# Wopr

Peter Berck

January 16, 2012

# Contents


1 Intro 4

2 Walk Throughs 5
- 2.1 Memory Based Language Model . . . 5
  - 2.1.1 Multiple Test Files . . . 8
  - 2.1.2 Combining Commands . . . 9
- 2.2 *n*-gram-model . . . 10
- 2.3 Text Generation . . . 12
- 2.4 Predictive Editing . . . 12
- 2.5 Combined Predictive Editing . . . 14
- 2.6 Spelling Correction . . . 16
- 2.7 Global Context . . . 17
- 2.8 Multi-classifiers . . . 20
  - 2.8.1 md/md2 . . . 20
  - 2.8.2 multi_gated, mg (for wex) . . . 23
- 2.9 Server . . . 23

3 Installation 25

4 Performance 26
- 4.1 Memory . . . 26
- 4.2 Comparison with SRILM . . . 26
  - 4.2.1 Logprob Comparisons . . . 27
  - 4.2.2 pplx . . . 30
  - 4.2.3 WOPR in an external task . . . 30

5 Reference 32
- 5.1 Data . . . 32
  - 5.1.1 `window_lr` . . . 32
  - 5.1.2 `window_letters` . . . 32
  - 5.1.3 `hapax` . . . 33
  - 5.1.4 `lexicon` . . . 33
  - 5.1.5 `rfl` . . . 34
  - 5.1.6 `lcontext` . . . 34
  - 5.1.7 `gap` . . . 34
- 5.2 Training . . . 36
  - 5.2.1 `make_ibases` . . . 36
  - 5.2.2 `ngl` . . . 36

5.3 Testing . . . 37
5.3.1 `pplxs` . . . 37
5.3.2 `gt` . . . 38
5.3.3 `ngt` . . . 39
5.3.4 `correct` . . . 39
5.4 Miscellaneous . . . 40
5.4.1 `generate` . . . 40
5.4.2 `pdt` . . . 41
5.4.3 `server4` . . . 41

# 1
# Intro

Wopr is a wrapper around the $k$-NN classifier in Timbl, offering word prediction and language modeling functionalities. Trained on a text corpus, wopr can predict missing words, report perplexities at the word level and the text level, and generate spelling correction hypotheses.

Besides the Timbl-related functionality, wopr can create and manipulated data sets.

Memory based word prediction illustrates an interesting idea. Complex abstraction levels are not needed for (some) linguistic tasks. In the case of word prediction, the words themselves provide enough information. To put it bluntly, we generalise without abstracting.

This manual is divided into several parts. Chapter Two contains a walk through illustrating the main use cases of wopr. Chapter Three shows how to compile and install wopr. Chapter Four has some information about the performance of wopr, and compares the output to srilm output. Finally, Chapter Five contains a more detailed reference or some of the wopr functions.

This manual assumes familiarity with language modeling in general. Experience with memory based based learning techniques (or with Timbl) is an distinct advantage.

# 2

# Walk Throughs

Without further ado, we present a number of common tasks which can be performed with WOPR. We'll jump right in, and explain the different steps and option on the way.

## 2.1 Memory Based Language Model

A memory based version of a trigram model. The following text corpora are used in the experiments; `rmt.5e5` for our training data, and `rmt.t1000` to test on. The former consists of the first 500 000 lines of the Reuters Newspaper corpus, while the latter contains the last 1 000 lines.

The following steps are typically taken when creating a model from scratch.

1. Create a lexicon
2. Create a windowed data set (both for training and testing)
3. Train the instance base
4. Run a prediction test on test data set

The lexicon is created as follows.

Listing 2.1: Creating a lexicon

```
1 wopr -r lexicon -p filename:rmt.5e5
```

This will create a frequency list of all the tokens in the specified file. Before we explain what we did, we will show the second step.

Listing 2.2: Creating a data set

```
1 wopr -r window_lr -p filename:rmt.5e5,lc:2,rc:0
```

These examples shows how WOPR is typically used. The two main points to note are `-r window_lr`, and the `-p ...`. The `-r` tells WOPR to run the `window_lr` function. Everything that follows after `-p` are the parameters passed to the function. The parameters are specified as a comma seperated list of `keyword:value` pairs. In this case, we specify a `filename`, `rmt.5e5` and the size of the left- and right contexts. The context consists of two words on the left of the target, and none on the right. This is abbreviated as `l2r0`.

WOPR is quite verbose:

Listing 2.3: WOPR output

```
wopr -r window_lr -p filename:rmt.5e5,lc:2,rc:0
06:58:37.21: Timbl support built in.
06:58:37.22: /Users/pberck/local/
06:58:37.22: Starting wopr 1.29.3
06:58:37.22: PID:   7433 PPID:   2142
06:58:37.22: Starting.
06:58:37.22: Running: window_lr
06:58:37.22: window_lr
06:58:37.22:  filename:  rmt.5e5
06:58:37.22:  lc:        2
06:58:37.22:  rc:        0
06:58:37.22:  to:        0
06:58:37.22:  OUTPUT:    rmt.5e5.l2r0
06:59:15.61: SET filename to rmt.5e5.l2r0
06:59:15.61: Result = 0
06:59:15.61: Running for 38s
06:59:15.61: Ready.
```

Note lines 13 and 14. In line 13, WOPR prints the name of the output file it creates. These filenames are generated automatically by WOPR and contain information about the parameters used. In the case of `window_lr`, it incorporates the context size in the file name. This allows WOPR to skip steps in its processing if they have been done before. Running the exact same command again will show the following:

```
wopr -l -r window_lr -p filename:rmt.5e5,lc:2,rc:0
07:15:36.38: Running: window_lr
07:15:36.38: window_lr
07:15:36.38:  filename:  rmt.5e5
07:15:36.38:  lc:        2
07:15:36.38:  rc:        0
07:15:36.38:  to:        0
07:15:36.38:  OUTPUT:    rmt.5e5.l2r0
```

```
07:15:36.38: OUTPUT exists, not overwriting.
07:15:36.38: SET filename to rmt.5e5.l2r0
07:15:36.38: Result = 0
```

Another thing to notice is the SET statement in line 10. This is useful when combining the different steps. This will be explained later.

The next step is the creation of the instance base (output has been reduced to improve readability).

Listing 2.4: Creating an instance base

```
wopr -l -r make_ibase -p filename:rmt.5e5.l2r0,timbl:"-a4␣+D"
10:02:16.99: Running: make_ibase
10:02:16.99: make_ibase
10:02:16.99:  timbl:     -a4 +D
10:02:16.99:  filename:  rmt.5e5.l2r0
10:02:16.99:  ibasefile: rmt.5e5.l2r0_-a4+D.ibase
...
Size of InstanceBase = 3366271 Nodes, (134650840 bytes), 29.46 % compression
Learning took 527 seconds, 732 milliseconds and 483 microseconds
Writing Instance-Base in: rmt.5e5.l2r0_-a4+D.ibase
10:12:57.87: SET ibasefile to rmt.5e5.l2r0_-a4+D.ibase
10:12:57.87: Result = 0
```

Creating the instance based is delegated to Timbl. We refer to [Daelemans et al., 2009] for a thorough explanation. The variables specified in the timbl parameter are passed verbatim to Timbl.

And finally, we use this instance base to run a perplexity calculation on a test set. The test set has been prepared in a step not shown here, in the same format as the training data. It contains the last 1 000 lines of the Reuters data. Here we also specify the lexicon created in step 1. The lexicon is used to determine if a word is known or unknown to the system.

Listing 2.5: Running a test

```
wopr -l -r pplxs -p ibasefile:rmt.5e5.l2r0_-a4+D.ibase,
                    filename:rmt.t1000.l2r0,timbl:"-a4␣+D",
                    lexicon:rmt.5e5.lex
...
11:50:10.93:  Correct:        3368 (20.5893)
11:50:10.93:  Correct Distr: 6250 (38.2076)
11:50:10.93:  Correct Total: 9618 (58.7969)
11:50:10.93:  Wrong:          6740 (41.2031)
```

```
9  11:50:10.93:  Timbl took: 11m47s
10 11:50:10.93:  SET px_file to rmt.t1000.l2r0_11615.px
11 11:50:10.93:  SET pxs_file to rmt.t1000.l2r0_11615.pxs
12 11:50:11.09: Result = 0
```

WOPR generates two output files. The important one is `rmt.t1000.l2r0_11615.px`. That contains the prediction and statistics for each test instance we processed.

An extra perl script, `pplxs_px.pl` has been supplied to post process the output. It can be found in the `etc/` directory in the WOPR distribution. It computes perplexity and other statistics on the output. It is run as follows.

