Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Diplomarbeit

Very large language models for machine translation

vorgelegt von

Christian Federmann

am 31.07.2007

angefertigt unter der Leitung von
Prof. Dr. Hans Uszkoreit, Saarland University/DFKI GmbH

betreut von
Dr. Andreas Eisele, Saarland University/DFKI GmbH

begutachtet von
Prof. Dr. Hans Uszkoreit, Saarland University/DFKI GmbH
Prof. Dr. Reinhard Wilhelm, Saarland University

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und alle verwendeten Quellen angegeben habe.

Saarbrücken, den 31. Juli 2007

Christian Federmann

Einverständniserklärung

Hiermit erkläre ich mich damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek der Fachrichtung Informatik aufgenommen wird.

Saarbrücken, den 31. Juli 2007

Christian Federmann

Abstract

Current state-of-the-art statistical machine translation relies on statistical language models which are based on n-grams and model language data using a Markov approach. The quality of the n-gram models depends on the n-gram order which is chosen when the model is trained. As machine translation is of increasing importance we have investigated extensions to improve language model quality.

This thesis will present a new type of language model which allows the integration of very large language models into the Moses MT framework. This approach creates an index from the complete n-gram data of a given language model and loads only this index data into memory. Actual n-gram data is retrieved dynamically from hard disk. The amount of memory that is required to store such an indexed language model can be controlled by the indexing parameters that are chosen to create the index data.

Further work done for this thesis included the creation of a standalone language model server. The current implementation of the Moses decoder is not able to keep language model data available in memory, instead it is forced to re-load this data each time the decoder application is started. Our new language model server moves language model handling into a dedicated process. This approach allows us to load n-gram data from a network or internet server and can also be used to export language model data to other applications using a simple communication protocol.

We conclude the thesis work by creating a very large language model out of the n-gram data contained within the Google 5-gram corpus released in 2006. Current limitations within the Moses MT framework hindered our evaluation efforts, hence no conclusive results can be reported. Instead further work and improvements to the Moses decoder have been identified to be required before the full potential of very large language models can be efficiently exploited.

Zusammenfassung

Der momentane Stand der Technik im Bereich der statistischen Maschinenübersetzung stützt sich unter anderem auf die Verwendung von statistischen Sprachmodellen. Diese basieren auf N-grammen und modellieren Sprachdaten mittels eines Markow Modells. Die Qualität eines solchen N-gram Modells hängt von der verwendeten Ordnung des Modells ab, die beim Training des Sprachmodells gewählt wurde. Da Maschinenübersetzung von wachsender Bedeutung ist, haben wir verschiedene Möglichkeiten zur Verbesserung der Qualität von statistischen Sprachmodellen untersucht.

Diese Diplomarbeit stellt eine neue Art von Sprachmodell vor, die es uns erlaubt, sehr große Sprachmodelle in das Moses MT Framework zu integrieren. Unser Ansatz erstellt zuerst einen Index der N-gram Daten eines gegebenen Sprachmodelles und lädt später nur diese Indexdaten in den Speicher. Die tatsächlichen N-gram Daten werden zur Laufzeit des Decoders dynamisch von der Festplatte in den Speicher geladen. Hierbei kann die Größe des Speichers, der für die Verwendung eines solchen indizierten Sprachmodells benötigt wird, gezielt durch entsprechend gewählte Indizierungsparameter kontrolliert werden.

Neben dem indizierten Sprachmodell beinhaltete diese Diplomarbeit ebenso die Erstellung eines unabhängigen Sprachmodellservers. Die gegenwärtige Implementierung des Moses Decoders ist nicht dazu in der Lage, die N-gram Daten eines Sprachmodells im Speicher zu halten. Stattdessen ist Moses gezwungen, diese Daten bei jedem Neustart erneut von der Festplatte zu laden. Unser neuer Sprachmodellserver verschiebt die Verarbeitung der Sprachmodelldaten in einen eigenen, dedizierten Prozess. Dieser Ansatz erlaubt es dann, jene Daten aus dem Netzwerk oder sogar von einem Server aus dem Internet zu laden und kann zudem verwendet werden, um die Sprachmodelldaten anderen Anwendungen zugänglich zu machen. Hierfür steht ein einfaches, text-basiertes Kommunikationsprotokoll bereit.

Abschließend haben wir versucht, ein gigantisches Sprachmodell aus den im letzten Jahr von Google veröffentlichten N-gram Daten des Google 5-gram Corpus zu generieren. Momentane Beschränkungen des Moses MT Frameworks behinderten unsere Evaluationsbemühungen, daher können keine abschließenden Ergebnisse angegeben werden. Stattdessen haben wir weitergehende Verbesserungsmöglichkeiten am Moses Decoder indentifiziert, die notwendig sind, bevor das gesamte Potenzial sehr großer Sprachmodelle ausgeschöpft werden kann.

Acknowledgements

I would like to express my gratitude to my supervisors Andreas Eisele and Hans Uszkoreit who provided me with a challenging and interesting topic for my diploma thesis. In the same way I want to thank Reinhard Wilhelm for his willingness to examine this thesis.

Andreas constant support and mentorship, his encouragement and continuous guidance have greatly contributed to the success of this work and are highly appreciated. Thanks a lot!

I also want to take the time to thank all those who have aided me in the creation of this thesis work. In alphabetical order, these are Stephan Busemann, Bertold Crysmann, Bernd Kiefer, Marc Schröder, and Hendrik Zender. I am also indebted to everyone at the Language Techonology lab at DFKI.

Last but not least, I want to thank all my family and friends who have supported me over the years and thus had an important share in the successful completion of this thesis. In random order, these would be my grandmother, my grandfather, my parents, my sister Maike, my brother Alexander, my girlfriend Kira and the whole crazy bunch of friends out there, you know who you are. Thank you very much, your help is greatly appreciated.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Statistical machine translation (SMT) has proven to be able to create usable translations given a sufficient amount of training data. As more and more training data has become available over the last years and as there exists an increasing demand for shallow translation systems statistical machine translation represents one of the most interesting and active topics in current research on natural language processing [Callison-Burch and Koehn, 2005], [Koehn et al., 2007].

A typical SMT system is divided into two core modules: the *translation model* and the *language model*. The translation model takes a given source sentence and creates possible translation hypotheses together with corresponding scores which describe the probability of each of the hypotheses with regard to the source sentence. Thereafter, the hypotheses are sent to a statistical language model which rates each of the possibilities and assigns an additional score describing the likeliness of the given hypothesis in natural language text. The weighted combination of these scores is then used to determine the most likely translation.

1.1.1 Statistical Language Models

The term *statistical language model* describes a family of language models which model natural languages using the statistical properties of n-grams. N-grams are sequences of single words. For these models it is assumed that each word depends only on a context of $(n-1)$ words instead of the full corpus. This **Markov model assumption** greatly simplifies the problem of language model training thus enabling us to use language modeling for statistical machine translation systems.

First applications of n-gram language models were developed in the field of automatic speech recognition (ASR) [Jelinek, 1999]. The general approach of statistical language modeling has the advantage that it is not bound to a certain application area, in fact it is possible to use the same language model for SMT, ASR or OCR [1].

There exist several known **limitations of n-gram models**, most notably they are not able to model long range dependencies as they can only explicitly support dependency ranges up to $(n-1)$ tokens. Furthermore Markov models have been criticized for not capturing the **performance/competence distinction** introduced by Noam Chomsky.

However empirical experiments have shown that statistical language modeling can be applied to create usable translations. While pure linguistic theory approaches usually only model language competence the statistical n-gram models also implicitly include language performance features. Hence whenever *real world applications* are developed the **pragmatic n-gram approach** is preferable.

The quality of n-gram language modeling can directly be improved by training the models on larger text corpora. This is a property of the statistical nature of the approach. As we stated above large amounts of training data have become available over the last years, data which can now be used to build better statistical language models. This thesis work will concentrate on improving statistical language models for SMT.

1.2 State-of-the-art Language Models

As we have argued before, it has become a lot easier to collect large amounts of monolingual training data for language model creation. This enables us to use fourgrams or even fivegrams instead of just trigrams. In order to show an improved translation quality using such higher order n-grams, we have trained a baseline MT system and evaluated the overall performance using three different English language models: a trigram model, a fourgram model, and a fivegram language model.

n-gram order	BLEU score	runtime [s]
3	37.14	111
4	39.14	136
5	39.7	147

Table 1.1: Influence of increasing n-gram order

Table 1.1 lists the **BLEU scores** [Papineni et al., 2002] and the **overall runtime** in seconds.

[1] Optical Character Recognition

All three language models have been trained from the same fraction of the Europarl corpus [Koehn, 2005], only the maximum n-gram order differed. The tests were performed using a set of 100 sentences.

We can observe an improved translation quality with increasing language model n-gram order. At the same time, performance decreases as decoding becomes more complex when n-grams of higher order are used. Considering the fact that the performance loss is not too severe and an improvement to the translation quality is very welcome, it seems to be a sound assumption that fivegrams should be considered state-of-the-art for statistical machine translation.

As current language models are generated from enormous amounts of training data they will also need large amounts of memory to be usable within current machine translation systems. The drawback of the SRI [Stolcke, 2002] language model implementation lies in the fact that all n-gram data has to be loaded into memory at the same time, even if only a fraction of all this data would be needed to translate the given source text.

Computer memory is becoming cheaper, however it remains an expensive resource, especially when compared to the low cost of fast hard disks. Therefore it seems to be a valuable effort to investigate new methods for language model handling which try to reduce memory usage. Instead of using a single machine with large amounts of memory to translate a given source text, it is perfectly possible to distribute this task to several machines which only translate parts of the source text.

With the standard SRI language model, all these machines would require the same large amount of memory to handle a large language model. If instead we could use another language model with reduced memory requirements even for large n-gram sets, all computers within our cluster could be equipped with less memory and would still be able to translate their share of the source text. As this cluster solution would redistribute the whole workload to several machines, it would be even be possible for the new language model to take more time to complete compared to the original SRILM implementation.

Often clusters of several dozens or hundreds of machines are already available in large companies or institutions. These could easily be used to translate a given source text with such a new language model, the only requirement would be an adaption to the actual amount of memory available on each of the cluster nodes. Our new language model would create an index on the full n-gram data set and only load the index data to memory, the size of this index could be controlled to fit the memory limitations within our cluster.

As we have shown above, fivegram language models help to create better translations. However they require more memory which could lead to problems once really large language models are built. As memory is an expensive resource, we propose a new, indexed language model instead which can be adapted to require less memory than the corresponding language model in SRI format.

1.3 Decoder Startup Times

The current implementation of the Moses decoder [Koehn et al., 2007] also suffers from slow startup times which are caused by language model loading. Even if the decoder would use the same language model for a certain amount of translation requests, it would still have to load the full n-gram data from hard disk each time it is started.

The following table shows how much of the actual decoder runtime is required to perform **startup** tasks which include the loading of language model data and phrase-table handling. It compares these values to the full decoder **runtime** and shows the procentual **runtime loss** caused by loading the language model. The tests were performed using the same set of 100 sentences as in section 1.2.

n-gram order	model size	startup [s]	runtime [s]	loss
3	90 MB	172	274	63%
4	145 MB	157	273	58%
5	185 MB	170	293	58%

Table 1.2: Performance loss introduced by language model loading

These results clearly show that the Moses MT framework could benefit from a new language model class which would only interact with a preloaded language model instead of loading all n-gram data again and again. Such a preloaded language model could be hosted by a dedicated server, access could be possible from the same machine or even from a network.

1.4 Thesis Goals

In this diploma thesis we will describe a new language model class within the Moses MT framework. This language model class differs from the well known SRI model as it does not load all n-gram data into memory but only an index of all n-grams contained within the model data. Actual n-gram data is loaded dynamically from hard disk. This aims to reduce memory requirements while trying to ensure full compatibility to the original SRI model.

We will implement the indexed language model as a new class inside the existing Moses MT framework. The Moses decoder was chosen as it is the current state of the art in machine translation. Even better, it is developed and maintained as an open source community project and can be easily extended.

Next to the new language model class, we will develop a second extension, a language model server, which can be used to host language model data using a dedicated server. As we have seen above, decoder startup times are slowed down by the loading of n-gram data. These could be reduced by the introduction of a language model server.

Another advantage lies in the fact that decoder and language model would not have to be located on the same machine anymore, the language model server could also be hosted on a local network or the internet. Finally, this could enable language model usage in other applications which are not built using the Moses MT framework.

1.5 Thesis Overview

The complete thesis is divided into 7 chapters which are described below:

- **Chapter 1: Introduction** motivates the development of a new indexed language model and a language model server application to be used with the Moses MT framework. These enable the usage of very large language models with the Moses decoder.
- **Chapter 2: Building A Baseline System** describes how a baseline translation system can be setup on top of the Moses MT framework. This includes information on how to compile the SRILM toolkit and the Moses code. Both language model training and phrase-table generation are discussed and examples how to translate texts and to evaluate these translations are given.
- **Chapter 3: N-gram Indexing** defines the notion of character-level n-gram prefixes. These can be used to efficiently index large sets of n-gram data and can hence be used for the indexed language model. Several indexing methods are discussed and compared with respect to compression rate and actual compression gain. Additionally, the implementation of the Indexer tool is described.
- **Chapter 4: An Indexed Language Model** shows the design and implementation of an indexed language model class within the Moses MT framework. Several possible data structures for the index data are presented and compared, integration into and interaction with the Moses decoder are discussed, and compatibility to the original SRILM implementation is evaluated.

- **Chapter 5: A Standalone Language Model Server** describes how we created a standalone language model server application which can be queried from the Moses decoder or other applications. Access to the server is possible using either TCP/IP connections or IPC shared memory methods. A simple server protocol is designed which can be used to lookup n-gram data.
- **Chapter 6: A Google 5-gram Language Model** shows how a very large language model can be created using the Google fivegram corpus which was released in late 2006. As training with the SRILM toolkit failed, only a basic language model could be created. Translation quality is evaluated and limitations are discussed.
- **Chapter 7: Conclusion** summarizes what we have learned while working on this diploma thesis. It describes what has been achieved and what has not been possible to do. Finally it discusses possible future extensions of the indexed language model and the language model server application.

Chapter 2

Building A Baseline System

2.1 Motivation

Not too long ago, statistical machine translation software was expensive, closed source and inflexible. The long time standard, the Pharaoh decoder [Koehn, 2004a] by Philipp Koehn, was only available in binary form and so even small modifications to the decoding system were not possible. Luckily, things have changed and improved in recent years.

With the introduction of the Moses decoder which was developed by a large group of volunteers lead by Philipp Koehn and Hieu Hoang [Koehn et al., 2007], a fully compatible MT system became available that was open for community modifications. This enables us to integrate new ideas into the decoding system and to try to create better translations.

However, before actually integrating support for very large language models into the Moses code, it is necessary to create a baseline system and evaluate performance and translation quality of this system. These values can then later be used to compare the newly created language model code to the current state of Moses decoder.

We will briefly discuss the steps which are needed to create such a baseline system on the following pages.

2.2 Requirements

The baseline system uses the **Moses decoder**, language models are created using the **SRILM toolkit**, word alignment during the training is done with **GIZA++**. All these can be freely downloaded from the internet and are robust, well-tested tools.

Our baseline system has been installed on a Linux machine with 32GB of RAM and four Dual Core AMD Opteron 885 CPUs. A second system has been installed on a 1.83 GHz MacBook with 2GB of RAM. It should also be possible to setup and train such a system on a Windows machine as all software is available for Windows as well, however this will not be explained in this document.

2.3 SRILM Toolkit

2.3.1 Description

The **SRILM toolkit** [Stolcke, 2002] allows to create and apply statistical language models for use in statistical machine translation, speech recognition and statistical tagging and segmentation. For machine translation, SRILM language models are currently the *gold standard*, support for them is already available inside the Moses MT system. The SRILM toolkit was designed and implemented by Andreas Stolcke.

2.3.2 Software

The **SRILM toolkit** can be obtained from the internet at:

`http://www.speech.sri.com/projects/srilm/`

The toolkit may *be downloaded free of charge under an* open source community license meaning that it can be used *freely for non-profit purposes.* A Mailing list for support and exchange of information between SRILM users is available through:

`srilm-user@speech.sri.com`

2.3.3 Installation

Assuming the downloaded SRILM archive file is named `srilm.tar.gz` it can be installed as follows:

```
$ export $BASELINE=/home/cfedermann/diploma
$ mkdir $BASELINE/srilm
$ cp srilm.tar.gz $BASELINE/srilm
$ cd $BASELINE/srilm
$ tar xzf srilm.tar.gz
```

This will create a folder named `srilm` containing the SRILM code inside the $BASELINE folder. In order to compile this code, the `SRILM` variable inside `$BASELINE/srilm/Makefile` has to be configured.

```
SRILM = /home/cfedermann/diploma/srilm
```

It is also necessary to disable TCL usage and any special optimizations (like `-mtune=pentium3`) inside the Makefile for the specific target machine. For example, if we try to build the SRILM toolkit on Mac OS X, we change `$BASELINE/srilm/common/Makefile.machine.macosx`. We add the following line to the Makefile and remove the existing `TCL_INCLUDE` and `TCL_LIBRARY` definitions if available.

```
NO_TCL = X
# TCL_INCLUDE =
# TCL_LIBRARY =
```

Now, the SRILM code can be compiled and installed:

```
$ cd $SRILM
$ make World
```

It is crucial to verify that the compilation process worked correctly otherwise the baseline system will not function properly. We can check this by calling the tests inside the `$BASELINE/srilm/test` folder:

```
$ cd $SRILM/test
$ make all
```

In case of errors or any other problems during the installation process, there exists more detailed documentation in the file `INSTALL` inside the SRILM folder. It is recommended to add the SRILM binary folder, e.g. `$SRILM/bin/macosx`, to the global `$PATH` variable.

2.3.4 Usage

The most important command of the SRILM toolkit is the `ngram-count` tool which counts n-grams and estimates language models. There exist several command line switches to fine-tune the resulting language model, we will explain only some of them here. For more information refer to the respective man page:

```
$ man ngram-count
```

A sorted 5-gram language model from a given English corpus in `en.corpus` can be created using the following command:

```
$ ngram-count -sort -order 5 -interpolate -kndiscount\
  -text en.corpus -lm en.srilm
```

The command line switches are explained below:

- `-sort` outputs n-gram counts in lexicographic order. This can be required for other tools within the SRILM toolkit and is mandatory for the indexed language model that will be presented later in this thesis.
- `-order n` sets the maximal order (or length) of n-grams to count. This also determines the order of the language model.
- `-interpolate` causes the discounted n-gram probability estimates at the specified order n to be interpolated with estimates of lower order.
- `-kndiscount` activates Chen and Goodman's modified Kneser-Ney discounting for n-grams.
- `-text` specifies the source file from which the language model data is estimated. This file should contain one sentence per line, empty lines are ignored.
- `-lm` specifies the target file to which the language model data is written.

2.4 GIZA++ & mkcls

2.4.1 Description

GIZA++ [Och and Ney, 2003] is a tool to compute word alignments between two sentence aligned corpora, **mkcls** [Och, 1999] is a tool to train word classes using a maximum-likelihood criterion. Both tools were designed and implemented by Franz Josef Och.

2.4.2 Software

The original version of GIZA++ and mkcls can be downloaded at:

```
http://www.fjoch.com/GIZA++.html
http://www.fjoch.com/mkcls.html
```

These versions of GIZA++ and mkcls will not compile with newer **g++ 4.x** compilers which are standard on modern computer systems. There exist patched versions by Chris Dyer which resolve these issues. They are available from:

```
http://ling.umd.edu/~redpony/software/
```

2.4.3 Installation

Assuming the GIZA++ and mkcls archive files are named `GIZA++.tar.gz` and `mkcls.tar.gz`, they can be installed like this:

```
$ export $BASELINE=/home/cfedermann/diploma
$ cd $BASELINE
$ tar xzf GIZA++.tar.gz
$ tar xzf mkcls.tar.gz
```

This will create two folders `GIZA++-v2` and `mkcls-v2` inside the `$BASELINE` folder. In order to compile the code for mkcls, it is just necessary to type:

```
$ cd $BASELINE/mkcls-v2
$ make
```

For GIZA++, a small change is required. The compiler flag `-DBINARY_SEARCH_FOR_TTABLE` has to be added to the `CFLAGS` definition inside the original Makefile:

```
CFLAGS = $(CFLAGS_GLOBAL) -Wall -W -Wno-deprecated\
         -DBINARY_SEARCH_FOR_TTABLE
```

GIZA++ and its accompanying tools can then be compiled using:

```
$ cd $BASELINE/GIZA++-v2
$ make opt snt2cooc.out
```

The resulting binaries are called `mkcls`, `GIZA++`, `snt2plain.out`, `plain2snt.out`, and `snt2cooc.out`. In order to make life a little easier, these will be copied to a central `tools` folder which is placed inside the `$BASELINE` folder:

```
$ export $TOOLS=$BASELINE/tools
$ cd $BASELINE/GIZA++-v2
$ cp GIZA++ *.out $TOOLS
$ cp $BASELINE/mkcls-v2/mkcls $TOOLS
```

2.4.4 Usage

Both GIZA++ and mkcls will be called by Moses training scripts, you should not have to invoke them yourself. Documentation is available in the corresponding `README` files.

2.5 Moses Decoder

2.5.1 Description

Moses is a statistical machine translation system which allows to train translation models for any given language pair for which a parallel corpus, i.e. a collection of translated texts, exists. The Moses decoder works using a **beam search** [Koehn, 2004a] algorithm to determine the best translation for a given input. Translation is **phrase-based** [Koehn et al., 2003] and allows words to have a **factored** representation. Moses has been designed by a team headed by Philipp Koehn, implementation was mainly done by Hieu Hoang.

