat. The program should be built with the same platform specifics as the chat, which means that the Java programming language and Oracle database functionality should be used. Further, the thesis should give a good overview of the area of language modeling.

A full-scale evaluation of the word predictor is not within the scope of this thesis. However a minor test of the developed program will be carried out, and based on the conclusions of the test several possible improvements to the word predictor will be suggested.

1.3 Plan

Chapter two introduces word predictors in general, where issues such as what environment is most suitable for word prediction and what kinds of users are most likely to benefit from word prediction are discussed. The chapter will also cover the area of language modeling, which is the theory on which word predictors rely. Here we will look at several statistical language models (LMs) and
how they can be interrelated. Chapter two gives good background knowledge that is needed in order to follow the discussions in the subsequent two chapters, which are more specific to the word predictor implementation in detail. In chapter three we will look at the environment of the DoBeDo chat and the word predictor, which has been integrated into this environment. We will see how the word predictor works with the user in the chat and also have a first look at the LM that it uses to make its predictions. There is a strong link between chapter three and the discussion in chapter two about word predictors and the different kinds of LMs that can be used in word predictors. Chapter three can be viewed as a user-oriented chapter, where functionality and usability is in focus. Chapter four on the other hand, is more technical by nature. Here we will see how the different programs are implemented, and what kinds of algorithms have been used. The client-server architecture, design of the lexicon, performance and database tuning issues are areas that will be dealt with in this section. Chapter five contains a small evaluation of the word predictor and chapter six is a concluding chapter with a discussion around possible improvements to the word predictor regarding the LM which it is based on, usability and technical issues such as where the lexicon of the predictor should be located.
Chapter 2

Word prediction

To begin with, word prediction should be separated from word completion in that it is more complex and not only base the predictions of words on already typed letters from the word to be completed. The most common information source for word prediction is the history of words preceding the word to be predicted, but there are also word predictors that use concepts such as word recency, long-distance dependencies, topic-guidance and many other sources. Some of these concepts will be dealt with in the section on language modeling. But first we will look closer at the nature of a word predictor, where issues such as functionality and usage are discussed.

2.1 Word predictors

A typical interface to a word predictor would be a list holding a limited number of relevant words picked out by the predictor. The list is then modified and pruned from incorrect alternatives as the user types more letters of the word. If the word prediction list contains the correct word the user can select it and go on to type the next word. For example, a user may have typed the sequence “I want to s” and a word predictor presents the suggestion list shown in figure 2.1.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>say</td>
</tr>
<tr>
<td>2</td>
<td>stay</td>
</tr>
<tr>
<td>3</td>
<td>stop</td>
</tr>
<tr>
<td>4</td>
<td>stand</td>
</tr>
<tr>
<td>5</td>
<td>shout</td>
</tr>
<tr>
<td>6</td>
<td>scream</td>
</tr>
<tr>
<td>7</td>
<td>spell</td>
</tr>
</tbody>
</table>

Figure 2.1: Example of word predictor suggestion list

Let us assume that the user wanted to type the word “spell”. The user can choose the word directly from the list, e.g. with the mouse or by pressing the
correct number. If the word is not in the suggestion list the user can type a new letter of the word, in this case “p” and then get a new list with suggestions. The latter would render “spell” to be the first (and only) word in the suggestion list. Often a space is inserted after the suggested word, which allows the user to continue typing immediately after the prediction. A possible, and often used, choice when implementing a word predictor could be to make the insertion of an alternative automatic when there is enough information to make that decision. “Enough information” is of course something that cannot be easily determined, but one obvious situation is when all other possible alternatives are excluded and the predictor has only one lexicon entry to choose from.

There are a number of important user-oriented issues to look into when building a word predictor and these are mainly:

- **Users**
  What kinds of users are best suited to use the word predictor?
- **Input environment**
  What kind of environment is the word predictor integrated with?
- **Invocation of predictions**
  When does the word predictor make its predictions and how are these triggered?
- **Presentation of suggestions**
  How are the word predictions shown to the user?
- **Learning**
  Both short-term learning (during one session with the word predictor and its related environment software) and long-term learning (the word predictor’s ability to learn over more than one session).

Apart from these matters there is of course the vital question of which sources the word predictor should base its predictions on, which will be discussed in the section on language modeling.

### 2.1.1 Users

The first word prediction software was developed as a means of speeding up typing and assisting individuals with physical disabilities by saving keystrokes (Heinisch & Hecht, 1993). Apart from being for people with physical disabilities word prediction can also assist individuals with poor spelling to use a greater variety of words (Hunt-Berg & Rankin, 1994; Laine & Follansbee, 1994).

However, although word prediction can develop writing skills regarding aspects such as word fluency, variety of words and motivation of writing, the tests completed so far have been “inadequate in scope and design” (DiGiovanni, 1995). More research is needed to be able to draw any really meaningful conclusions according to DiGiovanni (1995).

Further Heinisch & Hecht (1993) say that the ideal candidate for word prediction is someone who types with less than 20 WPM (words per minute), can recognise alternatives from a list or understand synthesised speech and can shift focus between keyboard, typing environment and word list environment.
2.1.2 Environment

As for the environment, or input software, that word predictors can be integrated with, most implementations of word predictors are linked to word processors, where the user of the word processing software can get assistance in the typing process. Word prediction could be useful in virtually all environments where there are many keystroke presses, and another example, which is the input environment of the word predictor developed within this thesis, is to use it in an Internet chat.

Another, and new area where word prediction has come in handy, is in the new devices with limited keyboard access, such as hand computers and cellular phones (Kronlid & Nilsson, 2000).

2.1.3 Invocation of predictions and presentation of suggestions

There are basically two different ways in which a word predictor can be triggered to give word suggestions:

1. user triggered word prediction
2. automatic or program-defined triggered word prediction

In a user triggered word predictor it is the person typing that invokes the predictor with an action of some sort, for instance by pressing a specific key on the keyboard. On the other hand, automatic triggering is handled by the word predictor itself, where the program gives suggestions whenever it feels it has enough information to perform such an action. In automatic triggering the word suggestion list could be constantly visible and modified for each new character the user types of the current word. If user-triggered, the word list could be in a popup format, which is not visible until the user has pressed the specific prediction key.

To have a word predictor that is constantly calculating what words are most likely to follow the words written so far, making computationally expensive look-ups against some kind of stored dictionary (e.g., in RAM or in a database) would require quite a lot of computational power. The obvious advantage of this approach is that the user can see the correct alternative in the word list as soon as the word predictor has found it and perhaps choose it at an earlier stage than with a user-oriented approach of predictions. On the other hand, to let the user decide when the word predictor should predict saves a lot of computational power, since the lookups against the dictionary are performed much more seldom. The disadvantage is that the user may prompt for a suggestion too early or too late. Too early would mean that the word predictor has not got enough information to go on and gives wrong suggestions in the word list. Too late would mean that key strokes could have been saved since the word predictor could have found the correct suggestion with fewer letters typed.

As for the presentation of the words in the suggestion list, there are two important aspects to consider:

1. Ordering of the suggestions
2. Number of suggestions

**Ordering of the suggestions**
The word list received from a prediction can be ordered in three plausible ways:

1. *Top down*, most to least probable word according to the word predictor
2. *Alphabetically*
3. By *length*

When the word predictor produces a list of outputs of possible words it uses a large dictionary, and sometimes there are quite a few alternatives that the predictor will present to the user. It would seem most natural to arrange the suggestions by how likely they are. However, since there sometimes are many suggestions in the list, it is not certain that this approach is the best one. Some users scan the list from top to bottom, while others intuitively search for some kind of pattern in the list in order to find the alternative they are looking for. Another way of ordering the suggestions could be alphabetically, which hypothetically can ease the user's scanning of the suggestion list. The same reasoning goes for the ordering of suggestions by length, which might improve the users ability to scan the list.

**Number of suggestions**
Finally, one must also decide how many suggestions should be presented to the user. Too many suggestions might confuse the user, and slow down the process where the user finds the correct alternative, or even make the user miss the correct alternative. Too few suggestions might render the correct alternative to stay outside of the suggestion list. A good way of handling this balance act is to let the users decide the number of alternatives for themselves.

**2.1.4 Learning**
Any really useful word predictor should use some kind of short-term and long-term learning.

Short-term learning is the feature that handles the fact that a word previously mentioned in a discourse (spoken or written) has an increased chance of occurring again later in that discourse. We can say that a word triggers itself, thus making it more probable in the remainder of the discourse. For instance, in this thesis, the word "language model" has a much higher probability of occurring than it would in any of August Strindberg's novels. A word predictor should increase the probability of words previously mentioned in a given discourse, since these are more likely to occur again in the remainder of the discourse.

Long-term learning can be viewed as a continuous modification of the word predictor over more than one discourse occasion, with incorporation of new words and alterations of the way the predictor gives word suggestions. Every user of a word predictor will have a specific vocabulary, and it is impossible for a
CHAPTER 2. WORD PREDICTION

word predictor in its initial state to have covered all the words every unique user may use. Thus, it is necessary for a word predictor to be able to incorporate new words (and sequences of words), which have not been encountered in the training and build up process of the language model.

This is a very difficult feature to implement. To make changes that affect the output of a word predictor permanently and not just during one session (as is the case with short-term learning) requires caution. The user can for instance produce ungrammatical sequences (e.g. “I has ached feet”) or misspellings (e.g. “Do yuo like vegetables?”), and a word predictor need some kind of detection system for avoiding the insertion of erroneous words (and sequences) into its dictionary. The simplest way to do this is to require that the word, or word sequence is used a decided number of times before inserted into the lexicon. More complex solutions include the use of spell checkers or grammar checkers.

Assuming one could come to an acceptable solution regarding inserting incorrect parameters into the language model there is another related and at least as serious problem with using long-term learning: to know exactly how new sequences and words should be related to the already existing model. Training of word predictors (more on this in the next section) is often performed on many million words (even hundreds of millions) and incorporation of new words or parameters into the existing language model of the word predictor must be done with this in consideration. Some kind of weighting of new information against that which is already in the lexicon is necessary. As far as the author knows there is no really satisfactory solution to how this should be done.

2.2 Language modeling

As has been mentioned previously, language modeling is the knowledge base for word predictors.

In language modeling one tries to find regularities and patterns in natural language and build models for the language that is explored. In the case of word predictors it is thus a question of estimating what word is most likely to follow a sequence of other words or parts of the language in question (often pieces such as punctuation marks are also considered when building a word predictor). Another example of a language model could be a set of grammar rules, which describe regularities in natural language, group words together in classes based on their behaviour and set up rules for the word classes interaction with each other. More areas where language models are useful are in speech recognition, where one tries to find the most likely word sequence uttered by an acoustic signal, spelling correction, where ambiguities could be solved with the assistance of a language model and optical character recognition, where the idea is basically the same as for speech recognition, only the output is letters and not sound. In short “every computer application that must process natural language with less then complete knowledge may benefit from language modeling” (Rosenfeld, R. 1994 pp. 3).

There are basically two ways of approaching language modeling. Rosenfeld (1994) uses the terms “knowledge based” and statistical language modeling.
Knowledge based models are built with the use of human linguistic knowledge and are hand coded, whereas statistical models are run on large amounts of data (natural language text in this case) that automatically shape the models’ parameters. With the high complexity and diversity of human language, which makes it difficult to get an overall picture of the language parameters even for a trained human eye, the absolutely most widely used approach is that of statistical language modeling. The most common statistical LM is the N-gram model which we will take a closer look at in the sections “N-gram word-based partitioning of the history” and “N-gram class-based partitioning of the history”. Several improvements have been proposed to the traditional word and class-based N-gram models and some of these will be dealt with later in this chapter, with references to more indepth articles.

2.2.1 Training of a LM

A statistical language model is, as has been mentioned, trained on large amounts of text. It is important that the model is trained on a similar domain as that in which it is then to be used. Rosenfeld (1994) shows that the results of a language model are significantly decreasing if the training text and the test text have different domains. He trained his LM on Wall Street Journal data, and then measured the perplexity the LM gave on test data taken from both the WSJ corpus and another corpus of AP wire stories. Perplexity can here be seen as the size of the set of possible words that a LM produces when predicting the next word given the history of words. More about this concept can be found in the section “Evaluating word predictors”. The perplexity of the AP data is twice that of the WSJ data. Further, Niesler & Woodland (1997) show that this is especially so when the LM is word-based. It is in other words easier to make a LM specific for some domain if it is word-based than if you use a category-based model. Rosenfeld (1994) shows in his PhD thesis of ways that you can train LMs both regarding within-domain adaptation and cross-domain adaptation. He mainly uses caches of different kinds and we will look a little bit closer on these in the following sections.

The fact that we need so many words to train our LM presents us with two other problems - the size of the LM and uncertainty regarding sparse data. According to Zipf’s law (Zipf, 1949) the product of the frequency of a word and its rank based on this frequency is approximately constant, \( f(w) \cdot r(w) \approx c \). This means that a few words occur many times and many words are very infrequent. So, the more words included in a training text, the more new words with low frequency counts there will be. In other words, the tail of the distribution increases and we get significantly more low-frequency words with uncertain probability estimates as a consequence when we increase the training text. For instance, if we have a text of 200 million words and we have only seen the word sequence “foot soldiers” once in all of this massive text chunk, then we do not really have much evidence that the word “soldiers” is likely to follow the word “foot”.

One way of handling this is to prune the training text of low frequency data, thus making the distribution smaller and base the LM on more reliable data. In an N-gram LM, the most common method is to remove the least frequent N-grams, which would be those with frequency 1 in most cases. According to Rosenfeld
& Seymore (1997b) cut-offs saves quite a lot of space while not affecting the
performance significantly. They used cut-offs of bigrams and trigrams on Wall
Street Journal data (45 million words) and saved memory required to store
the LM from 104 MB to 29 MB simply by removing bigrams and trigrams
occurring only once. 78 percent of the trigrams and 61 percent of the bigrams
were removed. The word error rate (WER) of course rises, but not disturbingly
much for lower cut-off values, which makes it a good method for pruning training
data.

There are other ways of pruning a LM and Rosenfeld & Seymore (1997b) for
instance show on a different approach based on weighted differences, which is
not based on the number of times an N-gram occurs in the training text. This
approach is used in a trigram backoff model (see the section “Smoothing” below)
and improves somewhat on the cut-off model based on counts alone, especially
when there is not a lot of training data to use.

We will now proceed and look at some different ways of building LMs in the
next few sections.

2.2.2 N-gram word-based partitioning of the history
First of all, when we speak of the probability of a word hereafter, we will use
the maximum likelihood estimate for the probability of an event, which states
that an event \( E \) which occurs \( r \) times out of a possible \( R \) is \( P(E) = r/R \).

In any given text written (or utterance spoken) in natural language there are
groups of words behaving in approximately the same way. There are further
specific words that tend to stand more often together than other words. For
instance, the words “why do” would form a bigram considerably more often
than the two words “why zero” and “do” is thus much more likely to follow
“why” than “zero”. To estimate the probability of a word sequence \( W \) one can
use the formula:

\[
p(W) = \prod_{i=1}^{n} p(w_i|w_1, \ldots, w_{i-1})
\]  

(2.1)

In other words, the probability of the sequence \( W \) is the product of the proba-
bilities that the separate words occurring in the sequence will follow each other.
\( w_1, \ldots, w_{i-1} \) is often referred to as the history \( h \). The word N-gram approach
uses a simplification of the use of \( h \) when predicting a word. \( P(w|h) \) with \( h \)
being the entire history of words up until the current word is not really possi-
table to use. Most histories would never occur more than once, rendering the
values to make the probability calculations on insufficient. Further, there is no
evidence to support the use of the entire history. Rosenfeld (1994) tells of an
investigation made in Huang et al. (1993) where it was examined if the use of
long-distance bigrams (see the section “Long-distance N-grams”) could decrease
the training-set perplexity, which is an indication of the average mutual informa-
tion (Rosenfeld, 1994) between words. For instance, a distance-3 trigram is
one which predicts \( w \) based on \( h = (w_{i-4}, w_{i-3}) \). The lower the training set per-
plexity the more information conveyed by the long-distance bigram \( (w_{i-4}, w_{i-3}) \)
regarding \( w \). The perplexity when using distances 1,2...10,1000 increased significantly from 1 through 5 and then remained virtually constant to 10. The test distance 1000 had however a higher perplexity than distances 6-10, indicating that there is still some more information conveyed at the 6-10 level.

Given these facts, the most widely used way of handling the history of a word \( w \) is to partition the last \( N - 1 \) words of the history \( h \) into equivalence classes \( Q \).

\[
p(w) = p(w|Q(h)) \tag{2.2}
\]

The choice of which number \( N \) to use should be decided based on the amount of training material that is available. The more refined equivalence class being used (i.e. increasing \( N \)), the less evidence there will be of the specific equivalence class for predicting a word, since there will be fewer examples of the more detailed equivalence classes. If we for instance set \( N=3 \), we have the most known of all statistical language models, the trigram model,

\[
p(W) = \prod_{i=1}^{n} p(w_i|w_{i-2}, w_{i-1}) \tag{2.3}
\]

Under this view, a string can be considered an Nth order Markov chain (Jelinek, 1999; Sang, 1998). A Markov chain is a well studied stochastic process that deals with the linear sequence of events, where the next event does not depend on all the previous events but is limited to a certain decided time-horizon. Directly translated to language modeling and the trigram language model, the output of the next word is merely depending on the two words preceding it.

### 2.2.3 N-gram class-based partitioning of the history

Even when the history of a word is partitioned into a shorter equivalence class, say bigrams, there will be many cases when a word of a new text has a history that has never been encountered when training the language model on some training text, thus giving the word a zero probability of occurring along with the specific history. Jelinek (1999) tells of an experiment IBM researcher performed in the 1970s where there was a test set of 300 000 words and a training set of 1 500 000, based on a corpus of patent descriptions with a 1000-word vocabulary. Of the trigrams in the test set, 23 % never occurred in the training set. The conclusion here is that a LM based on word trigrams would be clueless in 23 % of the test trigram cases, since it had never encountered these trigrams in the training data.

One way of remedying this is to cluster words into larger classes which reduces the parameter space spanned by word-based N-gram models. These classes could basically be of three different types:

- Linguistic
- Domain specific
- Data driven
Linguistic classes are most often built with part of speech (POS) tags, which try
to capture words’ linguistic roles, i.e. their relationship to the other words in
their context. Jelinek (1999) tells us of one such model built by IBM (Derouault
& Merialdo, 1986).

Domain specific classes are often built with the fact in mind that language
models in reality are built to function in specific domains where specific groups
of words behave in approximately the same way.

Data driven clustering entirely overlooks all knowledge based approaches to
creating the word classes. Instead one uses a large amount of data and derive
the classes by some statistical method.

A comparative evaluation of word-based and category-based LMs can be found
in Niesler & Woodland (1997), where they among other things came to the
conclusion that word-based language models outperform class-based models.
If we only use one or the other model, it is really not all that strange, since
the class-based model is much more general in its suggestions, and it therefore
has many more alternatives to choose from. Category-based N-gram language
modeling is most often used in conjunction with word-based language modeling,
which means that we can combine the accuracy of the word-based model with
the generality of the category-based model. There are many ways of combining
the two, but one feasible approach could be:

\[
p(W) = \prod_{i=1}^{n} p(w_i|w_{i-2}, g(w_{i-1}))
\]

where the part \( g(w_{i-1}) \) stands for the cluster into which \( w_{i-1} \) has been grouped.

Dupont & Rosenfeld (1997) discusses an interesting approach to language mod-
eling where they use a combination of several hierarchical N-gram models, rang-
ing from word-level to a very general category which are combined in a two-
dimensional lattice. The first dimension consists of the number of words in
the history (N-gram factor) and the other dimension consists of hierarchically
ordered word classes.

2.2.4 Smoothing

As we have discussed earlier, language is such a complex and wide information
source that it is very difficult to model with merely an N-gram word-based
LM. However, not even the use of a clustering model will remedy the fact that
very many sequences of higher N-grams will be completely new to any model,
regardless of how much training data we have built the model with. Interpolation
and back off are two ways of combining higher order N-gram models with
lower order N-gram models to make up for the scarcity of data.

The interpolated trigram formula looks like this:

\[
p(w_i) = (p(w_i|w_{i-2}, w_{i-1}) \cdot \lambda_3) + (p(w_i|w_{i-1}) \cdot \lambda_2) + (p(w_i) \cdot \lambda_1)
\]

A language model with this behaviour could be considered a hidden Markov
model (HMM) according to Jelinek (1999). With this LM, given any specific sequence of words, we try to find the alternative that gives the highest combination of all three N-gram probability estimates when predicting the next word. The 3 weights \( \lambda_i \) give the N-gram parameters different importance in the overall results of the model. These weights can be calculated with the Baum-Welch algorithm (also referred to as Baum or forward-backward algorithm) (Jelinek, 1999; Sang, 1998; Baum, 1972).

The convergence of this algorithm to a local maximum for the training text can be proven with for instance the Expectation-Maximisation algorithm (Jelinek, 1999). A major, unsolved problem with the algorithm is that is does not compute maximum values for the weights for yet unseen data (global maximum).