Listing 2.6: Post processing of WOPR output

```
1 perl pplx_px.pl -f rmt.t1000.l2r0_11615.px -l2 -r0
```

The file to process is specified with the `-f` parameter. The context size is specified with the `-l` and `-r` parameters. It contains a line for each instance, followed by a summary. A fragment of the output:

```
0.02119694  -5.5600  -1.6737  1 1 [01010] ways
0.41176411  -1.2801  -0.3854  1 1 [01010] of
```

It shows the probability of the classification, followed by the $log_2$ and $log_{10}$ of the probability. For an explanation of the other values, see the reference chapter. The summary at the end shows, amongst other statistics, the perplexity on the test data.

```
Wordcount: 16358 sentencecount: 0 oovcount: 542
Wopr ppl:    251.41 Wopr ppl1:    251.41  (No oov words.)
```

To get performance per word, the `-w` flag can be added.

```
winner: cg:0 (0.00%) cd:0 (0.00%) ic:1 (100.00%)
with: cg:11 (18.33%) cd:22 (36.67%) ic:27 (45.00%)
```

This show that the word `winner` appeared once in the test set, and was incorrectly predicted. The word `with` was predicted correctly in 11 cases, et cetera.

### 2.1.1 Multiple Test Files

Instead of processing just one file, a directory argument can be given to WOPR to process all files matching a certain expression.

Listing 2.7: Processing multiple files

```
wopr -l -r pplxs -p ibasefile:rmt.5e5.l2r0_-a4+D.ibase,
                    dir:expdir,timbl:"-a4␣+D",
                    lexicon:rmt.5e5.lex
...
13:28:05.56:  dir:             expdir
13:28:05.56:  dirmatch:        .*
13:28:05.56: Processing 1 files.
...
13:28:34.65: Processing: expdir/rmt.t1000.l2r0
13:28:34.65: OUTPUT:     expdir/rmt.t1000.l2r0_4713.px
13:28:34.65: OUTPUT:     expdir/rmt.t1000.l2r0_4713.pxs
...
```

Lines five and six show the directory, and the expression used to match the files. The default is to take all the files in the directory. To specify something else the parameter `dirmatch` can be specified. To process all files ending in `l2r0`, the expression `l2r0$` can be supplied.

Line seven shows there is only one file, and the rest of the output shows that each file is processed in a similar manner to the single file method.

### 2.1.2 Combining Commands

As we mentioned before, the WOPR-commands can be combined. WOPR has a mechanism to automatically generate output filenames based on certain parameters. It will 'create' come of these parameters after each run, so they will be available to the next function as input without having to specify them from the start. We can combine several of the steps we shows in the previous part like this.

Listing 2.8: Combining commands

```
wopr -r lexicon,window_lr,make_ibase,pplxs -p filename:rmt.5e5,
                                              timbl:"-a4␣+D",lc:2,rc:0
...
09:11:54.15: SET ibasefile to rmt.5e5.l2r0_-a4+D.ibase
09:11:54.15: Result = 0
09:11:54.15: Running: pplxs
09:11:54.15: pplxs
09:11:54.15:  ibasefile:      rmt.5e5.l2r0_-a4+D.ibase
09:11:54.15:  lexicon:        rmt.5e5.lex
09:11:54.15:  counts:         rmt.5e5.cnt
...
```

If we look at the last step, `pplxs`, we see that the instance base filename and the other parameters have been taken from the previous steps. This example also shows theproblem with this mechanism; we cannot generate the windowed test set in the same chain. That has to be done seperately. (To be clear, one would not run the example in real life. It was to show both the strength and weakness of the combined commands.)

For complex WOPR usage, *scripting* can (should) be used.

## 2.2 *n*-gram-model

WOPR can also create a classical *n*-gram-model. This functionality has been included so as to be able to compare the memory based language model to a 'pure' *n*-gram-model. It has neither been optimised for speed, nor for memory efficiency.

Creation of the model consists of one step; `ngl`.

Listing 2.9: Creation of an *n*-gram model

```
1  wopr -r ngl -p filename:rmt.5e5
2  11:19:40.16: Running: ngl
3  11:19:40.18: ngl
4  11:19:40.18:  filename:  rmt.5e5
5  11:19:40.18:  n:         3
6  11:19:40.18:  fco:       0
7  11:19:40.18:  OUTPUT:    rmt.5e5.ngl3f0
8  11:19:40.18: Reading...
9  11:21:33.15: Writing...
10 11:22:36.35: SET ngl to rmt.5e5.ngl3f0
11 11:22:45.65: Result = 0
```

The output contains a list with all the *n*-grams followed by a frequency count and a conditional probability. The default *n*-gram size is three. Instead of the plain probability, the $log_2$ or $log_{10}$ of the probabilities can be output as well. Use the `log` parameter to specify the base of the log.

The command to process a test file is called `ngt`.

Listing 2.10: testing with an *n*-gram model

```
1  wopr -r ngt -p ngl:rmt.5e5.ngl3f0,testfile:rmt.t1000,topn:3
2  11:31:50.24: Running: ngt
3  11:31:50.24: ngt
4  11:31:50.24:  filename:  rmt.t1000
5  11:31:50.24:  ngl file:  rmt.5e5.ngl3f0
6  11:31:50.24:  ngc file:
7  11:31:50.24:  n:         3
```

```
8  11:31:50.24:  id:
9  11:31:50.24:  topn:      3
10 11:31:50.24:  mode:      wopr
11 11:31:50.24:  OUTPUT:    rmt.t1000.ngt3
12 11:31:50.24:  OUTPUT:    rmt.t1000.ngp3
13 11:31:50.24:  OUTPUT:    rmt.t1000.ngd3
14 11:31:50.24: Reading ngrams...
15 11:31:59.71: Writing output...
16 11:32:00.02: Total words: 16358
17 11:32:00.02: Total oovs: 542
18 11:32:00.02: Total log2prob: -121551
19 11:32:00.02: Average log2prob: -7.6853
20 11:32:00.02: Average pplx: 205.828
21 11:32:00.02: SET ngt_file to rmt.t1000.ngt3
22 11:32:00.02: SET ngp_file to rmt.t1000.ngp3
23 11:32:00.02: SET ngd_file to rmt.t1000.ngd3
24 11:32:01.90: Result = 0
```

This produces three output files. One, `rmt.t1000.ngt3`, contains statistics for each classification, as shown in the following fragment. It shows a word, the $n$-gram probability, the size of the matching $n$-gram and the $n$-gram itself.

```
and 0.0928793 2 puts and
calls 0.5 3 puts and calls
```

The second, `rmt.t1000.ngp3`, contains statistics per line of text.

```
-81.9518 293.087 10 0 The officers quickly managed to calm down ...
```

The output contains the entropy, perplexity, word count and number of unknown words, followed by the text line itself.

The third file, `rmt.t1000.ngd3`, is a file containing distributions of possible answers.

```
million 0.6 3 9 45 1 [ million 27 mln 6 to 3 ]
eight-year 0.000502681 2 680 5678 0.0208333 [ ) 342 . 342 in 326 ]
bond 0.142857 2 5 7 0.5 [ Eurobond 3 bond 1 contract 1 ]
```

It contains the target word, the probability of the classification (or logprob, if the `log` parameter has been specified), followed by the $n$ of the matching $n$-gram, the size of the distribution, the sum of the frequencies in the distribution, and the reciprocal rank of the target in the distribution. This is followed by the top–$n$ of the distribution.

The `ngt` function can also read and process a language model generated by SRILM. The SRILM model needs to have been generated with the `-write` option to generate a list with $n$-gram frequency counts. That file needs to be read with the `ngc` option.

Listing 2.11: Using an SRILM language model

```
1 ngram-count -no-eos -no-sos -text TXT -sort -write TXT.ngc -lm TXT.lm
2 wopr -r ngt -p ngl:TXT.lm,testfile:rmt.t1000,topn:3,mode:srilm,ngc:TXT.ngc
```

Using the `srilm` option sets the system to read and write the $log_{10}$ of the probabilities. This works also with the WOPR generated $n$-grams generated with `ngl`. In both the `ngl` and `ngt` functions the base of the logarithms can be specified with the `log` parameter.

## 2.3 Text Generation

WOPR can also be used in reverse, and generate text from a language model. The command is called `generate`.

Listing 2.12: Text generation

```
1 wopr -r generate -p ibasefile:rmt.5e5.l2r0_-a4+D.ibase,timbl:'-a4 +D',
2                     filename:out1,ws:2
```

The command still uses the older specification for context, `ws`. It needs to be set to the sum of the left and right context specifiers.

```
Kodjo took the loss , no delays .
USA : U.S. farm commodities to Russia , China .
The state had to evacuate the church for two consecutive terms in an
   abandoned car in the market moved in and out of parliament Koigi
   wa Wamwere and two hawks on to Baltimore .
The other 14 first list equities were unchanged in higher management
   ranks .
```

## 2.4 Predictive Editing

The text generation explained in the previous section can also be used to simulate *predictive editing*. Predictive editing is a tool to help authors write texts by preparing possible continuations of their text which can be insterted with a single key press or button click. The predictive editing simulation in WOPR generates possible strings after each word in a test set, and calculates how may key presses could have been saved.

The following example shows a typical run.