2.5.2 Software

The Moses source code can be obtained from the project website or the SourceForge Subversion repository. The latest development version of the source code can be checked out using the following command:

```
$ svn co https://mosesdecoder.svn.sourceforge.net/svnroot/\
  mosesdecoder/trunk mosesdecoder
```

Stable releases of the software can be downloaded at:

```
http://mosesdecoder.sourceforge.net/download.php
```

2.5.3 Installation

Assuming that the Moses source code is available in the `$BASELINE` folder, inside a subfolder `mosesdecoder`, it can be configured and compiled using the following commands. Please note that we have to specify the `--with-srilm` switch to enable SRILM language model usage:

```
$ cd $BASELINE/mosesdecoder
$ ./regenerate-makefiles.sh
$ ./configure --with-srilm=$BASELINE/srilm
$ make -j 4
```

After compilation has finished, the `moses` binary should be copied to the `tools` folder:

```
$ cp $BASELINE/mosesdecoder/moses-cmd/src/moses $TOOLS
```

Moses includes a set of support tools which should be put inside a `scripts` folder inside the `tools` folder. To correctly configure the path settings for the scrips, we have to edit the file `mosesdecoder/scripts/Makefile`:

```
TARGETDIR=$(TOOLS)/scripts
BINDIR=$(TOOLS)
```

The support tools can be generated using:

```
$ cd $BASELINE/mosesdecoder/scripts
$ make release
```

This will generate (yet) another subfolder `scripts-YYYYMMDD-HHMM` inside `$TOOLS/scripts` containing the current versions of the Moses scripts. To enable Moses to use them, the `SCRIPTS_ROOTDIR` variable has to be exported and set:

```
$ export SCRIPTS=$TOOLS/scripts
$ export SCRIPTS_ROOTDIR=$SCRIPTS/scripts-YYYYMMDD-HHMM
```

2.5.4 Usage

The Moses decoder relies on a fully trained translation model for a given language pair. Before we will explain the steps necessary to create such a model, we will show the basic usage of the decoder. Assuming we have our Moses configuration file available in `moses.ini` and want to translate `source.text`, the corresponding call to Moses looks like this:

```
$ moses -config moses.ini -input-file source.text
```

2.5.5 Additional Requirements

Moses requires some additional tools for the training and evaluation processes. For training, a `tokenizer`, and a `lowercaser` are necessary. These tools can be obtained from the website of the 2007 ACL Workshop on Statistical Machine Translation at:

```
http://www.statmt.org/wmt07/scripts.tgz
```

Evaluation is done using the notion of BLEU [Papineni et al., 2002] scores. An appropriate tool for this is the NIST BLEU scoring tool which is available here:

```
ftp://jaguar.ncsl.nist.gov/mt/resources/mteval-v11b.pl
```

2.6 Basic Training

2.6.1 Preparational Steps

Assume we want to train a translation model for German to English. The first step is to prepare the parallel corpus data. It has to be tokenized, lowercased and sentences which would be too long to handle (and their correspondences in the other language) have to be removed from the corpus.

Corpus preparation can be done as follows:

```
$ tokenizer.perl -l de < corpus.de > tokens.de
$ tokenizer.perl -l en < corpus.en > tokens.en
$ clean-corpus-n.perl tokens de en clean-corpus 1 40
$ lowercase.perl < clean-corpus.de > lowercased.de
$ lowercase.perl < clean-corpus.en > lowercased.en
```

2.6.2 Training Step

Once the corpus data is prepared, the actual training process can be started. Moses offers a very helpful training script which is explained in more detail on the Moses website:

```
http://www.statmt.org/moses/?n=FactoredTraining.HomePage
```

In a nutshell, translation model training is done using:

```
$ train-factored-phrase-model.perl PARAMETERS
```

The training process takes **a lot** of time and memory to complete. Actual settings for the PARAMETERS are discussed on the Moses website.

2.7 Minimum Error Rate Training

It is possible to perform an optimization on the scoring weights which are used to produce translations. This is called *minimum error rate training* (MERT) [Och, 2003] and works by iteratively optimizing scoring weights for a test document which is translated and then compared to a reference translation. Assuming we want to optimize weights using the files `tuning.de` and `tuning.en` and a trained translation model from German to English is available in the baseline folder, a MERT can be performed using:

```
$ tokenizer.perl -l de < tuning.de > input.tokens
$ tokenizer.perl -l en < tuning.en > reference.tokens
$ lowercase.perl < input.tokens > input
$ lowercase.perl < reference.tokens > reference
$ mert-moses.pl input reference moses moses.ini\
  --working-dir $BASELINE/tuning --rootdir $SCRIPTS
```

This will produce optimized weights for the given tuning document and store results inside a `tuning` subfolder in the baseline folder. As with the basic training script, the `mert-moses.pl` script will take a long time to finish. The final step is to insert the new weights into the basic `moses.ini`:

```
$ reuse-weights.perl tuning/moses.ini < moses.ini\
  > tuning/optimized-moses.ini
```

This results in an optimized version of the original `moses.ini` which is stored in a new configuration file `optimized-moses.ini` inside the `tuning` folder.

2.8 Evaluation

Evaluation is done using the NIST BLEU scoring tool. Assuming a reference translation `evaluation.en` exists, we can evaluate the translation quality of `output.en` which was translated from `evaluation.de` as follows:

```
$ mteval-v11b.pl -r evaluation.en -t output.en -s evaluation.de
```

The different command line switches have the following meanings:

- ▷ `-r` denotes the reference translation
- ▷ `-t` denotes the translation output
- ▷ `-s` denotes the translation input

It is also possible to use the `multi-bleu.perl` script to evaluate the translation quality. This is easier to use as it does not require to wrap the translation files into SGML tags. Our example would look like this if `multi-bleu.perl` was used:

```
$ multi-bleu.perl evaluation.en < output.en
```

2.9 Summary

In this chapter we have shown how to **setup a baseline** system for statistical machine translation based upon open source software which is available at no charge. The system is built using the **SRILM** toolkit for language modeling and **GIZA++** for both sentence alignment and phrase-table generation. Sentence decoding is done with the **Moses** decoder, evaluation can be performed using several **BLEU** scoring tools.

Chapter 3

N-gram Indexing

3.1 Motivation

Whenever statistical machine translation is done this involves statistical language models of the target language [Brown et al., 1993]. These models are used to rate the quality of intermediate translation results and help to determine the best possible translation for a given sentence. Current language models are often generated using the SRILM toolkit [Stolcke, 2002] which was designed and implemented by Andreas Stolcke.

The standard file format for these models is called *ARPA or Doug Paul format for N-gram backoff models*. Informally spoken, a file in ARPA format contains lots of n-grams, one per line, each annotated with the corresponding conditional probability and (if available) a backoff weight. A single line within such a file might look like this:

```
-0.4543365      twilight zone .
```

or like this, if a backoff weight is present:

```
-2.93419        the big       -0.5056427
```

In recent years, the creation of high quality language models resulted in files up to 150 MB in size. Often these models only contained n-grams up to a maximum order of 3, i.e. trigrams. Such models could easily be stored inside the memory of a single computer, access was realized by simple lookup of n-gram sequences from memory.

Nowadays things have changed. Due to an enormous amount of available source data, it is now possible to create larger and larger language models which could help to improve translation quality. These **huge language models** [Brants et al., 2007] will require a more clever way to handle n-gram access as several GB of memory could be necessary to store the full language model.

If not all of these n-grams are actually used for a specific translation task, parts of the allocated memory are just blocked without purpose and thus wasted. However, even if all superfluous n-grams inside a given large language model would be filtered out *a priori*, it is still very likely that we will experience problems with the amount of memory which is required to store the full language model.

3.1.1 Possible Solutions

There exist several possible solutions to integrate support for very large language models into the existing frameworks for statistical machine translation. For instance, it might be interesting to use a hash function based approach to reduce the n-gram data size requirements. It is also possible to think of a method which loads only very frequent n-grams into memory and handles rare n-grams by hard disk access.

For this thesis, we decided to investigate a different approach which we will call **character-level n-gram indexing**. This technique has been chosen because it is easy to understand and as it allows a very fine-grained control on the amount of n-gram data which has to be loaded into memory. We will describe and define this approach on the following pages.

3.1.2 Character-level N-gram Indexing

Instead of loading the full n-gram data into the computer's memory, this data is indexed using some (hopefully) clever indexing method and only the resulting index data is stored in memory. That way only a small fraction of the original data has to be handled *online*, the vast majority of the n-gram data will remain *offline* and can be handled *on demand*. As the name implies, we will create the index data based on character-level n-gram prefixes.

It is quite clear that all n-gram computations then require lookup of the respective n-grams from hard disk. This will take more time than equivalent lookup operations from memory but otherwise it would not be possible to utilize a large language model at all once the computer cannot provide enough dedicated memory. Hard disk access can also be reduced by caching of already processed n-grams and by using a good indexing method which minimizes the average number of n-grams to read from hard disk for any given index key.

3.1.3 Definition: N-gram Prefix

We define n-gram indexing using the notion of so called **n-gram prefixes**. Given an arbitrary n-gram $w_1 w_2 ... w_n$ and a set of parameters $\{\Gamma_1, \Gamma_2, ..., \Gamma_n\}$, $\Gamma_i \in \mathbb{N}_0^+$, the character-level n-gram prefix set is computed as follows:

$$Key_i = w_i[0 : \Gamma_i] \tag{3.1}$$

$$Prefix_n = \{Key_1, Key_2, ..., Key_n\} \tag{3.2}$$

where $w[0 : m]$ denotes the m-prefix of a given string w. Thus the indexing method takes the first Γ_i characters for each of the words w_i which create the full n-gram and creates the index key set out of them. A parameter value of 0 will give an empty index key ϵ, any parameter value $\Gamma_i > length(w_i)$ will return the full word w_i as index key.

This is what we formally define as the **character-level n-gram prefix**. Any indexing method of this family can be uniquely defined by the corresponding set of parameters $\{\Gamma_1, \Gamma_2, ..., \Gamma_n\}$. It is convenient to represent the complete n-gram prefix set as a string with spaces inserted as delimiters.

3.1.4 An Indexing Example

An Example should make the general idea clear. Assume we want to index the 5-gram "`the big ogre wants food`" using n-gram prefixes. Given a uniform weighting of $\Gamma_1 = ... = \Gamma_5$, the character-level n-gram prefixes for this n-gram, ordered by increasing parameters Γ_i, would look like this:

Γ_i	character-level n-gram prefix
1	`"t b o w f"`
2	`"th bi og wa fo"`
3	`"the big ogr wan foo"`
4	`"the big ogre want food"`
5	`"the big ogre wants food"`

Table 3.1: Character-level n-gram prefixes example

When applied onto an actual language model file, each of the n-gram prefixes represents exactly the subset of n-grams which match the respective n-gram prefix. Each subset can be uniquely determined by the position of the first matching n-gram line within the original language model file and the total number of matching n-grams. Figure 3.1 illustrates the basic relationship between n-gram prefixes and n-gram subsets:

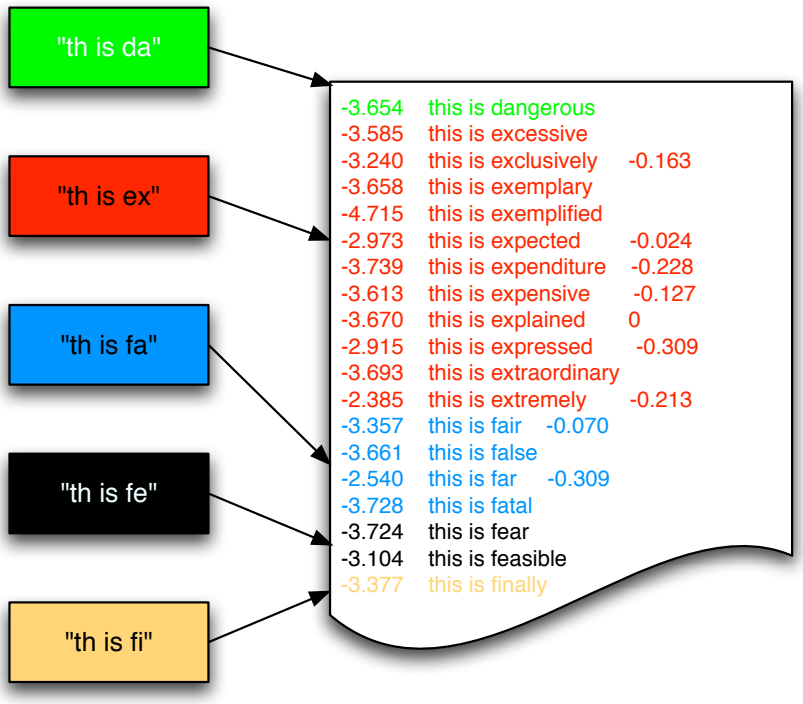

Figure 3.1: Relation between n-gram prefixes and n-grams in a language model file

To index a language model using n-gram prefixes, the file position and the n-gram line count are stored as the index value for each of the n-gram prefixes. The n-gram prefixes themselves serve as index keys. In practice that might look similar to this:

```
"th is fe" → "393815:2"
```

The above representation encodes that the n-gram prefix "th is fe" occurs at position 393,815 in the corresponding language model file and is valid for the following 2 lines of content.

Whenever we encounter an n-gram with this n-gram prefix and want to lookup the corresponding n-gram data, we would have to jump to position 393,815 inside the respective language model file, read in the following two lines and check whether the given n-gram is contained in any of them. If that is the case, we have found our n-gram and can return its conditional probability and backoff weight. If not, the n-gram is unknown as it is not contained within the language model data.

3.2 Basic Algorithm

The basic algorithm for language model indexing based on character-level n-gram prefixes is explained on the next page. In fact, it is quite simple: it requires a given language model in ARPA format and a set of parameters which will be used for computing the index keys of the n-gram prefix for each n-gram inside the language model.

Algorithm 1 character-level n-gram indexer

Require: language model file LM in ARPA format,
set of parameter values $\Gamma = \{\Gamma_1, \Gamma_2, ..., \Gamma_n\}$

1: index = $\emptyset$
2: **while** $\exists$ some n-gram in LM **do**
3: current-ngram $\leftarrow LM$
4: index = index $\cup$ NGRAM-PREFIX(current-ngram, Γ)
5: **end while**
6: **return** index

The pseudo-code does the following:

- ▷ **line 1** initializes the index set.
- ▷ **lines 2-5** represent the main loop which iterates over all n-grams.
- ▷ **line 3** reads the current ngram from the language model file.
- ▷ **line 4** computes the n-gram prefix for this n-gram and adds the result to the index set.
- ▷ **line 6** finally returns the index set as result of the algorithm

The most important part of the algorithm is **the choice of the actual indexing method** used inside NGRAM-PREFIX, i.e. the choice of the set of parameters Γ. We will describe and define several possible indexing methods in the next section.

3.3 Indexing Methods

3.3.1 Increasing Indexing

This indexing method creates increasingly larger index keys Key_i for the words w_i of a given n-gram $w_1\ w_2\ ...\ w_n$:

$$\Gamma_1 = 1 \tag{3.3}$$
$$\Gamma = \{\Gamma_1, \Gamma_2, ..., \Gamma_n \quad | \quad \forall i > 1 : \Gamma_i = \Gamma_{i-1} + 1\} \tag{3.4}$$

For our "`the big ogre wants food`" example this would yield the n-gram prefix:

Γ	character-level n-gram prefix
`[1, 2, 3, 4, 5]`	`"t bi ogr want food"`

Table 3.2: Increasing Indexing example

This method could be optimized by setting some maximum index position i_{max} after which all $\Gamma_i,\ i\ >\ i_{max}$ are clipped and set to 0 so that they can be neglected when constructing the index.

3.3.2 Decreasing Indexing

Decreasing Indexing effectively means *reverse increasing indexing*. Here the first index key is the largest and all subsequent index keys are of decreasing size.

$$\Gamma_1 = \Gamma_{max}, \quad \Gamma_{max} \in \mathbb{N}^+ \tag{3.5}$$
$$\Gamma = \{\Gamma_1, \Gamma_2, ..., \Gamma_n \quad | \quad \forall i > 1 : \Gamma_i = \Gamma_{i-1} - 1, \quad \Gamma_i \in \mathbb{N}_0^+\} \tag{3.6}$$

Assuming $\Gamma_{max} = 4$, the n-gram prefix for the example sentence changes to:

Γ	character-level n-gram prefix
`[4, 3, 2, 1, 0]`	`"the big og w` ϵ`"`

Table 3.3: Decreasing Indexing example

Depending on the actual implementation of the indexing method, the *empty* index key ϵ can either be represented as a blank space `" "` or simply be left out of the index key.

3.3.3 Uniform Indexing

This is the easiest possible indexing method. It assumes uniform parameters $\Gamma_1 = \Gamma_2 = ... = \Gamma_n$ which means that all index keys Key_i are of equal size:

$$\Gamma_1 = \Gamma_{max}, \quad \Gamma_{max} \in \mathbb{N}^+ \tag{3.7}$$

$$\Gamma = \{\Gamma_1, \Gamma_2, ..., \Gamma_n \quad | \quad \forall i > 1 : \Gamma_i = \Gamma_{i-1}\} \tag{3.8}$$

Given $\Gamma_{max} = 3$, the character-level n-gram prefix for our example looks like this if we apply uniform indexing:

Γ	character-level n-gram prefix
`[3, 3, 3, 3, 3]`	`"the big ogr wan foo"`

Table 3.4: Uniform Indexing example

Again, it may be clever to define a maximum index position i_{max} to decrease index size and to reduce index construction time.

3.3.4 Custom Indexing

It is also possible to design a *custom* indexing method which simply relies on the given set of Γ_i parameter values and does not assume any relation between these. Formally:

$$\Gamma = \{\Gamma_1, \Gamma_2, ..., \Gamma_n \quad | \quad \forall i \geq 1 : \Gamma_i \in \mathbb{N}_0^+\} \tag{3.9}$$

Example:

Γ	character-level n-gram prefix
`[3, 2, 0, 2, 1]`	`"the bi` ϵ `wa f"`

Table 3.5: Custom Indexing example

Depending on the actual application of the indexing method, such a custom indexing approach might be a better choice than the three methods defined above as the custom Γ_i parameters give a more fine-grained control on the creation of the index data. However it is important to **take care of the empty index key** ϵ. The decision to represent it as a blank space `" "` or to simply leave it out of the n-gram prefix has to be taken based on application needs.

3.4 Evaluation

For evaluation of these indexing methods, we will define the notions of **compression rate**, **large subset rate** and **compression gain**. These can be used to effectively rate any possible indexing method.

3.4.1 Definition: Compression Rate

$$CompressionRate = 1 - \frac{SIZE(index\ set)}{SIZE(ngram\ set)} \tag{3.10}$$

The compression rate compares the size of the generated index data with the size of the original n-gram collection and can be used to determine the *compression factor* of the indexing operation. A value of 0 represents no compression, any larger value represents some actual compression of the n-gram data.

3.4.2 Definition: Large Subset

Each index entry refers to a certain part of the original language model, *a certain subset*. In order to access this data, the corresponding lines within the language model have to be looked up from hard disk. As hard disk access is relatively slow, it makes sense to keep the average subsets small and to prevent the creation of very large n-gram subsets.

For evaluation, we define a certain threshold, the so-called **subset threshold** which divides the n-gram data subsets into *small* and *large subsets*.

3.4.3 Definition: Large Subset Rate

$$LargeSubsetRate = \frac{COUNT(large\ subsets)}{SIZE(ngram\ set)} \tag{3.11}$$

The large subset rate describes the ratio between the line count of all *large subsets* and the size of the original n-gram collection. Effectively, this is the part of the language model which is *considered too costly to process*. Hence a good indexing method should always try to minimize the large subset rate.

3.4.4 Definition: Compression Gain

$$CompressionGain \quad = \quad CompressionRate - LargeSubsetRate \tag{3.12}$$

Compression gain represents a measure to rate the quality of an indexing method. It compares the compression rate and the large subset rate. The first one is optimal if the index is very small, the second one is optimal if the number of large subsets is very small. As these two values are diametrically opposed, their difference represents the amount of language model data which does not have to be handled *online* and can be *efficiently processed*. Effectively, this is the amount of saving, the indexing method has achieved.

3.4.5 Evaluation Concept

An optimal indexing method should minimize both the size of the generated index (which maximizes the compression rate) and the line count of all *large subsets* (which maximizes the compression gain).

3.4.6 Evaluation Results

Indexing methods have been evaluated using an English language model containing 10,037,283 n-grams which was about 324MB in size. The maximum n-gram order was 5. The results showed that:

- ▷ increasing indexing yields good and robust compression rates and relatively good large subset values. Indexing up to 3 words seemed to suffice even for the 5-gram model.
- ▷ decreasing indexing gives the best compression rates at the cost of more large subsets. Again, indexing only the first 3 words of each n-gram created the best results which were even better than the corresponding results from increasing indexing.
- ▷ uniform indexing constructs large index sets which results in a bad compression rate. Small Γ values seem to work best, but compared to the other indexing methods, uniform indexing does not perform too well.
- ▷ custom indexing heavily relies on the chosen set of Γ_i parameters. While some of the custom results were remarkably good, others performed outstandingly bad.

More detailed information on the evaluation results is available in the following figures. The x-axis represents the subset threshold, the y-axis shows the compression gain. The full evaluation results are available in Appendix D.