Let us use a small but explanatory example. Given these facts:

1. Text of 100 words
2. “you” occurs 5 times, “are” occurs 4 times, “are you” occurs 3 times, “how are you” occurs 4 times and “how are you” occurs 2 times

If we now write “how are “ and asks our language model to predict the next word, the word “you” would be given the following probability (given that we use the maximum likelihood estimate and the three different weights are \( \lambda_3 = 0, 40, \lambda_2 = 0, 40, \lambda_1 = 0, 20 \)):

\[
\frac{2}{4} \times 0, 4 + \frac{3}{4} \times 0, 4 + \frac{5}{100} \times 0, 2 = 0, 51
\]

(2.6)

If we have a new word sequence, say “strange things happen” and our LM has never encountered the history “strange things happen” in the training data, but has encountered the sequence “things happen” we get a situation where the interpolation formula would render the probability of this sequence to be the sum of the bigram “things happen” and the unigram “happen”. Had we not seen “things happen” either, the probability would have been the unigram “happen” on its own.

Backing off also takes advantage of the fact that it is easier to find examples of lower order N-gram models in training data. But instead of interpolating higher and lower N-grams it back off to lower order N-grams if there are unreliable frequency counts of higher order N-grams. In the above example, a trigram backoff model would have first tried to find the sequence “strange things happen” and then backed off to a lower order model when not finding this sequence. More about backing off can be found in Jelinek (1999) and Rosenfeld (1994).

Several improvements, apart from the already mentioned class-based models and the use of interpolation or backing off, have been proposed to improve on the LMs capability. Some of these are briefly covered in the following sections.

### 2.2.5 Long-distance N-grams

The use of an immediate history N-gram model has the obvious drawback of missing relationships further back in the history than the reach of the N-gram.
CHAPTER 2. WORD PREDICTION

To capture long-distance relations is not an easy thing to do. For instance, in the sentence “The meteorologist talked about the weather” the relation between “meteorologist” and “weather” is lost in anything less than a 5-gram model. This relationship would be captured by a distance-4 trigram model ($p(w_t)$ is based on $w_{t-5}, w_{t-4}$). However, the long-distance model has some serious deficiencies, where the fact that they do not capture relationships if these are on a distance separate from the decided distance (say 4). Had the word “meteorologist” been one step further back for instance, it would not have been captured by the distance-4 trigram.

In this aspect, long-distance models can be viewed static, i.e., they do not change over the period of a discourse but are set to predefined values from the beginning. In the next section we will look at dynamic models that use the previous information in the current discourse when trying to model the language.

2.2.6 Dynamic LMs - caches, triggers and topics

As we have mentioned earlier (in chapter two, section “Learning”), the previous use of a word (sequence) in a discourse increases the probability of that word of occurring again. One good way of capturing this is to use some sort of cache (for instance unigram-, bigram- or trigram cache). However, as Rosenfeld (1994) points out, the probability estimate deviates much less from that of a static model if it is a common word that is put in the cache. In this view, only words which are not so frequent and therefore much more surprising to encounter should be stored in the cache and hence get an increased probability. The same reasoning can be applied to bigrams and trigrams as well.

Moreover, there is of course a relationship between the distance of a word and its probability of occurring again. The further back in the history of a discourse, the less should one rely on the cache for predicting that word again, which a cache-based model should try to capture. For instance, Clarkson & Robinson (1995) has developed an exponentially decaying cache which ranks words closer in the history of the cache higher than those further back.

Rosenfeld (1994) uses another concept named triggers. In this approach, word A is said to trigger word B, thus increasing the probability of word B of occurring if word A occurs in a discourse. The strongest trigger according to the results yielded by the trigger approach is $A \rightarrow A$, i.e., the one word which is most likely to trigger another word is the word itself. One word can of course trigger many words, and this is also so in Rosenfeld’s trigger model.

One more way of improving a LM is to use topics, since different words and word sequences are more probable in different contexts. This could also decrease the size of the specific LM used for predicting a specific piece of text, since the different topic models could be made more directed towards some area. Two examples of LMs that use this approach are Weng et al. (1997) and Rosenfeld & Seymore (1997a).
2.3 Evaluating word predictors

The most common measurement of evaluating word predictors is to use the number of keystrokes saved when using the word predictor in the text typing process. This is also the measurement used when evaluating the word predictor within this thesis. The best predictors on the market save approximately 50 percent keystrokes (Carlberger, 1999) but most predictors save much less than this. To measure keystroke savings is the most straightforward approach of evaluating word predictors, although there are some issues that make it highly uncertain. For instance, it is directly related to the way each individual uses the word predictor (this is especially so if we have user-triggered word predictors) - how well the performance of the predictor can be exploited in other words.

Another way to measure the underlying model of a word predictor is to more directly evaluate the LM that it uses. From information theory we have the concepts of logprob and perplexity. If we have a text which a language model $p'$ has never encountered before, we can calculate the average logprob of that text with the following formula given that $K$ is the size of the text and $p'(w_i)$ is the probability that the language model gives to word $w_i$:

$$l_{p_k} = -rac{1}{K} \sum_{i=1}^{K} \log_2 p'(w_i|h_{i-1}) \tag{2.7}$$

Given this formula we can use the concept of perplexity, which is $2^{l_{p_k}}$, to determine the quality of a language model on a test text. Perplexity can be seen as a measurement of the average number of words the LM will choose from, when guessing every word in the test text, given the history of the test text. Typically, the lower perplexity a LM of a word predictor assigns to a text, the better that word predictor will predict the words of the text.

Since perplexity does not take into consideration the word error rate factor (in fact, a language model with a smaller vocabulary will give a better perplexity value since it will exclude rare words) Clarkson & Robinson (1995) has given a suggestion on a better model evaluation measure, which does take word error rate into consideration.
Chapter 3

DoBeDo and the word predictor

This section will give an overview of the environment the word predictor has been developed for - the DoBeDo chat. First we will see what the DoBeDo environment looks like, with an explanation of how the users communicate in the chat. Second, we will discuss the core of the chat - the language used by the users, which is unique in many ways, and discuss how this uniqueness may affect a word predictor to be used together with this language. Thirdly, we will look at the functionality added by the word predictor and see how it is used in the DoBeDo chat environment.

3.1 The DoBeDo environment

DoBeDo is a graphical world where you as a user is a small figure, in the DoBeDo world called “Avatar”. You can walk around in different “rooms”, and speak to other characters. When you start typing on the keyboard, or chooses to speak, shout, whisper or talk from a menu which pops up if you press the left mouse button on your Avatar, a talk bubble appears with the text you type in it. Figure 3.1 shows how it can appear in the DoBeDo chat:
DoBeDo is not a regular text-based chat since the users are actually different characters that can move around in a graphical environment, and not only “lines of text” appearing on the screen. However, DoBeDo is a regular chat if you pile off this extra spice and look at what is its core functionality - to work as a medium for people to meet and carry conversations with each other.

3.1.1 DoBeDo language

Internet chat language is very different from the language in a newspaper, a fiction novel or a children’s book. It could almost be said to be a genre of itself, with many specific letter sequences and symbols occurring that you would not see anywhere else. Some examples of sequences of symbols that occur in a chat can be found in figure 3.2. (all sequences have been taken from authentic DoBeDo chat language).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>:)</td>
<td>Smile</td>
</tr>
<tr>
<td>;)</td>
<td>Flirt</td>
</tr>
<tr>
<td>:=)</td>
<td>Teasing smile</td>
</tr>
<tr>
<td>&amp;-)</td>
<td>Glasses</td>
</tr>
</tbody>
</table>

An important aspect of chat language is the number of misspellings and grammatical errors that the chat users produce because of the online writing in the chat environment. The real-time factor naturally leads to a higher percentage of spelling typos and grammatical mind slips where people forget words or produce erroneous word sequences.

Moreover, the language used is much less formal than that produced in almost
any other text environment. People are basically talking to each other, only they use written language instead of spoken.

Another thing that can be of interest for a word predictor to be used in a chat is the fact that Internet chat language often is subject guided, with different rooms for different topics of conversation. For instance, one chat room may be for people that are in a party mood and another room for love sick people. Knowing this, it could be possible to develop a word predictor with specific vocabularies or to give different words higher probabilities depending on which room the predictor is triggered from. DoBeDo's chat is at the time being not really subject guided, but there are ways of expressing what mood one is in (e.g. happiness or sadness) and further ways of expressing how one makes a submission to the chat (e.g. talk, shout or think) and this information could very well be used in the prediction function. As of right now, it is however not used.

There are three summary remarks to keep in mind. First of all, it is crucial, in the author's opinion, that a word predictor that is to be used in an Internet chat is trained on chat language. In order to capture the specific language that is used in the chat environment the predictor needs to have encountered this kind of language. Second, which is a consequence of the first, we need to watch out when training the predictor on authentic chat language, since this language is full of errors. As far as possible, we should try to avoid that the errors made in the training material is reflected in the language model that the word predictor is based on. The third point to keep in mind is the fact that chat language is often subject guided with different rooms for different subjects. This is something which an Internet chat word predictor should try to take advantage of.

3.2 Word predictor interface and functionality

This section will describe the extra functionality added by the word predictor to the existing DoBeDo chat environment. It will be a user-oriented description of the word predictor, and not a description on how things work "behind the curtains". For a more technical description on how the word predictor is implemented and what kind of dictionary that is used the user is recommended to read chapter four, which describes the implementation in detail.

3.2.1 Language model - cache and static lexicon

There are two sources on which the DoBeDo word predictor makes its predictions.

1. Static main lexicon
2. Chat cache (short-term memory)

The static main lexicon is built using a statistical trigram model which has been trained on authentic chat material. It is the same for all users and stored in a database. More information on how the main lexicon is implemented can be found in the section on the main lexicon in chapter four.
CHAPTER 3. DOBEDO AND THE WORD PREDICTOR

The other part of the source of word prediction is a chat cache, which takes into consideration recency factors (see section “Dynamic Models - caches, triggers and Topics”) where a word is more likely to appear again within a discourse if it has been mentioned before in the same discourse. This dynamic chat cache is accumulated during one chat session. If the user types a bigram sequence for the second time during one session, this sequence has been stored in the chat cache and will be given as a suggestion if the user asks for a prediction. For instance, if the user has made one previous contribution to the chat saying: “I wanna talk to you” and during the same session has begun to type: “Because I wa” and then presses the key for a word prediction, the suggestion list will contain the word “wanna” regardless of whether this word is part of the main lexicon or not.

3.2.2 Cache and static lexicon interaction
Presently, the word predictor orders the chat cache suggestions before the static main lexicon suggestions, regardless of whether a specific chat cache bigram has been typed once or 5 times. However, it would be better to perform some form of calculation on the chat cache bigrams and integrate the suggestions from the chat cache into the descending order list of the lexicon suggestions.

3.2.3 Learning in the DoBeDo environment
There is no long-term learning in the word predictor developed for DoBeDo. There are many reasons for not incorporating new words into the already built static main lexicon, and these are discussed in the chapter two, section “Learning”.

The use of a chat cache can perhaps be viewed as a simple form of learning. The model “learns” a word, and at the same time increases the probability for this word to occur again within one chat session. The learning is not permanent though, meaning that the next time the user logs in to the chat, the word has not been incorporated into the main lexicon.

3.2.4 Invocation of the predictor
There is only one way in which the word predictor can be invoked right now, and this is a manual invocation done by the user by pressing <tab> (for a motivation on this, view chapter four, section “Invocation of the word predictor”). The word predictor will not carry out predictions, even when the user has pressed the specific word prediction key, if it feels it has not got enough information to make the prediction. For instance, the first word a user types need to have at least one letter typed if the predictor will even make a prediction. If there is only one word in the history which can be used to guess, the predictor requires that the word to be guessed has one letter. If there are two words in the history, the predictor can guess even if it has not been given any part of the word which it will try to predict.
<table>
<thead>
<tr>
<th>History</th>
<th>Prefix Length Needed to Predict</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>two words</td>
<td>0</td>
<td>Are you &lt;tab&gt;</td>
</tr>
<tr>
<td>one word</td>
<td>1</td>
<td>Are y&lt;tab&gt;</td>
</tr>
<tr>
<td>no history</td>
<td>1</td>
<td>A&lt;tab&gt;</td>
</tr>
</tbody>
</table>

Figure 3.3: History requirements of the word predictor

There is also another restriction model, which is less restrictive in its manner and make predictions based on less information. Some discussion around this can be seen in chapter five, or Appendix D, which reveals two different test runs with the two different restriction models.

We have now seen when the predictor will guess words, but not how they are presented to the user or how the user can choose from the suggestions the program has found. This will be covered in the next section.

### 3.2.5 Presentation and choice of suggestions

When the program has made a prediction, the results are presented in a list interface next to the talk bubble. If the user for instance has typed “Hello wo” and then asked for a prediction it could look like figure 3.4.

![DoBeDo word predictor suggestion list](image)

Figure 3.4: DoBeDo word predictor suggestion list.

The first word in the list is put in the talk bubble with a space after it, and if this suggestion is the correct one, the user can simply continue typing, which will render the suggestion list to disappear and the text written to appear after the inserted word in the talk bubble. However, if the first word in the list is not the correct one, the user has four different options to select from.

1. Choose another alternative from the list by pressing the correct number from the keyboard
2. Choose another alternative from the list by using the mouse
3. Move up and down the list by pressing <Page Up/Down> keys.
4. Press the <Esc> key if no suggestion is the wanted one.

If no suggestion at all are found, the program will alert the user of this with a pop up bubble which will inform the user of the failure to find any suitable alternatives. For more information on how this is implemented the reader is recommended to read the chapter four, section “Presentation of suggestions and choice”.

3.2.6 Language independence

The DoBeDo word predictor works for the English language only, since the main lexicon has been built using English chat text. This is, however, easily remedied, since the program is language independent. There are no grammar rules or syntactical classes in the lexicon, but it is strictly word based. Since there are a number of megabytes of new chat text produced each day in the chats (DoBeDo exist in Sweden, England and Germany), a new sufficiently large training corpora can be accumulated very fast.
Chapter 4

Implementation

4.1 Architecture

The DoBeDo chat is a typical client-server side system. The chat is written in Java. Here follow an overview of the system and the following sections will explain those parts that are of importance of the word predictor developed.

![Diagram of architecture](image)

Figure 4.1: Architecture.

First, we will talk briefly about the technical aspects of how the communications are set up between the DoBeDo chat clients and the DoBeDo chat servers, which is not really directly related to the word predictor implementation. The next sections will go into the different parts of that implementation in more detail.
The chat clients can operate as an application or an applet, and when entering the chat from inside the DoBeDo site, which is an Internet site, the client is run as an applet in the user’s browser. The client sends all user-related actions via a MUX server to a Chat server. The MUX server is a reflection of some chosen parts of the Chat server and also maintains network traffic regarding such things as when a client logs on or off (starting and ending sessions in the chat). The Chat server in its turn takes care of everything that happens in the chat, such as for instance when an avatar moves from one point to another, and sends the appropriate messages to those clients that are affected (in the case of a move, all clients in the same room as that clients which performs the move will be informed). “CMC_Message.java” is the class that makes up the messages that are sent back and forth between clients and servers. This message is also used for sending word prediction actions to the word predict server (via the chat server), which handles the prediction. We will look a little bit closer at this in the section “Implementation of the functionality of the Word Predictor”

4.2 Implementation of the LM

There are five different Java classes developed which are all part of the language model implementation (here we count both the database lexicon and the cache as parts of the language model):

- logExtract.perl - this program extracts pure text from the chat logs (more about the chat logs in the section “Training data”).
- ChatStatistics.java - the program that builds the database lexicon consisting of unigrams, bigrams and trigrams. The program uses the interpolated trigram formula when building the three different relations.
- UpdatePolwithTest.java - a program that can modify the unigram, bigram and trigram relations in the database regarding their interpolation values.
- ChatCache.java - accumulated short-term lexicon during one chat session of previous submissions the user has made to the chat. The information stored is in the shape of bigrams. The cache is located at the client side.
- Bigram.java - this class stores bigrams along with their frequencies.

The main lexicon built by “ChatStatistics.java”, and the other classes presented here, will be further investigated in the following sections.

4.2.1 Training data

The training data consists of authentic chat material, which is stored in log files. The files also contain a lot of information that is not chat language related, such as for instance information on when an avatar leaves a room, and the log files were therefore filtered to create pure chat language specific text files (see appendix A for an example section of a log file). In these files each line consists of one utterance of an avatar in the chat. The material used for training the LM is taken over a period of time of one month and ten days from the English
DoBeDo chat. Some facts about the text can be found in figure 4.2 (data generated with Unix < wc > command):

Before the language modeling program that builds the database has been applied to these text files, they have been appended to one large text file (≈ 60 MB).

Here follows some examples of utterances which can be found in a more thorough example in appendix A.

<table>
<thead>
<tr>
<th>Authentic chat submissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
</tr>
<tr>
<td>yeah she comes on dobedo eva seen her?</td>
</tr>
<tr>
<td>caaaaaalmmmmmmmm iiiiitititititit</td>
</tr>
<tr>
<td>need a swimsuit for this place.. lol</td>
</tr>
<tr>
<td>so is this supposed to be the Titanic I guess, huh?</td>
</tr>
<tr>
<td>i'll get you later booul</td>
</tr>
</tbody>
</table>

Figure 4.3: Example chat language taken from file testdata.txt

As discussed in the section on the DoBeDo environment, and as can to some extent be seen by these examples, chat language has many specific features, with special characters and symbols, and many typing and spelling errors. The former (the specificity of the language) is a question of how one should tokenise the text files in the language modeling program and this is discussed in the next section. The other part, with language errors that occur in the training data, is more difficult to handle. As was discussed in the section “The DoBeDo Language” one would like the language model to discard errors made by the users in the authentic training material, but this is one of the weaknesses of the language model developed in this thesis, it does not remove errors completely (or even to a satisfactory extent). There is mainly one reason for this and that is that chat language is so specific that it is difficult to know what is a language error or what is a stylistic variant used in the chat. In other words, it is not possible to compare the training data directly towards some lexicon, since many variants that are correct in the chat is simply not included in the lexicon.

There is however one way in which an effort has been made to remove spelling errors from the chat; by using a cut-off method (see section “Training of a LM“ in chapter two), which removes all unigrams, bigrams and trigrams only occurring once in the training set, many language errors have been removed. How this cutting-off is performed can be seen in the section “Static main lexicon”.

4.2.2 Tokenisation

When building the language model based on the training data, the first thing one has to decide on is what sequences of symbols make up words. As we have discussed earlier, there are very many kinds of sequences that are specific for the
language used in Internet chats. Hence, we have to be careful when construct-
ing a tokeniser for the chat text which is to be used for building the lexicon.
“ChatStatistics.java”, the program that builds the lexicon, has a tokeniser that
divides on the tokens in figure 4.4.

<table>
<thead>
<tr>
<th>Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>[space]</td>
</tr>
<tr>
<td>,</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>!</td>
</tr>
<tr>
<td>?</td>
</tr>
</tbody>
</table>

Figure 4.4: Tokenisation

With this tokenisation, words such as “:-)” will be part of the lexicon. Another,
quite difficult question to decide on, is whether punctuation marks should be
taken into consideration or not when building the lexicon. As of right now, they
are not used. In other words, in the sentence:
“I like you. Can you buy me a beer?” the word sequence “like you Can” is a
possible trigram, since the full stop is ignored.

It is, naturally, important to be consistent with the tokenisation of the words,
which are to be put in the lexicon, and the tokenisation of the sentences the
users write in the chat which are to be used in the lookup process. There are two
places in the chat client in which word sequences are tokenised. The chat cache
(see the section “Chat cache” in chapter four), which is accumulated during one
chat session, has the same tokenisation as the program that builds the main
lexicon. And similarly the tokenisation of the history that is used when making
a lookup is gathered using the same tokenisation as the other two.

As for the program that collects the main lexicon (“ChatStatistics.java”) it
works in such a way that it does not look for N-grams over line boundaries. So
if we have the sentences (on separate lines):
“Me and you forever baby!”
“Can we go somewhere private?”
There will not be any N-grams consisting of the sequences “forever baby Can”
or “baby Can” in the lexicon. This is so because every line of text in the file
that is processed for N-grams is supposed to be a whole chat text unit, i.e. a
submission to the chat of its own. And there is hence no reason to treat text
over line boundaries as units that have anything in common.

The next two sections will deal with the lexicons in more detail - the chat cache
and the database lexicon.