Listing 2.13: Predictive editing

```
1 wopr -r pdt -p filename:nyt.tail1000,timbl:"-a1␣+D",
```

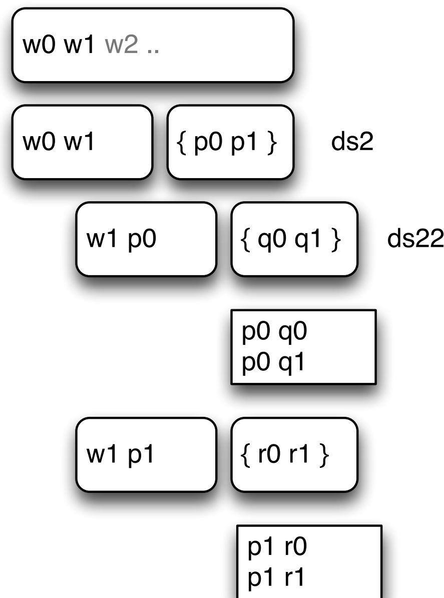


Figure 2.1: Predictions after `w0w1` (`n2ds22`)

```
2                  ibasefile:nyt.3e7.1000000.l2r0_-a1+D.ibase,
3                  lc:2,n:3,ds:511
4   Reading Instance-Base from: nyt.3e7.1000000.l2r0_-a1+D.ibase
5   ...
6   13:58:54.97: Keypresses: 136754
7   13:58:54.97: Saved      : 29119
8   ...
9   13:58:54.97: SET pdt_file to nyt.tail1000_n3ds511_25746.pdt
10  13:58:54.97: Result = 0
```

The `n` parameter specifies the length of the generated sequences. The `ds` parameter specifies the depth for each `n`. This works as follows. For each left context, there are a number of possible next words (predictions). These can be sorted by frequency, highest first. The depth parameter takes the top–$n$ of this distribution. For sequences longer than one word, each of the predictions is shifted into the context (the left most word is discarded), and a new prediction from the new context is made.

The output is quite verbose, and rather mechanical. The suffix of the output files is `pdt`. The first line is a summary line containing the length and branching settings, and the instance base used to run the experiment.

```
# l3 3 2 1 ../nyt.3e7.100000.l2r0_-a1+D.ibase
```

```
S0000 Instead , the stars were those whose congeniality put them at the
heart of the informal communication networks that would spring up during
times of crisis or innovation .  171
I0000.0000 _ Instead
P0000.0000.0000 , the former
M0000.0000.0000 , the 5
P0000.0000.0001 , he said
M0000.0000.0001 , 1
P0000.0000.0002 of continuing to
P0000.0000.0003 of seeing everything
P0000.0000.0004 he was a
P0000.0000.0005 he submitted to
E0000.0001 Instead ,
E0000.0002 , the
I0000.0003 the stars
...
R0000 5 14
...
T 136754 16222 11.8622
```

This is followed by the first sentence in the test set (S0000) followed by the length of the sentence. For each sentence, the instances are printed (I0000.0000). Each instance is followed by a number of predictions (P0000.0000.0000). If the prediction (partly) matches what is to come next, a line starting with an M is printed (M0000.0000.0000), containing the words matched plus a number indicating the key presses saved by that sequence. The 4-digit numbers are sentence number, word number, and prediction number. In the above example, there are two matches, M0000.0000.0000 and M0000.0000.0001. The former saves more keys (five) and is therefore chosen by the system. This means that two words are skipped, marked by the lines starting with an E (excluded). Then the system continues with the next word, I0000.0003.

At the end of each sentence, a result is printed (R0000) with the number of words and keypresses saved. The last line in the file displays the total score; total number of keypresses, the total number of keypresses saved, and the score as a percentage.

## 2.5 Combined Predictive Editing

The predictive editing system described in the previous section can be extended with a second classifier; one that is letter-based, and tries to predict the 'rest of the word being typed'. If the prediction matches, a further number of words are predicted with the word classifier, in the same way as explained in the previous paragraph. This way, we simulate a system that predicts a possible string of words after every letter being typed. This will save an additional number of keypresses over the first system. The following figure tries to show the system.

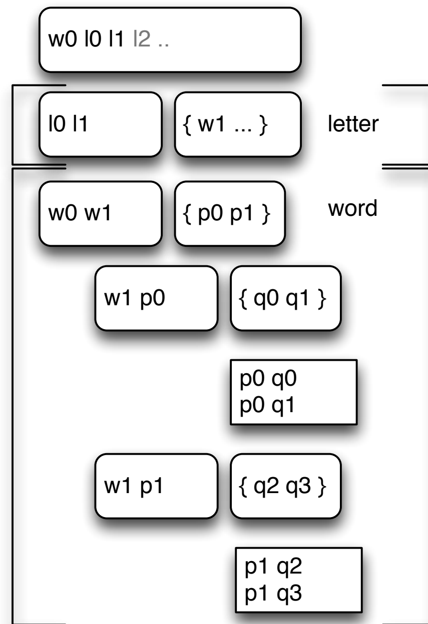


Figure 2.2: Predictions after `w0` and two letters

In figure 2.2, we show what happens after a word, `w0`, and two letters, `l0 l1`, have been typed. The letter classifier predicts predicts some words, including `w1` which is the correct word in the string that follows. At that point, the word predictor is called with `w1` plus some preceding words (`w0`) to predict a number of words to come after `w1`.

This variant is called `pdt2`, and a typical run could look like as shown in the following example. Note that there are two groups of parameters, one for each classifier. The parameters suffixed `0` are for the letter classifier, the ones suffixed with a `1` are for the word classifier.

Listing 2.14: Predictive editing

```
1  wopr -l -r pdt2 -p filename:austen.txt.l10,lc0:4,rc0:0,timbl0:'-a1 +D',
2                     ibasefile0:austen.txt.1e4.lt4m0c_-a1+D.ibase,
3                     lc1:2,rc1:0,timbl1:'-a1 +D',
4                     ibasefile1:austen.txt.1e4.l2r0_-a1+D.ibase,
5                     n:3,ds:5
```

The output produces is similar to the `pdt` output, but lines starting with `L` are added to show the predictions from the letter classifier. Line number 5 in the output example shows the letter classifier finding the word `communication` after the letters `commu` have been typed. It uses the last four letters typed as input, hence ony `ommu` is shown in the output. The user can save eight

keypresses by selecting the classifiers output at this point. But that is not all. In line seven, the system also predicts the string `of Mrs Elton` (which happens to be what the user wanted to write), saving another 12 letters. Combined, this will give a 21 letter saving (8 plus 12, plus the space after `communication`).

The last three lines of the file give a summary, `T0` for the letter classifier, `T1` for the word classifier, and a combined figure marked `T`.

Listing 2.15: Edited output

```
# l3 5 1 1
# austen.txt.1e4.lt4m0_-a1+D.ibase austen.txt.1e4.l2r0_-a1+D.ibase
S0000 communication of Mrs Elton and me 33
I0000.0000 _ communication
L0000.0000.0004 o m m u communication 8
P0000.0000.0001 of Mrs Elton
M0000.0000.0001 of Mrs Elton 12
E0000.0001 communication of
E0000.0002 of Mrs
E0000.0003 Mrs Elton
I0000.0004 Elton and
L0000.0004.0000 o n _ a and 2
P0000.0004.0004 now you have
I0000.0005 and me
L0000.0005.0000 n d _ m me 0
P0000.0005.0003 is quite a
R0000 3 12 11
T0 33 11 33.3333
T1 33 12 36.3636
T 33 23 69.697
```

The last line marked `L` in the example predicts zero keypresses saved. It actually finds `me` after `m`, but we need to subtract one for the user's 'select' action.

## 2.6 Spelling Correction

Wopr can also be used to do spelling correction. The correction algorithm is based on words in the predicted list of words. A filter is applied to the list, removing words which fall outside predefined parameters such as levenshtein distance, word length and frequency.

Listing 2.16: Spelling correction

```
1 wopr -r correct -p ibasefile:rmt.5e5.l2r0_-a4+D.ibase,timbl:"-a4␣+D",
2                    filename:rmt.t1000.l2r0
```

```
3  14:16:50.07: Running: correct
4  14:16:50.07: correct
5  14:16:50.07:  ibasefile:  rmt.5e5.l2r0_-a4+D.ibase
6  14:16:50.07:  lexicon:
7  14:16:50.07:  counts:
8  14:16:50.07:  timbl:      -a4 +D
9  14:16:50.07:  id:         7272
10 14:16:50.07:  mwl:        5
11 14:16:50.07:  mld:        1
12 14:16:50.07:  max_ent:    5
13 14:16:50.07:  max_distr:  10
14 14:16:50.07:  min_ratio:  0
15 14:16:50.07: Processing 1 files.
16 ...
17 14:17:13.02: Processing: rmt.t1000.l2r0
18 14:17:13.02: OUTPUT:     rmt.t1000.l2r0_7272.sc
19 ...
20 14:35:42.79:  SET sc_file to rmt.t1000.l2r0_7272.sc
21 14:35:42.80: Result = 0
```

The statistics printed (here removed) can be ignored. The output will contain a list with the words falling within the parameters. They are potential corrections of misspelt words.

Reuters doesn't contain misspellings, so we show output from another file.

```
_ _ Her ssiter (sister) -6.26454 0 76.88 1 [ sister 1 ]
```

It shows the instance, followed by the classifiers best guess between parenthesis. This is followed by the logprob, the entropy and the word logprob ($2^{-logprob}$). The final number is the size of the Timbl distribution returned, followed by a list with potential misspellings.