Figure 3.2: Increasing indexing: compression gain (y) for increasing subset threshold (x)

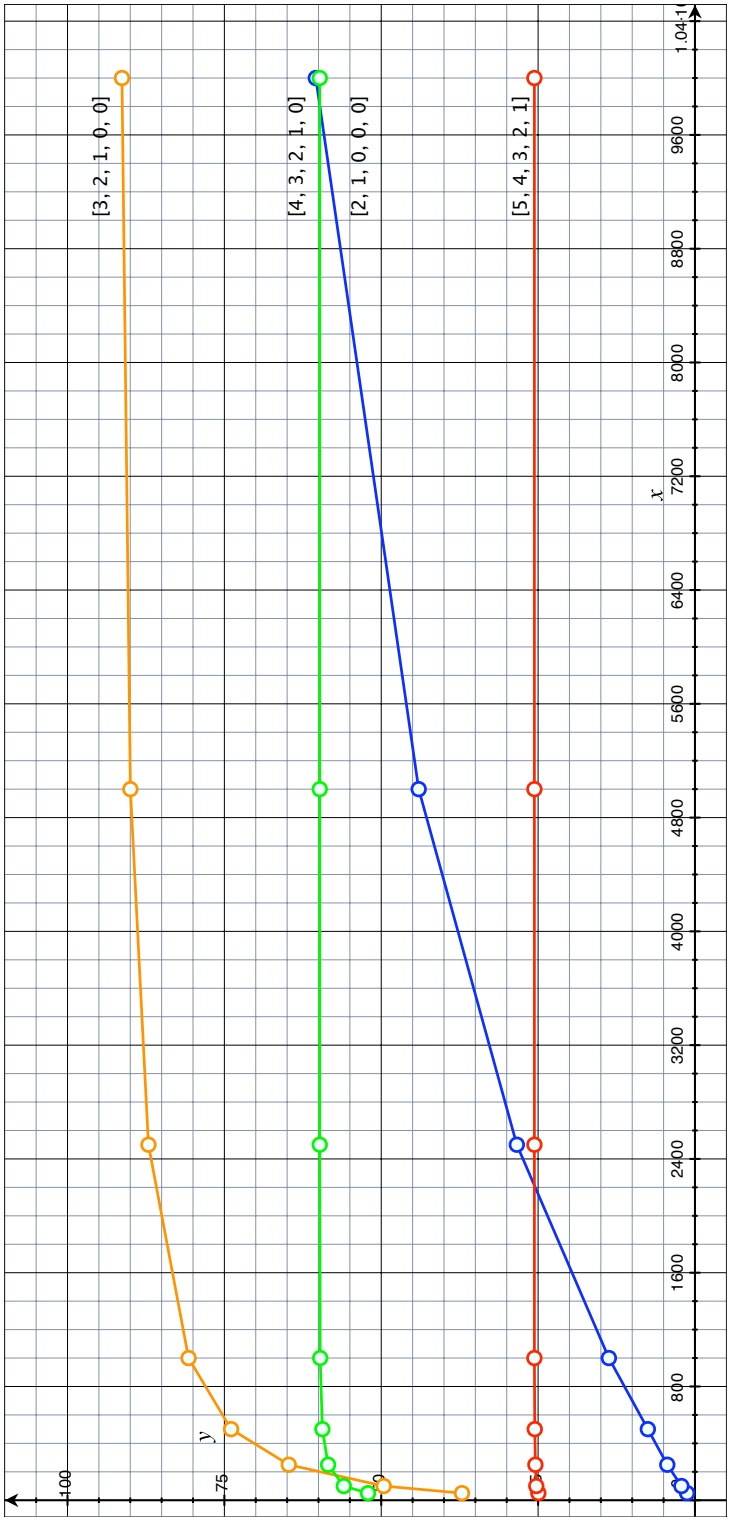

Figure 3.3: Decreasing indexing: compression gain (y) for increasing subset threshold (x)

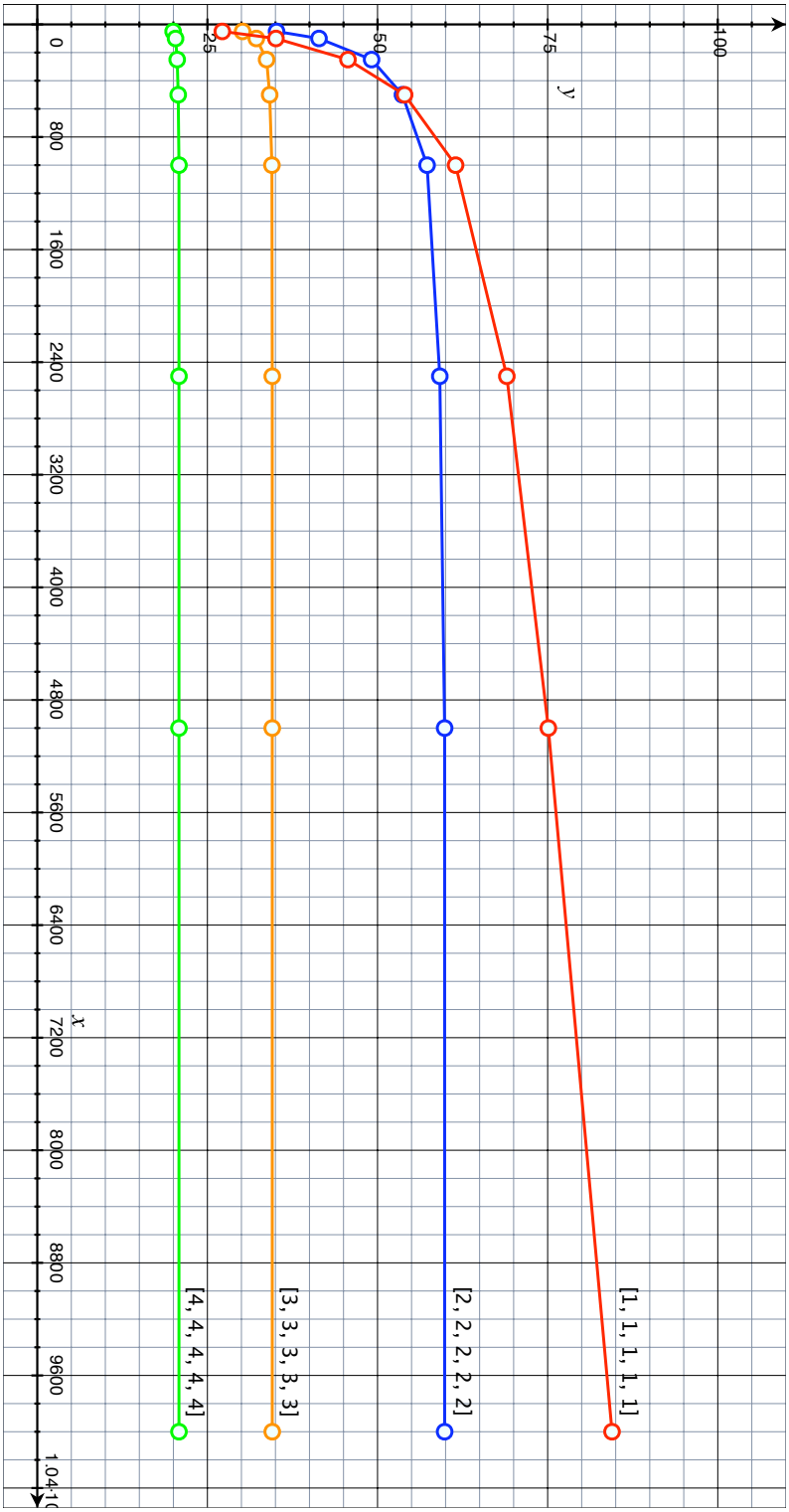

Figure 3.4: Uniform indexing: compression gain (y) for increasing subset threshold (x)

3.5 Indexer Tool

The **indexer** tool allows to create a character-level n-gram index for one or several language models in ARPA format. Furthermore it creates a unigram vocabulary which contains all unigrams included within the language model files. The Indexer tool provides the following command line options:

```
-l or --lm=<file #1>,[<file #2>,...,<file #n>]
-i or --index=<index file>
-v or --vocab=<vocabulary file>
-m or --method={increasing | decreasing | uniform | custom}
-g or --gamma=<gamma #1>[,<gamma #2>,...,<gamma #n>]
-n or --no-autoflush
-f or --file_id=<start_id>
```

These options are explained below:

- ▷ **Language Model parameter:** the Indexer tool works on language model files in ARPA format. These files can be specified using the `--lm` command line switch. It is possible to specify only a single language model `--lm=<file #1>` or multiple files `--lm=<file #1>,<file #2>,...,<file #n>`.

- ▷ **Index parameter:** all index data is written to the index file. The corresponding file name is specified using the `--index=<index file>` switch.

- ▷ **Vocabulary parameter:** the unigram vocabulary data is written to the vocabulary file which is specified using the `--vocab=<vocabulary file>` switch.

- ▷ **Method parameter:** this parameter allows to choose the actual indexing method which should be used to generate the index data from the language model file(s). The Indexer tool supports increased indexing for `--method=increasing`, decreasing indexing using `--method=decreasing`, uniform indexing with `--method=uniform` and custom indexing given `--method=custom`.

  For the first three of these it is only necessary to specify a **single Γ value** which is then interpreted as Γ_1 or as Γ_{max} depending on the actual choice of the indexing method. For custom indexing, a full set of Γ_i parameters has to be set using the `--gamma` switch.

- ▷ **Gamma parameter:** each indexing method needs some Γ_i parameters to be properly defined. These values can be specified using the `--gamma` switch. An increasing indexing method with $\Gamma_{max} = 3$ is set using the `--method=increasing --gamma=3` switch.

  Please note that the Indexer tool **assumes custom indexing** if the method parameter is not set. E.g. `--gamma=3` would result in custom indexing with $\Gamma = [3, 0, 0, 0, 0]$.

- **Autoflush parameter:** the `--no-autoflush` command line switch can be used to disable the *autoflush behaviour* of the Indexer tool. If set, all index and vocabulary data is collected in memory before it is finally written to the corresponding files on disk. This requires more memory to complete but it ensures that no duplicate index keys are available inside the index data.
- **File id parameter:** the Indexer tool assigns a numeric id to each of the given language model files, starting at 0. In order to allow merging of index files, it is also possible to manually define this *start id* using the `--file-id=<start-id>` switch.

The Indexer tool has been implemented using `C++`. The source code is freely available at `http://www.cfedermann.de/diploma` under the license terms printed on page 73.

3.6 File Formats

The n-gram Indexer tool generates both n-gram indexes and (unigram) vocabularies. The corresponding file formats are explained below:

3.6.1 Index Data Format

The index file contains character-level n-gram prefixes for unigrams, bigrams, trigrams and n-grams of higher order. Each n-gram prefix is stored in a single line, the index keys Key_i are separated by single blanks `" "`. A line within the index file contains the n-gram prefix and the corresponding index value, separated by a "\t" tab character. An example follows below:

```
<ngram-prefix>\t<index value>
```

As large language models may be stored in several smaller files, the index value is extended by a **file identifer** which uniquely determines the language model file for which the file position parameter inside the index data is valid.

The full index value is of the following form:

```
<file position>:<file identifier>:<line count>
```

File identifiers are defined in a list of file names (including the full path to the file) at the beginning of the index file:

```
0:<file name #1>
...
n-1:<file name #n>
```

Finally, the indexer includes the Γ_i parameter values when writing the index file. These values can later be used to index n-grams and look them up in the set of index keys. Γ_i parameters are placed in the first line of the index file, separated by `":"`. The $\Gamma = [3, 2, 1]$ set would look like this:

```
gamma:3:2:1
```

Empty or malformed lines are ignored when processing the index file. A *real world* example should give a somewhat better impression of the index file format:

```
gamma:3:2:1:0:0
files:1
0:/home/cfedermann/diploma/language-models/europarl-v3.srilm.sorted
!       0:0:1
! !     21:0:1
! ! !   45:0:2
...
```

3.6.2 Unigram Vocabulary Format

In order to use the character-level n-gram index data in a language model, it is necessary to know the set of unigrams which were used to generate the index. Typically, these are stored in a list and referenced by integer values to make lookup and comparison fast and easy. The Indexer tool allows to write out a simple vocabulary list which looks like this:

```
<word #1>
...
<word #n>
```

The vocabulary is stored one word per line, the full file contains all unigrams from all language model files which were used for creation of the index data. If multiple files have been used (and thus more than one entry is available inside the list of file names at the beginning of the index file) all unigrams are written into **only one vocabulary** file.

Vocabulary data is usually represented using unique numerical ids. It might be a very helpful extension to the Indexer tool if it was changed to generate these ids and to write them out to the vocabulary file. Again, we separate word form and word id using a "\t" tab character.

```
<word #1>\t<wordid #1>
...
<word #n>\t<wordid #n>
```

3.7 Summary

As we have seen in this chapter, large n-gram corpora can be efficiently indexed using **character-level n-gram indexing**. This technique has been defined based upon so called **n-gram prefixes**. The basic indexing algorithm generates index data by calling an **indexing method**. There are several possibilities to define these methods, we presented **increasing**, **decreasing**, **uniform** as well as **custom** indexing. For evaluation of index data, the notions of **compression rate**, **large subset rate** and **compression gain** have been defined and applied onto SRILM language models.

Chapter 4

An Indexed Language Model

4.1 Motivation

As we have seen in the previous chapter, it is possible to efficiently index large language models. In order to make use of the resulting index data, we now have to create a new language model class inside the Moses MT decoding system. This class will be able to load an n-gram index, the corresponding vocabulary data and will handle all n-gram requests from within the Moses code.

This chapter will describe the design of such an indexed language model and explain the steps which are necessary to integrate it into the Moses MT system. Once the language model is ready, we will compare its translation quality and overall system performance to the original SRILM language model implementation which is included in Moses.

4.2 General Design

The indexed language model is actually divided into two software layers:

On the one hand, there is the **LanguageModelIndexed** class which is derived from the **LanguageModel** base class which is part of the Moses framework. This class provides **high-level** access to the language model data from within the Moses framework.

On the other hand, we will describe the **IndexedLM** class which will handle all **low-level** access to the raw index data. The latter has been designed to provide effective methods which keep the former clean and easy to understand.

Figure 4.1 gives an overview on the general design of the indexed language model and its integration into the Moses MT system:

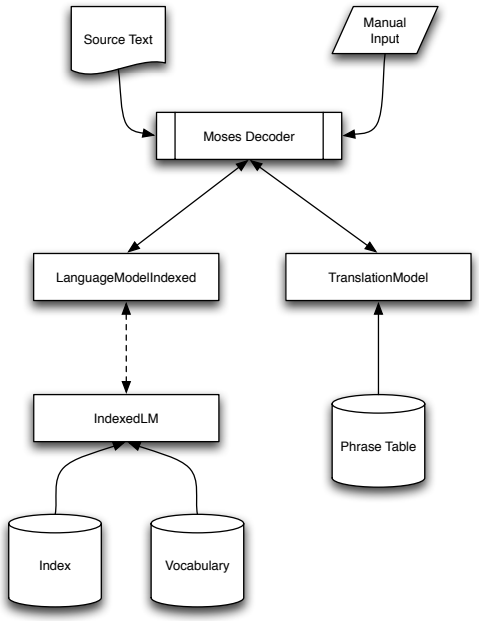

Figure 4.1: Design of an indexed language model

4.3 IndexedLM

We will first describe the low-level **IndexedLM** class and its design and implementation. The Moses MT system assigns a numeric id to each unigram which allows to uniquely identify that unigram. Hence the first thing our language model has to provide is some lookup structure which handles vocabulary data.

4.3.1 Vocabulary Data

To allow fast retrieval of word ids given a unigram and vice versa, two C++ `std::maps` will be used. These guarantee an amortized access time of $O(\log n)$. The unknown word is represented using the value 0, all ids $id \in \mathbb{N}^+$ represent a unique word inside the vocabulary.

The actual implementation of the **IndexedLM** class uses a 32bit `unsigned long` variable to encode the word ids which allows a maximum of 4,294,967,294 possible unigrams in the vocabulary which should suffice even for the largest language models we can currently work with. Word ids and words can be queried from outside the **IndexedLM** class as this is required for full integration inside the Moses MT framework.

4.3.2 Index Data

Next to the unigram vocabulary data the indexed language model has to load and handle the n-gram index data. As we have mentioned earlier, indexing is done using **n-gram prefixes**. Each index key represents a small fraction within the original, large language model file. This subset of the language model can be explicitly determined using the **file position**, the **line count** and the **file name** of the corresponding model file encoded as a numerical **file id**:

size in bytes	4	4	4
content	file id	file position	# of lines

Table 4.1: Subset data for language model files

As the actual implementation of the index data structures is of significant importance, several ideas have been designed, implemented and evaluated. We will only briefly mention the intermediate data structures here, compare them and then describe the final implementation in more detail.

C++ std::map with sorted model subsets

The index is defined using a C++ `std::map` and a custom `struct ExtendedIndexData` which allows to store subset information and the corresponding n-gram data once it has become available. As the n-gram data is volatile, only a pointer to it is stored inside the index data for a given index key. If this pointer is `NULL`, the n-gram data has not yet been loaded from disk or is not available in memory anymore. Otherwise the indexed n-grams are available in (yet another) C++ `std::map` containing `struct NGramData` values. These encode the final n-gram string, its conditional probability inside the language model and, if available, the respective backoff weight.

Language model data is indexed and sorted by the Indexer tool as described in chapter 3. All n-grams belonging to the same subset are stored in one continuous chunk of the sorted model file. As the complete subset is loaded and stored inside a C++ `std::map`, the n-gram lookup is in $O(\log n)$.

This implementation has the advantage that index lookup can be done with an amortized access time of $O(\log n)$ as the `std::map` guarantees this. However there have been performance problems when loading large index sets as the balanced tree behind the C++ `std::map` needed a relatively long time to re-balance itself. Furthermore, the actual amount of required memory was also unsatisfying.

Custom index tree with sorted model subsets

Due to the aforementioned problems with the `std::map` implementation, a custom index tree class has been designed and implemented. Effectively, this index tree stores all possible paths between all possible subsets within the language model. Each of those subsets is dependent on the parameters Γ_i of the n-gram prefixing algorithm and uniquely identified using **prefix ids**. Each of the nodes represents the subset information of the traversed prefix up to the respective depth of the tree.

As in the C++ `std::map` implementation, language model data is indexed and processed in sorted text form. Again, the n-gram lookup time is in $O(\log n)$.

The tree implementation allows faster traversal and lookup of index entries and requires less memory than the C++ `std::map`. Due to the tree structure, the index lookup time is also in $O(\log n)$.

Custom index tree with sorted binary model subsets

The creation of a custom index tree class improved overall system performance, yet subset loading and decoding the n-gram data was still a bottleneck. To speed up subset loading from hard disk, we changed the implementation of the Indexer tool to produce binary output following a simple binary format specification:

size in bytes	4	$4 * \alpha$	8	8
content	n-gram size α	n-gram ids	cp [1]	bow [2]

Table 4.2: Binary format for language model files

The usage of binary data allows easier subset lookup as it enables us to load the whole subset data in one single read operation. Afterwards, decoding the n-gram data can be performed faster as its format is fixed, costly `std::string::find()` calls are not necessary anymore.

Problems arose with respect to the `double` precision, each and every probability was a little bit different from the original value. Another update to the Indexer tool which encoded these probabilities using 16 bytes instead of just 8 bytes helped getting rid of the precision problems, yet the file size of the binary language model file nearly doubled.

[1] conditional probability
[2] backoff weight

Custom index tree with serialized binary tree models

As the binary model subsets did not achieve significant performance improvements, a further refinement of the Indexer tool was added. Instead of just writing out binary n-gram data for each of the indexed subsets, the subset data was collected from the original model file and then transformed into an n-gram tree in memory. In the final step of the index generation, we replaced the output of binary n-gram data lines by serialization of the n-gram trees to hard disk.

size in bytes	4	$(4+8+8+4)*\alpha$			
content	# of children α	child id	cp	bow	next_ptr

Table 4.3: Binary tree format for language model files

Serialization was realized in such a way that all word ids of the same depth inside the tree were written to file in order and with correctly computed next_ptrs to their children nodes. The n-gram trees could then be loaded from hard disk in a single read operation and efficiently be traversed using a quick search approach in $O(\log n)$.

In theory, this method should have the best properties and performance for our indexed language model as this implementation would allow us the determine full n-gram scores with a single tree traversal, similar to the SRILM implementation.

However, the necessary changes to the Indexer tool greatly decreased its performance while heavily increasing the memory requirements. The actual n-gram lookup performance from hard disk was also not as good as previously expected and the problems with binary precision which we already observed for sorted binary model subsets persisted.

Next to these shortcomings, the approach did also increase the overall complexity of the indexed language model implementation. Hence we decided to drop the idea of serialized binary tree models.

4.3.3 Comparison of the Different Index Data Structures

Let us now briefly compare the different possibilities to store and access index data. **Memory** refers to the amount of RAM which is needed for operation, **speed** describes the decoding performance and **disk** represents the size of the respective model file on hard disk. As the overall lookup times should all be in $O(\log n)$, these three columns provide a more meaningful way to compare the listed approaches:

data structure	index lookup time	n-gram lookup time	memory	speed	disk
`std::map`	$O(\log n)$	$O(\log n)$	$-$	$+$	$+$
slow construction for large index sets, relatively high memory usage					

Table 4.4: C++ std::map index data structure

data structure	index lookup time	n-gram lookup time	memory	speed	disk
Index tree	$O(\log n)$	$O(\log n)$	$+$	$-$	$+$
performs better than C++ `std::map`, subset loading slow					

Table 4.5: Custom index tree index data structure

data structure	index lookup time	n-gram lookup time	memory	speed	disk
Index tree	$O(\log n)$	$O(\log n)$	$+$	$+$	$-$
improves subset loading speed, but has `double` precision problems					

Table 4.6: Index tree with binary model index data structure

data structure	index lookup time	n-gram lookup time	memory	speed	disk
Index tree	$O(\log n)$	$O(\log n)$	$+$	$++$	$--$
best lookup performance, worst indexing performance					

Table 4.7: Index tree with binary tree model index data structure

4.3.4 Final Implementation

Empirical tests have shown that the **custom index tree** shown in table 4.5 performs best. As the usage of binary n-gram data makes the indexing process a lot more complex and lookup speed is only minimally improved, we have decided to keep standard sorted language model files as described in chapter 3. This also allows manual post-editing as these model files are still human-readable.