4.2.3 Static main lexicon

The main lexicon is stored as three different relations in an oracle 8i database
(McCullough-Dieter, 1998) and the lookups against the database is handled via
a word predict server (more about this in the section on the functionality of the
word predictor down below). The program that builds the database lexicon is,
as mentioned, “ChatStatistics.java” and there is also a program that updates the
relations regarding their interpolation values called “UpdatePolwithTest.java” (more about that program in the section “Modifications of the Main Lexicon”, and also in chapter five). The three relations are:

1. Unigram  
2. Bigram  
3. Trigram

The lexicon is based on the interpolated trigram formula described in the section “Interpolated Trigram Formula” and each row in the lexicon contains a value with how likely the last word in the relation is based on the words preceding it. For example, the bigram relation has the following four columns:

<table>
<thead>
<tr>
<th>WORD1</th>
<th>WORD2</th>
<th>FREQ</th>
<th>INTERPOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello</td>
<td>world</td>
<td>10</td>
<td>0.00027978</td>
</tr>
</tbody>
</table>

Figure 4.5: Bigram relation

The data definition language statements can be found in appendix B, which also contains the statements used to make lookups against these relations faster.

Here follows the algorithm used in “ChatStatistics.java” for creating the bigram relation (the algorithm is basically the same for all three relations so it is enough to show it for bigrams):

**Bigram frequencies:**

1. For each two words, calculate bigram frequencies:

   1.1. See if the bigram already exists in bigramONE (Java hash table)  
        - yes - put it in bifinal (Java hash table) with frequency 2.  
        - Remove it from bigramONE.  
        - Proceed with next bigram (1.1)  
        - no - go to 1.2.  

   1.2. See if the bigram already exists in bifinal  
        - yes - increase the bigrams frequency with one  
        - no - put the word in bigramONE.  
        - Proceed with next bigram (1.1)  

1.3. When finished counting bigrams - remove the bigramONE hashtable (cut-off to save memory)

**Bigram interpolation values:**

\[
\frac{C(word1, word2)}{C(word1) \ast INTPOLBIVALE} + \frac{C(word2)}{C \ast INTPOLUNIVALE}
\]

1. For each key/bigram in the hashtable:

   1.1. Get the bigram’s frequency \(C(word1, word2)\)  
   1.2. Get the first words frequency \(C(word1)\)  
   1.3. Get the second word’s frequency \(C(word2)\)
1.4. Calculate interpolation value (probability):
\[
(C(word_1, word_2)/C(word_1) \times \text{INTPOLBIVALE}) +
(C(word_2)/C) \times \text{INTPOLUNIVALUE}
\]

1.5. Put the words and the probability value into a BIGRAM relation in the database:
(see figure 4.5)

Using these two algorithms a lexicon consisting of 104 000+ unigrams, 480 000+ bigrams and 613 000+ trigrams has been built. As can be seen by viewing the algorithm, only words with a frequency of 2 or higher remains after the first frequency calculation, which in reality means that we have used a cut-off excluding all words occurring only once. This is so for all three relations.

Another thing worth mentioning is that the three relations all are in Boyce Codd Normal Form (Elmasri & Navathe, 2000), which was the minimum requirement set out before building the database LM. The tables also use some indexes to increase the speed of the lookup process, and these indexes together with an “explain plan” statement (which reveals what kind of searches the oracle SQL engine will perform) can be found in appendix B as well.

The weights have not been estimated with the baum-welch algorithm, but instead a test program (“updatePolwithTest.java”) has been developed that takes a test file as input and calculates the average position of each word in the test file as given by the LM. This program has then been used to see whether a change in the weights have produced better or worse results. This is not the optimal solution (which would be to train the LM with the baum-welch algorithm in this case) but it does give some credibility to the weights that are used.

We will soon look at the other component of the LM, the chat cache, but first we will see in what way the lexicon can be modified.

4.2.4 Modifications of the main lexicon

Since there is no long-time learning implemented in the word predictor, there is no way of adding new N-grams to the existing static database lexicon. The reason for this is because of the many difficult questions to answer regarding learning. For a closer discussion on this, see chapter two, section “Learning” and chapter three, section “Learning in the DoBeDo environment”.

But the weights, which are used when building the relations, can be changed to improve the accuracy of the word predictor. The program which is developed for this is, as said earlier, “updatePolwithTest.java”. It takes three new weight values and alters the probability value in all database relations. As mentioned in the previous section, this program does not use the baum-welch algorithm (see chapter three, section “Smoothing”) to optimize the weights. However, the LM can be tested against a test text, which we can see two examples of in chapter five and appendix D.

It is in, other words, only possible to rebuild the lexicon from scratch, or make changes to the interpolation weights if one would like to modify the main lexicon.
4.2.5 Chat cache

The chat cache is, unlike the database lookup process, located at the client side. There are two classes that make up the chat cache, “ChatCache.java” and “Bigram.java”.

As mentioned in chapter three the chat cache consists of all the bigrams a specific client has produced thus far in the discourse together with frequencies on how many times these bigrams have been produced (ChatCache.buildChatCache() called from Avatar). When a client presses the <tab> key for word prediction the first place to be searched is the chat cache (ChatCache.predictChatCache() called from Avatar). Keywords for the chat cache are:

- dynamic
- non-persistent
- user-unique

One immediate question to arise when implementing the chat cache was to decide on what N-gram model it should be built. As has been mentioned in the chapter on language modeling, the trigram formula suffers from the sparse data problem, with many trigrams only occurring once or a very few number of times. But to use unigrams (zero word history), which is the information unit with most data, was not really a good alternative either, since it is the most unreliable language model in the aspect that there is no previous words history to go on. Bigrams on the other hand are more common than trigrams while still preserving a certain amount of history (one word) which increases the chances of a correct prediction.

The chat cache has been implemented as a subclass to the Java Vector class. The cache will disappear as soon as the user logs out of the chat and is thereby not a persistent source of information for the word predictor as opposed to the main lexicon. The bigram class, which has been developed to be stored in the chat cache, consists of the two word strings making up the bigram and also holds information on how many times the bigram has been produced by the user. This way, chat cache suggestions can be ordered on how likely they are as well, with the simple use of frequency counts (this is not implemented as of now). The “ChatCache.java” and “Bigram.java” classes can be found in appendix C.

As of now, the cache suggestions are ordered before the main lexicon suggestions, and this is done when the two suggestion vectors with result alternatives are combined (Avatar.completeLMPrediction()).

4.3 Implementation of the functionality of the Word Predictor

The functionality of the word predictor can be divided between the client and server as we have seen earlier. We will first look at a flow schedule and then in more detail describe the different steps of the prediction process.
Figure 4.6: Prediction flow schedule.

**Client**

- **Avatar.java** - the class that represents the user’s graphical figure in the chat. All actions made by the Avatar are caught in this class. Here code was added that deals with word prediction actions by the user, including presentation and selection of suggestions. Furthermore, this is the class that builds and holds the chat cache

- **Text.java** - this class makes up the text that an Avatar submits to the chat. It handles, among other things, the typing in the text bubble (which is the container for the text). Here code was added that fetch the word history of a text backwards one or two words from the last word of the text, depending on how many words the history contains

- **PopupMenu.java** - this is merely a menu class and the graphical environment in which the word prediction suggestions are shown

**Server**

- **WordPredict.java** - server class which handles lookups against the main lexicon. This class sets up connections to the database and calls “DbWordPredictHandler.java”

- **“DbWordPredictHandler.java”** - this is the class that makes the actual query against the database
4.3.1 Invocation of the word predictor

The prediction function is not automatic but user-triggered (see chapter two, section “Invocation of predictions and presentation of suggestions”). The reasons are mainly two:

1. The client-server side architecture and the expensiveness of the prediction action
2. The graphical layout in the DoBeDo chat

Since the clients are connected to the mux server via a TCP/IP connection, which in its turn is connected to the chat server via a UDP connection, and the prediction lookup process is conducted against a fairly large database, there is simply no room for making predictions after every letter a client types. Every word prediction action takes between 500 and 3000 milliseconds on the server side, and with this in mind, plus the time for the prediction message to be transported back and forth between client and server, the server’s queue for handling the word predictions would rapidly increase to an unmanageable number.

Also, as we have seen in the chapter on the DoBeDo environment there is limited space to present anything to an Avatar, and when there are 50 different Avatars in a room, there should be as few distractions as possible for the users. Even if every client only see their own word prediction list (more about the presentation format in section “Presentation of suggestions and choice”) it would be enough to distract the user to have a constantly showing list. The list does take up some space, and would cover parts of the room that other Avatars produce talk bubbles in. Further, the prediction should not be something you have to use, but should be a feature that can be used if you choose to do so; therefore the predictions are user-triggered.

The reason why the key <tab> is used is simply because it is a key often used when getting completion suggestions (for instance in a Unix environment).

Next, we will look at what happens at the different stages in the flow schedule above, and what class take care of the specific stage.

4.3.2 History requirements and the predictions

When we have pressed <tab> “Avatar.java” catches this action and fetches the history of the text bubble from the class which models the bubble, “Text.java” (Text.getHistory() called from Avatar).

Since there is no meaning in trying to predict what a person is saying if there is no word history and no prefix letters (we simply have nothing to base our predictions on) the first thing that is checked is whether there is “enough history”. This is done both by the Avatar class and by the prediction function that is stored in the database. The function exists, as said earlier, in two different versions, but we will only look at the (more restrictive) function that is used in the chat (for a look at the other function, see appendices B and D). The reason why we have to use these restrictions is because of performance reasons, where a less restrictive function, which requires less prefix information, will take longer to execute.
1. Does history contain two words or does the prefix consist of more than zero prefix letters?
   yes - go to 2
   no - abort the prediction process and do not call the database function

2. wordpredict (oracle database stored function. See appendix B)
   2.1. does history contain two words?
      yes - go to 2.1.1.
      no - go to 2.2.

   2.1.1. does prefix contain at least two chars?
      yes - make a lookup against uni-, bi- and trigram relations
      no - go to 2.1.2

   2.1.2. does prefix contain at least one char?
      yes - make a lookup against bi- and trigram relations
      no - make a lookup against trigram relation only

   2.2. does history contain one word?
      yes - go to 2.2.1.
      no - go to 2.3.

   2.2.1. does prefix contain at least two chars?
      yes - make a lookup against bi-, and unigram relations
      no - go to 2.2.2

   2.2.2. does prefix contain at least one char
      yes - make a lookup against bigram relation
      no - we will not get here (see first condition of algorithm)

   2.3. does prefix contain one char?
      yes - make a lookup against unigram relation
      no - we will not get here (see first condition of algorithm)

When the prediction is finished on the server side (and the cache prediction is finished on the client side) the Avatar class will continue with displaying the suggestion, unless any of the following scenarios occur:

1. No predictions received from the lookup
2. The lookup process times out (it takes too long, a value set to 3 seconds)

If so, “Avatar.java” will present an information text bubble to the user stating that the prediction process has failed (Avatar.noPredictions()), go out of the prediction state and start listening for new prediction actions (and other types of actions as well).

However, if we have managed to stay on track this far, we will go to the next step, which is the presentation and choice of the suggestions. This is dealt with in the next section.

4.3.3 Presentation of suggestions and choice

The presentations are displayed in a class that is called “PopupMenu.java”. There are 10 suggestions displayed at a time (Avatar.showWordPredictions())
but you can scroll to see more suggestions if there are any. Maximum number of suggestions is set to 100, and more suggestions than this is not really plausible to think that a user cares to scroll through. The functionality of listening to what choice the user makes is entirely implemented in “Avatar.java”:

- Listen if the user presses a new letter which means that the suggestion in top of the suggestion list is the correct one (this suggestion is put in the text edit bubble as soon as the suggestions are shown)
- Listen to if the user presses a certain number in the popup and then put this suggestion in the talk bubble
- Listen to if the user presses <page up> or <page down> to see more alternatives
- Listen if the user presses <Esc> which will render the suggestion list to disappear

All these alternatives make the Avatar class go out of prediction mode and ready to receive new predictions. The two first alternatives will also insert a word into the talk bubble.
Chapter 5

Evaluation of the word predictor

Here we will look a little bit closer on the program that evaluates the behaviour of the word predictor. The program tries to use as much word history as possible, which, since the model developed is an interpolated trigram model, means that there are a maximum of two words in the history that can be used. There are the same restrictions on how the test program handles lookups as in the real chat word predictor, since they use the same oracle stored function, “wordpredict” (even if we have also used a second function “wordpredict2” in the test as well). For descriptions on the functions see appendix B.

The test program uses a text of previously unseen data to evaluate the word predictor. In other words, the data of the test text has not been used when building the LM. The text consists of 1000 chat submissions and a total of 4104 words. The tokenisation is the same as when building the LM (see chapter four, section “Tokenisation”) Two larger test run output sequences can be found in appendix D.

5.1 The prediction strategy of the program

We have tried to give the program some kind of strategy regarding when it makes predictions and how it chooses among the alternatives.

In the first test, where we have used the stored function “wordpredict”, the program tries to predict a word based on one prefix letter if the word consists of less than six letters. Otherwise the program uses two prefix letters to get prediction suggestions.

For the second test with “wordpredict2” we consistently use one prefix letter, unless the word consist of more than seven characters and there is no word history, when we will use two prefix letters. The test with “wordpredict2” will in other words generally have less history (regarding the prefix) when making its lookups.
As for scrolling and choosing among the suggestion alternatives (the program is assumed to have the same popup menu as the chat word predictor, meaning that there are 10 suggestions displayed at a time) there are a few rules the predictor follows. As the chat word predictor is implemented now, the only results presented to the user are the first 100 words given as suggestions from the LM. However, when conducting this test, we assume that a user would never scroll further back than 30 words. So, if the correct word is in the suggestion list at position > 30, it will not be discovered by the test program and consequently added to a OOV (Out-of-Vocabulary) rate instead.

The test program gives four different outputs:

- Data on how many times the suggested word is in position 1-10
- The average position in the suggestion list of the word (if it is among the first 30 suggestions)
- The average key saving percentage
- How many words the program does not predict (OOV)

The data on how many times the word given from the LM is in position 1-10 is a good estimation of the quality of the model in the chat environment, since the first 10 results are the most easily accessible for the user (the prediction presentation format shows a popup menu with the first 10 results).

The key saving percentage will be affected negatively every time the word in the test text is OOV. This is so because we have actively asked for a prediction by using a keystroke (since the predictor is implemented as a user-oriented triggered predictor) and will still have to type all of the letters of the word after the failed prediction. This is especially so if the program gives alternatives back from the prediction but the correct alternative is not in the list, since we need to scroll to the word at position 30 to discover this. If the program does not give any suggestion back from the prediction process, which could happen if the stored function feels there is not enough information and does not perform a lookup, or if the prefix letters used in the lookup process are not part of any beginning of any word in the three N-gram relations, the harm is lesser, since this will render only one extra keystroke (the triggering of the predictor).

So, a word 5 characters long in position > 30, would require:

<table>
<thead>
<tr>
<th>Number of keystrokes</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>typing the first letter of the word</td>
</tr>
<tr>
<td>1</td>
<td>asking for a suggestion</td>
</tr>
<tr>
<td>2</td>
<td>scrolling the suggestion list to position 30</td>
</tr>
<tr>
<td>1</td>
<td>press &lt;Esc&gt; and go out of prediction mode</td>
</tr>
<tr>
<td>4</td>
<td>the four remaining characters of the word</td>
</tr>
<tr>
<td>1</td>
<td>space after the word</td>
</tr>
</tbody>
</table>

Table 5.1: Calculation of number of keystrokes

This sums up to 10 instead of 6 keystrokes pressed, which is a considerable increase. However, this is a worst case scenario, and in reality most words will actually be in position 1-10, as we will see in the results from the two test runs.
5.2 Results from the tests

Here follow the results from two evaluations with the functions “wordpredict” and “wordpredict2”:

<table>
<thead>
<tr>
<th>Position</th>
<th>Counts wordpredict</th>
<th>Counts wordpredict2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1611</td>
<td>1586</td>
</tr>
<tr>
<td>2</td>
<td>404</td>
<td>414</td>
</tr>
<tr>
<td>3</td>
<td>279</td>
<td>271</td>
</tr>
<tr>
<td>4</td>
<td>230</td>
<td>221</td>
</tr>
<tr>
<td>5</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>6</td>
<td>87</td>
<td>104</td>
</tr>
<tr>
<td>7</td>
<td>72</td>
<td>76</td>
</tr>
<tr>
<td>8</td>
<td>57</td>
<td>66</td>
</tr>
<tr>
<td>9</td>
<td>54</td>
<td>58</td>
</tr>
<tr>
<td>10</td>
<td>42</td>
<td>56</td>
</tr>
</tbody>
</table>

Table 5.2: Suggestions top 10

As can be seen, very many words are among the first ten suggestions. The test text consists of 4104 words, and of these the program chooses an alternative 3060 times for “wordpredict” and out of these 3060 times the correct word is among the first 10 suggestion 2964 times. For “wordpredict2” the overall number of predictions is 3391 and out of these the program finds the suggestion among the first 10 alternatives 2980 times, which is slightly better.

<table>
<thead>
<tr>
<th>wordpredict average position</th>
<th>wordpredict2 average position</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.8792539</td>
<td>4.3850017</td>
</tr>
</tbody>
</table>

Table 5.3: Average position in suggestion list

“wordpredict2” gives a slightly higher average position in the alternative list, since it makes guesses based on less information. The average position here is for the words that actually are predicted and does not take words in positions above 30 into consideration.

<table>
<thead>
<tr>
<th>Keystrokes wordpredict</th>
<th>Keystrokes wordpredict2</th>
</tr>
</thead>
<tbody>
<tr>
<td>16002/19045 ≈ 16 % saved</td>
<td>15903 instead of 19045 ≈ 17 % saved keystrokes</td>
</tr>
</tbody>
</table>

Table 5.4: keystroke savings

These results will be discussed in chapter six.

<table>
<thead>
<tr>
<th>OOV wordpredict</th>
<th>OOV wordpredict2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1044</td>
<td>713</td>
</tr>
</tbody>
</table>

Table 5.5: OOV

The main reason why so many words are out of the vocabulary is because of the restrictions on the functions that make the lookups, where “wordpredict2” is less restrictive, and the fact that we are only interested in a suggestion if it
is among the first 30 alternatives. All words beyond this are, as said earlier, counted as OOV words.
Chapter 6

Discussion

6.1 The results

As can be seen, the test program developed for the word predictor in chapter five does not perform very well. It saves on an average less than 20 % of the keystrokes which is not all that good. Since the test is conducted in such a manner that only the first 30 alternatives are scanned and the word otherwise is considered OOV the OOV rate is very high (almost 1/4 of all words are not predicted). And since every OOV word actually increases the keystrokes with more than the length of the word, this contributes to the poor results.

Another problem is the test text itself. It is full of grammatical and spelling errors. A word that is misspelled in the test data causes the word predictor to fail, since the LM has cut off most spelling errors. Further, the simple fact that it is chat language is probably the largest problem for the word predictor. Chat language often consist of abbreviations and the chat submissions are most of the time very short (approximately 4 words per submission) which renders the predictor to have little word history to go on. Moreover the words used are on an average 4.75 characters long (19504/4104) which makes it difficult for a word predictor to save any keystrokes.

The chat cache is not used in this test program either, since there is no way to separate one persons chat submissions from all others in the logs. If this was possible, we could have stored previous submissions from one user and used the cache as well to get predictions.

Apart from the objections above, there is also the objection that the behaviour of the test program is very static. It tries to predict all words, regardless of whether the word that is to be predicted is a low-frequency word. A human could probably sense what words the predictor would best predict and not even try to get predictions when typing low-frequency words. This has also proven to be so when the author have used the word predictor, and in many cases received a much higher keystroke saving percentage (up to 50 %).

The test does not say anything about whether the typing in the chat is faster or slower with this word predictor. A reasonable guess would be that it does not
CHAPTER 6. DISCUSSION

speed up the typing in the chat environment. When using the word predictor, it takes approximately one to three seconds to get any suggestions back from the predictor. This is due to the large database lookups and the fact that we must transport the requests and answers over a network.

6.2 Improvements

Some improvements to the word predictor come rapidly in mind. First of all, the speed of the predictions are of essential importance for the program to be really useful, and it would therefore make much sense to place the dictionary at the client-side. Related to the speed issue is the design of the database where much could probably be done. As for the language model that is used, the first thing one would like to improve is to use classes in conjunction with the word-based approach. This would make the model more general and decrease the OOV rates. Another quite important thing is to improve on how the weights are decided, and use the baum-welch algorithm instead of a simple test program as is the case now. Finally, the implementation of the cache leaves many question marks. It does not take into consideration factors such as how recent a bigram has been used, how common that bigram is in the static language model, or how many times it has been used so far in the current discourse. In order for the word predictor to be really useful the submissions a user makes in one chat session should be learnt and somehow incorporated into the model over a longer period of time as well.

6.3 Summary remarks

The fact that the language we are trying to model is chat language has proven to be our worst enemy when developing the word predictor. Chat language is most of the time very specific. The words are short and the chat submissions equally so. In order for a word predictor to function really well in this environment, it is the author’s belief that we need to change the way people use their language in Internet chats. Longer submissions and longer words would naturally improve on the results of a word predictor. However, why should people change the way they express themselves because that would make a word predictor perform better? Still though, it would be interesting to investigate if the functionality of using a word predictor could change the way people express themselves in Internet chats.
Bibliography


Carlberger J. 1999, Word predict: Design and implementation of a probabilistic word prediction program, Master’s Thesis, Department of Speech, Music, and Hearing, Royal Institute of Technology (KTH), Stockholm.