## 2.7 Global Context

Wopr contains a mechanism to include a larger, *global* context in the classifier. This is done by marking words from a list with words that are deemed important. When going through a text to make instances, we put the words from the list we encounter in the text in the global context. The global context has a certain size, so that older words are pushed out by newer words. We also put a decay factor on the words, so that they will disappear even if they are not pushed out by newer words. These global context instances can be seen as instances with gaps in them (after all, we don't include every word), and the idea is that they capture the essence of the (preceeding) text. The gaps are of variable length, so they have been dubbed *elastigrams*. To create the instances for Timbl, we add the global context instances to already existing 'normal' instances (in fact, the global context instances are created from a regular data set).

There is a function to create the global context called `lcontext`, and one to create a list with important words from the lexicon, called `rfl`.

The steps involved are the following.

1 create training data

2 create a list

3 create the global context from the list and training data

The first steps have been explained before, and will just be mentioned here.

Listing 2.17: Creating lexicon and training data

```
1 wopr -r lexicon -p filename:rmt.5e5
2 wopr -r window_lr -p filename:rmt.5e5,lc:2,rc:0
```

In the next step, we create a list with words which we want to include in our global context. We take a part of the lexicon, namely from the 50th most frequent word to the 550th most frequent word. We skip the top-50 words because these are function words, and deemed less important when trying to catch the essence of the text. The list will still not be perfect, but serves our purpose.

Listing 2.18: Creating list for global context training data

```
1  wopr -r rfl -p lexicon:rmt.5e5.lex,m:50,n:550
2  ...
3  10:53:37.62: range_from_lex
4  10:53:37.62:  lexicon: rmt.5e5.lex
5  10:53:37.62:  m:       50
6  10:53:37.62:  n:       550
7  10:53:37.62:  OUTPUT:  rmt.5e5.lex.r50n550
8  10:53:37.62: Reading lexicon.
9  10:53:37.95: Read lexicon.
10 10:53:38.19: Frequency list: 1786 items.
11 10:53:38.25: Range file contains 539 items.
12 10:53:38.25: SET range to rmt.5e5.lex.r50n550
13 10:53:38.34: Result = 0
```

The first five word in the file are:

```
year 17964
U.S. 17870
market 17175
but 16501
after 16004
```

Now we can combine the list and the training data to form the global context training data.

Listing 2.19: Creating global context training data

```
wopr -r lcontext -p range:rmt.5e5.lex.r50n550,filename:rmt.5e5.l2r0,
                    gcd:50,gcs:10
...
10:58:39.91: lcontext
10:58:39.91: Reading range file.
10:58:39.91: Loaded range file, 539 items.
10:58:39.91:  filename:  rmt.5e5.l2r0
10:58:39.91:  range:     rmt.5e5.lex.r50n550
10:58:39.91:  gcs:       10
10:58:39.91:  gcd:       50
10:58:39.91:  gct:       0
10:58:39.91:  from_data: true
10:58:39.91:  gc_sep:    true
10:58:39.91:  OUTPUT:    rmt.5e5.l2r0.gc10d50t0
11:01:34.19: SET filename to rmt.5e5.l2r0.gc10d50t0
11:01:34.72: Result = 0
```

The following shows a fragment of the resulting data set. In line two, the word `markets` is (again) added to the head of the global context part of the instance (fourth word from the right). In line three `into` is added, and everything shifts a place to the left.

```
_ _ _ _ _ _ markets markets economy back sent Mexican markets
_ _ _ _ _ markets markets economy back markets Mexican markets into
_ _ _ _ markets markets economy back markets into markets into a
_ _ _ _ markets markets economy back markets into into a buzz
```

Finally, we create a new instance base, and run a word prediction experiment.

Listing 2.20: Creating global context training data

```
wopr -r make_ibase -p filename:rmt.5e5.l2r0.gc10d50t0,timbl:'-a4 +D'
11:07:57.72: Timbl support built in.
11:07:57.72: /Users/pberck/local/
11:07:57.72: Starting wopr 1.29.3
11:07:57.72: PID:   9161 PPID:   4617
11:07:57.72: Starting.
11:07:57.72: Running: make_ibase
11:07:57.72: make_ibase
11:07:57.72:  timbl:     -a4 +D
```

```
10 11:07:57.72:  filename:  rmt.5e5.l2r0.gc10d50t0
11 11:07:57.72:  ibasefile: rmt.5e5.l2r0.gc10d50t0_-a4+D.ibase
12 ...
13 11:20:12.39: SET ibasefile to rmt.5e5.l2r0.gc10d50t0_-a4+D.ibase
14 11:20:12.39: Result = 0
```

Listing 2.21: Creating global context training data

```
1  /exp/pberck/wopr/wopr -l -r pplxs -p filename:rmt.t1000.l2r0.gc10d50t0,
2              timbl:"-a4 +D",ibasefile:rmt.5e5.l2r0.gc10d50t0_-a4+D.ibase,
3              lexicon:rmt.5e5.lex
4  ...
5  11:58:55.32: Processing: rmt.t1000.l2r0.gc10d50t0
6  11:58:55.32: OUTPUT:     rmt.t1000.l2r0.gc10d50t0_30826.px
7  11:58:55.32: OUTPUT:     rmt.t1000.l2r0.gc10d50t0_30826.pxs
8  12:51:30.18:  Correct:       3141 (19.2016)
9  12:51:30.18:  Correct Distr: 866 (5.29405)
10 12:51:30.18:  Correct Total: 4007 (24.4957)
11 12:51:30.18:  Wrong:         12351 (75.5043)
12 12:51:30.18:  Timbl took: 52m25s
13 12:51:30.18:  SET px_file to rmt.t1000.l2r0.gc10d50t0_30826.px
14 12:51:30.18:  SET pxs_file to rmt.t1000.l2r0.gc10d50t0_30826.pxs
15 12:51:30.24: Result = 0
```

There are a number of parameters that can be tweaked to get a better peformance. We refer to the reference section for details.

## 2.8 Multi-classifiers

WOPR contains functionality to apply more than one classifier on a task. One modus operandus is to apply multiple classifiers and combine the different outputs, another is to use a gated approach, and choose a certain classifier based on context features.

The instance bases and parameters are stored in a configuration file which is read by WOPR.

### 2.8.1 md/md2

The following shows a configuration file to apply two different classifiers to a test set in two different formats (to match the training set). The test sets need to be generated from the same data, and so must the instance bases.

Listing 2.22: Example kvs file

```
classifier:T0
```

```
ibasefile:austen.train.l2r0_-a1+D.ibase
timbl:-a1 +D
testfile:austen.test.l2r0

classifier:T1
ibasefile:austen.train.l3r0_-a1+D.ibase
timbl:-a1 +D
testfile:austen.test.l3r0
```

```
wopr -r md2 -p kvs:austenmd2.kvs
10:21:33.93: Running: md2
...
10:21:33.93: Reading classifiers.
Reading Instance-Base from: austen.train.l2r0_-a1+D.ibase
Feature Permutation based on Data File Ordering :
< 2, 1 >
Reading weights from austen.train.l2r0_-a1+D.ibase.wgt
10:21:34.77: T0/1
10:21:34.77: T0/austen.train.l2r0_-a1+D.ibase/austen.test.l2r0/wgt=1/type=1
Reading Instance-Base from: austen.train.l3r0_-a1+D.ibase
Feature Permutation based on Data File Ordering :
< 3, 2, 1 >
Reading weights from austen.train.l3r0_-a1+D.ibase.wgt
10:21:36.13: T1/1
10:21:36.13: T1/austen.train.l3r0_-a1+D.ibase/austen.test.l3r0/wgt=1/type=1
10:21:36.13: Read classifiers. Starting classification.
10:21:52.30: T0: 1092/8308
10:21:52.31: T1: 1068/8308
10:21:52.31: SET filename to austenmd2.kvs_8713.mc
10:21:52.31: Result = 0
```

Output:

```
Mrs [ and 0.0629821 1 true and 0.0629821 1 true ]
Dashwood [ Weston 0.138182 1 false Weston 0.138182 1 false ]
could [ was 0.113636 2 true and 0.133333 3 true ]
think [ find 1 2 true find 1 2 true ]
of [ of 0.192746 1 false of 0.705882 2 false ]
no [ it 0.125 2 true nothing 0.6 3 true ]
other [ consequence 0.192308 2 true other 1 3 true ]
```

It shows the target followed by the output of each classifier between square brackets. Each classification is followed by its probability, which is calculated by taking the frequency of the

answer divided by the sum of all the frequencies in the distribution, the depth at which TIMBL matched in the tree, and a boolean value indicating if we matched at a leaf. Four values for each classifier.