4.3.5 N-gram Cache

The **IndexedLM** class features an n-gram cache which is designed to minimize hard disk access times. Whenever a request is handled, the resulting conditional probability and backoff weight for the given n-gram are stored in a so-called *n-gram cache*. Subsequent requests for the same n-gram will be looked up from the n-gram cache instead of being loaded from hard disk. The data will remain inside this cache until it is cleared, for example after a complete sentence has been processed.

The following figure gives an idea how the indexed language model, the index and vocabulary files and the n-gram cache interact. After the **IndexedLM** object has been instantiated, it loads both the index data using `loadIndex()` and the vocabulary using `loadVocab()`. Only the index information is loaded and n-gram data is handled on demand. Once the data inside the n-gram cache is not needed anymore, the cache can be cleared using `clearCache()`.

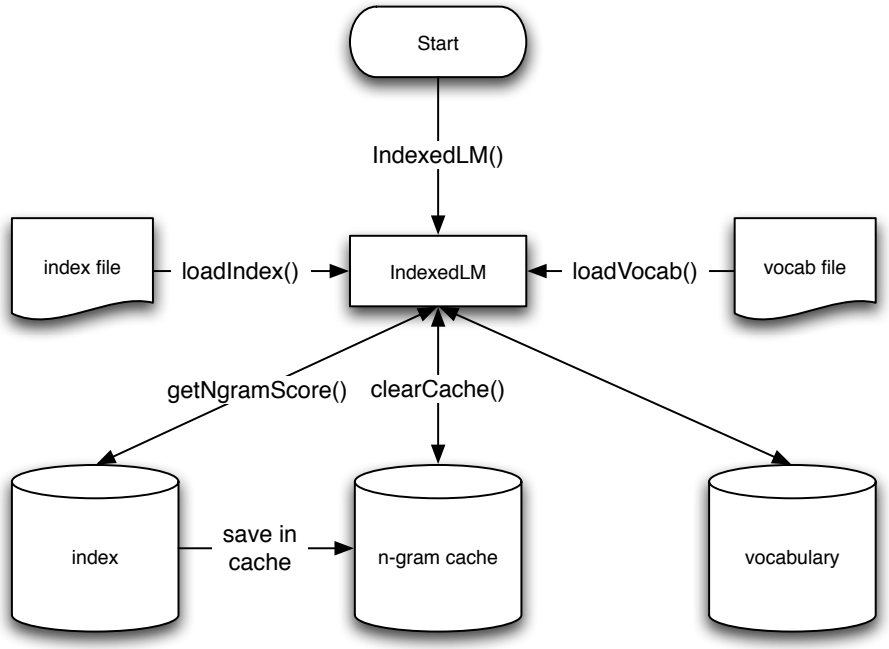

Figure 4.2: IndexedLM cache overview

4.3.6 N-gram Retrieval

After the **IndexedLM** object has been created, it allows to lookup conditional probabilities and backoff weights for any given n-gram. The correct index key for any given n-gram can be determined using the `model_gammas` parameters. Using this information, it is possible to retrieve the n-gram in the language model if it is available.

Typically if a given n-gram $w_1 w_2 ... w_n$ cannot be found inside the language model data, the language model tries to backoff to a smaller n-gram $w_2 w_3 ... w_n$ to approximate the probability of the full n-gram. The indexed language model was designed to be compatible to the lookup algorithm in the original SRILM language model implementation.

However, while the original SRILM model is tree based and can do the n-gram lookup by traversing this tree, the indexed language model is only partially available in memory and can only lookup single n-grams. Therefore it has to emulate the SRILM behaviour performing multiple calls to `loadNgram()`. Furthermore it is not possible to keep a pointer to the maximal n-gram retrieved, the Moses concept of an n-gram `State` will be omitted.

4.3.7 Retrieval Algorithm

The following pseudo code illustrates how the indexed language model tries to lookup a given n-gram inside the language model. The algorithm is the main connection between the low-level **IndexedLM** object and the high-level **LanguageModelIndexed** object. It has been implemented in the `getNgramScore()` method:

Algorithm 2 retrieve largest n-gram inside the language model

Require: n-gram $w_1 w_2 ... w_n$
1: current-ngram $\leftarrow$ n-gram, $\alpha \leftarrow 1$
2: result $\leftarrow$ {probability = 0, backoff-weight = 0}
3: **while** current-ngram not found and $n > 0$ **do**
4: current-ngram $\leftarrow w_\alpha w_{\alpha+1} ... w_n$
5: $\alpha \leftarrow \alpha - 1$
6: **end while**
7: result.probability $\leftarrow$ PROB(current-ngram)
8: **if** order(current-ngram) < order(n-gram) **then**
9: result.backoff-weight $\leftarrow$ COMPUTE-BACKOFF
10: **end if**
11: **return** result

The algorithm gets an n-gram and tries to look it up in the language model file. If the full n-gram is not available in the model file, the algorithm gradually reduces the n-gram size and tries to look up the smaller n-grams. Once an n-gram has been found and it is not the full n-gram, the backoff weight is determined using COMPUTE-BACKOFF. The inner details of this algorithm will be shown below, ($\oplus$ denotes string concatenation).

Algorithm 3 retrieve backoff weight for a given n-gram and context position α

Require: n-gram $w_1 w_2 ... w_n$, $1 \leq \alpha < n$

```
backoff-weight ← 0
current-ngram ← w_α
while current-ngram found and α > 0 do
    backoff-weight += BOW(current-ngram)
    current-ngram ← w_{α-1} ⊕ current − ngram
    α ← α − 1
end while
return backoff-weight
```

The backoff weight is determined by searching for the longest n-gram prefix which is available in the language model. For each of the increasingly larger prefixes which is found during the search the corresponding backoff weight is retrieved and accumulated.

A short example will help to illustrate n-gram retrieval. Assume we are given the n-gram "`<s> the house </s>`" and the following logarithmic language model probabilities:

probability	n-gram	backoff-weight
$-\infty$	`<s>`	-1.444851
-3.786017	`<s> small`	-0.1383325
-4.330768	`</s>`	
-0.8830299	`<s> the`	-0.5427635
-2.439504	`<s> the house`	-0.1765614
-4.24368	`house`	-0.2198115
-2.468234	`house this`	0
-3.485961	`small`	-0.389709
-1.934778	`the`	-0.7377257
-2.707532	`the house`	-0.3511544
-2.460786	`this`	-0.5616246

Table 4.8: N-gram retrieval example, all scores in $\log_{10}$ format

The largest connected n-gram contained within language model is "`</s>`". Hence the basic logarithmic conditional probability is -4.330768.

Now we will compute the backoff weight: the token "`house`" is our first approximation. It can be found within the language model and so we memorize its backoff weight, −0.2198115.

We then try to gradually increase the n-gram prefix from "`house`" to "`the house`" and finally to "`<s> the house`". As all these prefixes exist, we also keep their backoff weights, namely −0.3511544 and −0.1765614.

In the end, we get a logarithmic conditional probability of −5.078295:

n-gram	probability	backoff-weight	Σ
`</s>`	−4.330768		−4.330768
`house`		−0.2198115	−4.5505795
`the house`		−0.3511544	−4.9017339
`<s> the house`		−0.1765614	−5.0782953

Table 4.9: N-gram probability construction, all scores in $\log_{10}$ format

If we were given the n-gram "`my house`" instead, this would give a final logarithmic probability of just −4.24368 as the word "`my`" is not contained within the language model which makes it impossible to add anymore backoff weight.

4.4 LanguageModelIndexed

Now we will briefly introduce the high-level **LanguageModelIndexed** class. This class connects the **IndexedLM** class with the Moses MT system. It has been derived from the **LanguageModelSingleFactor** base class and provides methods to lookup word ids given a string and to compute n-gram probabilites.

4.4.1 Interaction with IndexedLM

The volatile cache inside **IndexedLM** is cleared (using `clearCache()`) after a single sentence has been processed. In order to construct an own vocabulary map for **LanguageModelIndexed**, the class directly accesses the low-level vocabulary `std::map` provided by **IndexedLM**. This is necessary as the Moses MT system has to know word ids for the vocabulary it processes, otherwise translation would not rely on the language model anymore and be simplified to phrase-table transfer. Actual n-gram probabilities and backoff weights are computed using the `getNgramScore()` method.

The following figure illustrates the relations and interactions between high-level class LanguageModelIndexed and its low-level counterpart IndexedLM:

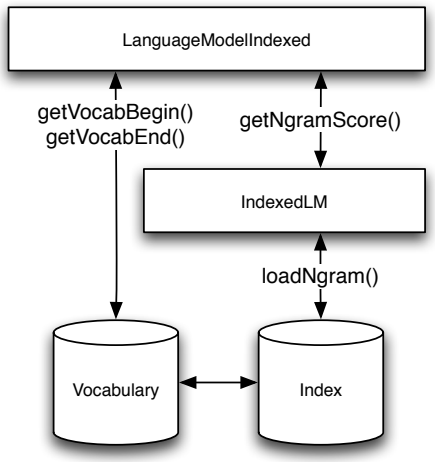

Figure 4.3: Interactions between LanguageModelIndexed and IndexedLM

4.4.2 Interaction with Moses

The **LanguageModelIndexed** class implements the **LanguageModelSingleFactor** interface defined in the Moses MT framework. It provides full compatibility to the **LanguageModelSRI** class provided the same base SRILM model file is used.

Figure 4.4 gives an overview on the dependencies between the Moses framework and the high-level LanguageModelIndexed class:

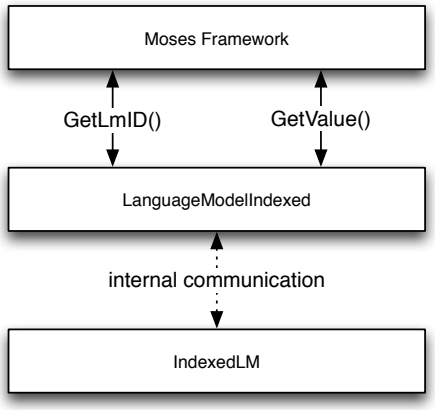

Figure 4.4: Interactions between Moses and LanguageModelIndexed

4.4.3 Moses Integration

To integrate the indexed language model code into the Moses MT system the following changes and additions had to be done:

file name	status	description
configure.in	modified	added conditional which allows to switch on/off IndexedLM support
LanguageModelFactory.cpp	modified	added actual construction of IndexedLM objects
Makefile.am	modified	added IndexedLM, IndexTree and NGramTree files to list of object files
Parameter.cpp	modified	changes to allow loading of IndexedLM files
TypeDef.h	modified	added IndexedLM to the list of available language models

Table 4.10: Changes to the Moses framework

file name	status	description
LanguageModelIndexed.cpp	added	code which connects Moses and IndexedLM
LanguageModelIndexed.h	added	
indexedlm.cpp	added	implementation of the IndexedLM class
indexedlm.h	added	
indextree.cpp	added	implementation of the IndexTree class
indextree.h	added	
ngramtree.cpp	added	implementation of the NGramTree class
ngramtree.h	added	

Table 4.11: Additions to the Moses framework

4.5 Comparison to the SRILM Model

The indexed language model has been designed and implemented with full compatibility to the original SRILM implementation in mind. The `getNgramScore()` method emulates the tree traversal inside SRILM and generates both the same n-gram scores and backoff weights. This allows to use the indexed language model as a drop-in replacement for any given SRI language model.

4.5.1 Performance

When evaluating the performance of the indexed language model, it is important to think about the differences between memory and hard disk access. While the first is measured in *nano seconds (ns)*, the latter is performed in *micro seconds (µs)*, there is a difference of three orders of magnitude. Hence there is an implicit performance loss whenever loading from hard disk is involved.

Let us now briefly reconsider why the indexed language model has been designed and implemented. Large amounts of computer memory are still reasonably expensive yet they would

be required to make use of large language models inside the Moses MT framework. As large language models can help to create better and stylistically more satisfying translations it makes sense to use them but the costs for upgrading RAM can be a difficult obstacle.

Now the indexed language model provides an implementation which allows to actually use those large language models without the need to provide more computer memory. The downside lies in the fact that translations will take longer when performed using the indexed language model which is due to the slower hard disk access times.

4.6 IndexedLM vs. SRILM

We will compare the processing time, memory usage and BLEU scores of several language models on the following pages. The models have been evaluated using a small test set of 10 sentences drawn from the Europarl corpus. We chose this number of sentences as it represents a common choice for *real life experiments*.

The language models have been generated from Europarl data and the second release of the LDC [3] Gigaword corpus. The respective model files differ in size and n-gram order as shown in the following table:

language model	SRI file size	index file size	n-gram order
en-3gram	90 MB	14 MB	3
en-4gram	145 MB	13 MB	4
en-5gram	185 MB	13 MB	5
cna_eng	183 MB	17 MB	5
europarl-v3	324 MB	19 MB	5
xin_eng	2.3 GB	85 MB	5
afp_eng	2.4 GB	125 MB	5
nyt_eng	3.6 GB	211 MB	5
apw_eng	6.0 GB	193 MB	5

Table 4.12: Overview of all evaluation language models

[3] Linguistic Data Consortium

language model	startup time [s]	processing time [s]	processing time / sentence [s]	memory usage [MB]	BLEU score
en-3gram.srilm	34	10	1.0	0.9 GB	21.13
en-4gram.srilm	37	11	1.1	1.0 GB	19.25
en-5gram.srilm	37	11	1.1	1.0 GB	19.80
cna_eng.srilm	47	14	1.4	1.0 GB	16.87
europarl-v3.srilm	61	11	1.1	1.2 GB	20.46
xin_eng.srilm	255	12	1.2	3.7 GB	17.33
afp_eng.srilm	272	13	1.3	5.0 GB	17.87
nyt_eng.srilm	398	17	1.7	4.5 GB	15.20
apw_eng.srilm	614	18	1.8	8.1 GB	15.58

Table 4.13: SRI language model performance within the Moses MT framework

language model	startup time [s]	processing time [s]	processing time / sentence [s]	memory usage [MB]	BLEU score
en-3gram.index	162	276	27.6	0.9 GB	21.99
en-4gram.index	244	289	28.9	1.0 GB	18.95
en-5gram.index	253	293	29.3	1.0 GB	20.04
cna_eng.index	263	277	27.7	1.0 GB	16.85
europarl-v3.index	312	224	22.4	1.0 GB	20.17
xin_eng.index	319	282	28.2	1.4 GB	17.19
afp_eng.index	351	256	25.6	1.5 GB	17.37
nyt_eng.index	420	317	31.7	1.8 GB	15.21
apw_eng.index	513	322	32.2	2.0GB	15.62

Table 4.14: Indexed language model performance within the Moses MT framework

4.7 Summary

In this chapter, we have shown how an **efficient indexed language model** can be designed and implemented. We decided to use a **tree-based index** data structure and look up n-gram probabilities from sorted model files which are stored in ASCII format. For evaluation, processing time, memory usage and BLEU scores have been determined. Overall, the indexed language models **performed slower** than the corresponding SRI models, however they required **less memory**. The actual performance of the indexed language models depends on the selection of the respective indexing parameters Γ.

Chapter 5

A Standalone Language Model Server

5.1 Motivation

The current Moses decoder has a simple yet disturbing drawback. Every time the decoder is started it loads the language model data from disk and processes the phrase-table data. Even if the same language model is used to translate a set of documents there exists no possibility to keep the language model data available in memory. As experimentation with machine translation often involves only little changes of the scoring weights while the actual language model file remains unchanged, this clearly is a drawback of the current implementation.

A new approach can help to improve this situation. It is possible to create a dedicated language model server which loads a given language model when started and makes the n-gram data contained within the language model available via shared memory or using network communication. This chapter will describe the implementation of such a language model server and will also present a new language model class inside the Moses framework which connects the Moses decoder to the new language model server.

5.1.1 Further Applications

The language model server can also be used from within other applications. This is possible due to a flexible and extendable protocol which can be used to remotely control the language model server. It is hence possible to use language models in new ways and different application fields which makes the whole idea interesting for several research projects. For example, one might think of a natural language generator which rates its output using the language model server discarding any text which does not get a certain score from the language model.

5.2 General Design

The Moses framework defines its language model classes using multiple class inheritance. As implementations for several language model types such as SRI or IRST models already exist within the Moses code base, we have designed the language model server to build upon these foundations. Hence we can support all language model types supported by Moses *out of the box*. Furthermore, we could also integrate the indexed language model developed in chapter 4 with only little effort.

To allow easy integration of the language model server into existing or new applications, a simple, text-based protocol will be developed. N-gram data can be queried using either TCP networking or shared memory IPC methods. This guarantees fast and portable access to the language model data.

The following figure illustrates the general design of the language model server:

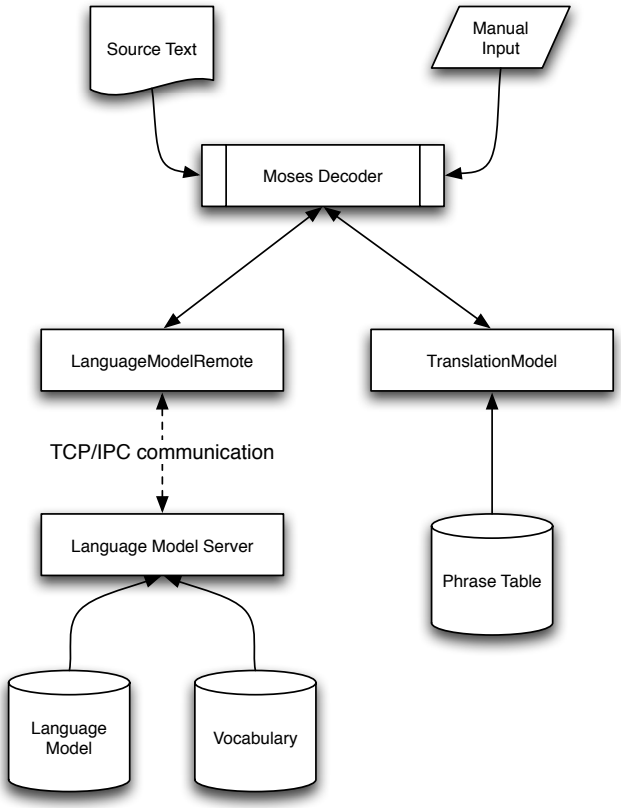

Figure 5.1: Design of a language model server

5.3 Server Modes

As we have mentioned above the server supports TCP and IPC communication. It can either be started in pure TCP or IPC mode, additionally there exists supports for a *mixed* mode which allows both TCP and IPC queries. The language model server handles access from multiple clients at the same time.

5.4 TCP Server

5.4.1 Advantages

The TCP server creates a new listen socket at a given port once it has been started. All incoming connection requests are monitored and handled according to the commands defined in the server protocol. As TCP communication itself is operating system independent, this server mode provides a very flexible way to share language model data within a network. Even shell scripts can easily send requests to the server.

Next to this flexibility, the usage of TCP also allows to process queries from multiple clients at the same time. Depending on the actual language model implementation, this can greatly improve the performance of the server. However, to make use of this feature, it is mandatory to check whether the language model can be used in parallel to handle several requests.

Finally the TCP protocol guarantees reliable, in-order processing of client queries.

5.4.2 Disadvantages

Processing speed can become the main problem of the TCP server. In fact all data sent to the server has to be transformed into TCP packets. This adds a memory overhead and also takes some time to finish. Furthermore, requests originating from distant machines will need a certain time to be routed to the language model server.

Also TCP packet size is limited and hence huge vocabularies will create a certain amount of congestion when queried. This could be circumvented by sharing vocabulary data with the clients as the *vocabulary download* could be omitted then. Standard requests to the language model server will most likely never get split into several TCP packets.

5.4.3 Overview

In practice the aforementioend shortcomings of the TCP server have not become an issue as the advantages prevail and reliable access from large networks or the internet represents a valuable extension to language modeling.

The approach has the general advantage that only little configuration data is needed on the client side, all other data is exchange at processing time. The following figure shows how the TCP server mode works:

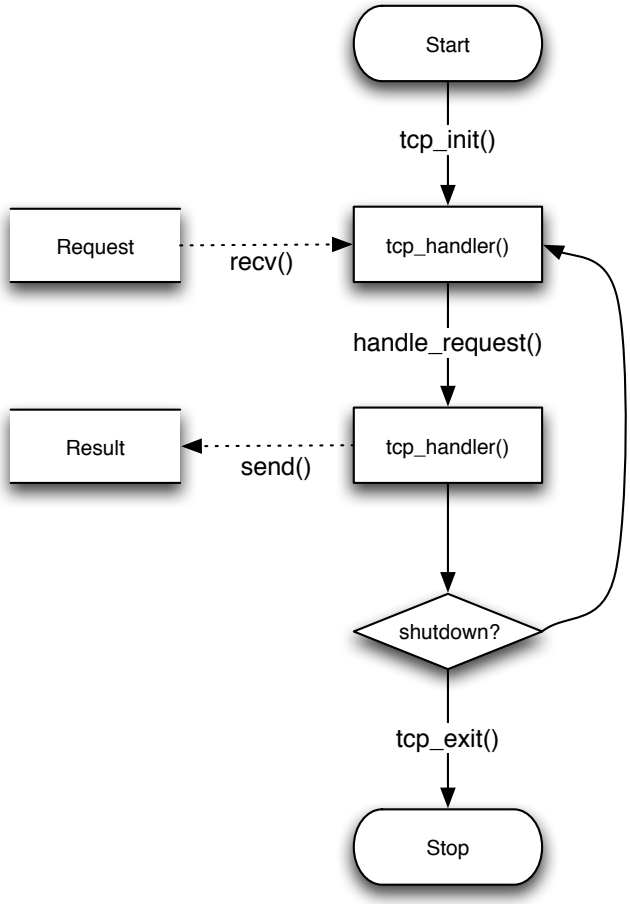

Figure 5.2: Flow chart of the TCP server mode

5.5 IPC Server

5.5.1 Advantages

In order to speed up communication in situations where both the client and the server reside on the same machine, the IPC server mode allows to query the language model server using a shared memory approach. Instead of opening a port this mode creates two shared memory areas, one for requests, one to write back results. The shared memory areas are generated using a numeric seed value which helps to create unique identifiers for them. Access to the request and result memory is managed using a semaphore which locks the resources once a read/write operation is initiated.