Appendix A

Omitted

In the published version of the thesis, appendix A has been removed. Please contact DoBeDo AB if you would like to get this appendix.
Appendix B

Database operations

In order to fully understand the content of this appendix, the reader is recommended to have at least some experience of SQL, where areas such as optimization is of most importance.

Anything proceeded by a sequence of “−” here should be considered a comment, and not sql-code.

B.1 Data definition commands

Here follow the commands with which the three relations UNIGRAM, BIGRAM and TRIGRAM have been created.

```sql
create table unigrams
( word1 VARCHAR2(30),
  freq NUMBER,
  pol FLOAT);

create table bigrams
( word1 VARCHAR2(30),
  word2 VARCHAR2(30),
  freq NUMBER,
  pol FLOAT);

create table trigrams
( word1 VARCHAR2(30),
  word2 VARCHAR2(30),
  word3 VARCHAR2(30),
  freq NUMBER,
  pol FLOAT);
```

As can be seen, there are no primary keys or constraints specified. The reason why no primary keys are specified is because the indexes which then are created (primary keys automatically get indexes created on them) are too large and need to be treated a little bit specially. They are created later, after the tables are created instead (see the optimality section below). The reason why there are
no constraints on the bigram and trigram tables (that the words in them most occur in the unigram table) is because the program which inserts the N-grams already take care of this in its inner structure.

B.2 Stored functions

These are the functions (actually only one of them) used in the prediction process on the server side. See the class “DbWordPredictorHandler” for a more detailed view on how it is called from the java code.

B.2.1 wordpredict

This is the function that is used in the chat word predictor.

CREATE OR REPLACE FUNCTION wordpredict
(  
ORD1 IN UNIGRAMS.WORD1TYPE,  
ORD2 IN UNIGRAMS.WORD1TYPE,  
ORD3 IN UNIGRAMS.WORD1TYPE)  
RETURN cv_types.word_cursor IS  
resultset1 cv_types.word_cursor;
BEGIN  
-- If there are two words in the history and prefix  
-- is at least two letters, search all tables  
IF ORD1 IS NOT NULL AND ORD2 IS NOT NULL AND LENGTH(ORD3) > 2 THEN  
-- DBMS_OUTPUT.PUT_LINE(‘searching for trigrams and bigrams and unigrams’);  
OPEN resultset1 FOR  
select w1, max(poli) maxpol  
from  
(select word1 w1, pol pol1 from unigrams  
where word1 like ORD3  
union  
select word2 w1, pol pol1 from bigrams  
where word1 like ORD2 and word2 like ORD3  
union  
select word3 w1, pol pol1 from trigrams  
where word1 like ORD1 and word2 like ORD2 and word3 like ORD3  
)  
group by w1  
order by maxpol desc;
RETURN(resultset1);  
-- Else if there are two words in the history and LENGTH(ORD3) > 1,  
-- search for bigrams and trigrams  
ELSIF ORD1 IS NOT NULL AND ORD2 IS NOT NULL AND LENGTH(ORD3) > 1 THEN  
-- DBMS_OUTPUT.PUT_LINE(‘searching for trigrams and bigrams’);  
OPEN resultset1 FOR  
select w1, max(poli) maxpol  
from  
(select word2 w1, pol pol1 from bigrams  
where word1 like ORD2 and word2 like ORD3  
union  
select word3 w1, pol pol1 from trigrams  
where word1 like ORD1 and word2 like ORD2 and word3 like ORD3  
)  
group by w1  
order by maxpol desc;
RETURN(resultset1);
-- Else, if ORD2 is not null, search for trigrams only....
-- Too many results from bigram if one allows a zero prefix here
ELSIF ORD1 IS NOT NULL AND ORD2 IS NOT NULL THEN
-- DMIS_OUTPUT.PUT_LINE('searching for TRIGRAMS');
OPEN resultset1 FOR
select words3 from trigrams
where word1 like ORD1 and word2 like ORD2
order by pol desc;
RETURN(resultset1);
-- Else, if ORD2 is not null and prefix contains at least two chars,
-- search for bigrams and unigrams.
ELSIF ORD2 IS NOT NULL AND LENGTH(ORDs) > 2 THEN
-- DMIS_OUTPUT.PUT_LINE('searching for bigrams and unigrams');
OPEN resultset1 FOR
select w1, max(pol) maxpol
from
(select word1 w1, pol pol1 from unigrams
where word1 like ORD3
union
select word2 w1, pol pol1 from bigrams
where word1 like ORD2 and word2 like ORD3 )
group by w1
order by maxpol desc;
RETURN(resultset1);
-- else, if ORD2 is not null and prefix is at least one char,
-- search for bigrams only
ELSIF ORD2 IS NOT NULL AND LENGTH(ORDs) > 1 THEN
-- DMIS_OUTPUT.PUT_LINE('searching for bigrams');
OPEN resultset1 FOR
select word2
from bigrams
where word1 like ORD2 and word2 like ORD3
order by pol desc;
RETURN(resultset1);
-- else, if prefix is one or more chars, search unigrams only
ELSIF LENGTH(ORDs) > 1 THEN
OPEN resultset1 FOR
select word1
from unigrams
where word1 like ORD3
order by pol desc;
RETURN(resultset1);
END IF;
END wordpredict;

B.2.2 wordpredict2

This function will make predictions more often, based on less information than
the wordpredict function.

CREATE OR REPLACE FUNCTION wordpredict2
(
ORD1 IN UNIGRAMS.WORD1TYPE,
ORD2 IN UNIGRAMS.WORD1TYPE,
ORD3 IN UNIGRAMS.WORD1TYPE)
RETURN cv_types.word_cursor IS
resultset1 cv_types.word_cursor;
BEGIN
-- If there are two words in the history,
-- search all tables
IF ORD1 IS NOT NULL AND ORD2 IS NOT NULL THEN
-- DMIS_OUTPUT.PUT_LINE('searching for trigrams and bigrams and unigrams');
OPEN resultset1 FOR
select wi, max(pol1) maxpol
from
(select word1 wi, pol pol1 from unigrams
where word1 like ORD3
union
select word2 wi, pol pol1 from bigrams
where word1 like ORD2 and word2 like ORD3
union
select word3 wi, pol pol1 from trigrams
where word1 like ORD1 and word2 like ORD2 and word3 like ORD3
)
group by wi
order by maxpol desc;
RETURN(resultset1);
-- Else, if ORD2 is not null,
-- search for bigrams and unigrams.
ELSIF ORD2 IS NOT NULL THEN
-- DMIS_OUTPUT.PUT_LINE('searching for bigrams and unigrams');
OPEN resultset1 FOR
select wi, max(pol1) maxpol
from
(select word1 wi, pol pol1 from unigrams
where word1 like ORD3
union
select word2 wi, pol pol1 from bigrams
where word1 like ORD2 and word2 like ORD3
)
group by wi
order by maxpol desc;
RETURN(resultset1);
-- else, if prefix is one or more chars, search unigrams only
ELSIF SUBSTR(ORD3) > 1 THEN
OPEN resultset1 FOR
select word1
from unigrams
where word1 like ORD3
order by pol desc;
RETURN(resultset1);
END IF;
END wordpredict2;

B.3 Table optimization

- Create the plan table which is used to analyze costs of the above stored function on the three tables UNIGRAM, BIGRAM and TRIGRAM:

```
CREATE TABLE plan_table
  (statement_id VARCHAR2(30),
   timestamp DATE,
   remarks VARCHAR2(90),
   operation VARCHAR2(30),
   options VARCHAR2(30),
   object_node VARCHAR2(128),
```
APPENDIX B. DATABASE OPERATIONS

object_owner VARCHAR2(30),
object_name VARCHAR2(30),
object_instance NUMERIC,
object_type VARCHAR2(30),
optimizer VARCHAR2(256),
search_columns NUMERIC,
id NUMERIC,
pid NUMERIC,
position NUMERIC,
cost NUMERIC,
cardinality NUMERIC,
bytes NUMERIC,
other_tag VARCHAR2(256),
other LONG);

- Analyze before any indexes have been applied

-- First delete any previous entry with the same statement_id:
delete from plan_table where statement_id = 'f_s';
-- Run the query which is then stored in the plan table
explain plan set statement_id = 'f_s' for
select w1, max(p1) maxp1
from
(select word1 w1, pol1 pol1 from unigrams
where word1 like 'a%'
union
select word2 w1, pol2 pol2 from bigrams
where word1 like 'how' and word2 like 'a%'
union
select word3 w1, pol3 pol3 from trigrams
where word1 like 'hello' and word2 like 'how' and word3 like 'a%'
) group by w1
order by maxp1 desc;

- See the costs for the query above:

SELECT LPAD(' ',2*(LEVEL-1))||operation||' ||options
||' ||object_name ||' || object_type ||
DECODE(id, 0, 'Cost = ||COST3 expl_plan
FROM plan_table
CONNECT BY PRIOR id = parent_id AND statement_id = 'f_s'
START WITH id=0 AND statement_id = 'f_s'

EXPL_PLAN
-----------------------------------------------
SELECT STATEMENT
Cost =
SORT ORDER BY
SORT GROUP BY
VIEW
SORT UNIQUE
UNION-ALL
TABLE ACCESS FULL UNIGRAMS
TABLE ACCESS FULL BIGRAMS
TABLE ACCESS FULL TRIGRAMS
9 rader.

- Comments: No indexes used, full table scan of all three tables
- Create indexes for the words in the tables:
-- Unigrams:
CREATE UNIQUE INDEX "WORDPREDICT"."UNIGRAM_INDEX1" ON
"WORDPREDICT"."UNIGRAMS" ("WORD1") TABLESPACE "WORDPREDICT"
STORAGE ( INITIAL 200K NEXT 400K MAXEXTENTS 1000 PCTINCREASE 20);

-- Bigrams:
CREATE UNIQUE INDEX "WORDPREDICT"."BIGRAM_INDEX12" ON
"WORDPREDICT"."BIGRAMS" ("WORD1", "WORD2") TABLESPACE "WORDPREDICT"
STORAGE ( INITIAL 200K NEXT 400K MAXEXTENTS 1000 PCTINCREASE 20);

-- Trigrams:
CREATE UNIQUE INDEX "WORDPREDICT"."TRIGRAM_INDEX123" ON
"WORDPREDICT"."TRIGRAMS" ("WORD1", "WORD2", "WORD3") TABLESPACE "WORDPREDICT"
STORAGE ( INITIAL 400K NEXT 800K MAXEXTENTS 10000 PCTINCREASE 20);

• Analyze after unigram, bigram and trigram indexes have been created:

-- See the new costs for the query above:
SELECT LPAD(' ',2*(LEVEL-1))||operation||' '||options
|| object_name || ' ' || object_type ||
DECODE(id, 0, 'Cost = '||COST) expl_plan
FROM plan_table
CONNECT BY PRIOR id = parent_id AND statement_id = 'f_s'
START WITH id=0 AND statement_id = 'f_s'

EXPL_PLAN
---------------------------------------------------------------------------------
SELECT STATEMENT
Cost =
SORT ORDER BY
SORT GROUP BY VIEW
SORT UNIQUE
UNION-ALL

TABLE ACCESS BY INDEX ROWID UNIGRAMS
INDEX RANGE SCAN UNIGRAM_INDEX1 UNIQUE
TABLE ACCESS BY INDEX ROWID BIGRAMS
INDEX RANGE SCAN BIGRAM_INDEX12 UNIQUE
TABLE ACCESS BY INDEX ROWID TRIGRAMS
INDEX RANGE SCAN TRIGRAM_INDEX123 UNIQUE

12 rader.

• Comment: Now we use indexes instead when searching the three tables. Much better!
Appendix C

Program files

The programs are not presented in any specific order. Some of the programs are merely displayed with the code that is of interest for the word prediction functionality.

C.1 logExtract.java

# File: logExtract.pl
# Date: 2000-04-03
#
# (c) 2001, DoBeDo AB. All Rights Reserved

# This program goes through a log text file and extracts chat submission
# The values of interest are in columns 11 and 16
#extract('@ARGV');

sub extract
{
    # Get filename and open the file
    my($inFile) = @ARGV[0];
    open IN, "<$inFile";

    # Open output file with the same name as the infile + EXTRACTED as prefix
    open OUT, ">EXTRACTED$inFile";

    # Print a confirmation
    print "Now starting extraction....\n";

    # Go through the input file line by line
    while(<IN>)
    {
        # Split current line on tabs. This will create a list with each
        # column of the log file as an array element
        @currLine = split(/\t/,$.);

        # If the sixth column is a 2 - something is being communicated
        if($currLine[5] == 2)
        {
            # Add the communicated string to the output file
            print OUT,
        }
    }
}
C.2 ChatStatistics.java

This is the program that collects N-grams from the training data and puts them into three relations in an oracle database.

```java
/*
 * File: ChatStatistics.java
 * Date: 2000-07-04
 *
 * (c) 2001, DoBeDo AB. All Rights Reserved
 */
import java.util.*;
import java.awt.*;
import java.io.*;
import java.sql.*;

/** This program will go through a file, line by line, and collect N-grams (uni-, bi- and trigrams) together with their frequencies and also calculate a interpolation value
   It takes a text file as argument
 */
public class ChatStatistics
{
   // Database related variables
   // Connection
   private Connection con;

   // developing.dobedo.com=

   // Prepared statements for inserting values into the relations
   // UNIGRAM, BIGRAM and TRIGRAM
   private PreparedStatement insertUnigram;
   private PreparedStatement insertBigram;
   private PreparedStatement insertTrigram;

   // The tables which I make insertions to
   private String UNIGRAM_TABLE = "UNIGRAMS";
   private String BIGRAM_TABLE = "BIGRAMS";
   private String TRIGRAM_TABLE = "TRIGRAMS";

   // Counter of total number of words (the dictionary)
   private int C=0;

   // Input file
   private static String fileName;
```
// HashTables that will contain Ngrams
private Hashtable unigramONE = new Hashtable();
private Hashtable bigramONE = new Hashtable();
private Hashtable trigramONE = new Hashtable();
private Hashtable unigramFINAL = new Hashtable();
private Hashtable bigramFINAL = new Hashtable();
private Hashtable trigramFINAL = new Hashtable();

// Variables used when calculating the interpolated trigram formula
private static final float INTPOL_UNI_VALUE = 0.03f;
private static final float INTPOL_BI_VALUE = 0.47f;
private static final float INTPOL_TRI_VALUE = 0.50f;

// This creates the tables that are used for storing the LM
// UNIGRAM | BIGRAM | TRIGRAM
public void createTables()
{
    try {
        Statement stat = con.createStatement();

        // Drop tables (if the program has been run before)
        // This also drops indexes on the tables
        stat.executeUpdate("drop table " + BIGRAM_TABLE);
        stat.executeUpdate("drop table " + TRIGRAM_TABLE);
        stat.executeUpdate("drop table " + UNIGRAM_TABLE);

        // Create tables (without primary keys)
        stat.executeUpdate("create table " + UNIGRAM_TABLE +
"( word1 VARCHAR2(30)," +
"freq NUMBER," +
"pol FLOAT" +
")");

        stat.executeUpdate("create table " + BIGRAM_TABLE +
"( word1 VARCHAR2(30)," +
"word2 VARCHAR2(30)," +
"freq NUMBER," +
"pol FLOAT" +
")");

        stat.executeUpdate("create table " + TRIGRAM_TABLE +
"( word1 VARCHAR2(30)," +
"word2 VARCHAR2(30)," +
"word3 VARCHAR2(30)," +
"freq NUMBER," +
"pol FLOAT" +
")");

        System.out.println("**** Tables created! **** \n");
    } catch(SQLException sqle) {
        System.out.println("TABLE ERROR " + sqle.getMessage());
    }
}
public void createIndexes()
{
    try {
        Statement stat = con.createStatement();

        // Create indexes on these tables
        stat.executeUpdate("create index POL_UNIS_INDEX on "+ UNIGRAM_TABLE + " (POL)");
        stat.executeUpdate("create index POL_BIS_INDEX on "+ BIGRAM_TABLE + " (POL)");
        stat.executeUpdate("create index POL_TRIS_INDEX on "+ TRIGRAM_TABLE + " (POL)";

        // These indexes are instead of the primary key indexes
        stat.executeUpdate("create index UNIGRAM_INDEX_i on " + UNIGRAM_TABLE + " (WORDS)";
        stat.executeUpdate("create index BIGRAM_INDEX_12 on " + BIGRAM_TABLE + " (WORD1,WORD2)";
        stat.executeUpdate("create index TRIGRAM_INDEX_123 on " + TRIGRAM_TABLE + " (WORD1,WORD2,WORD3)";

        stat.executeUpdate("create index BIGRAM_INDEX_WORD1 on " + BIGRAM_TABLE + " (WORD1)";
        stat.executeUpdate("create index BIGRAM_INDEX_WORD2 on " + BIGRAM_TABLE + " (WORD2)";
        stat.executeUpdate("create index TRIGRAM_INDEX_WORD1 on " + TRIGRAM_TABLE + " (WORD1)";
        stat.executeUpdate("create index TRIGRAM_INDEX_WORD2 on " + TRIGRAM_TABLE + " (WORD2)";
        stat.executeUpdate("create index TRIGRAM_INDEX_WORD3 on " + TRIGRAM_TABLE + " (WORD3)";

    } catch(SQLException sqle)
    { System.out.println("INDEX ERROR "+sqle.getMessage()); }
}

public void countNgrams()
{
    // Threads
    Uni uni = new Uni();
    Bi bi = new Bi();
    Tri tri = new Tri();

    try {
        // Start threads that will count Ngrams (accumulates a lot of memory!)
        uni.start();uni.join();
        bi.start();bi.join();
        tri.start();

        // The method should not finish until they have all finished
        //uni.join();
        //bi.join();
        tri.join();
    } catch(InterruptedException ie)
{  
  System.out.println("Interruped thread");
  }
}

// Method which starts three threads that interpolates the values  
public void interpolateNgrams()  
{  
  Uni2 uni2 = new Uni2();  
  Bi2 bi2 = new Bi2();  
  Tri2 tri2 = new Tri2();  
  
  try  
  {  
    uni2.start(); uni2.join();  
    bi2.start(); bi2.join();  
    tri2.start(); tri2.join();  
  }  
  catch(InterruptedException ie)  
  {  
    System.out.println("Interruped thread");  
  }
}

// Constructor, starts submethods which builds the language model  
public ChatStatistics()  
{  
  try  
  {  
    // load the driver  
    Class.forName(driver);  
    con = DriverManager.getConnection(url, schema, pw);  
  }  
  catch(SQLException sqle){System.out.println("SQL "+sqle.getMessage());}  
  catch(ClassNotFoundException cle){System.out.println("CLASS "+cle.getMessage());}
  
  // Create tables UNIGRAMS, BIGRAMS and TRIGRAMS used for  
  // storing the predictor dictionary  
  createTables();  
  System.out.println("**** Start counting N-grams ****\n");  
  // Start count of word sequences  
  countNgrams();  
  System.out.println("**** Start Interpolation process ****\n");  
  // Count interpolation values  
  interpolateNgrams();  
  System.out.println("**** Create Indexes ****\n");  
  // Create indexes on the tables UNIGRAMS, BIGRAMS and TRIGRAMS  
  //createIndexes();  
  System.out.println("\n************* FINISHED! *************\n");
}
// Get the filename and start the collection process
public static void main(String args[]) {
    if (args.length > 0) {
        fileName = args[0];
        new ChatStatistics();
    } else {
        System.out.println(" Ange fil som argument");
    }
}

// Inline classes for counting word sequences
public class Uni extends Thread {
    public void countUnigrams() {
        // Variables for keeping account of bigrams, trigrams etc
        int count = 0;

        // The current line and current word being processed
        String newLine;
        String word;

        // The tokenizer which divides lines into words
        StringTokenizer tok;

        // Create a stream to the infile
        BufferedReader stream;
        try {
            stream = new BufferedReader(
                new FileReader(fileName));

            // Go through input file
            while ((newLine = stream.readLine()) != null) {
                // Split the line on different delimiters.
                // These delimiters must be the
                // same as those used in Text.java and
                // Avatar.java when dealing with the
                // history of what the user says.
                tok = new StringTokenizer(newLine, "\n\r\t\f .!?"");

                // Reset count to 0 on each iteration
                // (if I want bigrams and trigrams that
                // stretch across lines, don’t reset this counter)
                // As for now, the input file contains one users
                // submission on each line. Every
                // line is thus an independent piece of text
                count = 0;

                // Build unigrams with the current line
                while (tok.hasMoreElements()) {
                    word = tok.nextToken();
                }
            }
        } finally {
            stream.close();
        }
    }
}
// See if the word already exists in unigramONE
// (which the word is put
// in the first time it is encountered)
// yes - put it in unifinal with frequency 2.
// Remove it from unigramONE.
// Increase total number of words count
// Proceed with next word
if(unigramONE.containsKey(word))
{
    unifinal.put(word,new Integer(2));
    unigramONE.remove(word);
    C++;
}
// no -
// 1.2. See if the word already exists in unifinal
// yes - increase the words frequency in the unifinal hash
// no - put the word in unigramONE with frequency one.
// Proceed with next word
else
{
    if(unifinal.containsKey(word))
    {
        Integer freq = (Integer) unifinal.get(word);
        unifinal.put(word,new Integer(freq.intValue()+1));
    }
    else
        unigramONE.put(word,new Integer(1));
}
}// End of line