Extra parameter `c:1`, combines the distributions, and prints the best of the combined between braces.

```
Mrs [ and 0.0629821 1 true and 0.0629821 1 true ] { and 0.125964 }
Dashwood [ Weston 0.138182 1 false Weston 0.138182 1 false ] { Weston 0.276364 }
could [ was 0.113636 2 true and 0.133333 3 true ] { was 0.24697 }
think [ find 1 2 true find 1 2 true ] { find 2 }
of [ of 0.192746 1 false of 0.705882 2 false ] { of 0.898628 }
no [ it 0.125 2 true nothing 0.6 3 true ] { nothing 0.634091 }
other [ consequence 0.192308 2 true other 1 3 true ] { other 1.07692 }
```

Instead of applying several classifiers, a pre-made distribution can be added to the classifiers output. Think of this as a way to highlight/emphasise classes in the classifiers output. The extra distribution file needs to contain as many instances as the regular test set. We can, for example, create a file with global information about each instance in the test file. When these occur in the distribution of the classifier, they can be brought to the front.

An example `kvs`.

```
classifier:T0
ibasefile:austen.train.l2r0_-a1+D.ibase
timbl:-a1 +D
testfile:austen.test.l2r0

classifier:TG
distfile: austen.test.l0r0.gc10d50t0
distprob: 0.1
```

Note that WOPR assumes that the data in the distfile is similar to a data file with instances - that is, the last token on each line is considered to be the target value and is ignored. The _ value is also ignored, and doubles are removed.

Then we run WOPR.

```
1 wopr -r mc -p kvs:austen3.kvs,c:1,topn:5
```

The output looks like this. Note that the output of the TG "classifier" is not saved in the output file.

```
coolness [ a 0.181818 1 false ] { between 0.190909 each 0.190909 ... }
between [ of 0.25 1 false ] { between 0.35 arise 0.25 of 0.25 ... }
their [ the 0.214286 1 false ] { the 0.214286 them 0.193277 ... }
```

Note the second line, where the combined distribution contains the right answer.

### 2.8.2 multi_gated, mg (for wex)

## 2.9 Server

Wopr can be run in *server* mode where it will listen on a socket for input and return a classification result.

Several implementations have been tried, the most recent and working function is called `server4`. It has a few obscure options to make it compatible with third party software like `moses` and pbmbmt. The following command line shows an example of how it can be used.

```
%wopr -r server4 -p ibasefile:OpenSub-english.train.txt.l2r0_-a1+D.ibase,
%            timbl:'-a1 +D',mode:1,keep:1,verbose:1,
%            lexicon:OpenSub-english.train.txt.lex,port:8888

wopr -r server4 -p ibasefile:rmt.5e5.l2r0_-a4+D.ibase,lexicon:rmt.5e5.lex,
                   keep:1,mode:1,verbose:1
16:47:42.60: Timbl support built in.
16:47:42.60: /Users/pberck/local/
16:47:42.60: Starting wopr 1.29.8
16:47:42.60: PID:  34818 PPID:  34459
16:47:42.93: Starting.
16:47:42.93: Running: server4
16:47:42.93: server4. Returns a log10prob over sequence.
...
```

Wopr will respond like this to input.

```
16:48:24.22: Connection 4/www.wopr.dev [127.0.0.1]
16:48:24.23: Listening...
16:48:26.31: |1 2 3|
16:48:26.77: result sum=-6.98105/1.04459e-07
16:48:26.77: result ave=-2.32702/0.00470958
```

The esoteric options are `mode`, and `keep`. The `mode:1` argument specifies that we will send an *instance* to wopr. The default, `mode:0` indicates we send a whole sentence which will be windowed first. For this to work, the context size must be specified as well, with the parameters `lc` and `rc`.

For pbmbmt, a special version of the server has been created, called `mbmt`. It is similar to `server4`, but has several of the parameters hard-coded and doesn't 'fork', making it easier to run from scripts in combination with pbmbmt.

The following snippet illustrates this. The wopr language model is started in the background, and then pbmbmt is started. Wopr is terminated after the translation is ready.

```
1  # Run mbmt server on port 8888
2  nohup wopr -r mbmt -p ibasefile:file.ibase,timbl:"-a1␣+D",lc:2,rc:0, \
3      port:8888 > lmlog.txt 2>&1 &
4
5  # Wait a little for LM to become ready
6  sleep 30
7
8  # Run pbmbmt, point to LM server port 8888
9  pbmbmt.py -DL8888 -- dutch english test exp1 > mtlog.txt 2>&1
10
11 # Exit wopr by sending the _QUIT_ command
12 kill `echo "_QUIT_" | nc localhost 8888`
```

# 3 Installation

TIMBL is required to run WOPR.

Assuming TIMBL is configured and installed locally with `-prefix=/home/pberck/local`. We also install WOPR locally:

Listing 3.1: Configuring WOPR locally

```
sh bootstrap
./configure --prefix=/home/pberck/local \
            --with-timbl=/home/pberck/local
make
make install
```

If TIMBL is installed system wide, the `-with-timbl` parameter can be omitted. The configure script should be able to figure out where the libraries are. Wopr can be built without TIMBL by specifying `-without-timbl`. This will leave you with a WOPR which can create datafiles and run an $n$-gram model only. The `LD_LIBRARY_PATH` and `PATH` variables might need to be set as well. A short test to see if all went well:

Listing 3.2: Running WOPR

```
./wopr
10:44:32.19: Timbl support built in.
10:44:32.19: /Users/pberck/local/
10:44:32.19: Starting wopr 1.29.3
10:44:32.20: PID:   4173 PPID:   1136
10:44:32.20: Starting.
10:44:32.21: Running for 00s
10:44:32.21: Ready.
```

The second line shows the path tot the TIMBL libraries.

# 4 Performance

## 4.1 Memory

Memory based classifiers tend to, nomen est omen, use a lot of memory. To illustrate, we show two plots of memory usage; one for a training task (Figure 4.1) and one for a classification task. The training corpus consisted of 1 000 000 lines (22 855 429 instances) of text taken from the Gigaword Newspaper corpus. The context size was `l3r0`. The scale of the $y$-axis in both graphs is in kB. The $x$-axis shows the time taken in minutes.

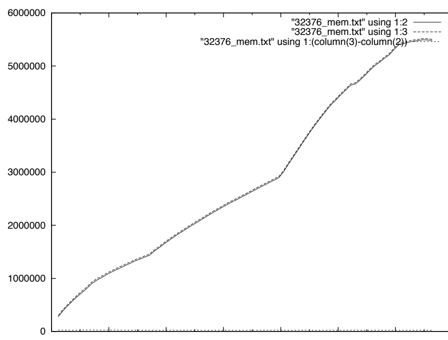


Figure 4.1: nyt.1000000.l3r0

The following plot (Figure 4.2) shows the memory use when using the above instance base for a classification task. The test set consisted of 1 000 lines of text from the same Gigaword newspaper corpus.

## 4.2 Comparison with srilm

The memory based approach to language modeling has several advantages over a classical $n$-gram-based approach. First of all, we are not limited to $n$-grams but instead are able to use any

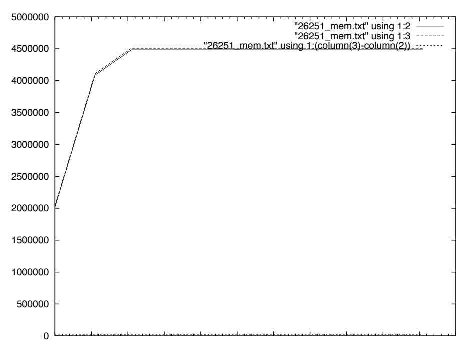


Figure 4.2: nyt.1000000.l3r0 ibase, test tail1000

number and type of features in the instances. The instances can for example contain grammatical and non-local context information. It is also possible to use a right-context together, or even without, the words on the left.

Secondly, the tree-based approach has an inherent back-off mechanism. In classical $n$-gram-models, a back-off to smaller sequences (from trigrams to bigrams, to unigrams) is performed when an $n$-gram is not found in the training data, typically when unseen words are encountered. The IGTREE algorithm returns the distribution stored at that point in the tree when an unknown word is encountered. The TRIBL2 algorithm is more robust in that sense; it continues to try to match the remaining feature values.

In the following examples, we'll compare a memory based language model with a language model generated by SRILM.

### 4.2.1 Logprob Comparisons

We created two language models, one with WOPR and one with SRILM. We used 100 000 lines of text from the Reuters corpus. For WOPR, the context was set to `l2r0`. For SRILM, a third order language model was created. Test set was 1 000 lines taken from the same corpus.

We take a sentence from the test data. First we show the WOPR output. For each word in the test sentence we show the probability, the $log_2$ and $log_{10}$ of the probability, followed by some stats we will ignore for the moment.

Listing 4.1: WOPR output

```
0.07905  -3.6611  -1.1021  1 0 [00100] The
0.00324  -8.2712  -2.4899  1 0 [01000] only
0.05263  -4.2479  -1.2788  2 1 [01010] reason
0.37984  -1.3965  -0.4204  1 0 [00100] for
0.00008 -13.6354  -4.1047  1 0 [01000] caution
0.00451  -7.7941  -2.3462  2 1 [10010] is
0.01645  -5.9258  -1.7838  1 0 [01000] that
```

```
0.02150   -5.5394   -1.6675  1 0 [01000] we
0.01473   -6.0853   -1.8319  1 0 [01000] do
0.33673   -1.5703   -0.4727  1 0 [00100] not
0.03030   -5.0444   -1.5185  2 1 [01010] know
0.04545   -4.4594   -1.3424  2 1 [01010] what
0.00072 -10.4446   -3.1441  1 0 [10000] other
0.00004 -14.7434   -4.4382  1 0 [10000] demands
0.00002 -15.5113   -4.6694  1 0 [10000] Colston
0.00016 -12.6280   -3.8014  1 0 [10000] might
0.10623   -3.2348   -0.9738  1 0 [01000] have
0.00052 -10.8982   -3.2807  1 0 [01000] .
```

And the SRILM output. The output is taken from the debug output from SRILM. It shows the word (in context), followed by the size of the matching $n$-gram, the probability and the $log_{10}$ of the probability.