It is possible to create a large result area which even allows to write back a huge vocabulary in a single step. This is a clear advantage over the TCP server mode which is constricted by the maximum TCP packet size. Another, even more important, advantage lies in the pure speed of the IPC approach: there is no transformation into packets needed, there is no overhead but requests can be processed very quickly as only local memory access is involved.

Due to the utilization of semaphore locking, we can guarantee reliable, in-order processing of client requests.

5.5.2 Disadvantages

The IPC server is dependent on the Linux/Sys V shared memory implementation and hence not easily portable to different operating system architectures (e.g. Windows). Here we can observe a clear advantage of the TCP server mode which is portable to any operating system which operates a TCP stack, including exotic systems like Haiku or SkyOS.

Furthermore, the semaphore locking forces our language model server to process client requests sequentially. It is not possible to allow parallel queries from multiple clients. However, as access to the language model server should be quite fast when using the IPC server mode this does not have a serious impact on overall system performance.

Figure 5.3 illustrates the IPC server mode:

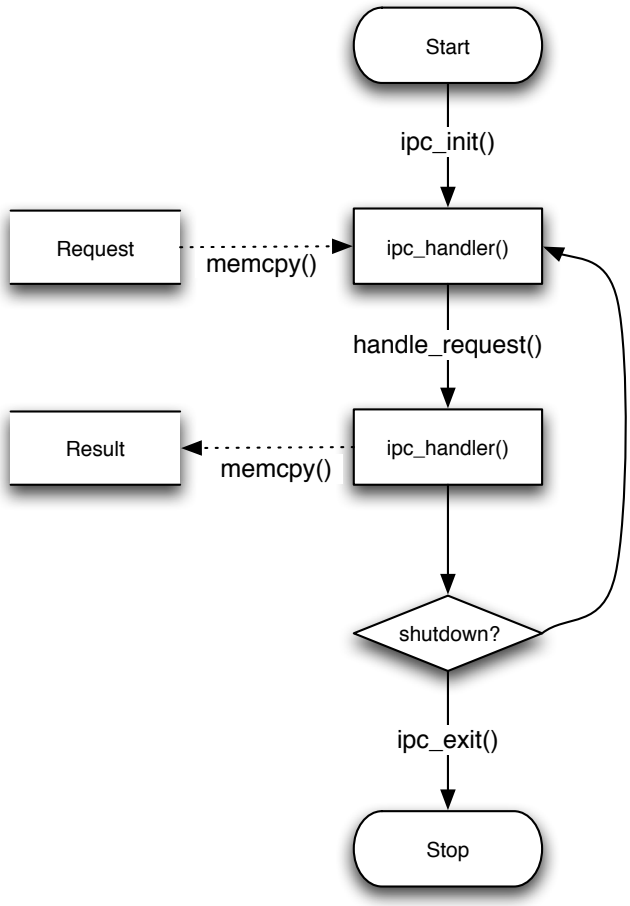

Figure 5.3: Flow chart of the IPC server mode

5.6 Server Mode Comparison

To finish the inspection of the different server modes we provide a tabular summary of their respective features. This compares whether the mode supports sequential or parallel access to the language model, it lists possible limitations concerning memory and rates the processing speed. Finally the code portability is indicated.

server mode	sequential / parallel access	memory limitations	processing speed	portable
TCP	yes / yes	TCP packet size	fast	yes
IPC	yes / no	no	very fast	no
MIXED	yes / yes	TCP packet size	fast	no

Table 5.1: Comparison of language model server modes

5.7 Protocol

To remotely control the server a simple, text-based communication protocol has been designed. This protocol allows to query n-gram scores from the language model and also makes the vocabulary data available to the clients. Queries are composed of pre-defined keywords and n-gram data. N-grams can be specified using the actual word forms or the corresponding word ids. Results are returned in text form and have to be post-processed by the respective application.

5.7.1 Request Format

Requests to the server are composed of a unique **command** code and a set of **arguments**, if applicable. Multiple arguments should be separated by blank spaces `" "`. This results in the following format:

```
<command>:<argument #1> <argument #2> ... <argument #n>
```

5.7.2 Result Format

After a client request has been processed by the language model server, it returns some result to the client. Regardless of the actual type of the result it is returned as text data. The clients have to post-process result data according to the protocol specifications themselves after receiving a result from the server.

5.7.3 Protocol Commands

The following table lists all protocol commands which are currently implemented in the language model server code base. It lists the respective arguments and results and specifies the type of the result. A detailed description of the commands follows below the table.

command code	arguments	result	result type
SHUTDOWN	none	`"SHUTDOWN"`	string
TYPE	none	`"SRILM", "INDEXEDLM"`	string
MODE	none	`"TCP", "IPC", "MIXED"`	string
GETID	word	`$WORD_ID`	integer
GETWORD	word id	`$WORD`	string
GETVOCAB	none	`$VOCAB_DATA`	string
UNKNOWNID	none	`$WORD_ID`	integer
NGRAM	words	`$SCORE`	float
SCORE	word ids	`$SCORE`	float

Table 5.2: Protocol commands for the language model server

5.7.4 Description

We will now describe the required parameters and actual effect of the protocol commands. Whenever no information regarding arguments is given, the respective command does not support/require arguments.

- **SHUTDOWN:** this command tells the server to close all open connections and exit the server loop. Afterwards no connections to this server instance are possible anymore and the server process quits. Any resources used by the server are freed and can be re-used at once. This is especially important for the port used in TCP server mode.
- **TYPE:** sending the TYPE command to the server returns the type of the language model to the client computer. At the moment, only SRI and Indexed language models are supported, so either `"SRILM"` or `"INDEXEDLM"` is returned. Later revisions of the language model server may allow additional language model types, the TYPE command would then be updated accordingly.
- **MODE:** each instance of the language model server is configured for one of the three possible server modes. The MODE command returns `"TCP"`, `"IPC"` or `"MIXED"` with respect to the current server configuration.

- **GETID** ⟨*word*⟩**:** this command can be used to query the vocabulary id for the word specified in the argument parameter ⟨*word*⟩. As we have already described in previous chapters, words are internally represented using unique numeric identifiers, i.e. integers. If ⟨*word*⟩ is contained within the vocabulary, the corresponding word id is returned, otherwise the unknown id is returned.

- **GETWORD** ⟨*word id*⟩**:** given a numeric word id, this command returns the corresponding word contained within the vocabulary. If ⟨*word id*⟩ can not be found the empty string `""` is returned.

- **GETVOCAB:** the GETVOCAB command tells the server to send a list of all words contained within the vocabulary. Additionally it returns the corresponding word ids for each of these words. The list is formatted like this:

  ```
  <id #1>=<word #1>:<id #2>=<word #2>: ... :<id #n>=<word #n>
  ```

  It is worth noting that this command generates **a lot** of data which can result in network congestion or runtime delays. For IPC connections, larger shared memory areas can help to overcome this problem.

- **UNKNOWNID:** it is clear that no vocabulary can ever contain every possible word. In fact it is very likely to find unknown words when working with *real life* data. Internally, these unknown words are represented using a special, designated id for the **unknown word**. This id can be queried by the client using the UNKNOWNID command. Usually, the unknown word corresponds to 0.

- **NGRAM** ⟨*words*⟩**:** the central task of a language model is to rate or weight n-grams according to their likeliness in natural language. Of course, this should also be the central operation for our language model server. Every client can send n-grams using the NGRAM command. Here, the n-gram data is specified in text form, i.e. word after word. The language model server will then convert the n-gram into word ids, send them through the language model and return the resulting score to the client. If the n-gram is not contained within the language model, the largest n-gram and backoff weight are determined. In case of errors, the empty string `""` is returned.

  An example request will help to illustrate usage of the NGRAM command:

  ```
  "NGRAM:this is a test"
  ```

  If this request would be sent to an active language model server, the server would try to look up the n-gram `"this is a test"` inside the language model data.

- ▷ **SCORE** ⟨*word ids*⟩**:** it is also possible to query an n-gram score based on word ids instead of the actual words. This can be done using the SCORE command. Apart from the different arguments this command and its result are identical to the NGRAM command described above.

5.8 LanguageModelRemote

In order to integrate the language model server into the Moses MT framework, we have extended the Moses code base with a new **LanguageModelRemote** class. This class loads a configuration file which specifies the port, mode and type of an active language model server. Afterwards it establishes a connection to this language model server, either via TCP or using IPC methods, and loads the vocabulary data using the GETVOCAB command.

The LanguageModelRemote class behaves exactly as any other language model class would do, the only difference lies in the fact that actual language model operations are performed by a language model which is not available on the local machine but on a remote server.

5.8.1 Interaction with Moses

The **LanguageModelRemote** class implements the **LanguageModelSingleFactor** interface defined in the Moses MT framework. It maps all requests to the language model to requests which are sent to a language model server.

Figure 5.4 gives an overview on the dependencies between the Moses framework and the high-level LanguageModelRemote class:

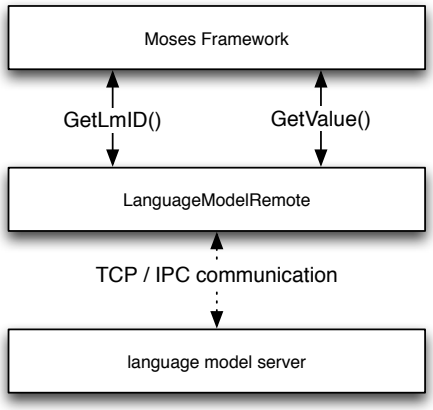

Figure 5.4: Interactions between Moses and LanguageModelRemote

5.8.2 Moses Integration

To integrate the remote language model code into the Moses MT system the following changes and additions had to be done:

file name	status	description
configure.in	modified	added conditional which allows to switch on/off RemoteLM support
LanguageModelFactory.cpp	modified	added actual construction of RemoteLM objects
Makefile.am	modified	added RemoteLM files to list of object files
Parameter.cpp	modified	changes to allow loading of RemoteLM files
TypeDef.h	modified	added RemoteLM to the list of available language models

Table 5.3: Changes to the Moses framework

file name	status	description
LanguageModelRemote.cpp	added	code which connects Moses and the language model server
LanguageModelRemote.h	added	

Table 5.4: Additions to the Moses framework

5.9 Comparison to the SRILM Model

As the language model server itself does not perform any language model computations but rather utilize existing language model classes from the Moses MT framework, we can guarantee that all scores and backoff weights returned by the language model server are identical to those generated by the original models.

5.9.1 Performance

The overall system performance depends on the server mode. While IPC connections are very fast, TCP connections may become slow depending on the distance between server and client within the network. Effectively, the language model server eliminates the need to instantiate a complete language model each time the Moses decoder is started. Instead only a connection to the (already running) language model server is established and used to perform language model computations.

5.9.2 Limitations

The main problem with the Moses MT framework is the missing support of *batched* calls to the language model methods. Each and every interaction with the language model results in a single call to the model. For our remote language model this generates a lot of single calls

which have to be transmitted over TCP or IPC. Even if all those single calls can be processed very quickly on the server side, the accumulated transmission time slows down the overall performance of the remote language model.

If the Moses decoder would instead support *batched* calls, i.e. collective processing of a certain number of calls to the language model, the transmission delay would decrease which would result in an improved system performance.

5.10 Summary

In this chapter, we have described the design and implementation of a **standalone language model server** which allows to **share language model data** over a network or on a single machine. The server supports **TCP connections** and **IPC shared memory** methods which makes the language model server fast, flexible and portable.

Furthermore we developed a **remote language model** class which allows to connect the Moses decoder to a shared language model server, all internal method calls are mapped to remote requests. The server is built upon the language model foundations of the Moses MT framework and hence **supports all Moses language model** types. Scores and backoff weights are **fully compatible to the original language models**.

It is also possible to make use of the language model data in completely new applications as the server can be **remotely controlled using a simple protocol**.

Chapter 6

A Google 5-gram Language Model

6.1 Motivation

In the previous chapters, we have designed and developed the means to handle very large language models inside the Moses MT framework. We have defined several indexing methods which enable the Indexer tool to create an indexed language model out of a large language model in SRI format. This indexed language model can then be loaded by the Moses decoder and be used to translate texts.

We now want to create such a large language model and evaluate its performance. In order to create the indexed language model, we first have to create a standard SRI model out of a large corpus. While the LDC Gigaword corpora are already pretty large, there exists an even larger corpus, the **Google 5-gram corpus**. This chapter will describe the steps taken to create an indexed language model from this corpus.

6.2 Google 5-gram Corpus

Google released a **huge collection of English n-gram data** in August 2006. This data contains **billions of n-gram types (based on a trillion of running words)** up to a maximum n-gram order of 5 and has been collected from public web pages in early 2006. It has later been made available by the Linguistics Data Consortium as a set of 6 DVDs. All together, the compressed n-gram data is about 24 GB in size, the uncompressed version takes more than 90 GB of space.

The n-gram counts for the Google 5-gram corpus are shown in the following table:

number of	count
tokens	1,024,908,267,229
sentences	95,119,665,584
unigrams	13,588,391
bigrams	314,843,401
trigrams	977,069,902
fourgrams	1,313,818,354
fivegrams	1,176,470,663

Table 6.1: Google 5-gram corpus counts

The Google corpus contains large lists of n-grams each single one annotated with its frequency relative to the total number of tokens. Both tokens and n-grams have been post-processed and a frequency cutoff has been applied. For more detailed information, please refer to the respective `readme.txt` [1] file.

6.3 Corpus Preparation

To prepare the Google corpus data to be used for a large language model, all n-gram data was first converted to lowercase and then sorted lexicographically. In order to keep the file size of the n-gram data files on a manageable level, we decided to create separate files for each of the different n-gram orders.

6.4 Language Model Generation

Before we can actually invoke the Indexer tool to create an indexed language model for the Google corpus, we have to create a language model in SRI format. Usually, we would use the `ngram-count` tool for this but the sheer amount of n-gram data already lets us doubt whether it can be used for this specific task.

Several attempts to compile a complete Google SRILM failed when `ngram-count` was used, even the special script `make-big-lm` which is dedicated to create large language models in a more memory efficient manner did not help to overcome these problems. The SRILM toolkit seemed to have severe problems with interpolation and smoothing of n-gram counts, hence

[1] http://www.ldc.upenn.edu/Catalog/docs/LDC2006T13/readme.txt

after some time of trial and error, we had to discard our initial plan to create the Google SRI language model file using the SRILM toolkit.

In fact, we decided to skip all interpolation and smoothing efforts for our new language model and simply used the given frequency counts to determine the corresponding conditional probability values for the n-grams. This approach does not support n-gram backoff anymore and could be compared to a large **n-gram memory** instead of a full-fledged language model. However as the complete Google 5-gram corpus is a lot larger than any language model currently known, we considered this worth a try. Current work also suggests a reduced importance of full interpolation and smoothing for very large language models. [Brants et al., 2007]

The final Google language model in SRI format was about 100 GB in size.

6.5 Indexing

After the creation of this basic *lookup model* in SRI format was finished, we used the Indexer tool to build an indexed version of the Google language model. In order to keep the index size small, we decided to use the following set of Γ_i parameters:

$$\Gamma_{GoogleLM} = \{3, 2, 1, 0, 0\} \tag{6.1}$$

As we had already seen in chapter 3, indexing up to the third word of the n-gram suffices even for n-grams of higher order. Hence we limited the Γ_i parameters to only three values.

To prevent memory issues when indexing the Google data, we split up the complete 40 GB file into several smaller files which could still be handled by the Indexer tool and would still fit into the 32 GB of the development machine. This was possible as the Indexer tool allows to customize the file ids it assigns to the different language model files. For more information on this, please refer to the Indexer tool documentation on page 29.

After the respective parts of the Google data had been indexed, the resulting index data and vocabulary files were merged to create a single index data file for the complete Google language model. Merging the index data files was done by a merging script which combined n-gram subsets for index keys which appeared in multiple index data files. The final index file still contained references to several n-gram files but after the merging process had been completed it was guaranteed that each and every index key referenced only to a single file containing n-gram data.

The full index data file was about 5.6 GB in size, the vocabulary about 0.1 GB.

6.6 Index Merging

The merging algorithm pseudo code is shown below. The algorithm requires a set of n index data files (represented by $IDX_1, ..., IDX_n$) which contain index keys generated using the same set of Γ_i parameters. It is not possible to join index data files which were created using different Γ_i sets. If an index key exists in more than one index data file, the corresponding n-gram data subsets will be merged into a single n-gram subset. All output is written to a new index data file which is represented by the $INDEX$ variable.

Algorithm 4 index data files merging

Require: index data files $IDX_1, ..., IDX_n$, index output file $INDEX$

```
1: combined-keys = ∅
2: for i = 1 TO n do
3:    while ∃ some key in IDX_i, key ∉ combined-keys do
4:       current-key ← IDX_i
5:       combined-keys = combined-keys ∪ current-key
6:       INDEX ← COMBINE(current-key, IDX_1, ..., IDX_n)
7:    end while
8: end for
9: return index
```

The pseudo-code does the following:

- ▷ **line 1** initializes the $combined-keys$ set which stores all index keys which have already been processed. If such a key is encountered a second time while the algorithm is running, it will be ignored.
- ▷ **line 2-8** represents the outer loop which iterates over all possible index data files.
- ▷ **line 3-7** represents the inner loop which iterates over all possible index keys within a single index data file. Please note that an index key is only processed if it is not contained within $combined-keys$.
- ▷ **line 4** reads an unknown index key from the index data file.
- ▷ **line 5** adds the current index key to the set of already processed keys $combined-keys$.
- ▷ **line 6** finally combines all n-gram data bound to the current index key. The $COMBINE$ operation checks all index data files and generates a new n-gram data file if the $current-key$ occurs multiple times.

6.7 Evaluation

When trying to evaluate the GoogleLM we quickly reached the limits of our workstation and the indexed language model developed in chapter 4. The current implementation of the indexed language model and its integration into the Moses MT framework are robust and well-tested, yet they should still be considered being in *prototype stage*.

As the index data for our Google language model is already as large as the biggest SRI language model which we used for evaluation in chapter 4 it is clear that huge amounts of computer memory and a long decoding runtime are needed to actually use this data for statistical machine translation. Additionally, a large amount of hard disk space is required to store the sorted model files.

By design, the Moses MT system creates translations using a translation and a language model. In this thesis, we have investigated and developed methods to allow very large language models to be used with the Moses decoder. Effectively, we have increased the amount of information which is available via the language model part of the decoding system. The translation model and its internal *phrase-table handling* have been treated as a *black box* and were used without modification.

Several experiments with the GoogleLM data revealed a serious problem with this design of the Moses decoder, a problem which actually hinders the usage of very large language models. As the decoder treats all tokens which are not contained within the phrase-table as *unknown words*, it is not able to access the amount of additional information which is contained in our large language model. Hence the usage of the Google language model did not help to improve the overall translation quality. The only effect we observed was a slower performance of the Moses decoder.

Because of this unsatisfying intermediate results and due to the enormous amounts of hard disk and computing power which would have been required to evaluate the Google language model in detail we decided not to invest further efforts as long as they would not show any measurable impact on the translation quality.

Therefore we cannot observe an improved translation quality for the very large Google language model. While an updated and more refined version of the indexed language model would help to reduce the overall performance loss, it would still not be able to resolve the problems with the translation model and its phrase-table.

More research on a better interaction between translation and language models seems to be an interesting and important effort as it could help to utilize the huge amount of language model data provided by large corpora such as the Google 5-gram corpus.

6.8 Summary

In this chapter, we have described how we created an **indexed Google language model** from the Google 5-gram corpus released in 2006. As the sheer amount of n-gram data made it **impossible to directly use the SRILM toolkit**, we decided to create a **n-gram lookup model** without any smoothing or interpolation applied to the n-gram data. This approach was also supported by current research results.

Index data was created in several steps, the intermediate index files were then merged using an **algorithm for index merging**. Final evaluation of the Google language model revealed that the current implementation of the Moses MT framework cannot be used with very large language models. This is due to the fact that the internal phrase-table of the Moses decoder **penalizes all unknown words** and is hence not able to access any corresponding n-gram data within the language model.

Further research and refinements to the Moses MT design seem to be necessary before a language model such as the **GoogleLM** can be used to create improved translations.

Chapter 7

Conclusion

In this chapter we summarize what we have achieved with regard to the initial ideas for this thesis (section 7.1), what lessons we have learnt about limitations and problems when using very large language models with the Moses MT framework (section 7.2). Finally, we propose a number of possible ways to improve and refine our work in future research (section 7.3).

7.1 Work Done

In this thesis work, we have investigated the current state of the art for statistical machine translation and statistical language models using the open source Moses MT framework and the SRI language modeling toolkit.

Initially, we have discovered that a better translation quality can be achieved when higher order n-gram language models are used. More specifically, we have shown that the usage of 5-grams instead of 3-grams has a measurable effect on the overall translation quality of a given set of test sentences.