// End of file

// When finished counting unigrams -
// remove the unigramONE hashtable (save memory)
System.out.println("Remove # words with frequency 1: "+
unigramONE.size());
unigramONE.clear();
System.out.println("Total number of unigrams: "+
unifinal.size());

// Close the stream
stream.close();

} catch(IOException ioe){System.out.println("Error opening file "+
fileName);}

}

// The threads run method, will finish when the
// count of unigrams is done
public void run()
{
    System.out.println("Start UNI");
    countUnigrams();
}

// Class for Bigram count
public class Bi extends Thread
{
public void countBigrams()
{
    // Variables for keeping account of bigrams, trigrams etc
    int count = 0;

    // Current line and words
    String prevW = "";
    String word = "";
    String newLine;

    // Tokenizer which divides lines to words
    StringTokenizer tok;
    BufferedReader stream;
    try
    {
        stream = new BufferedReader(
            new FileReader(fileName));

        // Go through input file
        while((newLine = stream.readLine()) != null)
        {
            tok = new StringTokenizer(newLine,",\n\r\t\f ,,.!?"");

            // Reset count to 0 on each iteration
            // (if I want bigrams and trigrams that
            // if stretch across lines, don’t reset this counter)
            count = 0;

            // Build bigrams with the current line
            while (tok.hasMoreElements())
            {
                // prevW contains the word before this word
                word = tok.nextToken();

                // For bigrams, must have at least one previous word read
                if(count > 0)
                {
                    // Create a "bigram" (a vector with two elements)
                    // which contains {prevW,word}
                    Vector bigr = new Vector();
                    bigr.add(prevW);
                    bigr.add(word);

                    // See if the bigram already exists in bigramONE
                    // yes - put it in bifinal with frequency 2.
                    // Remove it from bigramONE.
                    // Proceed with next bigram
                    if(bigramONE.containsKey(bigr))
                    {
                        bifinal.put(bigr,new Integer(2));
                        bigramONE.remove(bigr);
                    }
                    else
                    {
                        // yes - increase the bigrams frequency
                        // no - put the word in bigramONE.
                        if(bifinal.containsKey(bigr))
                        {
                            Integer freq = (Integer) bifinal.get(bigr);
                            bifinal.put(bigr,new Integer(freq.intValue()+1));
                        }
                    }
                }
            }
        }
    }
}
public void run()
{
    System.out.println("Start BI");
    countBigrams();
}

// For counting trigrams
public class Tri extends Thread
{
    public void countTrigrams()
    {
        // Variables for keeping account of bigrams, trigrams etc
        int count = 0;

        // The current line and words
        String prevW = "";
        String prevGW = "";
        String word = "";
        String newLine;

        // Tokenizer which divides lines into words
        StringTokenizer tok;

        // Open stream for reading
        BufferedReader stream;
        try
        {
            stream = new BufferedReader(
                    new FileReader(fileName));

            while ((newLine = stream.readLine())
                    != null) /* continue reading */
            {
                tok = new StringTokenizer(newLine);
                while (tok.hasMoreTokens())
                { /* tokens are coming */
                    word = tok.nextToken();
                    if (word.length() > 1 && !word.equals(prevW))
                    { /* do something */
                        if (count < 1)
                        { /* do something */
                            countBigrams(word,
                                    new Integer(1));
                        }
                        else
                        { /* do something */
                            bigramONE.put(bigr,
                                    new Integer(1));
                        }
                    }
                    prevW = word;
                }
                count++;
            }
        }
    }
}

// Increase count which holds information on which
// word is being processed
count++;

// Put current word in a prevW variable, used for next
// iteration
prevW = word;

} // End of try
stream.close();
System.out.println("Remove # <bigrams with frequency 1: "+
        bigramONE.size());
bigramONE.clear();
System.out.println("Total number of bigrams: "+ bifinal.size());
}

} // The threads run method, is run until all bigrams are counted
public void run()
{
    System.out.println("Start BI");
    countBigrams();
}

}
// Go through input file
while((newLine = stream.readLine()) != null) {
    tok = new StringTokenizer(newLine, "\n\t\f ,.?!");

    // Reset count to 0 on each iteration
    // (if I want bigrams and trigrams that
    // stretch across lines, don’t reset this counter)
    count = 0;

    // Build trigrams with the current line
    while (tok.hasMoreElements()) {

        // Current word
        word = tok.nextToken();

        // For trigrams, must have at least two previous words read
        // (which have been put in variables prevW and prev2W)
        if(count > 1) {
            // Create a bigram (vector {prev2W,prevW,word})
            Vector trigr = new Vector();
            trigr.add(prev2W);
            trigr.add(prevW);
            trigr.add(word);

            // See if the trigram already exists in trigramONE
            // yes - put it in trfinal with frequency 2.
            // Remove it from trigramONE.
            // Proceed with next trigram
            if(trigramONE.containsKey(trigr)) {
                trfinal.put(trigr,new Integer(2));
                trigramONE.remove(trigr);
            } else { // else - see if the trigram already exists in trfinal
                else {
                    // yes - increase the trigrams frequency
                    // no - put the word in trigramONE.
                    if(trfinal.containsKey(trigr)) {
                        Integer freq = (Integer) trfinal.get(trigr);
                        trfinal.put(trigr,new Integer(freq.intValue()+1));
                    } else
                        trigramONE.put(trigr,new Integer(1));
                }
            }
        }

        // If second -> N words have been processed -
        // save for trigram
        if(count > 0) {
            prev2W = prevW;
            prevW = word;
        }
        count++;
        prevW = word;
    }
}

} // End of try
stream.close();
System.out.println("Remove # trigrams with frequency 1: "+
trigramONE.size());
trigramONE.clear();
System.out.println("Total number of trigrams: "+
trifinal.size());
} catch (IOException ioe) {
System.out.println("No such file");
}

// Runs until all trigrams are counted
public void run() {
    System.out.println("Start TRI");
    countTrigrams();
}

// Class which counts a interpolation value for
// unigrams based on their counts
// Also puts the unigrams into a relation
// UNIGRAM (placed in schema wordpredict
// in the developing database)
public class Uni2 extends Thread {
    // Runs until all unigrams are counted
    public void run() {
        System.out.println("Start Uni2");
        interpUniToDatabase();
        System.out.println("End Uni2");
    }

    // Takes the hashtable unifinal and inserts it
    // into a database relation UNIGRAM, along
    // with an interpolation value which is estimated
    // based on the words frequencies and
    // the total number of words in the hashtable
    // (which can be seen as a dictionary in this
    // program)
    public void interpUniToDatabase() {
        // Increase the number of words value, since
        // so many have been discarded in the count
        // (only words with frequency 2 or higher have been counted)
        float freq;
        float interp1;
        int counter = 0;

        // The unigram
        String word = "";

        // Prepare statement used when inserting unigrams into UNIGRAM
        try (insertUniGram = con.prepareStatement("INSERT INTO ") +

UNIGRAM_TABLE + " VALUES (?, ?, ?)");
}
catch (SQLException sqle)
{
System.out.println("Error creating preparedstatement for unigram: " +
sqle.getMessage());

// Get an enumeration over the keys in the unifinal has
// The keys are to be inserted into a UNIGRAM relation:
// #COUNT | WORD | FREQ | INTERPOL
for (Enumeration e = unifinal.keys(); e.hasMoreElements(); )
{
// Increase the counter which will be added into the UNIGRAM relation
// (helps when we need to iterate over the elements in the database)
++counter;

// Get the next word from the key set
word = (String) e.nextElement();

// Get the word’s frequency
freq = Integer.parseInt(unifinal.get(word).toString());

// Calculate interpolation value (C(word)/C)
interpol = (freq/C) * INTPOL_UNI_VALUE;

// Insert the different values into the UNIGRAM relation
try
{
insertUnigram.setString(1, word);
insertUnigram.setInt(2, (int) freq);
insertUnigram.setFloat(3, interpol);
insertUnigram.executeUpdate();
}
catch (SQLException sqle)
{
System.out.println("Error inserting Unigram to database: " +
sqle.getMessage());
--counter;
}
}
}

// Class which counts a interpolation value for bigrams based on their counts
// Also puts the bigrams into a relation BIGRAM (placed in schema wordpredict
// in the developing database)
public class Bi2 extends Thread
{
public void run()
{
System.out.println("Start Bi2");
interpolBiToDatabase();
System.out.println("End Bi2");
}

// Iterates over the keys in the bifinal hash,
// performs an interpol calculation
// and inserts the results into a database BIGRAM relation
public void interpolBiToDatabase()
{
// Variables used when calculating the interpolation value
float freq;
float freqFirst;
}
float freqSecond;
float interpol;
int counter=0;

// Current bigram
String word1="";
String word2="";

// Statement which inserts bigrams into BIGRAM relation
try{
    insertBigram = con.prepareStatement("INSERT INTO " + BIGRAM_TABLE + " VALUES (?, ?, ?)");
    }
    catch(SQLException sqle)
    {
        System.out.println("Error creating preparedstatement for bigram: "+ sqle.getMessage());
    }

    // Get an enumeration over the keys in the bifinal hashtable
    // The keys are to be inserted into a BIGRAM relation:
    // WORD1 | WORD2 | FREQ | INTERPOL
    for (Enumeration e = bifinal.keys() ; e.hasMoreElements() ;)
    {
        // Increase the counter which will be added into the BIGRAM relation
        // (helps when we need to iterate over the elements in the database)
        counter++;

        // Get the bigram from the key set
        // (the bigram is stored as a vector with
        // two elements)
        Vector v = (Vector) e.nextElement();

        // Get the bigrams first and second words
        // (needed for the interpolation formula)
        word1 = (String) v.get(0);
        word2 = (String) v.get(1);

        // Get the bigram's frequency (C(word1,word2))
        freq = Integer.parseInt(bifinal.get(v).toString());

        // Get the first words frequency (C(word1))
        freqFirst = Integer.parseInt(unifinal.get(word1).toString());

        // Get the second word's frequency (C(word2)
        freqSecond = Integer.parseInt(unifinal.get(word2).toString());

        // Calculate interpolation value:
        // (C(word1,word2) / C(word1)* INTPOL_BI_VALUE) +
        // (C(word2) / C*INTPOL_UNI_VALUE)
        interpol = ((freq/freqFirst)*INTPOL_BI_VALUE) + ((freqSecond/C)*
                    INTPOL_UNI_VALUE);
    }
    catch(SQLException sqle)
    {
        System.out.println("Error inserting Bigram to database: "+ sqle.getMessage());
    }
counter--;}

}

}

}

// Class which counts a interpolation value
// for trigrams based on their counts
// Also puts the trigrams into a relation
// TRIGRAM (placed in schema wordpredict
// in the developing database)
public class Tri2 extends Thread
{
    public void run()
    {
        System.out.println("Start Tri2");
        interpTriToDatabase();
        System.out.println("End Tri2");
    }

    public void interpTriToDatabase()
    {
        // System.out.println(trifinal.toString());

        // Variables used when calculating the interpolation value
        float freqTri;
        float freqBi12;
        float freqBi23;
        float freqSecond;
        float freqThird;
        float interp1;
        int counter=0;

        // Current trigram
        String word="";
        String word2="";
        String word3="";

        try{insertTrigram = con.prepareStatement("INSERT INTO " +
            "TRIGRAM_TABLE " + " VALUES (?,?,?,?,?,?)");
        }
        catch(SQLException sqle)
        {
            System.out.println("Error creating prepared statement for trigram: " +
                sqle.getMessage());
        }

        // Get an enumeration over the keys in the trifinal hashtable
        // The keys are to be inserted into a TRIGRAM relation:
        // #COUNT | WORD1 | WORD2 | WORD3 | FREQ | INTERPOL
        for (Enumeration e = trifinal.keys() ; e.hasMoreElements() ;)
        {
            // Increase the counter which will be added into the TRIGRAM relation
            // (helps when we need to iterate over the elements in the database)
            counter++;

            // Get the trigram from the key set (a vector with three elements)
            Vector v3 = (Vector) e.nextElement();

            // Get the trigrams first, second and third words (needed for the
            // interpolation formula)
word1 = (String) v3.get(0);
word2 = (String) v3.get(1);
word3 = (String) v3.get(2);

// Get the trigrams frequency (C(word1,word2,word3))
freqTri = Integer.parseInt(trifinal.get(v3).toString());

// Get the two relevant bigram's frequencies (C(word1,word2)) and
// (C(word2,word3))
Vector v12 = new Vector();
v12.add(word1);
v12.add(word2);
freqBi12 = Integer.parseInt(bifinal.get(v12).toString());

Vector v23 = new Vector();
v23.add(word2);
v23.add(word3);
freqBi23 = Integer.parseInt(bifinal.get(v23).toString());

// Get the second word's frequency (C(word2))
freqSecond = Integer.parseInt(unifinal.get(word2).toString());

// Get the third word's frequency (C(word3))
freqThird = Integer.parseInt(unifinal.get(word3).toString());

// Calculate interpolation value:
// (C(word1,word2,word3) / C(word1,word2)) * INTPOL_TRI_VALUE +
// (C(word2,word3) / C(word2)) * INTPOL_BI_VALUE +
// (C(word3) / C) * INTPOL_UNI_VALUE
interpol = ((freqTri/freqBi12)*INTPOL_TRI_VALUE)*
((freqBi23/freqSecond)*INTPOL_BI_VALUE)+
((freqThird/C)*INTPOL_UNI_VALUE);

// Insert the different values into the TRIGRAM relation
try{
    insertTrigram.setString(1,word1);
    insertTrigram.setString(2,word2);
    insertTrigram.setString(3,word3);
    insertTrigram.setInt(4,((int)freqTri));
    insertTrigram.setFloat(5,interpol);
    insertTrigram.executeUpdate();
}
catch(SQLException sqle)
{
    System.out.println("Error inserting Trigram to database: " +
sqle.getMessage());
    counter--;
}
}
C.3 UpdatePolywithTest.java

This is the program used in the tuning of the weights process, and also to test
the predictor.

/*
 * File: updatePolywithTest.java
 * Date: 2000-10-04
 */

import java.util.*;
import java.awt.*;
import java.io.*;
import java.sql.*;
import oracle.jdbc.driver.*;

/**
 * This class updates the weights of the N-gram relations in
 * the database. It can also test the LM against a test text
 * where it calculates the average probability position in the
 * text of each word.
 */

class updatePolywithTest
{
  // Database related variables
  // Connection
  private static Connection con;

  // statements used in the program
  Statement stmt, stmt2, stmt3, stmt4;

  // Prepared statements
  PreparedStatement unigramInsert,bigramInsert,trigramInsert,
  unigramSelect,bigramSelect;

  // Callable statements used against a stored procedure in
  // the wordpredict schema
  private CallableStatement searchAll;

  // The jdbc connection
  private static String url =
    // censor

  // Variables that updates the three relations unigram,
  // bigram and trigram
  private static float INTPOL_UNI_VALUE, INTPOL_BI_VALUE,
  INTPOL_TRI_VALUE;

  // Resultsets which contain the different relations
  ResultSet unigram, bigram, trigram;

  // ResultSet for getting the size
  ResultSet size;

  // Inputfile
  private static String fileName;
// Number of words in database dictionary (unigram)
int C;
float keystrokes=0;
float keystrokesLw=0;

// Variables for the estimation process
// Suggestion list position 1-10
int one,two,three,four,five,six,seven,eight,nine,ten=0;

// Out of vocabulary, results average and number of words
// in test file
int OOV,numberOfPredictions=0;
float results=0;

/*
This program updates the three N-gram relations with three
new weights
*/
public void databaseUpdate()
{
System.out.println("*** Database connection, start updating ***");

float freq;
float interpol, freqFirst, freqSecond, freqThird, freqBi12,freqBi23;
String word1, word2, word3;

ResultSet freqF, freqS, freqT,freq12,freq23;

try {
   // Update unigrams
   unigram = stmt.executeQuery("select * from unigram3");

   // Get the number of words (multiply with 4 since
   // a lot of words are removed since we only use words
   // with frequency 2 or more)
   size = stmt4.executeQuery("select count(*) from unigram3");
   size.next();
   C = size.getInt(1) * 4;

   System.out.println(C);

   // Go through each row and update the interpolation column
   // WORD | FREQ | INTPOL
   while(unigram.next())
   {
      word1 = unigram.getString(1);
      freq = unigram.getInt(2);

      // Calculate interpolation value (C(word)/C)
      interpol = (float(freq) * INTPOL_UNI_VALUE);

      // Insert into unigram3
      // WORD1 | FREQ | POL
      unigramInsert.setString(1,word1);
      unigramInsert.setInt(2, (int)freq);
      unigramInsert.setFloat(3, interpol);
   }
}
unigramInsert.executeUpdate();
}
unigram.close();
}
catch(SQLException sqle){System.out.println("Unigram update error "+
sqle.getMessage());}

try
{
// Update bigrams
bigram = stmt.executeQuery("select * from bigram3");

// BIGRAM
// WORD1 | WORD2 | FREQ | INTPOL
while(bigram.next())
{
  // Get word1,word2 from bigram table
  word1 = bigram.getString(1);
  word2 = bigram.getString(2);

  // Get bigram frequency C(word1,word2)
  freq = bigram.getInt(3);

  // Get the first words frequency C(word1)
  unigramSelect.setString(1,word1);
  freqF = unigramSelect.executeQuery();
  freqF.next();
  freqFirst = freqF.getInt(1);

  // Get the second word's frequency C(word2)
  unigramSelect.setString(1,word2);
  freqS = unigramSelect.executeQuery();
  freqS.next();
  freqSecond = freqS.getInt(1);

  // Calculate interpolation value:
  // (C(word1,word2) / C(word1)* INTPOL_BI_VALUE) +
  // (C(word2) / C)*INTPOL_UNI_VALUE
  interpol = ((freq/freqFirst)*INTPOL_BI_VALUE) +
              ((freqSecond/freq)*INTPOL_UNI_VALUE);

  // Update row BIGRAM3
  // WORD1 | WORD2 | FREQ | POL
  bigramInsert.setString(1,word1);
  bigramInsert.setString(2,word2);
  bigramInsert.setInt(3,(int)freq);
  bigramInsert.setInt(4,(int)interpol);
  bigramInsert.executeUpdate();
}
bigram.close();
}
catch(SQLException sqle){System.out.println("Bigram update error "+
sqle.getMessage());}

// Trigrams
try
{
  trigram = stmt.executeQuery("select * from trigram3");
System.out.println(INTPOL_TRI_VALUE);
// WORD1 | WORD2 | WORD3 | FREQ | INTPOL

while(trigram.next())
{
    // Get word1,word2,word3 from trigram table
    word1 = trigram.getString(1);
    word2 = trigram.getString(2);
    word3 = trigram.getString(3);

    // Get trigram frequency C(word1,word2,word3)
    freq = trigram.getInt(4);

    // Get relevant bigram frequency (C(word1,word2))
    // and (C(word2,word3))
    bigramSelect.setString(1,word1);
    bigramSelect.setString(2,word2);
    freq12 = bigramSelect.executeQuery();
    freq12.next();
    freq12i2 = freq12.getInt(1);

    bigramSelect.setString(1,word2);
    bigramSelect.setString(2,word3);
    freq23 = bigramSelect.executeQuery();
    freq23.next();
    freq23i3 = freq23.getInt(1);

    // Get second words freq
    unigramSelect.setString(1,word2);
    freqS = unigramSelect.executeQuery();
    freqS.next();
    freqSecond = freqS.getInt(1);

    // Get the third word’s frequency (C(word3))
    unigramSelect.setString(1,word3);
    freqT = unigramSelect.executeQuery();
    freqT.next();
    freqThird = freqT.getInt(1);

    // Calculate interpolation value;
    // (C(word1,word2,word3) / C(word1,word2)) * INTPOL_TRI_VALUE +
    // (C(word2,word3) / C(word2)) * INTPOL_BI_VALUE +
    // (C(word3) / C) * INTPOL_UNI_VALUE
    interpol = ((freq/freq12i2)*INTPOL_TRI_VALUE)+
                ((freq23i3/freqSecond)*INTPOL_BI_VALUE)+
                ((freqThird/C)*INTPOL_UNI_VALUE);

    // Update row TRIGRAM3
    // WORD1 | WORD2 | WORD3 | FREQ | POL
    trigramInsert.setString(1,word1);
    trigramInsert.setString(2,word2);
    trigramInsert.setString(3,word3);
    trigramInsert.setInt(4,(int)freq);
    trigramInsert.setFloat(5,interpol);
    trigramInsert.executeUpdate();
}
}
catch(SQLException sqle){System.out.println("Trigram update error "+
    sqle.getMessage());}
APPENDIX C. PROGRAM FILES