Listing 4.2: SRILM output

```
        p( The |  )                 = [1gram] 0.00679429 [ -2.16786 ]
        p( only | The ...)          = [2gram] 0.00323679 [ -2.48989 ]
        p( reason | only ...)       = [3gram] 0.0205598 [ -1.68698 ]
        p( for | reason ...)        = [3gram] 0.384991 [ -0.41455 ]
        p( caution | for ...)       = [2gram] 2.32724e-05 [ -4.63316 ]
        p( is | caution ...)        = [1gram] 0.00198117 [ -2.70308 ]
        p( that | is ...)           = [2gram] 0.0164503 [ -1.78383 ]
        p( we | that ...)           = [3gram] 0.0625 [ -1.20412 ]
        p( do | we ...)             = [3gram] 0.00451603 [ -2.34524 ]
        p( not | do ...)            = [3gram] 0.65 [ -0.187087 ]
        p( know | not ...)          = [3gram] 0.024045 [ -1.61898 ]
        p( what | know ...)         = [2gram] 0.0868785 [ -1.06109 ]
        p( other | what ...)        = [1gram] 0.000100046 [ -3.9998 ]
        p( demands | other ...)     = [1gram] 1.4466e-05 [ -4.83965 ]
        p( Colston | demands ...)   = [1gram] 5.8895e-06 [ -5.22992 ]
        p( might | Colston ...)     = [1gram] 6.02518e-05 [ -4.22003 ]
        p( have | might ...)        = [2gram] 0.106227 [ -0.973765 ]
        p( . | have ...)            = [2gram] 0.000162122 [ -3.79016 ]
```

Normally, SRILM wraps each sentence in a *start of sentence* and *end of sentence* marker. These have been switched off in this comparison. The start-of-sentence marker is a bit like our empty context markers which we insert before the first word(s) in a sentence.

Listing 4.3: $log_{10}$probablities compared

```
-1.10209 -2.16786 1   11.6 [001] The
```

```
-2.48988 -2.48989 2    1.0 [000] only
-1.27875 -1.68698 3    2.6 [001] reason
-0.42039 -0.41455 3    1.0 [000] for
-4.10466 -4.63316 2    3.4 [001] caution
-2.34625 -2.70308 1    2.3 [001] is
-1.78383 -1.78383 2    1.0 [000] that
-1.66753 -1.20412 3    0.3 [010] we
-1.83187 -2.34524 3    3.3 [001] do
-0.47271 -0.18709 3    0.5 [000] not
-1.51851 -1.61898 3    1.3 [000] know
-1.34242 -1.06109 2    0.5 [000] what
-3.14414 -3.99980 1    7.2 [001] other
-4.43822 -4.83965 1    2.5 [001] demands
-4.66936 -5.22992 1    3.6 [001] Colston
-3.80140 -4.22003 1    2.6 [001] might
-0.97376 -0.97376 2    1.0 [000] have
-3.28069 -3.79016 2    3.2 [001] .
```

Some predictions show a similar logprob from both WOPR and SRILM. For example, both language models give the same likelyhood to the second word in the sentence, `only`. Also the seventh word, `that` gets similar logprobs.

The difference is most likely due to *smoothing* and *hapaxing*, which is done by SRILM but not by WOPR.

The trigrams generated by SRILM can also be read by WOPR, and used instead of the WOPR-generated $n$-gram-model. The output can then be compared with the WOPR $n$-gram-model. Doing this allows WOPR to determine the distribution of the classifications, and reciprocal ranking of the correct classification in the distribution. The following table shows a comparison of WOPR trigram-model scores and SRILM scores. The first two columns show the $log_{10}$-probabilities of the models. These are followed by the length of the matching $n$-gram, and some statistics on the distributions, the last two numbers are the reciprocal rank.

Listing 4.4: $n$-gram-models compared

```
 -2.1679   -2.1679 1 1  77513  77513 1728063 1728063 0.062 0.062 The
 -2.4899   -2.4899 2 2   3096   3096   11740   11740 0.040 0.040 only
 -1.2788   -1.6870 3 3     24      4      38      18 0.333 0.333 reason
 -0.2218   -0.4145 3 3      3      1       5       3 1.000 1.000 for
 -4.1053   -4.3423 2 2   2630   2630   12725   12725 0.017 0.017 caution
 -2.3462   -2.3462 1 1  77513  77513 1728063 1728063 0.043 0.043 is
 -1.7841   -1.7838 2 2   1466   1466    7781    7781 0.125 0.125 that
 -1.2041   -1.2041 3 3     45     21     128     104 0.500 0.500 we
 -1.9370   -2.3452 3 3     37     20     173     156 0.100 0.100 do
 -0.1871   -0.1871 3 3      7      2      20      15 1.000 1.000 not
```

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| -1.5185 | -1.6190 | 3 | 3 | 69 | 25 | 165 | 121 | 0.333 | 0.333 | know |
| -1.3424 | -1.0212 | 3 | 2 | 13 | 47 | 22 | 147 | 0.333 | 1.000 | what |
| -3.1441 | -3.1441 | 1 | 1 | 77513 | 77513 | 1728063 | 1728063 | 0.008 | 0.008 | other |
| -4.4382 | -4.4382 | 1 | 1 | 77513 | 77513 | 1728063 | 1728063 | 0.001 | 0.001 | demands |
| -4.6694 | -4.6694 | 1 | 1 | 77513 | 77513 | 1728063 | 1728063 | 0.001 | 0.001 | Colston |
| -3.8014 | -3.8014 | 1 | 1 | 77513 | 77513 | 1728063 | 1728063 | 0.002 | 0.002 | might |
| -0.9738 | -0.9738 | 2 | 2 | 101 | 101 | 273 | 273 | 0.500 | 0.500 | have |
| -3.2807 | -3.5967 | 2 | 2 | 750 | 750 | 3817 | 3817 | 0.029 | 0.029 | . |

### 4.2.2 pplx

The IGTREE algorithm produces the following perplexity graph on the English data (Figure 4.3).

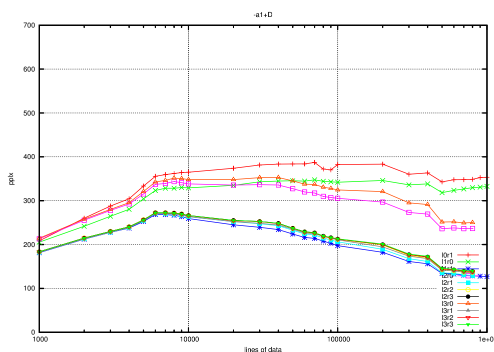


Figure 4.3: Perplexity, IGTREE

The perplexity score is low in the beginning, due to the large number of unknown words. The only known words are the function words, making up the skeleton of the sentence. Most of the content words will still be unknown at this point. As we train the classifier on more and more data, more of the content words become known. This causes a rise in the perplexity score first, before it starts to fall again.

The corresponding SRILM graph looks like this (Figure 4.4). Note that SRILM calculates two figures, one with (labeled pplx) and one without sentence markers (labeled pplx1).

### 4.2.3 wopr in an external task

We evaluate WOPR in a MT task. We run WOPR as the language model in PBMBMT, and compare the score (BLEU-score) with SRILM as the language model. We run PBMBMT on the included OpenSubtitles (Dutch to English) data.

The following table (Table 4.1) shows the scores of the experiment, run with a third order SRILM model, a `l2r0` WOPR IGTREE and a WOPR TRIBL2 language model. The second part of the table shows the scores with a larger context.