Further experiments with the MT system led to the conclusion that the current design of the Moses MT decoder lacks support for *persistent language model handling*. Each and every time the system is started, it has to re-load the complete language model from disk which slows down the overall performance and hinders experimentation.

7.1.1 Indexed Language Model

For the diploma thesis, we tried to integrate these findings into the machine translation framework. To allow larger language models which could potentially improve the quality of the translated sentences, we decided to design a *new indexed language model* class and a corresponding *indexer tool*.

The basic idea was to create this model such that the amount of required computer memory could be flexibly controlled by the chosen indexing method. The integration should result in a *prototype implementation* which could later be extended for further dissemination and actual usage.

For the indexer tool, we then designed a basic indexing algorithm which was only dependent on a set of suitable indexing parameters. We chose to index n-gram data based on the notion of *character-level n-gram prefixes* and defined several possible indexing methods. These were then applied to several language models and the resulting indexes were compared and evaluated. To allow an efficient evaluation we defined the notions of *compression rate, large subset rate, and compression gain.*

We then created a new language model class inside the Moses MT framework which should be able to load index data and map all n-gram requests to the corresponding parts of the language model files on hard disk. This indexed language model was built and tested to be fully compatible to the SRILM format which is a common choice for language modeling with the Moses system.

Evaluation of the indexed models has shown that the actual amount of computer memory which is required to utilize a given language model can be efficiently controlled by the chosen indexing parameters. More precisely this has enabled us to use very large language models without having to load them completely into memory. While the translation quality remained unchanged, we experienced a severe impact on the overall decoding performance. The current *prototype implementation* can still be improved to reduce the performance loss and to make the indexed language model usable on a larger scale.

7.1.2 Language Model Server

Next to the indexed language model, we have also designed and implemented a language model server which allows to separate the Moses MT decoder from the language model data. A new language model class has been integrated into the framework and now enables us to access n-gram data from the same machine or even the internet. That way, we wanted to improve the decoder startup times while keeping the translation quality unmodified.

We also saw a potential for new applications which could be enriched by language model data. In order to allow these to connect to the language model server, we designed a simple communication protocol. Even shell scripts or web applications could now interact with the language model server which further adds to its value.

When evaluating the performance of the remote language model, we also experienced unexpected problems. As the Moses decoder is not designed to interact with a remote language model server, it is not able to *collect multiple requests* to the language model but sends them all one after the other. This creates a lot of delay and hence lessens the actual usability of the language model server. Again, it is possible to further improve the interaction between the Moses MT framework and the remote language model class to create a better integration with the language model server.

7.2 Lessons Learnt

7.2.1 Indexed Language Model

The current implementation of the indexed language model and the accompanying indexer tool requires us to invest a lot more time to create translations with the Moses decoder. This is *not an error* of the chosen implementation but implicitly introduced *by design*. While this was expected behaviour, the actual amount of performance loss was still surprising. However, when put into the right perspective, i.e. when comparing the speed of computer memory to the access times of hard disks, this drawback seems less serious.

7.2.2 Language Model Server

When evaluating the language model server, we experienced serious performance problems which were caused by the inability of the Moses MT framework to support *batched n-gram requests*. Also as both TCP communication and IPC shared memory methods are inherently slower than direct memory access, the remote language model *by design* cannot be as fast as the original SRILM implementation.

7.2.3 Google Language Model

An extremely inconvenient finding was observed when we tried to build and evaluate a very large language model based up the Google 5-gram corpus. It was not clear that the SRILM toolkit would fail for such a huge amount of data as well as the phrase-table problem did effectively hinder any actual usage of the Google language model. Hence we could not finish our evaluations and thus will have to wait for later research results on the impact of a Google language model on translation quality.

7.3 Future Work

After having finished work on this thesis, we have shown that indexed language models within the Moses MT framework are feasible and can be used to utilize very large language models. As we have seen, the memory requirements can be adapted using a suitable set of Γ_i parameters. The whole implementation should now be used under *real world conditions* to improve the overall stability and performance of the system. The following sections propose several ideas for improvements and future work in the field of indexed language models.

7.3.1 Improved Performance

First and foremost it seems to be of significant importance to optimize the processing speed of the indexed language model class and its underlying foundations. If the performance loss can be reduced this will make the complete system more usable for experimentation. This would also allow for a broader dissemination of this work.

To improve the efficiency of the indexed language model and its integration into the Moses MT framework it seems to be reasonable to develop an improved n-gram cache inside the top-level Moses class `LanguageModelIndexed` replacing the n-gram cache which is currently located in the low-level `IndexedLM` class.

7.3.2 Separation of Language Model Data

It would also be interesting to separate language model data into a small fraction which is always available inside memory and a large fraction that is accessed using the indexed language model paradigm. For instance it might reduce the number of hard disk accesses if we kept all unigram data available in memory while larger n-grams would still be loaded from disk. This *hybrid approach* would not require much additional memory, yet the possible performance gain is tempting.

7.3.3 Batched N-gram Requests

When trying to work with our language model server we experienced problems with the internal design of Moses language model handling. At the moment there exists no way to collect multiple n-gram requests which are then sent in a *batched request*. As the availability of such batched requests could greatly improve the overall system performance for the remote language model class it seems to be an interesting area for future work.

It might be possible to collect all n-gram requests on phrase or sentence level or to use batches of a pre-defined size. However this approach will most likely require several complex changes to the Moses decoder.

7.3.4 More Flexible Phrase-tables

The large Google language model did not yield any measurable improvement in translation quality as the phrase-table prevented the Moses decoder to access any of the additional information contained within the language model. The current implementation only works for tokens that are contained within the phrase-table, all other words are treated as unknown words and do not contribute to the overall translation quality.

As we want to utilize the vast amounts of n-gram data which are provided by n-gram corpora such as the Google 5-gram corpus we have to find new ways to handle words that are unknown to the phrase-table. It is perfectly possible that an unknown token is contained within the language model data and could thus be used to create a better translation. This would require changes to Moses internal phrase scoring.

7.3.5 Hybrid Language Models

Last but not least, it also seems a worthwhile effort to explore the advantages and problems of *combined hybrid language models*. Instead of using only a single language model we could use a small *in-domain* language model in SRI format and combine that with a large *out-domain* indexed language model. Together both models could improve translation quality and reduce the amount of error caused by the domain of the source text.

Appendices

Appendix Introduction

The following appendices try to give a more detailed insight into the program code and class design which have been developed as part of this diploma thesis. As the full source code is way to large to be wholly included into this document, only a chosen subset of important code is printed and documented. For more details, refer to comments within the source code.

Source Code License

Appendix A

N-gram Indexing Code

The indexing tool has been designed and implemented using `C++`. It is based on the `Indexer` class and built using the `CmdLine` class and the `Debug` macros. This appendix will briefly introduce the nuts and bolts of the class design and provide further informations on the program implementation.

The source code is freely available at `http://www.cfedermann.de/diploma` under the license terms printed on page 73.

A.1 Class: Indexer

The `Indexer` class takes care of parsing a single or multiple language model files in ARPA format. It uses a set of given Γ_i parameters to create index data out of the language model data conforming to one of the indexing methods defined in chapter 3. It also handles unigram vocabulary creation and writes out sorted model files for each of the given language models.

A.1.1 Constants

The `Indexer` class defines the following constants:

```
#define FLUSH_LINES 1024
#define MAX_NGRAM_SIZE 5
```

Description:

- ▷ the constant `FLUSH_LINES` defines how many lines are written at once.
- ▷ the constant `MAX_NGRAM_SIZE` sets the maximum order for n-grams.

A.1.2 Typedefs

The `Indexer` class defines the following types:

```
struct IndexData { ... };
typedef std::map<std::string, bool> VocabType;
typedef std::map<std::string, IndexData> IndexType;
typedef VocabType::iterator VocabIterator;
typedef IndexType::iterator IndexIterator;
```

More information on the `IndexData` struct can be found on page 80.

Description:

- ▷ the struct `IndexData` stores all n-gram prefix data.
- ▷ the type `VocabType` defines the vocabulary structure. The `bool` is used to memorize which unigrams have already been written to the vocabulary file and which have not. This is necessary to prevent creating double entries inside the vocabulary when multiple language model files are processed.
- ▷ the type `IndexType` defines the index structure.
- ▷ the type `VocabIterator` defines an iterator over `VocabType`
- ▷ the type `IndexIterator` defines an iterator over `IndexType`

A.1.3 Public Interface

The `public` interface is shown below:

```
public:
    Indexer(std::map<std::string, std::string>& arguments);
    ~Indexer();

    void createIndex();
    void writeIndex();
    void writeVocab();

    IndexType& getIndex();
    VocabType& getVocab();
```

Description:

- ▷ the constructor `Indexer(std::map<std::string, std::string>&)` expects a `std::map` of command line arguments which is used to configure the `Indexer` class before processing the language model files.
- ▷ the destructor ∼`Indexer()` releases all allocated heap memory which has been used to store the language model contents.
- ▷ the method `void createIndex()` takes care of the actual index creation. It iterates over all given language model files, generating index data and unigram vocabulary for each of them. After this method has returned, all index and vocabulary data is processed and can be written to the corresponding files using `writeIndex()` and `writeVocab()`. Additionally, it is possible to post-process the data using `getIndex()` and `getVocab()`.
- ▷ the method `void writeIndex()` writes index data to the index file. It flushes content to file every `FLUSH_LINES` lines. This can be configured in `indexer.h`.
- ▷ the method `void writeVocab()` writes the unigram vocabulary to the vocabulary file. Again, lines are flushed to the file every `FLUSH_LINES` lines.
- ▷ the method `IndexType& getIndex()` can be used to access the contents of the `index` map for additional post-processing.
- ▷ the method `VocabType& getVocab()` allows to access the `vocabulary` map.

A.1.4 Private Interface

The `private` interface looks like this:

```
private:
    Indexer();

    void createIndex(const std::string& model_file, int file_id);
```

Description:

- ▷ the default constructor `Indexer()` is private to prevent any call to it.
- ▷ the method `void createIndex(const std::string&, int)` takes care of the actual construction of both index data and unigram vocabulary for the language model file specified by the `const std::string&` parameter. The second parameter `int` encodes the `file_id` of this model file which is used internally.

A.1.5 Data Members

The available data members of the `Indexer` class are listed below:

```
std::vector<std::string> language_models;
std::string index_file;
std::string vocab_file;
std::string indexing_method;
std::vector<int> gammas;

VocabType vocabulary;
IndexType index;

bool autoflush;
std::fstream _index, _vocab;
int start_id;
```

Description:

- ▷ the `language_models` vector stores the list of language model files which should be processed when creating the index data.
- ▷ the `index_file` string specifies the name of the index file.
- ▷ the `ivocab_file` string specifies the name of the vocabulary file.
- ▷ the `indexing_method` string specifies the chosen indexing method which is used to setup the Γ_i parameters during class instantiation. It has to be one of the following four values: `increasing, decreasing, uniform, custom`. For more information on these methods refer to the corresponding section on page 22.
- ▷ the `gammas` vector specifies the Γ_i parameters for the index creation.
- ▷ the `vocabulary` map stores all unigrams contained in the language model files.
- ▷ the `index` map contains the index data which is stored as `IndexData` structs.
- ▷ the `autoflush` bool shows if the *autoflush* feature is enabled. For more information on this refer to page 80.
- ▷ the `_index` fstream is used to open the `index_file`.
- ▷ the `_vocab` fstream is used to open the `vocab_file`.
- ▷ the `start_id` int defines the file_id for the first given language model.

A.2 Program: Main Loop

A.2.1 Code

```
 1:  try {
 2:      // create indexer instance and hand over arguments
 3:      Indexer* indexer = new Indexer(arguments);
 4:
 5:     // create index data by parsing the language model files
 6:      indexer->createIndex();
 7:
 8:      // write out index data
 9:      indexer->writeIndex();
10:
11:      // write out vocabulary data
12:      indexer->writeVocab();
13:
14:      // clean up and exit
15:      delete(indexer);
16:  }
17:  catch (std::string e) {
18:      std::cout << "[exception] " << e << std::endl;
19:      return -1;
20:  }
```

Description:

- ▷ **lines 2-23** of the main loop are embedded into a `try ... catch` construct which catches all expceptions that might be thrown during program execution.
- ▷ **line 3** creates the `Indexer` instance and configures it using the command line arguments which have been processed earlier. See the file `indexer.cpp` for more detailed information on command line argument handling.
- ▷ **line 6** creates the `index` map and fills it with indexed contents from the given language model files.
- ▷ **line 9** writes the computed index data to the `index_file`.
- ▷ **line 12** writes the unigram vocabulary to the `vocab_file`.
- ▷ **line 15** takes care of correct destruction of the `Indexer` instance.
- ▷ **lines 17-20** catch any exceptions thrown and report them.

A.3 Struct: IndexData

The `index` map stores all index data created by `createIndex()`. For each of the entries the corresponding position inside the language model file, the file id of the language model and the number of lines being referenced by the corresponding index key have to be stored. As the Indexer tool has to write out a *sorted model file* for each of the given language model files, it is also necessary to collect the corresponding lines for each index entry. A pointer is used to keep the `index` map small and fast. For more information have a look at page 81.

A.3.1 Struct Definition

```
struct
IndexData {
    unsigned long file_pos;
    unsigned int file_id;
    unsigned long line_count;
    std::string* lines;
};
```

A.4 Features

A.4.1 Autoflush

As the Indexer tool is designed to be capable of indexing multiple files, it is configured to *autoflush* contents to the corresponding output files. After a language model has been processed by `createIndex(const std::string&)`, the contents of both the `index` and the `vocabulary` maps are written to disk. The two maps are cleared before the next language model file is processed which reduces the overall memory requirements of the tool and allows to handle very large language model files.

The *autoflush* mode can be disabled using the `-n` or `--no-autoflush` command line parameters. If any of these is present, the whole index and vocabulary data is read into memory instead of being written to disk. The `writeIndex()` method can be used to write index data into the `index_file`, `writeVocab()` creates the vocabulary. If it becomes necessary to post-process either index or vocabulary data before their contents are written, the `getIndex()` and `getVocab()` methods can be used.

A.4.2 Sorted Model Files

To allow fast access to each subset of an indexed language model, it is necessary to create a sorted version of the original model file. This ensures that all lines which are indexed by a given n-gram prefix can be looked up in an efficient way as in the sorted file it is guaranteed that all these n-gram lines will follow one after the other. Please note that the resulting sorted model files are **not in ARPA format** anymore as all unnecessary data such as "\data\" or "\1-grams:" is removed during the creation process. Sorted model files are simply a way to speed up n-gram lookup.

A *real world* example of such a sorted model file looks is shown on the left, the corresponding original language model file (which is not sorted with regard to the n-gram prefixes) can be seen on the right:

file: europarl-v3.srilm.sorted

```
-2.916459       !       -2.05075
-2.76344        ! !     -0.2412159
-0.5342772      ! ! !   -0.0409897
-0.1764681      ! ! ! </s>
-0.1985702      ! ! </s>
-1.864807       ! "     -0.5895861
-1.755537       ! " )
-1.513885       ! " ,
-0.9381909      ! " .   -0.6377587
-0.0001343653   ! " . </s>
-0.2022471      ! " </s>
-1.308389       ! ’     -0.9424482
-1.839963       ! ’ )   -0.3197733
-0.2494251      ! ’ ) </s>
-1.099362       ! ’ ,   -0.04221211
-0.8973418      ! ’ , and
-1.010028       ! ’ , but
-2.236138       ! ’ -   -0.22873
-0.3250176      ! ’ - and
-0.9864706      ! ’ .   -1.138361
```

file: europarl-v3.srilm

```
-2.916459       !       -2.05075
-3.028812       "       -0.5158522
-6.265231       #       -0.1555177
-4.620731       $       -0.2908757
-3.471805       %       -0.7785481
-4.364939       &       -0.6885265
-2.451123       ’       -0.6866914
-5.726556       ’a      -0.1555177
-6.265231       ’abandon        -0.1555177
-6.265231       ’abord  -0.1555177
-6.265231       ’aborder        -0.1555177
-6.265231       ’aboville       -0.1555177
-6.265231       ’absolue        -0.1555177
-6.265231       ’acceptation    -0.1555177
-6.109078       ’accesso        -0.1555177
-6.265231       ’accompagnement -0.1555177
-6.109078       ’accord -0.1856652
-6.265231       ’accords        -0.1555177
-6.265231       ’accueille      -0.1555177
-6.265231       ’accuse -0.1555177
```

Appendix B

Indexed Language Model Code

The indexed language model classes have been designed and implemented using `C++`. The central code can be found inside the `IndexedLM` and the `LanguageModelIndexed` source code files. In order to efficiently store and access index and n-gram data, two additional classes `IndexTree` and `NgramTree` have been developed. The following appendix will briefly describe these classes and provide further informations on the actual implementation.

The source code is freely available at `http://www.cfedermann.de/diploma` under the license terms printed on page 73.

B.1 Class: IndexedLM

The `IndexedLM` class cares for low-level access to a given index file and provides methods to query the indexed language model. It is possible to look up n-gram probabilities and backoff weights if the n-gram is available inside the language model, an internal cache is available to improve performance. Queries can be sent based on n-gram surface forms or word ids.

B.1.1 Typedefs

The `IndexedLM` class defines the following types:

```
struct NGramData { ... };
typedef std::map<std::string, VocabId> VocabType;
typedef std::map<std::string, PrefixId> PrefixType;
typedef std::vector<std::string> IdType;
typedef VocabType::iterator VocabIterator;
typedef PrefixType::iterator PrefixIterator;
typedef IdType::iterator IdIterator;
```

Description:

- ▷ the struct `NGramData` is used to store all n-gram data in a single place. More detailed information on this struct can be found on page 87.
- ▷ the type `VocabType` defines the vocabulary structure, a mapping from word surface forms to a unique, numeric identifier. The `VocabId` type is defined as part of the `NgramTree` class.
- ▷ the type `PrefixType` defines the structure of the *prefix vocabulary*. As we have explained in chapter 3, we use n-gram prefixes to index a set of n-grams. Similar to the vocabulary which stores numeric ids for each single word, we store an id for each possible n-gram prefix. This helps to convert n-gram words into prefix keys very quickly.
- ▷ the type `IdType` is used to store the list of possible words inside the language model vocabulary. The corresponding, unique id is determined by the index position inside the `vector`. This allows fast retrieval of a word surface form given the word id without the need to instantiate a second `map`.
- ▷ the type `VocabIterator` defines an iterator over `VocabType`.
- ▷ the type `PrefixIterator` defines an iterator over `PrefixType`.
- ▷ the type `IdIterator` defines an iterator over `IdType`.

B.1.2 Public Interface

The `public` interface is shown below:

```
public:
    IndexedLM();
    ~IndexedLM();

    void clearCache();
    void loadIndex(const std::string& index_file);
    void loadVocab(const std::string& vocab_file);

    VocabId getUnknownId();
    VocabId getVocabId(const std::string& word);
    std::string getVocabString(VocabId id);

    NGramData getNgramScore(const std::string& ngram);
    NGramData getNgramScore(VocabId word, VocabId* context);

    IndexTree* getIndexTree();
```

```
VocabIterator getVocabBegin();
VocabIterator getVocabEnd();
IdIterator getIdsBegin();
IdIterator getIdsEnd();
```

Description:

- ▷ the constructor `IndexedLM()` creates a new instance of the `IndexedLM` class.
- ▷ the destructor `~IndexedLM()` releases all n-gram and index data and destroys the `IndexedLM` object.
- ▷ the method `void clearCache()` can be used to clear the internal n-gram cache. This could for instance be done once a sentence has been fully translated or after a given number of sentences have been translated.
- ▷ the method `void loadIndex(const std::string&)` takes care of loading an index from the given file which should have been generated by the `Indexer` discussed in chapter 3. After the index data has been loaded, the `IndexedLM` object can be actually used.
- ▷ the method `void loadVocab(const std::string&)` loads the language model vocabulary from the given file.
- ▷ the method `VocabId getUnknownId()` returns the numeric id for the unknown word. Usually, the unknown word is mapped to `0`.
- ▷ the method `VocabId getVocabId(const std::string&)` can be used to look up the numeric id for a given word. If the word is not contained within the vocabulary, the id of the unknown word is returned instead.
- ▷ the method `std::string getVocabString(VocabId)` returns the surface form of the word specified by the given word id. If the word id is invalid, the empty string `""` is returned.
- ▷ the method `NGramData getNgramScore(const std::string&` tries to look up the given string inside the language model. It will determine the maximum match and return the corresponding probability and backoff weight.
- ▷ the method `NGramData getNgramScore(VocabId, VocabId*)` is identical to the aforementioned `NGramData getNgramScore(const std::string&`. However it does not get an n-gram string but word ids.
- ▷ the method `IndexTree* getIndexTree()` returns a pointer to the root of the tree.
- ▷ the method `VocabIterator getVocabBegin()` returns an iterator to the first element

of the vocabulary `map`. This is used to allow the `LanguageModelIndexed` class to access the vocabulary data.

- ▷ the method `VocabIterator getVocabEnd()` returns an iterator to the end of the vocabulary `map`. Again, this is available for `LanguageModelIndexed`.
- ▷ the method `IdIterator getIdsBegin()` returns an iterator to the first element of the ids `vector`. As with the previous iterator methods, this allows the `LanguageModelIndexed` class to access the ids `map`.
- ▷ the method `IdIterator getIdsEnd()` returns an iterator pointing to the end of the ids `vector`. Again, this is available for `LanguageModelIndexed`.

B.1.3 Private Interface

The `private` interface is shown below:

```
private:
    NGramData loadNgram(IndexTree*, std::vector<VocabId>, unsigned int);
```

Description:

- ▷ the method `NGramData loadNgram(IndexTree*, std::vector<VocabId>, unsigned int)` tries to find an n-gram inside the language model data. The first parameter defines the subset inside the index which could contain the n-gram. The n-gram itself is specified by a `vector` of word ids. Finally, the length of the n-gram or some prefix of the n-gram is defined by an `integer`.