*/
this program goes through a test text and calculates the average score the language model give each of the words in the test text
The program makes a prediction against the language model lexicon after one letter of the word to be predicted, and tries to use up to two words as the history which it bases its suggestions on.*/
public Vector estimateWeights()
{
System.out.println("*** Start testing ***");
System.out.println("----------------------------------");

// Tells if we have managed to get a suggestion or not boolean prediction;
// Put the suggestions in this vector
Vector suggestions = new Vector();
// Variables for keeping account of bigrams, trigrams etc
int count = 0;
// The current line and words
String prevW = "";
String prev2W = "";
String word = "";
String newLine;

// Tokenizer which divides lines into words
StringTokenizer tok;

// Open stream for reading
BufferedReader stream;
try
{
stream = new BufferedReader(
    new FileReader(fileName));

// Go through input file and check for each word which position // it ends up in (if any) after sent to the database LM
while((newLine = stream.readLine()) != null)
{
tok = new StringTokenizer(newLine,"\n\r\t\f ,.!?");

// Reset count to 0 on each iteration (if I want bigrams // and trigrams that stretch across lines, don't reset this // counter)
count = 0;

// Build trigrams with the current line
while (tok.hasMoreElements())
{
  // Current word
  word = tok.nextToken();

  // If we have no words in the history, make lookup // with the assistance of the first letter of the word
if(count == 0)
{
    searchAll.setString(2,"");
    searchAll.setString(3,"");
    if(word.length() > 5)
        searchAll.setString(4,word.substring(0,2).
        concat(""));
    else
        searchAll.setString(4,word.substring(0,1).
        concat(""));

    searchAll.executeQuery();

    System.out.println("Unigram LM:");
    // Send the set to a help estimation method
    estimateValues(word,searchAll);
}

    // Bigram prediction (when there is only one word in
    // the history) If we have no words in the history,
    // make lookup with the assistance of
    // the two first letters of the word
    if(count == 1)
    {
        
        searchAll.setString(2,"");
        searchAll.setString(3,prevW);
        if(word.length() > 5)
            searchAll.setString(4,word.substring(0,2).
            concat(""));
        else
            searchAll.setString(4,word.substring(0,1).
            concat(""));

        searchAll.executeQuery();

        System.out.println("Bigram LM:");
        // Send the set to a help estimation method
        estimateValues(word,searchAll);
    }

    // For trigrams, must have at least two previous words read
    // (which have been put in variables prevW and prev2W)
    // If we have no words in the history, make lookup with
    // the assistance of the two first letters of the word
    if(count > 1)
    {
        searchAll.setString(2,prev2W);
        searchAll.setString(3,prevW);
        if(word.length() > 5)
            searchAll.setString(4,word.substring(0,2).
            concat(""));
        else
            searchAll.setString(4,word.substring(0,1).
            concat(""));

        searchAll.executeQuery();

        System.out.println("Trigram LM:");
        // Send the set to a help estimation method
APPENDIX C. PROGRAM FILES

```java
estimateValues(word,searchAll);
}

// If second -> N words have been processed - save for
// trigram
if(count > 0)
{
    prevW = prevW;
}

count++;
prevW = word;

System.out.println(word + ": gives total number of keystrokes: "
+ keystrokes);
System.out.println("LM keystrokes: " + keystrokesLM);
System.out.println("-----------------------------------------------");

} // End of word while loop
} // End of text/line while loop
} // End of try
catch(SQLException sql){System.out.println("SQL! "+sql.getMessage());}
catch(IOException ioe)
{
    System.out.println("No such file");
}

// Calculate the average position of the word to be predicted
results = (float) results/numberOfPredictions;
Vector ret = new Vector();
ret.addElement(new Float(results));
ret.addElement(new Integer(OGV));
return ret;
}

/*
 * This method calculates:
 * average suggestion position
 * keystroke savings
 * how many words which are in position 1-10 in the suggestion list
 * /
 * public void estimateValues(String word, CallableStatement stat)
 {
    boolean prediction=false, prediction2 = false;

    // This is used to see if we get to a result at all
    boolean equality=false;

    int position=1;

    // The suggestions are returned here
    ResultSet res;

    // The number of keystrokes needed to type this word
    // (added one stroke for space after the word)
    keystrokes+=word.length()+1;

    try{
        res = (ResultSet) stat.getObject(1);

        // Check the words in the suggestion vector
```
// Maximum of 30 words
while (res.next() & position < 31)
{
    // We have managed to start predicting at all
    // There are in other words results to scroll
    prediction2=true;

    // See if the word in the results set is the word we are trying to predict
    if(res.getString(1).equals(word))
    {
        equality=true;
        // Calculate keystrokesLM
        // we have used two keystrokes before predicting mode
        // (plus one for predicting)
        if(word.length() > 5)
            keystrokesLM+=3;
        // we have used one keystroke before predicting mode
        // (plus one for predicting)
        else
            keystrokesLM+=2;

        // Number of words between 1-10
        if(position < 11)
        {
            // Increase the number of predictions
            numberOfPredictions++;

            // Increase the results of the suggestions
            results = position

            // Set prediction to true, we have managed to
            // find our word
            prediction = true;

            // See if we are among the first 10 suggestions
            switch(position)
            {
                // if it is the first word in the list,
                // we can simply continue typing the next
                // word (no more keystrokesLM needed)
                case 1:
                    one++;
                    break;
                // If it is the second to ten words, we can
                // chose the word
                // by pressing the correct number in the list
                // (= one more keystrokesLM)
                case 2:
                    keystrokesLM+=1;
                    two++;
                    break;
                case 3:
                    three++;
                    keystrokesLM+=1;
                    break;
                case 4:
                    four++;
                    keystrokesLM+=1;
                    break;
                case 5:
                    five++;
                    keystrokesLM+=1;
break;
case 6:
    six++;
    keystrokesLM+=1;
    break;
case 7:
    seven++;
    keystrokesLM+=1;
    break;
case 8:
    eight++;
    keystrokesLM+=1;
    break;
case 9:
    nine++;
    keystrokesLM+=1;
    break;
case 10:
    ten++;
    keystrokesLM+=1;
    break;
}

// If the length of the word is more than five characters,
// try to look at position 11-30
else if(word.length() > 5)
{
    // If we are among 11-30 add appropriate sum to
    // keystrokesLM
    if(position > 10 && position < 21)
    {
        // Increase the number of predictions
        numberOfPredictions++;

        // Increase the results of the suggestions
        results += position;

        // Set prediction to true, we have managed
        // to find our word
        prediction = true;
        keystrokesLM+=1;
    }
    else if(position > 20 && position < 31)
    {
        // Increase the number of predictions
        numberOfPredictions++;

        // Increase the results of the suggestions
        results += position;

        // Set prediction to true, we have managed
        // to find our word
        prediction = true;
        keystrokesLM+=2;
    }
    else
    {
        // We did not find the word among the first
        // 30 words, abort the prediction with some
        // loss of keystrokes...
        keystrokesLM += 3 + word.length() - 2 ;
        UV++;
    }
// Else, abort the prediction process with minimum
// "waste" of keystrokes There are not enough characters
// to save by looking at any more options
// than the first 10
else
{
    // Pressing the esc character and then typing
    // the rest of the word
    keystrokesLM += 1 + word.length() - 1;
    00V++;
}

    // Break the while statement
    break;
}

    // The current words position in the suggestion list
    position++;
}

    // If there were no alternatives returned when asking for a prediction
    // (we do not have to dispose of the popup menu with alternatives, but
    // can simply go on typing the word we wanted to be predicted)
    if(!prediction2)
{
    // Out of the vocabulary
    00V++;
    System.out.println("No alternatives returned from the LM: "+
    word + " total of: " + 00V);

    // The keystrokes used on the LM increases by 1
    // (the invocation of the predictor) +
    // word.length() + space (1)
    keystrokesLM += word.length() + 2;
}

    // If no prediction, don’t count this word in the average prediction sum
    else if(!prediction)
{
    if(!equality)
{
    // We did not find the word among the first 30 words,
    // some loss of keystrokes..
    00V++;
    keystrokesLM += 4 + word.length();
}

    System.out.println("The word: " + word +
" was not among the first thirty suggestions. "+"00V = " + 00V);
}

    // Else, we managed to find a prediction
else
{
    float average = results/numberOfPredictions;

    System.out.println("word: "+ word + ", LM position: "+ position);
    System.out.println("new average = " + average);
APPENDIX C. PROGRAM FILES

```java
} 
res.close();
}
catch(SQLException sql)
{System.out.println("Buh...");}

/*
 * Constructor
 */
public updatePolwithTest(int choice)
{
try {
//load the driver
Class.forName("oracle.jdbc.driver.OracleDriver");
con = DriverManager.getConnection("// censur);

// Statement which select full tables
stmt = con.createStatement();

// Statement which selects number of words in unigram table
stmt4 = con.createStatement();

// Statements for selects from unigram and bigram tables
unigramSelect = con.prepareStatement("select freq from unigram3 where word like ?");
bigramSelect = con.prepareStatement("select freq from bigram3 where word1 like ?
"and word2 like ?");

// Statements for inserting into unigram3, bigram3 and trigram3
unigramInsert = con.prepareStatement("INSERT INTO UNIGRAM VALUES (?,?,?)");
bigramInsert = con.prepareStatement("INSERT INTO BIGRAM VALUES (?,?,?)");
trigramInsert = con.prepareStatement("INSERT INTO TRIGRAM VALUES (?,?,?,?)");

// Statement which calls stored function that returns a resultset
// consisting of all word prediction alternatives for a given word history
// and two letters of the prefix of the word to be predicted
searchAll = con.prepareCall("@ = call wordpredict(??,?,?)");
searchAll.registerOutParameter(1, OracleTypes.CURSOR);
if(choice == 4)
{
// update database with new weights
databaseUpdate();
}

// Check the results with the new weights
Vector results = estimateWeights();
System.out.println("New overall average: " + results.get(0) 
+ ": 00V: " + results.get(1));
int overall = one + two + three + four + five + six + seven +
eight + nine + ten;
System.out.println("position 1-10 = : " + overall);
System.out.println("position 1 = " + one);
```
System.out.println("position 2 = " + two);
System.out.println("position 3 = " + three);
System.out.println("position 4 = " + four);
System.out.println("position 5 = " + five);
System.out.println("position 6 = " + six);
System.out.println("position 7 = " + seven);
System.out.println("position 8 = " + eight);
System.out.println("position 9 = " + nine);
System.out.println("position 10 = " + ten);

System.out.println("keystrokes = " + keystrokes + " LNM keystrokes = " + keystrokesLM);

float saved = keystrokesLM/keystrokes;
System.out.println("keystrokes saved = " + saved);
}
catch(SQLException sql){System.out.println("SQL Exception: " + sql.getMessage());}
catch(ClassNotFoundException cl){System.out.println("CLASS " + cl.getName());}
/*. The program can take 4 different parameters when started */

public static void main(String args[]) {
    // We have given weights and a file to be tested
    if (args.length == 4) {
        try {
            INTPOL_UNI_VALUE = Float.parseFloat(args[0]);
            INTPOL_BI_VALUE = Float.parseFloat(args[1]);
            INTPOL_TRI_VALUE = Float.parseFloat(args[2]);
            fileName = args[3];
            new updatePolWithTest(4);
        } catch(NumberFormatException e) {
            System.out.println("Error" + e.getMessage());
        }
    // We have only given a file to be tested
    else if(args.length == 1) {
        fileName = args[0];
        new updatePolWithTest(1);
    }
    else
        System.out.println("Give at least one file name as argument. If you also want to " + 
            "update the database weights, first give the three weights as " + 
            "arguments, then the file to be tested");
}
C.4 ChatCache.java

This is the cache which stores bigrams during one chat session. It is a client side program.

```java
/*
 * File: ChatCache.java
 * Date: 2000-06-07
 * (c) 2001, DoBeDo AB. All Rights Reserved
 */
package com.dobedo.client.world;

import com.dobedo.client.*;
import com.dobedo.client.gui.*;
import com.dobedo.misc.*;
import java.util.*;

/*
 * A vector class which contains bigrams collected from
 * the Avatar with assistance of the Text class. This is
 * the cache which a chat session for a user is stored in
 */
public class ChatCache extends Vector
{
    // frequencies in the chat cache
    private int freq;

    // creates a ChatCache with the Bigrams in the Bigram-vector.
    public ChatCache()
    {
        freq = 0;
    }

    // A method that builds a vector consisting of bigrams of the text
    // and a count of the bigrams. The counts are not used as of now.
    // in the predictor
    public void buildChatCache(String text)
    {
        text = text.trim();

        // This tokenizer need to divide on exactly the same things as the
        // tokenizer which builds the database lexicon (ChatStats.java)
        StringTokenizer st = new StringTokenizer(text,\"\r\n\t\",\"\",\"\?");
        int count = 0;
        String prevW = new String();
        while (st.hasMoreElements())
        {
            String word = st.nextToken();
            if(count > 0)
            {
                // Add a bigram to the chat buffer
                addBigramChatCache(prevW,word);
            }
            count++;
            prevW = word;
        }
    }
```
APPENDIX C. PROGRAM FILES

} // Add a bigram to the chat buffer if it does not exist, otherwise
// increase the frequency of the already existing bigram. As of now
// the frequencies of bigrams are not used, but they may be in the future.
public void addBigramChatCache(String one, String two)
{
    boolean exist = false;
    // Go through the chat buffer to see whether the bigram has been
    // encountered before
    for(Enumeration e = this.elements(); e.hasMoreElements();)
    {
        Bigram b = (Bigram) e.nextElement();
        // If the bigram matches one in the chatbuffer, increase its frequency.
        // and break the loop
        if(b.existBigram(one, two))
        {
            exist = true;
            b.increaseBigram();
            break;
        }
    }
    // Add a bigram last to the chat buffer if the Avatar has not written
    // this bigram before
    // (it is not contained in the chat buffer before now)
    if(!exist)
    {
        // System.out.println("Add bigram to chatbuffer: "+one+" "+two);
        Bigram b = new Bigram(one, two);
        this.addElement(b);
    }
}

// Searches all of the bigrams in the chatbuffer for the specified prefix
// and previous word
public Vector predictChatCache(String prefix, String prevW)
{
    // Add all the matching bigrams second words in this vector
    Vector results = new Vector();
    // Pick out all the bigrams with a correct first word and matching
    // prefix on the second word
    for(Enumeration e = this.elements(); e.hasMoreElements();)
    {
        Bigram b = (Bigram) e.nextElement();
        // If there exists a bigram with prevW and the second word beginning with
        // prefix, add the word that begins with the prefix
        if(b.existBigram(prevW, prefix))
        {
            // System.out.println("Searching ChatBuffer and finding/adding bigram");
            results.addElement(b.getSecondWord());
        }
    }
    return results;
}
C.5 Bigram.java

/*
 * File: Bigram.java
 * Date: 2000-06-07
 * 
 * (c) 2001, DoBeDo AB. All Rights Reserved
 */

package com.dobedo.client.world;

import com.dobedo.client.*;
import com.dobedo.client.gui.*;
import com.dobedo.misc.*;
import java.util.*;
import java.awt.*;

/**
 * A class which stores bigrams and their frequencies
 *
 */

class Bigram
{
    // The frequency of the bigram in the ChatBuffer
    private int freq;

    // The two Strings making up the bigram
    private String word1, word2;

    // A class to create a new bigram
    public Bigram(String one, String two)
    {
        word1 = one;
        word2 = two;
        freq = 1;
    }

    // Increases a bigrams freq (called if the bigram
    // already exists in the ChatBuffer)
    public void increaseBigram()
    {
        freq += 1;
    }

    // Checks whether the specified prefix occurs in a
    // bigram with a previous word
    // (e.g. "are y" should return true if the bigram is "are you")
    public boolean existBigram(String prevW, String prefix)
    {
        if(word1.equals(prevW) && word2.startsWith(prefix))
            return true;
        else
            return false;
    }

    // Fetches the bigram in a vector {word1,word2}
    public Vector getBigram()
    {
        Vector v = new Vector();
        return v;
    }
}
v.add(word1);  
v.add(word2);  
return v;  
}  
// Get second word  
public String getSecondWord()  
{  
    return word2;  
}  
}  

C.6 Avatar.java

This class is several thousand lines of code long, and therefore the only thing shown here are the lines of code of interest for the word prediction program.

/**
 * The Avatar class represents an avatar object. It defines
 * how an avatar will look in the GUI and communicate with
 * the underlying object model.
 * *
 * @author Andreas Arrgård
 */

public class Avatar extends ChatObject implements ImageObserver
{
    /**
     * ADDON by Patrik Sjöberg
     */

    // ADDON by Patrik Sjöberg  
    private ChatCache chatcache_ = new ChatCache();  

    // ADDON by Patrik Sjöberg - the current word to be guessed  
    private String guessWord_;  

    // ADDON by Patrik Sjöberg - history vector containing the previous two  
    // words and the prefix of the word to be guessed when predicting  
    // {prefix_prev4,prev2W}  
    private Vector history_;  

    // ADDON by Patrik Sjöberg - the prefix of the word to be guessed  
    // when predicting  
    private String prefix_;  
    private String prevW_;  
    private String prev2W_;  

    // ADDON by Patrik Sjöberg - the text before inserting a prediction  
    // suggestion (inserted back if the user regrets word prediction and  
    // presses <esc> or <left arrow>)  
    private String oldText_;  

    // ADDON by Patrik Sjöberg - Vector which holds words that are  
    // returned from the word prediction class after prediction  
    Vector dSuggs_;  
    Vector cacheSuggs_;
Vector suggs;

// ADDON by Patrik Sjöberg - the number current words shown in the
// vector of suggestions (0-9,10-19,20-29...). Words are shown in a popup menu.
private int suggPos_ = 10;

// ADDON by Patrik Sjöberg - a variable which keeps track of whether
// the Avatar is in prediction mode or not (if the user has pressed
// <tab> and not turned prediction off with <esc> or left <arrow>
// the Avatar is in prediction mode)
private boolean predicting_ = false;

// A popup menu for showing word prediction suggestions
PopupMenu popup_ = null;

// ADDON by Patrik - states what kind of listener the rows of the
// popup_ should listen to
public static String ACTION_PREDICT = "PREDICT";

//-- End of addons of variables by Patrik --/

/**
 * This method handles the local notifications sent to this
 * object. Depending on what is sent, different actions will be
 * taken.
 */
public boolean handleNotification( Object notification ) {
    // if the event is an instance of an Event then
    // this means that this client is bound to this avatar
    // and we can scan for keyboard input.

    if ( notification instanceof Event ) {
        // cast
        Event e = ( Event ) notification;

        // trace
        Application.getInstance().logTrace( "Avatar received: " + e );

        // is this a keyboard input
        if ( e.id == Event.KEY_ACTION || e.id == Event.KEY_PRESS ) {
            // trace
            Application.getInstance().logTrace( "Avatar key modifiers: " +
                    // e.modifiers + " key:" + e.key + " time:" + e.when );

            // ADDON by Patrik
            // If we are in predicting mode and press a key (not a
            // number between 0-9 or esc) and want to continue writing
            // after a suggestion has been given, remove the popup with
            // suggestions and set predicting to false
            if ( predicting_ && e.key != 27 && e.key != 1002 && e.key != 1003 &&
                 ( e.key > 47 && e.key < 58 ) ) {
                popup_.unregister();
                predicting_ = false;
            }
        }
    }
}
editBubble_.repaint();
}

if(e.key == 10)
{
   // ADDON by Patrik
   // If the avatar is in predicting_ mode, remove the
   // popup_menu with suggestions and continue with
   // displaying the talk bubble instead of edit bubble
   if(predicting_)
   {
      popup_.unregister();
   }
   // End of Addon

   // enter was pressed - does it exist a speach bubble?
   // get the text in the bubble...
   // if the edit bubble is null -
   // check the talk bubble for text

   String text = null;
   int textType = -1;
   if( editBubble_ != null )
   {
      // get the text and type...
      text = editBubble_.getText().trim();
      textType = editBubble_.getType();

      // remove the bubble
      removeEditBubble();
   }
   else if( talkBubble_ != null &&
            talkBubble_.bgColor == editColor_ )
   {
      // get the text and type...
      text = talkBubble_.getText().trim();
      textType = talkBubble_.getType();

      // remove the bubble
      say( null );
   }

   // does the avatar say anything
   if( text != null && text.length() > 0 )
   {
      // Shall the text be sent to the other clients
      // or shall it be interpreted as a command?
      if( text.startsWith("=") )
      {
         // trace
         Application.getInstance().
         logTrace("Trying to handle the command "+
         text + ", ");

         // Send this command to the parser...
         parseEvent( text.substring(i, text.length()) );
      }
      else
      {
         // get the bound object and talk target
   }
ObjectRep boundOR = ObjectSpace.getInstance().getObject(oId);
OID talkTarget = (OID) boundOR.getProperty("talktarget").value;