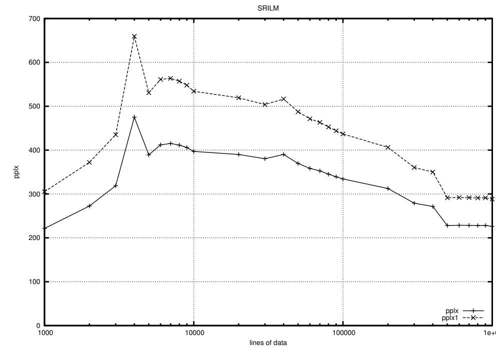


Figure 4.4: Perplexity, SRILM

Table 4.1: PBMBMT scores

| lm | BLEU | mtr | NIST | TER | WER | PER | h:m |
|---:|---:|---:|---:|---:|---:|---:|---:|
| SRILM | 0.2552 | 0.4964 | 5.5436 | 0.5604 | 56.18 | 48.15 | 0:82 |
| IGTREE | 0.2608 | 0.4963 | 5.3051 | 0.6036 | 60.40 | 52.15 | 0:39 |
| TRIBL2 | 0.2603 | 0.4969 | 5.3080 | 0.6036 | 60.38 | 52.04 | 0:66 |
| IGTREE `l2r1` | 0.2592 | 0.4968 | 5.3056 | 0.6085 | 60.61 | 52.25 | 0:47 |
| TRIBL2 `l2r1` | 0.2609 | 0.4967 | 5.3051 | 0.6066 | 60.40 | 52.36 | 3:06 |
| TRIBL2 `l2r2` | 0.2609 | 0.4963 | 5.3046 | 0.6092 | 60.80 | 52.31 | 9:35 |

# 5
# Reference

## 5.1 Data

### 5.1.1 `window_lr`

Creates a *windowed* data set. The `lc` and `rc` parameters set the size (number of words) of the left and right context. Out of sentence feature values will be filled with as many `_` (underscore character) as needed. WOPR assumes the input file contains one sentence per line. It will slide a window over each sentence, and each word will be at the target position once. There is an extra parameter `to`, which when set will let you predict the next word after the next word. That is, if set to 1, it will not take the token at the window position as target, but the one to the right. Setting it to 2 skips one more, et cetera. The default value for `to` is 0 so it can be left out if so desired.

filename The file to window.

lc Left context. Number of words on the left side of the target.

rc Right context. Number of words on the right side of the target.

to target offset. Shift the target position to the right. Setting this to one will skip one word, that is, a b -> d, c will be skipped.

The output filename will have the context appended to it, in the form `l3r0`. The `to` parameter will be appended if non-zero.

### 5.1.2 `window_letters`

Creates instances for a letter based classifier (for e.g word completion). Takes a normal text file as input. The input data can be treated as a continuous stream of words, or per line. In the latter case, the letter context is reset at the start of every new sentence. The target is the ‘current’ word at each point.

filename The file to window.

lc Left context. Number of letters on the left side of the target.

lm Lettering mode, 0 will treat the data as a continuous stream of words. 1 will reset every sentence.

The output filename will have the context appended to it, in the form `l3`. The `lm` parameter will be appended after the context.

### 5.1.3 `hapax`

Hapaxing translates all tokes with a frequency less than or equal to the `hpx` parameter to the string <`unk`>. For hapaxing we use a previously created lexicon on a *windowed* data set. The target (i.e the last token on each line of data) is not hapaxed.

filename The windowed data set to hapax.

lexicon The lexicon file to use.

hpx The frequency value.

hpx_sym The symbol to use for hapaxed tokens. Defaults to <`unk`>.

### 5.1.4 `lexicon`

Creates a lexicon file for the given text file. The lexcion consists of a list of tokens, followed by its frequency.

filename The data set to read.

It creates three output files, one with the suffix `.lex` containing the lexicon, and one with `.cnt` containing counts of counts. The first lines consists, for historical reasons, of three zeroes. The next line shows the number of tokens that occur once, followed by the number of tokens that occur twice, et cetera. Each line contains the number of occurences, followed by the frequency, followed by the number of occurences again. Only the first two columns are interesting, the third was part of a now abandoned smoothing experiment. The third file has the extension `.av` and contains a anagram value for each word. The value is the sum over each letters' ASCII value to the power 5. This allows the rapid searching of anagrams in a lexicon; the same letters will give the same value, regardless of the order of the letters.

```
alerting 114508940294
altering 114508940294
integral 114508940294
relating 114508940294
triangle 114508940294
```

### 5.1.5 `rfl`

This command outputs a range of words taken from a lexicon. The entries in the lexicon are sorted by frequency. The highest frequency comes first and has rank 1. Because we take ranks, a range of ten ranks can contain more than ten items as a number of words can have the same rank.

lexicon The lexicon file to process.

n Starting rank.

m Last rank.

### 5.1.6 `lcontext`

Create the global context part of the instances, and optionally merge them with a normal data set.

filename The normal data set or text file, depending on the value of the `gct` parameter.

range The file containing the words for the global context.

gcs Global context size

gcd Global context decay

gco Global context offset. Default to `0` which means that the global context starts directly after we have seen a word. This means that overlap between the normal and the global context is possible. The `gco` parameters introduces an extra offset of the specified number of instances.

gct Global context type. Type `0`, the default, creates a global context consisting of words. Type `1` creates binary features. The latter means a binary value for every word in the `range` list supplied. The `gcs` parameter is ignored in this case. Type `2` creates a 'hashed' value, that is, one value representing the words in the global context, in a position independent order.

fd Create from data (default `1`). Takes the words from a normal, windowed data set. Setting this to `0` creates global context from a normal text file.

id Identifier to append to the output filenames. If not specified, it defaults to the empty string.

gc_sep Defaults to `1`, a space. This space is inserted between the binary (type `1`) feature values. Setting it to `0` concatenates all the binary features into one binary string feature.

### 5.1.7 `gap`

A function to examine how words are distributed in a text. Can even be used to generate data for the *global context* data.

This function counts the number of tokens between the occurrences of the words we are examining. A `gap` parameter is used to determine whether words are 'close' together or not. Two output files are produced, one with suffix `gap` and one with suffix `gs`. The gap size is appended to the suffixes. The first file contains a list with words plus the gaps and a few statistics. Groups of small gaps are enclosed between parenthesis, giving an idea of how the words are clustered. The second file contains just the words and the statistics, and can optionally be filtered on several parameters.

An example of the contents of the first file.

Listing 5.1: `gap` output

```
wines 13 ( 50 92 ) 395 ( 100 38 28 37 16 77 10 21 15 )
                                [ 2 3 0.666667 11 1 0.916667 484 44 8.97727 ]
```

The example shows the token `wines`, which occurs 14 times in the text. The first three (note that we show *gaps*) are 50 and 92 tokens apart (a *close* group). Then there is a large gap of 395 tokens until we come across the next occurrence of `wines`. Then there is another group of 10 occurrences close together.

The first two numbers in the statistics show the number of close groups, and the number of potential groups. We have the two bracketed groups represented by the first number. There could have been three groups if the large gap in the middle had had an occurrence of `wines` within the gap distance, so that makes the total three. The number after that shows the ratio between those two numbers. The next numbers, 11 and 1, show we have 11 close gaps and only one large gap. The ratio between the number of close gaps and the total number of gaps is shown by the next number. Note that with 13 occurrences of the token, we only have 12 gaps. The number 484 is the sum of the close gaps values, which in its turn is followed by the average close gap distance. The last number is the ratio between the average large gap size and the average small gap size.

The second file contains the same information as the first one, but without the gaps.

Listing 5.2: `gs` output

```
wines 13 2 3 0.666667 11 1 0.916667 484 44 8.97727
```

This list can be filtered by setting the `filter` parameter to `1`. Only tokens satisfying the conditions will be added to the `gs` file. The `gap` file will still contain all the tokens from the specified lexicon.

**filename** A normal text file.

**lexicon** List of words to examine, in WOPRs lexicon format.

**gap** The maximum gap between tokens to consider them 'together'.

**filter** A boolean to specify if we want to filer the `.gs` output. If set, the following conditions must be met for a word to be included in the output.

min_f Minimum word frequency of words to include. Defaults to 0. The token frequency must be larger or equal to this.

max_f Maximum word frequency of words to include. Defaults to the maximum *long* value the computer can handle. The token frequency must be less than this.

min_r Minimum ratio between small and large gaps. Defaults to 0.5. The ratio must be larger or equal to this.

max_r Maximum ratio between small and large gaps. Defaults to 1.1. It can never be more than 1, but the default will make sure it will be included. The ratio has to be less that this parameter.

min_a Minimum average small gap value.

max_a Maximum average small gap value.

min_p Minimum groups to potential groups ratio.

max_p Maximum groups to potential groups ratio.

min_g Minimum average large to small groups size ratio.

max_g Maximum average large to small groups size ratio.

## 5.2 Training

### 5.2.1 `make_ibases`

filename Filename of the windowed data set.

timbl Settings for TIMBL. The parameters will be included in the name of the saved instance base file, with exception of the `k` parmeter which only affects processing a test set.

### 5.2.2 `ngl`

This creates a file with uni- to $n$-grams for the given text file. For each $n$-gram, the absolute frequency count and a conditional probability is calculated. The `fco` parameter is a frequency cut off value; only $n$-grams which occur more often than the specified `fco` parameter are included in the model.

filename The text file to read.

n The largest $n$-gram to include. Default is `3`.

fco The frequency cut-off value, defaults to `0`.

log Output the $log_2$ or $log_{10}$ of the probability. Only two and ten are allowed.

The output file will have the extension `.ngl3f0`, both the `n` and `fco` value are included. When specifying the `log` parameter, the log value is appended too (`.ngl3f0l10`).

## 5.3 Testing

### 5.3.1 pplxs

The `pplxs` function is geared towards word prediction. It takes an instance base, and calls TIMBL to process the test file. A number of word prediction related values are calculated and written to the output file.