B.1.4 Data Members

```
IndexTree* index;
VocabType* vocab;
IdType* ids;
PrefixType* prefixes;
std::map<std::pair<VocabId, unsigned int>, PrefixId> word_to_prefix;
std::map<int, std::string> model_files;
std::map<int, std::fstream*> model_handles;
std::vector<int> model_gammas;
NGramTree* model_cache;
VocabIterator vocab_end;
const VocabId model_unknown;
```

Description:

- ▷ the `index` pointer stores the address of an `IndexTree` object. This object contains all index data. For more information on the `IndexTree` class, refer to page 88.
- ▷ the `vocab` pointer stores the address of the vocabulary `map`.
- ▷ the `ids` pointer stores the address of the id `vector`.
- ▷ the `prefixes` pointer stores the address of the prefix `map`.
- ▷ the `word_to_prefix` `map` stores the mapping from word surface forms to the corresponding n-gram prefix ids. This allows faster conversion from surface form to n-gram prefix.
- ▷ the `model_files` `map` stores the file ids and names of all index files.
- ▷ the `model_handles` `map` stores the `std::fstream*` pointers to the (opened) file objects.
- ▷ the `model_gammas` `vector` contains the Γ_i parameter set for the current index files.
- ▷ the `model_cache` pointer stores the address of an `NGramTree` object. This object collects n-gram data and improves overall system performance. More information on the `NGramTree` class is available on page 90.
- ▷ the `vocab_end` iterator points to the end of the vocabulary `map`. This is used to optimize access time by avoiding superfluous calls to `map::end()` which is static in our context.
- ▷ the `model_unknown` value represents the numeric id for the unknown word.

B.1.5 Struct: NGramData

All n-gram data is stored in `struct NGramData` variables. This includes both the conditional probability and the backoff weight of an n-gram entry, the size of the n-gram and a `bool` attribute which encodes the validity of the specific n-gram data object. In order to avoid any memory overhead we chose a `struct` instead of a dedicated `class` implementation.

The struct definition is shown below:

```
struct
NGramData {
    bool invalid;
    unsigned int size;
    double cp, bow;
};
```

B.2 Class: IndexTree

The `IndexTree` class has been built to store large collections of index entries in an efficient manner. Each node inside the index tree represents an n-gram prefix inside the index, defined by the corresponding prefix id. If there exists some continuation from the current node, a `map` of pointers to subsequent nodes is available.

Starting from the root of the index tree it is very easy to locate the n-gram subset information for any given n-gram. First, all words of the n-gram are converted to prefix ids which only requires lookups from a `map`. Afterwards, a single tree traversal will determine whether the given n-gram can be found within the language model or not. If the n-gram subset exists, the final node contains the relevant subset information which is of type `ExtendedIndexData`.

B.2.1 Typedefs

The `IndexTree` class defines the following types:

```
struct ExtendedIndexData { ... };
typedef unsigned long PrefixId;
```

Description:

- ▷ the struct `ExtendedIndexData` holds all information for a single index entry. More information on the actual definiton of this `struct` can be found on page 90.
- ▷ the type `PrefixId` is used to assign each possible n-gram prefix a unique, numeric id. This allows to implement fast `map` lookup of n-gram prefixes without the need to actually compute the prefix.

B.2.2 Public Interface

The `public` interface of the `IndexTree` class is shown below:

```
public:
    IndexTree();
    IndexTree(ExtendedIndexData*);
    ~IndexTree();

    bool root();
    unsigned int size();
    IndexTree* prev();
    IndexTree* next(PrefixId);
    IndexTree* insert(PrefixId);
```

```
IndexTree* insert(PrefixId, ExtendedIndexData*);
void update(ExtendedIndexData*);
ExtendedIndexData* data();
```

Description:

- ▷ the default constructor `IndexTree()` creates an "empty" index node which does not have any n-gram subset information attached.
- ▷ the constructor `IndexTree(ExtendedIndexData*)` allows to create a new index node and binds it to the subset information specified by the given `ExtendedIndexData*` pointer.
- ▷ the destructor ∼`IndexTree()` cares for correct deletion of the tree nodes and releases any allocated memory.
- ▷ the method `bool root()` checks whether the current index node is the root of the whole index tree or not. The root of the tree is the node which does not have any precedent nodes, i.e. the node where `prev() == NULL`.
- ▷ the method `unsigned int size()` returns the number of nodes of the index tree.
- ▷ the method `IndexTree* prev()` returns an `IndexTree*` pointer to the parent node.
- ▷ the method `IndexTree* next(PrefixId)` tries to find a child node of the current tree node with the given prefix id. If no such node exists inside the index tree, the empty pointer `NULL` is returned, otherwise the method will return the address of the next node.
- ▷ the method `IndexTree* insert(PrefixId)` can be used to insert a new node into the index tree. This node will become a child of the current tree node, its id will be the given prefix id. No subset information will be attached to the new node. The method will return an `IndexTree*` pointer to the new node.
- ▷ the method `IndexTree* insert(PrefixId, ExtendedIndexData*)` basically does the same as the previous except that it also attaches subset information to the new node. Effectively this creates a new, "active" node, i.e. a node carrying information. Again, an `IndexTree*` pointer to the new node is returned.
- ▷ the method `void update(ExtendedIndexData*)` updates the subset information of the current index node with the given `ExtendedIndexData*` pointer.
- ▷ the method `ExtendedIndexData* data()` returns an `ExtendedIndexData*` pointer to the subset information attached to an index node. If no subset information is available, the empty pointer `NULL` is returned instead.

B.2.3 Data Members

The available data members of the `IndexTree` class are listed below:

```
IndexTree* prev_ptr;
std::map<PrefixId, IndexTree*> next_ptr;
ExtendedIndexData index_data;
```

Description:

- ▷ the `prev_ptr` pointer stores the address of the parent of the current index node.
- ▷ the `next_ptr map` stores pointers to all subsequent index nodes.
- ▷ the `index_data struct` stores all subset information related to the current index node.

B.2.4 Struct: ExtendedIndexData

N-gram subset information is stored inside `ExtendedIndexData struct` variables. This includes the **id of the file** which contains the n-gram subset, the **position** inside this file and the **number of lines** which form the n-gram subset.

The struct definition is shown below:

```
struct
ExtendedIndexData {
    int file_id;
    unsigned long file_pos, line_count;
};
```

B.3 Class: NgramTree

The `NgramTree` class allows to store large sets of n-grams in a space efficient way. Each node inside the n-gram tree represents a word inside the vocabulary, defined by the corresponding word id. If there exists some n-gram continuation for this word id, a `map` of pointers to subsequent nodes is built. Starting from the root of the n-gram tree, it is then easily possible to search for any given n-gram. Only a single tree traversal is required to determine whether the n-gram is known or not.

Any n-gram inside the original language model is represented by some node within the n-gram tree. As it is possible that some n-gram prefixes are not available and do not carry any probability or backoff weight, we distinguish active and inactive nodes. An active tree node represents a valid n-gram inside the language model and carries the corresponding information. Inactive nodes carry no information.

B.3.1 Typedefs

The `NgramTree` class defines the following types:

```
typedef unsigned long VocabId;
```

Description:

- ▷ the type `VocabId` is used to assign numeric ids to words inside the vocabulary. The `unsigned long` type allows vocabularies of up to 4,294,967,294 words.

B.3.2 Public Interface

The `public` interface of the `NGramTree` class is shown below:

```
public:
    NGramTree();
    NGramTree(double, double);
    ~NGramTree();

    bool root();
    bool active();
    unsigned int size();
    unsigned int children();

    NGramTree* prev();
    NGramTree* next(VocabId);
    NGramTree* insert(VocabId);
    NGramTree* insert(VocabId, double, double);

    double cp();
    double bow();
    VocabId word();
    std::vector<VocabId> ngram();
```

Description:

- ▷ the default constructor `NGramTree()` creates a new node inside an n-gram tree. No n-gram information is attached to the new node hence this method is usually called to create intermediate nodes without any information.
- ▷ the constructor `NGramTree(double, double)` can also be used to create a new node inside an n-gram tree. Additionally, the n-gram data section of the node will be initialized using the given parameters.

- ▷ the destructor ∼`NGramTree()` releases any allocated memory, cleans up and destroys the current `NGramData` object.
- ▷ the method `bool root()` checks whether the current n-gram node is the root of the complete n-gram tree or not. The root of the tree is the node which does not have any precedent nodes, i.e. the node where `prev() == NULL`.
- ▷ the method `bool active()` checks whether the current n-gram node has n-gram data attached or not. This checks the `bool` attribute `active_node`.
- ▷ the method `unsigned int size()` returns the number of nodes of the n-gram tree spanned by the current n-gram node.
- ▷ the method `unsigned int children()` returns the number of children of the current n-gram node.
- ▷ the method `NGramTree* prev()` returns an `NGramTree*` pointer to the parent node.
- ▷ the method `NGramTree* next(VocabId)` tries to find a child node of the current tree node with the given word id. If no such node exists inside the n-gram tree, the empty pointer `NULL` is returned, otherwise the method will return the address of the next node.
- ▷ the method `NGramTree* insert(VocabId)` can be used to insert a new node into the n-gram tree. This node will become a child of the current tree node, its id will be the given word id. No n-gram data will be attached to the new node. The method will return an `NGramTree*` pointer to the new node.
- ▷ the method `NGramTree* insert(VocabId, double, double)` basically does the same as the previous except that it also attaches n-gram data to the new node. Effectively this creates a new, "active" node, i.e. a node carrying information. Again, an `NGramTree*` pointer to the new node is returned.
- ▷ the method `double cp()` returns the conditional probability of the current n-gram node or 0 if the current n-gram node is not active.
- ▷ the method `double bow()` returns the backoff weight of the current n-gram node or 0 if the current n-gram node is not active.
- ▷ the method `VocabId word()` returns the word id of the current n-gram node.
- ▷ the method `std::vector<VocabId> ngram()` returns the `vector` of word ids which form the n-gram starting from the root of the n-gram tree up to the current n-gram node. This method calls the `private` method `_ngram()` to compute this `vector`.

B.3.3 Private Interface

The `private` interface of the `NGramTree` class is shown below:

```
private:
    std::vector<VocabId> _ngram(std::vector<VocabId>&);
```

Description:

- ▷ the method `std::vector<VocabId> _ngram(std::vector<VocabId>&)` is an internal helper method which is called by `std::vector<VocabId> ngram()`.

B.3.4 Data Members

The available data members of the `NGramTree` class are listed below:

```
NGramTree* prev_ptr;
std::map<VocabId, NGramTree*> next_ptr;
double cp_value, bow_value;
VocabId word_id;
bool active_node;
```

Description:

- ▷ the `prev_ptr` pointer stores the address of the parent of the current n-gram node.
- ▷ the `next_ptr map` stores pointers to all subsequent n-gram nodes.
- ▷ the `cp_value` double contains the conditional probability of the current n-gram node.
- ▷ the `bow_value` double contains the backoff weight of the current n-gram node.
- ▷ the `word_id VocabId` contains the word id of the current n-gram node.
- ▷ the `active_node bool` is set to `true` if and only if the current n-gram node does have n-gram data attached.

Appendix C

Language Model Server Code

The language model server has been designed and implemented using `C++`. It is based upon the `LanguageModelServer` class and built using the `CmdLine` class and the `Debug` macros. This appendix will briefly introduce the nuts and bolts of the class design and provide further informations on the program implementation.

The source code is freely available at `http://www.cfedermann.de/diploma` under the license terms printed on page 73.

C.1 Class: LanguageModelServer

The `LanguageModelServer` class loads a language model into memory and makes its contents available via TCP communication or IPC shared memory methods. The server will take care of initializing the communication channels and then wait for client requests. It serves forever or until an explicit **SHUTDOWN** command is received.

If the server is started in `MIXED` mode, i.e. if both `TCP` and `IPC` client requests are supported, it will prefer `IPC` connections and handle them first. This will guarantee fastest performance for `IPC` requests. In order to support handling of these two communication channels, the `TCP` port is made non-blocking and both handler routines are called within the server loop.

C.1.1 Constants

The `LanguageModelServer` class defines the following constants:

```
const static int TCP_BUFFER_SIZE = 256;
const static int IPC_BUFFER_SIZE = 256 * 1024;
```

Description:

- ▷ the constant `TCP_BUFFER_SIZE` defines the buffer size for TCP connections. The maximum TCP buffer size is constrained by the maximum TCP packet size.
- ▷ the constant `IPC_BUFFER_SIZE` defines the buffer size for IPC connections. This should generally be larger than the TCP buffer size as large shared memory areas can help to improve language model server performance.

C.1.2 Typedefs

The `LanguageModelServer` class defines the following types:

```
enum Mode { ... };
enum Type { ... };
```

Description:

- ▷ the enum `Mode` encodes the language model server mode. Possible values are `IPC = 0`, `TCP = 1` or `MIXED = 2` depending on the actual server configuration.
- ▷ the enum `Type` is used to store information on the type of the language model which is handled by the language model server. At the moment, this can either be `SRI = 0` or `INDEXED = 1`.

C.1.3 Public Interface

The `public` interface is shown below:

```
public:
    LanguageModelServer(Mode mode, Type lm_type, int port,
        const std::string& model_file, int model_order);
    ~LanguageModelServer();

    bool is_usable();
    Mode get_mode();
    Type get_type();
    void shutdown();
    std::string handle_request(const std::string& request);

    void tcp_handler();
    void ipc_handler();
```

Description:

- ▷ the constructor `LanguageModelServer(Mode, Type, int, const std::string&, int)` creates a new instance of the `LanguageModelServer` class. The actual server configuration is set by the constructor arguments. In order to create an object, the constructor needs to know which communcation channel(s) it has to use, the port or key number, what kind of language model is served, where the language model file can be found and the order of the model file.
- ▷ the destructor `~LanguageModelServer()` closes all open connections, releases shared memory areas, cleans up and destroys a `LanguagModelServer` class instance.
- ▷ the method `bool is_usable()` can be used to query the status of a `LanguageModelServer` object. If the server is up and running, `true` is returned, otherwise `false`.
- ▷ the method `Mode get_mode()` returns the server mode.
- ▷ the method `Type get_type()` returns the type of the language model which is made available by the `LanguageModelServer` object.
- ▷ the method `void shutdown()` activates the shutdown procedure of the language model server. This exits the main server loop and calls the destructor. Afterwards, no connections to the server are possible until a new server instance has been created.
- ▷ the method `std::string handle_request(const std::string&)` takes a client request and computes the corresponding response to the client.
- ▷ the method `void tcp_handler()` checks if there exist incoming `TCP` client requests and handles them if possible. This internally calls the `handle_request()` method to compute the response and afterwards sends back the result to the client.
- ▷ the method `void ipc_handler()` checks if any `IPC` client has written a request to shared memory. If that is the case, `handle_request()` is called and a response for the client is generated and written back to shared memory. Deletion of the client request data in the shared memory block signals that the result has been written and causes the client to copy the response.

C.1.4 Private Interface

The `private` interface looks like this:

```
private:
    LanguageModelServer();

    bool tcp_init(int port);
    void tcp_exit();

    bool ipc_init(int port);
    void ipc_exit();
```

Description:

- ▷ the default constructor `LanguageModelServer()` is private to prevent any call to it.
- ▷ the method `bool tcp_init(int)` takes care the initialization of the `TCP` connection. It gets the port as `integer` parameter, opens the server socket and binds it to the given port. The method returns `true` if the connection has been established successfully and `false` if some error occured. The server socket is made **non-blocking** for `MIXED` language model servers to support requests from both `TCP` and `IPC` clients.
- ▷ the method `void tcp_exit()` closes the server socket and cleans up. Afterwards no `TCP` requests can be handled until a new connection has been created.
- ▷ the method `bool ipc_init(int)` creates the shared memory areas and gets a semaphore to control read/write access via `IPC`. The `integer` parameter is used to compute two unique numeric ids for the shared memory blocks. Again, `true` is returned if `IPC` setup was successful, `false` if some error occured.

C.1.5 General Data Members

The available data members of the `Indexer` class are listed below:

```
Mode server_mode;
Type server_type;
LanguageModelSingleFactor* server_lm;
bool server_usable;
```

Description of general attributes:

- ▷ the `server_mode Mode` attribute stores the server mode.
- ▷ the `server_type Type` attribute stores the type of the language model.
- ▷ the `server_lm` pointer stores the address of the actual language model instance. It is set by the `LanguageModelServer( ... )` constructor and cleared by the destructor `~LanguageModelServer()`.
- ▷ the `server_usable bool` attribute encodes the status of the language model server.

C.1.6 TCP Data Members

The available data members used for `TCP` communication are listed below:

```
// tcp attributes
int server_socket, client_socket;
struct sockaddr_in server_addr, client_addr;
int addr_size, data_size;
char tcp_data[TCP_BUFFER_SIZE];
```

Description of `TCP` attributes:

- ▷ the `server_socket` and `client_socket int` attributes are used to establish a `TCP` connection between client and server.
- ▷ the `server_addr` and `client_addr struct` attributes are used internally to store address details for client and server.
- ▷ the `addr_size` and `data_size int` attributes store the size of address data structures and the number of bytes received from the client.
- ▷ the `tcp_data char` buffer is used to send and receive data from server to client.

C.1.7 IPC Data Members

The available data members used for `IPC` communication are listed below:

```
// ipc attributes
key_t server_key, client_key;
int server_semid, server_shmid, client_shmid;
struct sembuf server_semop[2];
char *server_data, *client_data;
```

Description of `IPC` attributes:

- ▷ the `server_key` and `client_key` key_t attributes are used to create unique identifiers for server and client.
- ▷ the `server_semid`, `server_shmid` and `client_shmid` `int` attributes are used to identify the shared memory areas and the semaphore for both client and server.
- ▷ the `server_semop` `struct` attribute is used to perform semaphore operations.
- ▷ the `server_data` and `client_data` pointers store the addresses of the shared memory blocks for server and client.

C.2 Program: Main Loop

C.2.1 Code

```
 1: try {
 2:     LanguageModelServer* lmserver = new LanguageModelServer( ... );
 3:
 4:     while(lmserver->is_usable()) {
 5:         switch(lmserver->get_mode()) {
 6:             case LanguageModelServer::IPC:
 7:                 lmserver->ipc_handler();
 8:                 break;
 9:
10:             case LanguageModelServer::TCP:
11:                 lmserver->tcp_handler();
12:                 break;
13:
14:             case LanguageModelServer::MIXED:
15:                 lmserver->ipc_handler();
16:                 lmserver->tcp_handler();
17:                 break;
18:         }
19:     }
20:
21:     delete lmserver;
22: }
23: catch (std::string e) {
24:     std::cout << "[exception] " << e << std::endl;
25:     return -1;
26: }
```

Description:

- ▷ lines 2-21 of the main loop are embedded into a `try ... catch` construct which catches all exceptions that might be thrown during program execution.
- ▷ line 2 creates a new `LanguageModelServer` instance and makes it available using the `lmserver` pointer.
- ▷ lines 4-19 form the actual server main loop which continues to wait for requests while the server is usable. This is queried using the `is_usable()` method.
- ▷ lines 5-18 subdivide the server main loop into handler sections for each of the possible server modes. The current server mode is returned by `get_mode()`.
- ▷ lines 6-8 represent the `IPC` handler.
- ▷ lines 10-12 represent the `TCP` handler.
- ▷ lines 14-17 represent the `MIXED` handler which actually is a combination of both the `TCP` and the `IPC` handlers.

C.3 TCP Implementation

If the language model server is configured to handle incoming `TCP` requests, a normal `TCP` server is started. For this, a server socket is created and bound to the given server port. The language model server will listen on this port and wait for any incoming `TCP` clients requests.

Messages are exchanged using a small char buffer which may contain up to `TCP_BUFFER_SIZE` bytes. The small size of this buffer also explains the main disadvantage of the `TCP` server mode: it will take more time than the `IPC` mode to transmit large results back to the client.

C.4 IPC Implementation

The language model server implements `IPC` communication using two designated blocks of shared memory and a semaphore to manage read/write access to these. In order to identify memory two unique keys are generated when the `IPC` mode is initialized.

Clients will wait until the semaphore is free to be used and lock it once they have been granted access. The server will then process the query and write back the result to memory. The client finally releases the semaphore which enables other clients to send requests.

Appendix D

Tables

The tables on the following pages provide detailed information on the different indexing methods which were defined and evaluated in chapter 3. Each table lists the **index size** and the corresponding **compression rate**. Afterwards the **number of large subsets**, the **subset count**, the resulting **large subset rate**, and last but not least the final **compression gain** are given. All these values are evaluated for several **subset thresholds**.

The various indexing methods have been evaluated on a 5-gram English language model containing 10,037,283 tokens, the column which shows the best combination of subset threshold and compression gain is printed in bold face.