// get the type of the bubble
short commArg = CMS_Message.COMMUNICATE_ARG_SAY;
if( talkTarget != null &&
talkTarget.equals( boundOR.getProperty("owner").value ) == false )
  commArg = CMS_Message.COMMUNICATE_ARG_WHISPER;
else
  {
    if(textType == Text.TYPE_SHOUT_BUBBLE)
      commArg = CMS_Message.COMMUNICATE.Arg_SHOUT;
    else if(textType == Text.TYPE_THINK_BUBBLE)
      commArg = CMS_Message.COMMUNICATE.Arg_THINK;
  }

// create a new bubble...
createTalkBubble
  (text, keyboardType, transientColor_, true);

// send this message to all the other clients
boundOR.sendMessage(null, text, commArg, talkTarget);

Application.getInstance().
logTrace("The message " +
text + "' was sent to the server.");

// add the text to the last said sentences
addToLast SaidSentence( text );

// ADDON by Patrik Sjöberg -
// adds every bigram {word1, word2} in the bubble
// to the chat cache
// (i.e. "i like you"
// adds {i, like} and {like, you})
chatcache_.buildChatCache(text);
// End of Addon

}
}
else if( e.key == 27 )
{
  // the escape key was pressed

  // ADDON by Patrik Sjöberg
  // Now the edit and talkbubbles are only removed
  // if the avatar is not in prediction
  // mode since the prediction popup menu
  // is otherwise the only thing to be removed
  if(!predicting_)
    {
      // remove the bubbles
      removeEditBubble();
      say( null );
    }

  // remove the drop menu if one exists.
if (registeredToPaintServer_ != null)
registeredToPaintServer_.getPopupMenu().unregister();

// ADDON by Patrik Sjöberg
// If the Avatar is in predicting_mode, remove
// the suggestion that has been filled out in the
// edit bubble by inserting the old text, and set the boolean
// predicting_ to false
if (predicting_)
{
    predicting_ = false;
    editBubble_.setText(oldText_);
    editBubble_.cursorPos = oldText_.length();
    editBubble_.repaint();
    popup_.unregister();
}

// ----------- ADDON by Patrik Sjöberg --------------------------
else if (e.key == (int) '\t')
{
    // tab was pressed - predict what word will come next in
    // the edit bubble

    String text = null;
    int textType = -1;
    if (editBubble_ != null)
    {
        // get the text and type...
        text = editBubble_.getText();

        // Get the text before prediction.
        // In case the user doesn't want
        // any of the suggestions given by
        // the word prediction, this text will be
        // inserted back into the editbubble
        oldText_ = new String(text);
    }

    // does the avatar say anything
    if (text != null && text.length() > 0)
    {
        // set flag for in predict mode...
        predicting_ = true;

        // Reset the suggestion pointer to 10 indicating
        // that we are among the
        // first 10 suggestions.
        suggPos_ = 10;

        // Get the history {prefix_, prevW, prev2W}:
        // where prefix_ is the part of the word to be predicted
        // that has been written so far. If the cursor has a space
        // before it, the prefix_ is empty. prevW and prev2W are the
        // two words before the prefix_. These can also be empty if
        // there is not sufficient history in the edit bubble
        history_ = editBubble_.getHistory();

        // Remove any previous popup
        if (popup_ != null)
            popup_.unregister();

        // Get the words in the history
prefix_ = (String) history_.get(0).toString();
prevW_ = (String) history_.get(1).toString();
prev2W_ = (String) history_.get(2).toString();

// If the user has not pressed enough letters before starting the
// guessing, show bubble stating that more letters need to be typed
if (!(prefix_.length() != 0 || prev2W_.compareTo("") != 0))
{
    // Send message to the client with information that he need
    // to press more letters
    noPredictions(1);
}
else
{
    // Search the chatcache for bigrams matching the history and
    // the current prefix - and add them to the suggestions vector
    cacheSuggs_ = chatcache_.predictChatCache(prefix_, prevW_);

    // Create a synchronized object which the database predict function will
    // wait for a maximum of 5 seconds (otherwise the prediction takes
    // too long)
    MessageSynch messageSynch = new MessageSynch();

    // Call the predict function with the history and suggs from cache
    // with the assistance of a CMC_Message which is sent to the server
    CMC_Message predictMessage = new CMC_Message();

    // Add data to the CMC_Message (including the history vector)
    predictMessage.data = history_.
    predictMessage.action = CMC_Message.GET_COMPLETION;
    predictMessage.target=
    new UID(ClientSystem.getInstance().
        getParameter("wordpredict_target" ));
    predictMessage.argument = 0;
    predictMessage.source = ClientSystem.getInstance().getBoundUID();

    // Send the prediction message to the server
    ClientSystem.getMessageHandler().
        sendMessage(messageSynch, predictMessage);

    // Wait three seconds for the prediction
    try{messageSynch.waitForResponse(30000);
    }
    catch(InterruptedException ie) {
        // What to do?
        System.out.println("Interrupted exception");
    };

    // If the response from the word predict server is not null, check
    // further for the alternatives it has given
    if(messageSynch.response != null) {
        // Put the prediction data in a vector
        CMC_Message cmc_getCompletions = messageSynch.response;
        dbSuggs_ = (Vector) cmc_getCompletions.data;

        // Combine the two word prediction results
        suggs_ = completeLPrediction(cacheSuggs_, dbSuggs_);

        // If there are any results - display the first word suggestion in the
        // edit bubble and also show a popup_ with the first 10 suggestions
if(suggs_.size()>0)
{
    // Get the first suggestion to be put in the editBubble
    // i.e. if the user has written "i like y" and the program
    // has found "you" to be the most likely suggestion, the
    // editbubble should only be filled with "ou" since the prefix_
    // is already there.

    // Take out the prefix_ of the word to be predicted
    prefix_ = (String) history_.get(0).toString();

    // Take out the word to be inserted (the part gotten from
    // the WordPredict lookup)
    guessWord_ = (String) suggs_.get(0);

    // Put the two parts together (chop off from guessWord_ the part
    // of the prefix_ already in the editbubble). Add a space to the
    // end of the suggestion so that the user can continue writing after
    // a word has been inserted
    guessWord_=guessWord_.concat(" ").substring(prefix_.length());

    // Put the word in the bubble along with the text existing before,
    // and set the cursor
    editBubble_.setText(text + guessWord_);
    editBubble_.cursorPos = editBubble_.length();

    // show the popup_ with suggestions if the avatar is registered in
    // the GUI
    if(registeredToPaintServer_ != null)
    {
        // Maybe this method should be extracted from this
        // class - put in WordPredict?
        // the first 0 - ? number of suggestions

        // If there are 10 or more suggestions, display 10 first
        if(suggs_.size() > 9)
            showWordPredictions(0,10);
        // Else, show the fewer than 10 words
        else
            showWordPredictions(1,suggs_.size());
    }
}
else
{
    // No predictions available, tell the user
    noPredictions(2);
}

// The message times out
}
else
{
    noPredictions(3);
    predicting_ = false;
}

}
editBubble_.repaint();

}
APPENDIX C. PROGRAM FILES

// ADDON by Patrik Sjöberg
// If the Avatar is in predicting mode AND
// the user presses a number between zero and 9, he wants that word in
// the suggestion list
else if(predicting_ && (e.key > 47 && e.key < 58))
{
// Only pick out the right suggestion if the key is a valid one
//(e.g. if user presses 8 but there are only seven suggestions
// in the vector, nothing should be picked out)
if((e.key==47)&&(suggPos_ -= 10) <= suggs_.size())
{

// If it is one of the first 10 (0 through 9) words
// that is chosen, display that word in the editBubble
if(suggPos_ == 10)
{
// Get the correct suggestion word
prefix_ = (String) history_.get(0).toString();

// Get the correct number in the suggestion vector, the
// integers 0-9 are between 49 and upwards in the ascii
guessWord_ = (String) suggs_.get(e.key-48);
guessWord_ = guessWord_.concat(" ").substring(prefix_.length());

// Put the word in the bubble, and move the cursor
editBubble_.setText(oldText_ + guessWord_);
editBubble_.cursorPos = editBubble_.length();

// Remove suggestion popup_ and repaint avatar and set
// predicting_ to false
popup_.unregister();
predicting_ = false;
editBubble_.repaint();
}
// If the user has chosen to proceed further down the list
// (suggPos_ increases by ten for every new down/page down press)
else if(suggPos_ > 10)
{
// Get the correct suggestion word
prefix_ = (String) history_.get(0).toString();

// Get the correct number in the suggestion vector, the
// integers are between 49 and upwards in the ascii. Now
// however, we need the Nth word first, since we have
// moved down at least 10 words in the suggestion list by
// pressing the down key
guessWord_ = (String) suggs_.get(e.key+suggPos_ - 10-48);
guessWord_ = guessWord_.concat(" ").substring(prefix_.length());

// Put the word in the bubble, and move the cursor
editBubble_.setText(oldText_ + guessWord_);
editBubble_.cursorPos = editBubble_.length();

// Remove suggestion popupkey == Event.LEFT &&
// predicting_ and repaint avatar and set
// predicting_ to false
popup_.unregister();
predicting_ = false;
editBubble_.repaint();
}
}
}

// ADDON by Patrik
// If the user presses page up or page down
else if(e.key == 1002 || e.key == 1003) &
& predicting_
{
// Not pleased with the first suggestion - so remove it from the editbubble
// The most likely alternative is put in the bubble when the Avatar
// has requested for a word prediction (pressed <tab>).
editBubble_.setText(oldText_);
editBubble_.repaint();

// Move 10 suggestions up in the popup meny (ten preceding alts.)
// 1002 = page up
if(e.key == 1002)
{
// If the suggPos_ is larger than 10, make it 10 smaller
// and show the ten preceding alternatives
if(suggPos_ > 10)
{
// decrease the suggPos_ number by 10
suggPos_ = suggPos_-10;

// For new label
int suggten = suggPos_-10;
showWordpredictions(suggten,10);
}

// Move the suggestion popup meny down 10 positions
else if(e.key == 1003)
{
// If there are 10 more suggestions in the suggs_ vector, show these
if(suggs_.size() - (suggPos_+10) >= 0)
{
// Increase suggPos_ by 10
suggPos_ = suggPos_+10;

// New label
int suggten = suggPos_-10;
showWordpredictions(suggten,10);
}
// If there are less than 10 suggestions left in the list
else
{
// here we are at the last less than 10 words in the list
int remaining = suggs_.size()-suggPos_;
//System.out.println("Last remaining "+remaining);
if(remaining > 0)
{
suggPos_ = suggPos_+10;

// New label
int suggten = suggPos_-10;
showWordpredictions(suggten,remaining);
}
}
else if(e.key == Event.LEFT &
& predicting_)
{
// remove the suggested word
editBubble_.setText(oldText_);
editBubble_.setCursorPos=oldText_.length();
editBubble_.repaint();
popup_.unregister();
predicting_ = false;
}
// ---- End of Addon of word prediction actions when pressing <tab> --------
// Part of the "old" avatar class
else if( (e.key = Event.UP || e.key = Event.DOWN) &&
lastSaidSentences_.size() > 0 &&
editBubble_ == null)
{
if(e.key = Event.UP && lastSaidSentence_ > 0)
lastSaidSentence_ --;
else if(e.key = Event.DOWN &&
(lastSaidSentences_.size() - 1) > lastSaidSentence_)
lastSaidSentence_ ++;
// the up or down arrow was pressed
createTalkBubble(String) lastSaidSentences_.elementAt(lastSaidSentence_),
Text.TYPE_SAY_BUBBLE, editColor_, false );
}
else
{
// System.out.println("Key pressed:" + e.key);
// check if we have a last said sentence active
if( talkBubble_ != null &&
talkBubble_.bgColor == editColor_ )
{
// get the text of the talk bubble and create and edit bubble
createEditBubble( talkBubble_.getText(), Text.TYPE_SAY_BUBBLE );
say( null );
}
else if(editBubble_ == null)
createEditBubble( "", Text.TYPE_SAY_BUBBLE );
editBubble_.handleEvent(e);
}
return true;
}
return false;
}
else if(notification instanceof Notification)
{
Notification not = (Notification) notification;
// get the representation of this object.
ObjectRep thisUR = ObjectSpace.getInstance().getObject(old_);
// trace
//Application.getInstance().logTrace
//("The avatar "+id_" received the notification "+not.action+"" );
// add the offer capabilities
if(ACTION_BODY_LOADED.equals(not.action))
{
bodyImage_ = ClientSystem.getImageHandler().getImage( (String) not.argument );
}
APPENDIX C. PROGRAM FILES

//System.out.println( "BodyImage is "+bodyImage_.
//getWidth(null)+","+bodyImage_.getHeight(null)+"."");
repaint();
return true;
}
if(ACTION_FACE_LOADED.equals(not.action))
{
    faceImage_ = ClientSystem.getImageHandler().getImage((String) not.argument);
    repaint();
    return true;
}
else if(ACTION_OFFER_GIVE.equals(not.action) &&
    not.argument instanceof Vector)
{
    // cast
    Vector v = (Vector) not.argument;

    // get the object that is to be offered
    OID offeredOID = (OID) v.elementAt(0);

    // get the object that is to be offered the offer
    OID receivingOID = (OID) v.elementAt(1);

    // create the offer text
    String text = Language.printf(Language.TEXT_AVATAR_SPEECH_OFFER_THING,
        ObjectSpace.getInstance().getObject(offeredOID).getName());

    // Create the message that shall be offered
    offer(text, Avatar.GIVE_OBJECT, offeredOID, receivingOID);
    return true;
}
// ADDON by Patrik Sjöberg
// If the user selects an item in the popup_ with the mouse,
// display that item in the text bubble
else if("PREDICT".equals(not.action) && predicting_)
{
    Integer indexInt = (Integer) not.argument;
    int index = indexInt.intValue();
    // If it is one of the first 0 through 9 words (guided by up/down)
    // that is chosen, display that word in the editBubble
    if(suggestPos_ == 10)
    {
        // Get the correct suggestion word
        prefix_ = (String) history_.get(0).toString();
        // Get the correct number in the suggestion vector, the integers are between
        // 49 and upwards in the ascii
        guessWord_ = (String) suggest_.get(index);
        guessWord_ = guessWord_.concat(" ").substring(prefix_.length());

        // Put the word in the bubble, and move the cursor
        editBubble_.setText(oldText_ + guessWord_);
        editBubble_.cursorPos_ = editBubble_.length();

        // Remove suggestion popup_ and repaint avatar and set predicting_ to false
        popup_.unregister();
predicting_ = false;
editBubble_.repaint();
}
// If the user has chosen to proceed further down the list
else if(suggPos_ > 10)
{
// Get the correct suggestion word
prefix_ = (String) history_ .get(0) .toString();
// Get the correct number in the suggestion vector, the integers are between
// 49 and upwards in the ascii. Now however, we need the Nth word first, since
// we have moved down at least 10 words in the suggestion list
// by pressing the down key
System.out.println("Index: "+index);
guessWord_ = (String) suggs_ .get(index+suggPos_-10);
guessWord_= guessWord_ .concat(" ").substring(prefix_ .length());

// Put the word in the bubble, and move the cursor
editBubble_ .setText(oldText_ + guessWord_);
editBubble_ .cursorPos_ = editBubble_ .length();

// Remove suggestion popup_ and repaint avatar and set predicting_ to false
popup_ .unregister();
predicting_ = false;
editBubble_ .repaint();
}

return true;
}
// ------ End of Addon with mouse choice of prediction ------
.
.
.

// ------ ADDONS by Patrik Sjöberg OF EXTRA METHODS ----------
APPENDIX C. PROGRAM FILES

// Else, show the (less than 10) number of items that are in the suggs_vector
else
{
    popup_.setLabel(suggs_.size() + " words" + " of " + suggs_.size());
    addSuggItems(this, suggPos_, numToShow);
}

// move the popup_ to the correct position
NewRectangle nR = (NewRectangle) bounds_;
popup_.move( nR.x + nR.width, nR.y + nR.height / 2 + 50 );

// set the layer
popup_.setLayer( 10010 );
popup_.register(registeredToPaintServer_);
}

/* This method puts the suggestions from the cache vector in front of the suggestions from the db vector and returns the new complete suggestion vector */
public Vector completeLMPrediction(Vector bv, Vector dbv) {
    String tmp;
    for (Enumeration e = dbv.elements() ; e.hasMoreElements() ; ) {
        // If the element in the dbv (database vector) is not a member of the bv
        // (cache vector), insert it into the bv
        tmp = (String) e.nextElement();
        if (!bv.contains(tmp))
            bv.add(tmp);
    }
    return bv;
}

/* Adds the specified elements to the item vector for the popup_ menu giving word prediction suggestions */
public void addSuggItems(INotificationListener listener, int suggPos_, int fillOut) {
    // Remove all words from the predict popup_
    popup_.removeAll();

    // A vector that will contain the strings that are to be displayed in the popup Vector shorter;

    // If it is not the last, less than 10 words to be filled in the bubble
    // Fill the PopupMenu with the correct 10 words from the suggs_vector
    if(fillOut==10)
        {
            // If it is the first ten words of the suggs_vector that are to be filled into the popup menu
            if(suggPos_==10)
                { shorter = subVector(suggs_,0,10);
            }
            // else, pick out the correct ten words of the suggs_vector
            else
                shorter = subVector(suggs_,suggPos_-10,suggPos_);
        }
    // If it is less than 10 words left of the suggestion vector,
// pick out these last words
else {
    shorter = subVector(suggs_.suggs_.size()-fillOut,suggs_.size());
}
Integer i= new Integer(0);
int counter = 0;

// Iterate through and add ten, or fewer words if there are
// not ten words left to display of the suggs_vector, in the items vector
//while( iterate.hasNext() & & counter < fillOut)
Enumeration e = shorter.elements();
while( e.hasMoreElements() & & counter < fillOut)
{
    // Add "index" to the items in the popup menu
    i = new Integer(counter);
    String number = i.toString();
    number = number.concat(".");
    String st = new String(e.nextElement().toString());
    String numberPlusSugg=number.concat(st);

    // Create notification for this particular index in the popup
    Notification notification = new Notification(ACTION_PREDICT, this, this, i );

    popup_.addItem(notification,listener,numberPlusSugg);
    counter++;
}
popup_.setIsReadyToPaint(false);
}

// A helper method that returns a subvector with start and end index.
// Returns false if any of
public Vector subVector(Vector v, int indexStart, int indexEnd)
{
    Vector shorter = new Vector();
    int i;

    for(i=indexStart ; i < indexEnd ; i++)
    {
        shorter.addElement(v.elementAt(i));
    } return shorter;
}

// If we dont get any suggestions when trying to predict
public void noPredictions(int i)
{
    // Set the prediction state to false
    predicting_ = false;

    Text guess;

    // register a Text bubble with a message saying you need to
    // press more letters
    NewRectangle boundingBox = bounds_.getNewBoundingBox();
    if(i==1)
    {
        guess = new Text("Need more letters to guess your word",new Point (
C.7 Text.java

This class is very large, and the only method shown here is the method of particular interest for the word predictor.
// --------- ADDONS by Patrik SJöberg ----------------------
// Gets one or two or zero words backwards from where you are
// with the cursor right now from the Text depending on how many
// words can be found backwards
public Vector getHistory()
{
    // Vector to add the history to
    Vector history = new Vector(3);

    // Three stringbuffers which shall contain prefix of the word to
    // be predicted, the previous word and the word before that.
    String prefix = "";
    String prevW = "";
    String prev2W = "";

    // Get the text in the current Text bubble
    String s = new String(getText());

    // A check if the prefix is empty
    boolean prefEmpty = false;

    // Tokenize the text
    StringTokenizer histTok = new StringTokenizer(s, "\n\r\t\f ,!?");

    // If the string ends with a space, the prefix consists of the
    // empty string ""
    if (s.endsWith(" "))
    {
        prefEmpty = true;
        prefix = "";
    }

    // Two "memory strings"
    String oldW = "";
    String old2W = "";
    // Iterate through the string, find the last and next to last words
    while (histTok.hasMoreElements())
    {
        String w = histTok.nextToken();

        // If the string has no more elements, check the last three tokens
        if (!histTok.hasMoreElements())
        {
            // If the prefix is empty, the last two words are the previous and
            // second previous words
            if (prefEmpty)
            {
                prevW = w;
                prev2W = oldW;
            }
            else
            {
                // else, the prefix is the last word, and the two tokens before that are
                // the previous two words
                old2W = oldW;
                oldW = w;
            }
        }
        else
        {
            // else, the prefix is the last word, and the two tokens before that are
            // the previous two words
            old2W = oldW;
            oldW = w;
        }
    }
public static void notifyGetCompletion(CommandData cdata) {
}
}

class WordPredict {
    public static void notifyGetCompletion(CommandData cdata) {
        try {
            // Since this operation does not affect the chat world at
            // all it's not necessary to have the semaphore.
            ChatWorldSemaphore.getInstance().release();

            // Get the vector with prediction information
            Vector vecArgs = (Vector)cdata.data;
        }
    }
}

C.8 WordPredict.java

This is a server side program which will call for DbWordPredictHandler which
is the program that handles the actual lookup.