It creates two output files, one with suffix `.px` and one with suffix `.pxs`. The `.px` output contains the following:

```
85 bp bp 0 0 1 cg k 1 1 1 11 1 [ bp 11 ]
bp over over 0 0 1 cg k 1 1 1 11 1 [ over 11 ]
over LIBOR Libor -16.4148 0 87364.5 ic k 1 1 1 11 0 [ Libor 11 ]
LIBOR delayed . -13.915 2.502 15452.2 ic k 1 1 8 44 0 [ . 14 , 11 for 6 ]
```

The first tokens contain the instance (in this case `l1r0`), followed by the classification. This is followed by the logprob of the classification, the entropy of the distribution and the word level perplexity ($\overset{p}{\omega}$). The logprob is the $log_2$ of the probability of the classification. The entropy of the distribution $D$ is calculated as follows:

$$H(p) = -\sum_{x \in D} p(x) \log_2 p(x) \tag{5.1}$$

The word level perplexity is defined as:

$$\overset{p}{\omega} = 2^{-p_{classification}} \tag{5.2}$$

These three numbers are followed by an indicator (`cg`, `cd` or `ic`), followed by an known/unknown word indicator (`k/u`). This is followed by the match-depth and a boolean value indicating matched-at-leaf (one or zero). See [Daelemans et al., 2009] for a more thorough explanation. The following three values are the distribution size, the sum of the distribution frequencies, and the *reciprocal rank* of the answer. Finally, the output contains the top-$n$ of the distributions, specified as a list of `token frequency` pairs.

The `.pxs` output contains a summary per line of input. It contains the sentence number, followed by the number of words in the sentence, the sum of the logprobs of the words, the average perplexity, and the average $\overset{p}{\omega}$. This is followed by the number of unknown words in the sentence, the sum of the logprobs of the known words and the standard deviation of the word level perplexities. The final element in the output is a list with the $\overset{p}{\omega}$ of each word in the sentence.

```
0 14 -113.718 278.73 30724.6 14 -113.718 75772.5 [ 594.087 105.375
                          181718 28.3925 392.646 116.143 1.61151 216331
                          2616.16 28.3925 28129.8 68.2397 2 11.6667 ]
```

filename Test file to process.

dir Instead of one file, a whole directory with files will be processed if the `dir` parameter is specified instead of the `filename` parameter.

dirmatch A regular expression which determines which files will be read from the directory specified with `dir`.

ibasefile Filename of the instance base.

timbl Settings used to create the instance base.

lexicon Lexicon to determine known/unknown words.

counts Counts of ranks for the lexicon. This can be used to introduce smoothed or changed probabilities to the perplexity calculations.

hapax Used when reading the lexicon.

lc Used to determine when a new sentence starts (for the `.pxs` output).

rc Used to determine when a new sentence starts (for the `.pxs` output).

topn Number of elements from the TIMBL distribution to include in the output. Most frequent one first.

cache Number of elements in the cache (the default of three is good).

cth Cache threshold: A distribution is not cached until it contains this many items.

is Include the whole sentence in the output (pxs).

log The base of the logarithm to do the calculations in. Defaults to two, can be set to ten with `log:10`.

id Identifier to append to the output filenames. If not specified, it defaults to the PID.

### 5.3.2 gt

The `gt` function is a general test routine; it calls TIMBL and writes the output to a file. The word prediction calculations are not performed. The output filename is the name of the testfile, followed by the `id` and the suffix `gt`.

The output contains the following:

```
million million cg 1 1 20 224 [ million 139 mln 30 billion 11 ]
```

The first token is the target, the second the classification from TIMBL. This is followed by an indicator (cg, cd, or ic), followed by two values which indicate the match-depth, and a boolean value specifying matched-at-leaf (one or zero). If the `topn` parameter had been specified, this is followed by the distribution count, the sum of the frequency of the elements in the distribution, and the top-$n$ from the distribution. The latter is a list with `token frequency` pairs.

filename Test file to process.

dir Instead of one file, a whole directory with files will be processed if the `dir` parameter is specified instead of the `filename` parameter.

dirmatch A regular expression which determines which files will be read from the directory specified with `dir`.

ibasefile Filename of the instance base.

timbl Settings used to create the instance base.

cache Number of elements in the cache (the default of three is good).

cth Cache threshold: A distribution is not cached until it contains this many items.

cs Size of the cache which caches test instances. Default is 10 000. Note that this is a different cache from the cache which caches distributions.

topn Number of elements from the TIMBL distribution to include in the output. Most frequent one first.

id Identifier to append to the output filenames. If not specified, it defaults to the PID.

### 5.3.3 `ngt`

Apply an $n$-gram model to the test set. It finds the longest matching $n$-gram from the supplied $n$-gram-model that fits.

filename Filename of the plain text file to apply the model to.

ngl The $n$-gram model.

ngc File with counts from SRILM, see the `mode` parameter.

mode Switch to another mode, for example `mode:srilm` to use a language model generated by SRILM. When specifying an SRILM language model, a counts file needs to be specified with `ngc`.

n Only $n$-grams up to and including $n$ are used.

id Identifier to append to the output filenames. If not specified, it defaults to the empty string.

topn Number of elements from the distribution to include in the output. Most frequent one first.

It creates three output files, one with the extension `.ngp3`, one with the extension `.ngp3`, and one with the extension `.ngd3`. The `3` refers to the $n$-parameter supplied. The `.ngt` output contains information per word in the text. The `.ngp` file contains statistics per line of text, and the `.ngd` file contains info about the distributions.

### 5.3.4 `correct`

Word correction based on contents of the distribution returned by the classifier.

- **filename** Test file to process.
- **dir** Instead of one file, a whole directory with files will be processed if the `dir` parameter is specified instead of the `filename` parameter.
- **dirmatch** A regular expression which determines which files will be read from the directory specified with `dir`.
- **ibasefile** Filename of the instance base.
- **timbl** Settings used to create the instance base.
- **lexicon** Lexicon to determine known/unknown words.
- **counts** Counts of ranks for the lexicon. This can be used to introduce smoothed or changed probabilities to the perplexity calculations.
- **mwl** Minimum word length (guess added if > mwl). Default 5.
- **mld** Maximum levenshtein distance (guess added if <= mld). Default 1. Setting this to a larger value will introduce many potential misspellings.
- **max_ent** Max entropy of the distribution (guess added if <= max_entropy). Default 5.
- **max_distr** Maximum size of the distribution (guess added if <= max_distr). Default 10.
- **min_ratio** Ratio of the target lexical frequency versus the frequency of the words in the distribution. This is geared towards the disambiguation of confusibles and correctly spelt words on levenshtein distance one (like `woman` versus `women`). Default 0 (ignored).
- **id** Identifier to append to the output filenames. If not specified, it defaults to the PID.

## 5.4 Miscellaneous

### 5.4.1 `generate`

- **start** Fill the starting context with this string. The default is to start generating from an empty context. It needs to contain as many words as specified in the `ws` parameter.
- **filename** Output filename prefix.
- **ibasefile** The instance base.
- **timbl** The corresponding TIMBL settings.
- **end** The end of sentence marker, defaults to a full stop.
- **ws** The size of the left and right context added together.
- **mode** A parameter to control if the next word taken from a number of possibilities is weighted according to frequency. Default is on (`1`). Setting it to `0` gives equal probability to each possibility.
- **sc** Show the frequency count of each word.

len The maximum length of each sentence if the end of sentence marker is not reached.

n The number of sentences to generate.

### 5.4.2 `pdt`

filename File to run the simulation on, plain text.

ibasefile The instance base.

timbl The corresponding TIMBL settings.

n The length of generated sequences.

ds The depth for each $n$.

mo If set with `mo:1`, writes minimal output to the output file. Only the predictions which have a match will be output.

### 5.4.3 `server4`

ibasefile The trained instance base.

timbl The corresponding TIMBL settings.

lexicon The corresponding lexicon.

port Port to listen on.

mode `mode:0`: input is an instance, `mode:1`: input is a sentence and will be windowed. In this case, see the `resm` parameter.

resm `resm:0`: return average, `resm:1`: return sum, `resm:2`: return average, no OOV words.

keep `keep:1` does not close connection (for PBMBMT).

moses `moses:1`: return a fixed width string for use in moses.

lb Log base of the answer, `lb:0`, returns a straight probability, `lb:10` returns the $log_{10}$.

lc Left context size for windowing.

rc Right context size for windowing.

verbose More output.

hpx Hapax level.

skm Remove sentence markers.

cs Size of the cache. Received elements are cached, previous result is returned directly. Only useful when keep:1 is set.

subsection`mbmt`

ibasefile The trained instance base.

timbl The corresponding TIMBL settings.

lexicon The corresponding lexicon.

port Port to listen on.

lb Log base of the answer, `lb:0`, returns a straight probability, `lb:10` returns the $log_{10}$.

lc Left context size for windowing.

rc Right context size for windowing.

verbose More output.

cs Size of the cache. Received elements are cached, previous result is returned directly. Only useful when keep:1 is set.

# Bibliography

[Daelemans et al., 2009] Daelemans, W., Zavrel, J., Van der Sloot, K., and Van den Bosch, A. (2009). TiMBL: Tilburg memory based learner, version 6.2, reference guide. Technical Report ILK 09-01, ILK Research Group, Tilburg University.