Table D.1: Increasing Indexing with $\Gamma = [1, 0, 0, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	71							
compression rate	100.00%							
large subsets	61	59	55	53	50	47	42	**34**
subset count	10,037,181	10,037,027	10,036,345	10,035,603	10,033,195	10,028,742	10,009,452	**9,952,413**
large subset rate	100.00%	100.00%	99.99%	99.98%	99.96%	99.91%	99.72%	**99.15%**
compression gain	0.00%	0.00%	0.01%	0.02%	0.04%	0.08%	0.28%	**0.84%**

Table D.2: Increasing Indexing with $\Gamma = [1, 2, 0, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	22,789							
compression rate	99.77%							
large subsets	5,848	4,789	3,549	2,650	1,800	879	421	**194**
subset count	9,907,819	9,832,027	9,629,883	9,306,774	8,695,968	7,227,342	5,632,371	**4,023,928**
large subset rate	98.71%	97.96%	95.94%	92.72%	86.64%	72.00%	56.11%	**40.09%**
compression gain	1.06%	1.82%	3.83%	7.05%	13.14%	27.77%	43.66%	**59.68%**

Table D.3: Increasing Indexing with $\Gamma = [1, 2, 3, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	603,637							
compression rate	93.99%							
large subsets	28,618	14,452	5,670	2,661	1,073	219	54	**0**
subset count	6,650,123	5,658,116	4,312,623	3,263,117	2,157,998	901,438	353,170	**0**
large subset rate	66.25%	56.37%	42.97%	32.51%	21.50%	8.98%	3.52%	**0.00%**
compression gain	27.73%	37.62%	51.02%	61.48%	72.49%	85.01%	90.47%	**93.99**

Table D.4: Increasing Indexing with $\Gamma = [1, 2, 3, 4, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	2,273,241							
compression rate	77.35%							
large subsets	14,436	6,601	2,812	1,470	626	135	38	**0**
subset count	3,486,552	2,945,120	2,372,225	1,896,059	1,302,051	577,177	253,606	**0**
large subset rate	34.74%	29.34%	23.63%	18.89%	12.97%	5.75%	2.53%	**0.00%**
compression gain	42.62%	48.01%	53.72%	58.46%	64.38%	71.60%	74.83%	**77.35%**

Table D.5: Increasing Indexing with $\Gamma = [1, 2, 3, 4, 5]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	4,104,917							
compression rate	59.10%							
large subsets	10,944	5,424	2,642	1,445	622	135	38	**0**
subset count	3,111,001	2,728,780	2,303,750	1,876,015	1,296,372	577,177	253,606	**0**
large subset rate	30.99%	27.19%	22.95%	18.69%	12.92%	5.75%	2.53%	**0.00%**
compression gain	28.11%	31.92%	36.15%	40.41%	46.19%	53.35%	56.58%	**59.10%**

Table D.6: Decreasing Indexing with $\Gamma = [1, 0, 0, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	71							
compression rate	100.00%							
large subsets	61	59	55	53	50	47	42	**34**
subset count	10,037,181	10,037,027	10,036,345	10,035,603	10,033,195	10,028,742	10,009,452	**9,952,413**
large subset rate	100.00%	100.00%	99.99%	99.98%	99.96%	99.91%	99.72%	**99.15%**
compression gain	0.00%	0.00%	0.01%	0.02%	0.04%	0.08%	0.28%	**0.84%**

Table D.7: Decreasing Indexing with $\Gamma = [2, 1, 0, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size					25,965			
compression rate					99.74%			
large subsets	6,183	4,917	3,515	2,640	1,766	850	407	**171**
subset count	9,885,519	9,794,506	9,568,628	9,257,096	8,634,007	7,162,217	5,591,901	**3,945,685**
large subset rate	98.49%	97.58%	95.33%	92.23%	86.02%	71.36%	55.71%	**39.31%**
compression gain	1.25%	2.16%	4.41%	7.51%	13.72%	28.39%	44.03%	**60.43%**

Table D.8: Decreasing Indexing with $\Gamma = [3, 2, 1, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size					870,981			
compression rate					91.32%			
large subsets	32,182	14,206	4,229	1,554	541	109	21	**0**
subset count	5,439,109	4,186,632	2,668,544	1,745,449	1,064,872	423,944	134,718	**0**
large subset rate	54.19%	41.71%	26.59%	17.39%	10.61%	4.22%	1.34%	**0.00%**
compression gain	37.13%	49.61%	64.74%	73.93%	80.71%	87.10%	89.98%	**91.32%**

Table D.9: Decreasing Indexing with $\Gamma = [4, 3, 2, 1, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size					4,035,284			
compression rate					59.80%			
large subsets	7,786	2,106	336	58	6	**0**	0	0
subset count	771,143	384,384	133,373	41,568	6,941	**0**	0	0
large subset rate	7.68%	3.83%	1.33%	0.41%	0.07%	**0.00%**	0.00%	0.00%
compression gain	52.11%	55.97%	58.47%	59.38%	59.73%	**59.80%**	59.80%	59.80%

Table D.10: Decreasing Indexing with $\Gamma = [5, 4, 3, 2, 1]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	7,468,296							
compression rate	25.59%							
large subsets	615	140	40	10	**0**	0	0	0
subset count	63,974	30,717	16,818	6,459	**0**	0	0	0
large subset rate	0.64%	0.31%	0.17%	0.06%	**0.00%**	0.00%	0.00%	0.00%
compression gain	24.96%	25.29%	25.43%	25.53%	**25.59%**	25.59%	25.59%	25.59%

Table D.11: Uniform Indexing with $\Gamma = [1, 1, 1, 1, 1]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	1,020,477							
compression rate	89.83%							
large subsets	22,519	11,238	4,333	1,935	846	342	172	**41**
subset count	6,287,563	5,496,679	4,435,187	3,598,220	2,847,080	2,091,150	1,478,413	**538,948**
large subset rate	62.64%	54.76%	44.19%	35.85%	28.37%	20.83%	14.73%	**5.37%**
compression gain	27.19%	35.07%	45.65%	53.98%	61.47%	69.00%	75.10%	**84.46%**

Table D.12: Uniform Indexing with $\Gamma = [2, 2, 2, 2, 2]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	4,029,444							
compression rate	59.86%							
large subsets	16,102	7,155	2,019	690	160	21	**0**	0
subset count	2,478,529	1,852,836	1,076,425	621,432	258,950	71,178	**0**	0
large subset rate	24.69%	18.46%	10.72%	6.19%	2.58%	0.71%	**0.00%**	0.00%
compression gain	35.16%	41.40%	49.13%	53.66%	57.28%	59.15%	**59.86%**	59.86%

Table D.13: Uniform Indexing with $\Gamma = [3, 3, 3, 3, 3]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size					6,571,810			
compression rate					34.53%			
large subsets	4,234	1,239	190	55	4	**0**	0	0
subset count	437,486	234,120	82,886	38,013	4,419	**0**	0	0
large subset rate	4.36%	2.33%	0.83%	0.38%	0.04%	**0.00%**	0.00%	0.00%
compression gain	30.17%	32.19%	33.70%	34.15%	34.48%	**34.53%**	34.53%	34.53%

Table D.14: Uniform Indexing with $\Gamma = [4, 4, 4, 4, 4]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size					7,947,135			
compression rate					20.82%			
large subsets	769	227	58	16	**0**	0	0	0
subset count	87,132	50,198	26,289	11,104	**0**	0	0	0
large subset rate	0.87%	0.50%	0.26%	0.11%	**0.00%**	0.00%	0.00%	0.00%
compression gain	19.96%	20.32%	20.56%	20.71%	**20.82%**	20.82%	20.82%	20.82%

Table D.15: Uniform Indexing with $\Gamma = [5, 5, 5, 5, 5]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size					8,652,287			
compression rate					13.80%			
large subsets	403	48	9	4	**0**	0	0	0
subset count	35,942	9,534	3,873	2,330	**0**	0	0	0
large subset rate	0.36%	0.09%	0.04%	0.02%	**0.00%**	0.00%	0.00%	0.00%
compression gain	13.44%	13.70%	13.76%	13.78%	**13.80%**	13.80%	13.80%	13.80%

Table D.16: Custom Indexing with $\Gamma = [1, 1, 1, 1, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size					289,854			
compression rate					97.11%			
large subsets	30,487	16,641	6,405	2,576	955	348	172	**41**
subset count	8,011,439	7,027,179	5,436,976	4,103,280	2,999,655	2,112,103	1,478,413	**538,948**
large subset rate	79.82%	70.01%	54.17%	40.88%	29.89%	21.04%	14.73%	**5.37%**
compression gain	17.30%	27.10%	42.94%	56.23%	67.23%	76.07%	82.38%	**91.74%**

Table D.17: Custom Indexing with $\Gamma = [1, 1, 1, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size					42,297			
compression rate					99.58%			
large subsets	12,882	9,589	6,073	3,967	2,264	892	362	**87**
subset count	9,738,013	9,499,662	8,932,165	8,176,414	6,964,123	4,843,544	2,982,977	**1,099,108**
large subset rate	97.02%	94.64%	88.99%	81.46%	69.38%	48.26%	29.72%	**10.95%**
compression gain	2.56%	4.93%	10.59%	18.12%	30.20%	51.32%	69.86%	**88.63%**

Table D.18: Custom Indexing with $\Gamma = [1, 1, 0, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size					3,188			
compression rate					99.97%			
large subsets	1,725	1,483	1,149	928	725	516	382	**260**
subset count	10,020,854	10,003,090	9,949,436	9,871,776	9,728,160	9,402,176	8,918,805	**8,025,557**
large subset rate	99.84%	99.66%	99.12%	98.35%	96.92%	93.67%	88.86%	**79.96%**
compression gain	0.13%	0.31%	0.84%	1.62%	3.05%	6.30%	11.11%	**20.01%**

Table D.19: Custom Indexing with $\Gamma = [2, 2, 2, 2, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	2,313,163							
compression rate	76.95%							
large subsets	20,014	8,369	2,228	726	165	21	**0**	0
subset count	2,894,843	2,085,157	1,161,895	650,138	267,007	71,178	**0**	0
large subset rate	28.84%	20.77%	11.58%	6.48%	2.66%	0.71%	**0.00%**	0.00%
compression gain	48.11%	56.18%	65.38%	70.48%	74.29%	76.25%	**76.95%**	76.95%

Table D.20: Custom Indexing with $\Gamma = [2, 2, 2, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	759,912							
compression rate	92.43%							
large subsets	34,307	16,073	4,924	1,765	514	84	10	**0**
subset count	5,845,849	4,563,817	2,860,281	1,779,039	926,112	305,601	59,693	**0**
large subset rate	58.24%	45.47%	28.50%	17.72%	9.23%	3.04%	0.59%	**0.00%**
compression gain	34.19%	46.96%	63.93%	74.70%	83.20%	89.38%	91.83%	**92.43%**

Table D.21: Custom Indexing with $\Gamma = [2, 2, 0, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	92,705							
compression rate	99.08%							
large subsets	17,413	11,981	6,310	3,548	1,875	662	239	**84**
subset count	9,509,778	9,117,531	8,213,923	7,245,100	6,081,564	4,210,389	2,752,989	**1,692,081**
large subset rate	94.74%	90.84%	81.83%	72.18%	60.59%	41.95%	27.43%	**16.86%**
compression gain	4.33%	8.24%	17.24%	26.89%	38.49%	57.13%	71.65%	**82.22%**

Table D.22: Custom Indexing with $\Gamma = [3, 3, 3, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	2,269,882							
compression rate	77.39%							
large subsets	18,019	6,760	1,655	561	162	12	**0**	0
subset count	2,438,800	1,665,111	905,837	529,170	257,560	39,818	**0**	0
large subset rate	24.30%	16.59%	9.02%	5.27%	2.57%	0.40%	**0.00%**	0.00%
compression gain	53.09%	60.80%	68.36%	72.11%	74.82%	76.99%	**77.39%**	77.39%

Table D.23: Custom Indexing with $\Gamma = [3, 3, 0, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	677,224							
compression rate	93.25%							
large subsets	24,649	12,897	5,242	2,520	1,117	318	105	**37**
subset count	7,078,618	6,254,286	5,074,304	4,121,718	3,149,164	1,930,825	1,213,697	**746,685**
large subset rate	70.52%	62.31%	50.55%	41.06%	31.37%	19.24%	12.09%	**7.44%**
compression gain	22.73%	30.94%	42.70%	52.19%	61.88%	74.02%	81.16%	**85.81%**

Table D.24: Custom Indexing with $\Gamma = [4, 4, 0, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	1,323,716							
compression rate	86.81%							
large subsets	21,962	11,363	4,533	2,101	914	230	83	**33**
subset count	5,963,415	5,221,823	4,169,174	3,316,469	2,499,251	1,465,875	969,210	**633,039**
large subset rate	59.41%	52.02%	41.54%	33.04%	24.90%	14.60%	9.66%	**6.31%**
compression gain	27.40%	34.79%	45.28%	53.77%	61.91%	72.21%	77.16%	**80.51%**

Table D.25: Custom Indexing with $\Gamma = [2, 1, 1, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	154,813							
compression rate	98.46%							
large subsets	26,687	16,870	8,190	4,189	1,732	398	105	**6**
subset count	8,905,373	8,200,391	6,827,517	5,423,549	3,710,212	1,723,013	710,573	**70,708**
large subset rate	88.72%	81.70%	68.02%	54.03%	36.96%	17.17%	7.08%	**0.70%**
compression gain	9.73%	16.76%	30.44%	44.42%	61.49%	81.29%	91.38%	**97.75%**

Table D.26: Custom Indexing with $\Gamma = [3, 2, 2, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	1,281,536							
compression rate	87.23%							
large subsets	28,328	11,491	3,107	1,019	312	40	6	**0**
subset count	4,151,390	2,987,910	1,732,336	1,018,458	543,812	148,780	36,406	**0**
large subset rate	41.36%	29.77%	17.26%	10.15%	5.42%	1.48%	0.36%	**0.00%**
compression gain	45.87%	57.46%	69.97%	77.09%	81.81%	85.75%	86.87%	**87.23%**

Table D.27: Custom Indexing with $\Gamma = [3, 1, 1, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size	409,178							
compression rate	95.92%							
large subsets	33,546	17,970	6,669	2,771	999	234	49	**4**
subset count	7,488,193	6,385,161	4,637,587	3,285,455	2,075,593	942,805	326,034	**47,653**
large subset rate	74.60%	63.61%	46.20%	32.73%	20.68%	9.39%	3.25%	**0.47%**
compression gain	21.32%	32.31%	49.72%	63.19%	75.24%	86.53%	92.68%	**95.45%**

Table D.28: Custom Indexing with $\Gamma = [3, 1, 0, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size					111,273			
compression rate					98.89%			
large subsets	15,166	10,425	5,669	3,229	1,662	589	279	**108**
subset count	9,435,620	9,096,511	8,339,284	7,477,090	6,369,431	4,740,474	3,663,951	**2,477,582**
large subset rate	94.01%	90.63%	83.08%	74.49%	63.46%	47.23%	36.50%	**24.68%**
compression gain	4.89%	8.26%	15.81%	24.40%	35.43%	51.66%	62.39%	**74.21%**

Table D.29: Custom Indexing with $\Gamma = [1, 2, 2, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size					350,259			
compression rate					96.51%			
large subsets	30,999	16,595	6,762	3,124	1,236	267	71	**0**
subset count	7,659,580	6,641,427	5,122,894	3,853,868	2,543,401	1,121,270	469,308	**0**
large subset rate	76.31%	66.17%	51.04%	38.40%	25.34%	11.17%	4.68%	**0.00%**
compression gain	20.20%	30.34%	45.47%	58.11%	71.17%	85.34%	91.83%	**96.51%**

Table D.30: Custom Indexing with $\Gamma = [2, 3, 0, 0, 0]$

subset threshold	**50**	**100**	**250**	**500**	**1,000**	**2,500**	**5,000**	**10,000**
index size					283,784			
compression rate					97.17%			
large subsets	24,582	13,857	5,935	3,029	1,429	467	152	**57**
subset count	8,388,899	7,631,076	6,411,093	5,399,205	4,301,286	2,827,194	1,741,400	**1,093,170**
large subset rate	83.58%	76.03%	63.87%	53.79%	42.85%	28.17%	17.35%	**10.89%**
compression gain	13.60%	21.15%	33.30%	43.38%	54.32%	69.01%	79.82%	**86.28%**

Bibliography

[Brants et al., 2007] Brants, T., Popat, A. C., Xu, P., Och, F. J., and Dean, J. (2007). Large Language Models in Machine Translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 858–867.

[Brown et al., 1990] Brown, P. F., Cocke, J., Della Pietra, S. A., Della Pietra, V. J., Jelinek, F., Lafferty, J. D., Mercer, R. L., and Roossin, P. S. (1990). A Statistical Approach to Machine Translation. *Computational Linguistics*, 16(2):79–85.

[Brown et al., 1995] Brown, P. F., Cocke, J., Della Pietra, S. A., Della Pietra, V. J., Jelinek, F., Lai, J. C., and Mercer, R. L. (1995). Method and system for natural language translation. U.S. Patent 5,477,451.

[Brown et al., 1993] Brown, P. F., Della Pietra, S. A., Della Pietra, V. J., and Mercer, R. L. (1993). The Mathematics of Statistical Machine Translation: Parameter Estimation. *Computational Linguistics*, 19(2):263–311.

[Callison-Burch and Koehn, 2005] Callison-Burch, C. and Koehn, P. (2005). Introduction to Statistical Machine Translation. In *European Summer School for Language and Logic (ESSLLI)*.

[Callison-Burch et al., 2006] Callison-Burch, C., Osborne, M., and Koehn, P. (2006). Re-evaluating the Role of BLEU in Machine Translation Research. In *Proceedings of EACL*.

[Doddington, 2002] Doddington, G. (2002). The NIST Automated Measure and Its Relation to IBM's BLEU. In *Proceedings of LREC-2002 Workshop on Machine Translation Evaluation: Human Evaluators Meet Automated Metrics*, Gran Canaria, Spain.

[Hutchins, 1986] Hutchins, W. J. (1986). *Machine Translation: Past, Present, Future*. Ellis Horwood Series in Computers and their Applications. Ellis Horwood, Chichester, UK.

[Hutchins, 2002] Hutchins, W. J. (2002). Machine translation today and tomorrow. In *Computerlinguistik: was geht, was kommt?*, pages 159–162. Gardez!, Sankt Augustin, Germany.

[Jelinek, 1999] Jelinek, F. (1999). *Statistical Methods for Speech Recognition*. MIT Press, Cambridge, MA.

[Kirchhoff and Yang, 2005] Kirchhoff, K. and Yang, M. (2005). Improved Language Modeling for Statistical Machine Translation. In *Proceedings of the ACL Workshop on Building and Using Parallel Texts*, pages 125–128. Association for Computational Linguistics.

[Knight, 1999] Knight, K. (1999). A statistical MT tutorial workbook. Prepared for the 1999 JHU Summer Workshop.

[Koehn, 2004a] Koehn, P. (2004a). Pharaoh: A Beam Search Decoder for Phrase-Based Statistical Machine Translation Models. In Frederking, R. E. and Taylor, K. B., editors, *Machine Translation: From Real Users to Research*, pages 115–124. 6th Conference of the Association for Machine Translation in the Americas, AMTA-2004, Washington DC, USA, September 28 - October 2, 2004, Springer, Berlin, Germany.

[Koehn, 2004b] Koehn, P. (2004b). The Foundation for Statistical Machine Translation at MIT. In *Proceedings of Machine Translation Evaluation Workshop 2004*.

[Koehn, 2005] Koehn, P. (2005). Europarl: A Parallel Corpus for Statistical Machine Translation. In *Proceedings of MT Summit X, Phuket, Thailand*. Asia-Pacific Association for Machine Translation (AAMT).

[Koehn et al., 2007] Koehn, P., Hoang, H., Birch, A., Callison-Burch, C., Federico, M., Bertoldi, N., Cowan, B., Shen, W., Moran, C., Zens, R., Dyer, C., Bojar, O., Constantin, A., and Herbst, E. (2007). Moses: Open Source Toolkit for Statistical Machine Translation. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions*, pages 177–180, Prague, Czech Republic. Association for Computational Linguistics.

[Koehn and Monz, 2005] Koehn, P. and Monz, C. (2005). Shared Task: Statistical Machine Translation between European Languages. In *Proceedings of the ACL Workshop on Building and Using Parallel Texts*.

[Koehn et al., 2003] Koehn, P., Och, F. J., and Marcu, D. (2003). Statistical Phrase-Based Translation. In *NAACL '03: Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology*, pages 48–54, Edmonton, Canada. Association for Computational Linguistics.

[Och, 1999] Och, F. J. (1999). An Efficient Method for Determining Bilingual Word Classes. In *EACL99*, pages 71–76, Bergen, Norway.

[Och, 2002] Och, F. J. (2002). *Statistical Machine Translation: From Since-Word Models to Alignment Templates*. PhD thesis, RWTH Aachen Department of Computer Science, Aachen, Germany.

[Och, 2003] Och, F. J. (2003). Minimum Error Rate Training for Statistical Machine Translation. In *Proceedings of ACL*, Sapporo, Japan.

[Och et al., 2003] Och, F. J., Gildea, D., Khudanpur, S., Sarkar, A., Yamada, K., Fraser, A., Kumar, S., Shen, L., Smith, D., Eng, K., Jain, V., Jin, Z., and Radev, D. (2003). Syntax for Statistical Machine Translation. Johns Hopkins University 2003 Summer Workshop on Language Engineering, Center for Language and Speech Processing, Baltimore, MD, USA.

[Och and Ney, 2003] Och, F. J. and Ney, H. (2003). A Systematic Comparison of Various Statistical Alignment Models. *Computational Linguistics*, 29(1):19–51.

[Och et al., 1999] Och, F. J., Tillmann, C., and Ney, H. (1999). Improved Alignment Models for Statistical Machine Translation. In *EMNLP '99*, pages 20–28.

[Papineni et al., 2001] Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2001). Bleu: A Method for Automatic Evaluation of Machine Translation. IBM Research Report RC22176 (W0109-022), IBM.

[Papineni et al., 2002] Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of ACL*.

[Stolcke, 2002] Stolcke, A. (2002). SRILM - An Extensible Language Modeling Toolkit. In *Proceedings of the International Conference on Spoken Language Processing*, Denver, Colorado.

[Vauquois, 1968] Vauquois, B. (1968). A survey of formal grammars and algorithms for recognition and transformation in machine translation. In *Proceedings of the IFIP Congress 6*, pages 254–260.