/*
 * File: WordPredict.java
 * Date: 2001-01-03
 * (c) 2001, DoBeDo AB. All Rights Reserved
 */
package com.dobeo.chat;
import com.dobeo.chat.world.ObjEntity;
import com.dobeo.misc.CMC_Message;
import com.dobeo.misc2.ResourceException;
import com.dobeo.misc2.SM_Message;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Vector;
import java.sql.*;
import java.util.*;
import oracle.jdbc.driver.*;

/**
 * A helper class that handles all the WordPredict services.
 */
public class WordPredict {
    public static void notifyGetCompletion(CommandData cdata) {
        try {
            // Since this operation does not affect the chat world at
            // all it's not necessary to have the semaphore.
            ChatWorldSemaphore.getInstance().release();

            // Get the vector with prediction information
            Vector vecArgs = (Vector)cdata.data;
        }
    }
}
C.9 DbWordPredictHandler.java

This program sets up the connection against the database and handles the lookups.

/*

APPENDIX C. PROGRAM FILES

// Strings in the history of a Text bubble
// (since the language model is a trigram model, a maximum of two word +
// the prefix of the word to be predicted)
String prefix, prevW, prev2w;

// Get the words in the history
prefix = (String) vecArgs.get(0).toString();
prevW = (String) vecArgs.get(1).toString();
prev2W = (String) vecArgs.get(2).toString();

// Add % to the prefix
prefix = prefix.concat("%");

// Make the predictions (which end up in vecComp)
DbWordPredictHandler handler = DbWordPredictHandlerPool.getInstance().
getDbWordPredictHandler();
Vector vecComp = handler.getCompletions(prev2W, prevW, prefix);
handler.free();

// Create the message to send back to the client
SM_Message sm = new SM_Message();
CMC_Message cmc = new CMC_Message();
cmc.source = ChatSettings.getInstance().getWordPredictUid();
cmc.target = null;
cmc.data = vecComp;
cmc.msgId = cdata.msgId;
sm.sessionId = cdata.session;
sm.sender = null;
sm.destination = new Vector();
sm.destination.add(cdata.source.getUid());
try {
    sm.serverId = InetAddress.getLocalHost();
} catch (UnknownHostException e) {
    sm.serverId = null;
}
sm.mc_Message = cmc;
cmc.action = sm.action = SM_Message.UPDATE_COMPLETION;

// Send the message back to the client
ObjEntity.sendMessage(sm);
}
catch (ResourceException e) {
    // this means we're not able to get a db handler
}
finally {
    // better grab the semaphore before we exit
    ChatWorldSemaphore.getInstance().grab();
}
}
APPENDIX C. PROGRAM FILES

* File: DbWordPredictHandler.java
* Date: 2001-01-04
* (c) 2001, DoBeDo AB. All Rights Reserved
*
package com.dobedo.chat;

import com.dobedo.misc2.Application;
import com.dobedo.misc2.Resource;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Vector;
import oracle.jdbc.driver.OracleTypes;

/* This class handles the actual word predictions by using
 a callable statement which calls a stored oracle function */

public class DbWordPredictHandler extends Resource
{
    // The connection and statement used in the lookup against the database
    private Connection conn_ = null;

    /* Constructor
    Creates the connection to the database */
    public DbWordPredictHandler()
    {
        String strDriver = Application.getInstance().
            getPropertyString("chat.wordpredict.driver");
        String strUrl = Application.getInstance().
            getPropertyString("chat.wordpredict.url");
        String strUser = Application.getInstance().
            getPropertyString("chat.wordpredict.user");
        String strPwd = Application.getInstance().
            getPropertyString("chat.wordpredict.password");

        try {
            Class.forName(strDriver);
            conn_ = DriverManager.getConnection(strUrl, strUser, strPwd);
        }
        catch (SQLException e) {
            Application.getInstance().
                logError("Failed to create database connection. SQLException caught.", e);
        } catch (Exception e) {
            Application.getInstance().
                logError("Unable to load database driver ", + strDriver + ",.", e);
        }
    }
}
// ------------ public methods -----------

/*
* Inputstrings (of a history which is to be predicted) are:
* prevW
* prevV
* prevW
* This method in turn calls the submethod callWordPredict
*/
public Vector getCompletions(String str1, String str2, String str3)
{
    // Fill a vector with the alternatives received from
    // the callWordPredict method. There should never be
    // more than 100 results returned in the vector in this class
    Vector vec = new Vector();
    int max = 0;
    try {
        ResultSet res = callWordPredict(str1, str2, str3);
        while (res.next() && (max < 100)) {
            max++;
            String strWord = res.getString(1);
            vec.add(strWord);
        }
    } catch (SQLException e) {
        Application.getInstance().
        throwLogError("Failed to retrieve completions from database. SQLException caught.", e);
    }
    // Return the vector with database predictions
    return vec;
}

// ------------ private methods -----------

/*
* The actual lookup against the database function is made in this method
*/
private ResultSet callWordPredict(String str1, String str2, String str3) throws SQLException {

    // Create the callable statement
    if (csWordPredict_ == null) {
        csWordPredict_ = com_.prepareCall("? = call wordpredict(?, ?, ?)");
        csWordPredict_.registerOutParameter(1, OracleTypes.CURSOR);
        csWordPredict_.setMaxRows(100);
        csWordPredict_.setQueryTimeout(60);
    }
    // Insert the history strings into the statement
    csWordPredict_.setString(2, str1);
    csWordPredict_.setString(3, str2);
    csWordPredict_.setString(4, str3);

    // Execute the database lookup
    csWordPredict_.executeQuery();

    // Return the resultset
    return (ResultSet) csWordPredict_.getobject(1);
}
}
Appendix D

Selected test of the LM

These are two selected parts of test runs with the program “updatePolwithT-est.java”. The test text consist of 999 rows and 4104 words, which gives an average of almost 4 words per chat submission (since each line make up one chat submission). For a discussion around the results of these tests, see chapter 5.

D.1 A test with wordpredict

The first test was conducted with the wordpredict function (see appendix B), which means that it is more carefull about how many prefix letters that are needed when predicting.

*** Start testing ***
------------------------------------------------------------------------
Unigram LM:
The word: ello was not among the first thirty suggestions. OOV = 1
ello: gives total number of keystrokes: 5.0
LM keystrokes: 6.0
------------------------------------------------------------------------
Unigram LM:
The word: thanx was not among the first thirty suggestions. OOV = 2
thanx: gives total number of keystrokes: 11.0
LM keystrokes: 15.0
------------------------------------------------------------------------
Bigram LM:
word: mum. LM position: 10
new average = 10.0
mum: gives total number of keystrokes: 15.0
LM keystrokes: 16.0
------------------------------------------------------------------------
Unigram LM:
word: I. LM position: 1
new average = 5.5
I: gives total number of keystrokes: 17.0
LM keystrokes: 18.0
------------------------------------------------------------------------
Bigram LM:
word: CANT. LM position: 2
new average = 4.333333
CANT: gives total number of keystrokes: 22.0
LM keystrokes: 21.0

--------------------------------------

Unigram LM:
word: like. LM position: 2
new average = 3.75
like: gives total number of keystrokes: 27.0
LM keystrokes: 24.0

--------------------------------------

Bigram LM:
word: u. LM position: 1
new average = 3.2
u: gives total number of keystrokes: 29.0
LM keystrokes: 26.0

--------------------------------------

Trigram LM:
word: want. LM position: 1
new average = 2.633333
want: gives total number of keystrokes: 34.0
LM keystrokes: 28.0

--------------------------------------

Trigram LM:
word: to. LM position: 1
new average = 2.571428
to: gives total number of keystrokes: 37.0
LM keystrokes: 30.0

--------------------------------------

Trigram LM:
The word: still was not among the first thirty suggestions. 00V = 3
still: gives total number of keystrokes: 43.0
LM keystrokes: 37.0

--------------------------------------

Trigram LM:
word: be. LM position: 1
new average = 2.375
be: gives total number of keystrokes: 46.0
LM keystrokes: 39.0

--------------------------------------

Trigram LM:
word: with. LM position: 1
new average = 2.222222
with: gives total number of keystrokes: 51.0
LM keystrokes: 41.0

--------------------------------------

Trigram LM:
The word: chick was not among the first thirty suggestions. 00V = 4
chick: gives total number of keystrokes: 57.0
LM keystrokes: 50.0

--------------------------------------

Trigram LM:
word: but. LM position: 1
new average = 2.1
but: gives total number of keystrokes: 61.0
LM keystrokes: 52.0

--------------------------------------

Trigram LM:
word: cant. LM position: 3
new average = 2.181818
cant: gives total number of keystrokes: 66.0
LM keystrokes: 55.0

--------------------------------------
Unigram LM:
word: ure. LM position: 6
new average = 2.6
ure: gives total number of keystrokes: 70.0
LM keystrokes: 58.0

Bigram LM:
word: making. LM position: 8
new average = 2.9230769
making: gives total number of keystrokes: 77.0
LM keystrokes: 62.0

Trigram LM:
word: me. LM position: 1
new average = 2.7887144
me: gives total number of keystrokes: 80.0
LM keystrokes: 64.0

Trigram LM:
word: sad. LM position: 2
new average = 2.7333333
sad: gives total number of keystrokes: 84.0
LM keystrokes: 67.0

Trigram LM:
word: nov. LM position: 1
new average = 2.626
nov: gives total number of keystrokes: 86.0
LM keystrokes: 69.0

Unigram LM:
word: :) . LM position: 1
new average = 2.6294118
:) : gives total number of keystrokes: 91.0
LM keystrokes: 71.0

Unigram LM:
word: yo. LM position: 9
new average = 2.8688888
yo: gives total number of keystrokes: 94.0
LM keystrokes: 74.0

Unigram LM:
word: lol. LM position: 1
new average = 2.7694738
c: gives total number of keystrokes: 96.0
LM keystrokes: 76.0

Unigram LM:
The word: MIDNIGHT was not among the first thirty suggestions. QO5 = 5
MIDNIGHT: gives total number of keystrokes: 107.0
LM keystrokes: 88.0

Unigram LM:
word: just. LM position: 1
new average = 2.7
just: gives total number of keystrokes: 112.0
LM keystrokes: 90.0

Bigram LM:
word: want. LM position: 2
new average = 2.6666667
want: gives total number of keystrokes: 117.0
LM keystrokes: 93.0

Trigram LM:
word: 2. LM position: 1
new average = 2.590909
2: gives total number of keystrokes: 119.0
LM keystrokes: 95.0

Trigram LM:
word: know. LM position: 1
new average = 2.6217392
know: gives total number of keystrokes: 124.0
LM keystrokes: 97.0

Unigram LM:
word: and. LM position: 2
new average = 2.5
and: gives total number of keystrokes: 128.0
LM keystrokes: 100.0

Bigram LM:
word: i. LM position: 1
new average = 2.44
i: gives total number of keystrokes: 130.0
LM keystrokes: 102.0

Trigram LM:
word: would. LM position: 5
new average = 2.5384614
would: gives total number of keystrokes: 136.0
LM keystrokes: 106.0

Trigram LM:
word: newer. LM position: 7
new average = 2.6817735
never: gives total number of keystrokes: 16982.0
LM keystrokes: 15981.0

Trigram LM:
word: get. LM position: 1
new average = 2.6811557
get: gives total number of keystrokes: 16986.0
LM keystrokes: 15983.0

Trigram LM:
word: the. LM position: 2
new average = 2.8609665
the: gives total number of keystrokes: 16990.0
LM keystrokes: 15986.0

Trigram LM:
word: time. LM position: 3
new average = 2.8609966
time: gives total number of keystrokes: 16995.0
LM keystrokes: 15996.0

Trigram LM:
word: really. LM position: 1
new average = 2.8802866
really: gives total number of keystrokes: 19002.0
LM keystrokes: 15962.0

Unigram LM:
word: one. LM position: 5
new average = 2.8802866
one: gives total number of keystrokes: 19006.0
LM keystrokes: 15966.0

Bigram LM:
The word: mo was not among the first thirty suggestions. 0UV = 1042
mo: gives total number of keystrokes: 19009.0
LM keystrokes: 15969.0

Trigram LM:
word: brb. LM position: 3
new average = 2.8810227
brb: gives total number of keystrokes: 19013.0
LM keystrokes: 15972.0

Unigram LM:
word: how. LM position: 3
new average = 2.8810616
how: gives total number of keystrokes: 19017.0
LM keystrokes: 15976.0

Bigram LM:
word: old. LM position: 1
new average = 2.8804466
old: gives total number of keystrokes: 19021.0
LM keystrokes: 15977.0

Trigram LM:
word: are. LM position: 1
new average = 2.8796296
are: gives total number of keystrokes: 19025.0
LM keystrokes: 15979.0

Trigram LM:
word: you. LM position: 1
new average = 2.8792143
you: gives total number of keystrokes: 19029.0
LM keystrokes: 15981.0

Trigram LM:
The word: mini was not among the first thirty suggestions. 0UV = 1043
mini: gives total number of keystrokes: 19034.0
LM keystrokes: 15989.0

Unigram LM:
word: bye. LM position: 3
new average = 2.8792539
bye: gives total number of keystrokes: 19038.0
LM keystrokes: 15992.0

Bigram LM:
The word: yorkie was not among the first thirty suggestions. 0UV = 1044
yorkie: gives total number of keystrokes: 19045.0
LM keystrokes: 16002.0
New overall average: 2.8792839. 00V: 1044
position 1-10 = : 2964
position 1 = 1611
position 2 = 404
position 3 = 279
position 4 = 230
position 5 = 128
position 6 = 87
position 7 = 72
position 8 = 57
position 9 = 54
position 10 = 42
keystrokes = : 19045.0 LM keystrokes = 16002.0
keystrokes saved = 0.8402205

D.2 A test with wordpredict2

The second test was conducted with the wordpredict2 function (see appendix B),
which means that it is less careful about how many prefix letters that are needed
when predicting.

*** Start testing ***
------------------------------------
Unigram LM:
word: ello. LM position: 11
new average = 11.0
ello: gives total number of keystrokes: 5
LM keystrokes: 4
------------------------------------
Unigram LM:
word: thanx. LM position: 12
new average = 11.5
thanx: gives total number of keystrokes: 11
LM keystrokes: 8
------------------------------------
Bigram LM:
word: mum. LM position: 22
new average = 15.0
mum: gives total number of keystrokes: 15
LM keystrokes: 13
------------------------------------
Unigram LM:
word: I. LM position: 1
new average = 11.5
I: gives total number of keystrokes: 17
LM keystrokes: 16
------------------------------------
Bigram LM:
word: CANT. LM position: 2
new average = 9.6
CANT: gives total number of keystrokes: 22
LM keystrokes: 18
------------------------------------
Unigram LM:
word: like. LM position: 2
new average = 8.333333
like: gives total number of keystrokes: 27
LM keystrokes: 21

------------------------------------

APPENDIX D. SELECTED TEST OF THE LM

--------------------------------------------
Bigram LM:
word: u. LM position: 1
new average = 7.285714
u: gives total number of keystrokes: 29
LM keystrokes: 23
--------------------------------------------
Trigram LM:
word: want. LM position: 1
new average = 6.5
want: gives total number of keystrokes: 34
LM keystrokes: 25
--------------------------------------------
Trigram LM:
word: to. LM position: 1
new average = 5.868869
to: gives total number of keystrokes: 37
LM keystrokes: 27
--------------------------------------------
Trigram LM:
word: still. LM position: 20
new average = 7.3
still: gives total number of keystrokes: 43
LM keystrokes: 31
--------------------------------------------
Trigram LM:
word: be. LM position: 1
new average = 6.7272726
be: gives total number of keystrokes: 46
LM keystrokes: 33
--------------------------------------------
Trigram LM:
word: with. LM position: 1
new average = 6.25
with: gives total number of keystrokes: 51
LM keystrokes: 35
--------------------------------------------
Trigram LM:
The word: chick was not among the first thirty suggestions. G0V = 1
chick: gives total number of keystrokes: 57
LM keystrokes: 44
--------------------------------------------
Trigram LM:
word: but. LM position: 1
new average = 5.9461537
but: gives total number of keystrokes: 61
LM keystrokes: 46
--------------------------------------------
Trigram LM:
word: cant. LM position: 4
new average = 5.7142866
cant: gives total number of keystrokes: 66
LM keystrokes: 49
--------------------------------------------
Unigram LM:
word: ure. LM position: 6
new average = 5.733333
ure: gives total number of keystrokes: 70
LM keystrokes: 52
--------------------------------------------
Bigram LM:
word: making. LM position: 18
new average = 6.5
making: gives total number of keystrokes: 77
LM keystrokes: 56
-----------------------------------------------
TriGram LM:
word: me. LM position: 1
new average = 6.1764708
me: gives total number of keystrokes: 80
LM keystrokes: 58
-----------------------------------------------
TriGram LM:
word: sad. LM position: 2
new average = 5.944447
sad: gives total number of keystrokes: 84
LM keystrokes: 61
-----------------------------------------------
TriGram LM:
word: now. LM position: 1
new average = 5.6842103
now: gives total number of keystrokes: 88
LM keystrokes: 63
-----------------------------------------------
UniGram LM:
word: :). LM position: 1
new average = 5.45
:): gives total number of keystrokes: 91
LM keystrokes: 65
-----------------------------------------------
UniGram LM:
word: yo. LM position: 9
new average = 5.6190476
yo: gives total number of keystrokes: 94
LM keystrokes: 68
-----------------------------------------------
UniGram LM:
word: lol. LM position: 1
new average = 5.409091
lol: gives total number of keystrokes: 98
LM keystrokes: 70
-----------------------------------------------
UniGram LM:
The word: MIDNIGHT was not among the first thirty suggestions. 00V = 2
MIDNIGHT: gives total number of keystrokes: 107
LM keystrokes: 82
-----------------------------------------------
UniGram LM:
word: just. LM position: 1
new average = 5.2173915
just: gives total number of keystrokes: 112
LM keystrokes: 84
-----------------------------------------------
Bigram LM:
word: want. LM position: 2
new average = 5.06333335
want: gives total number of keystrokes: 117
LM keystrokes: 87
-----------------------------------------------
TriGram LM:
word: 2. LM position: 1
new average = 4.92
2: gives total number of keystrokes: 119
LM keystrokes: 89
APPENDIX D. SELECTED TEST OF THE LM

----------------------------------------
Trigram LM:
word: know. LM position: 1
new average = 4.769231
know: gives total number of keystrokes: 124
LM keystrokes: 91
----------------------------------------
Unigram LM:
word: and. LM position: 2
new average = 4.6666665
and: gives total number of keystrokes: 128
LM keystrokes: 94
----------------------------------------
Bigram LM:
word: i. LM position: 1
new average = 4.535714
i: gives total number of keystrokes: 130
LM keystrokes: 96
----------------------------------------
Trigram LM:
word: would. LM position: 5
new average = 4.551724
would: gives total number of keystrokes: 136
LM keystrokes: 99
----------------------------------------
.
.
----------------------------------------
Trigram LM:
word: really. LM position: 2
new average = 4.3836616
really: gives total number of keystrokes: 19002
LM keystrokes: 15862
----------------------------------------
Unigram LM:
word: one. LM position: 5
new average = 4.386423
one: gives total number of keystrokes: 19006
LM keystrokes: 15865
----------------------------------------
Bigram LM:
The word: mo was not among the first thirty suggestions. UV = 711
mo: gives total number of keystrokes: 19009
LM keystrokes: 15870
----------------------------------------
Trigram LM:
word: brb. LM position: 5
new average = 4.386623
brb: gives total number of keystrokes: 19013
LM keystrokes: 15873
----------------------------------------
Unigram LM:
word: how. LM position: 3
new average = 4.3884125
how: gives total number of keystrokes: 19017
LM keystrokes: 15876
----------------------------------------
Bigram LM:
word: old. LM position: 1
new average = 4.387411
old: gives total number of keystrokes: 19021
LM keystrokes: 15878
-------------------------------
Trigram LM:
word: are. LM position: 1
new average = 4.3864107
are: gives total number of keystrokes: 19025
LM keystrokes: 15880
-------------------------------
Trigram LM:
word: you. LM position: 1
new average = 4.3864103
you: gives total number of keystrokes: 19029
LM keystrokes: 15882
-------------------------------
Trigram LM:
The word: mini was not among the first thirty suggestions. 00V = 712
mini: gives total number of keystrokes: 19034
LM keystrokes: 15890
-------------------------------
Unigram LM:
word: bye. LM position: 3
new average = 4.3850017
bye: gives total number of keystrokes: 19038
LM keystrokes: 15893
-------------------------------
Bigram LM:
The word: yorkie was not among the first thirty suggestions. 00V = 713
yorkie: gives total number of keystrokes: 19045
LM keystrokes: 15903
-------------------------------
New overall average: 4.3850017. 00V: 713
position 1-10 = 2280
position 1 = 1586
position 2 = 414
position 3 = 271
position 4 = 221
position 5 = 128
position 6 = 104
position 7 = 76
position 8 = 66
position 9 = 58
position 10 = 56
keystrokes = : 19045 LM keystrokes = 15903
keystrokes saved = 0.83602233