UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

WeST People and Knowledge Networks
Institute for Web Science and Technologies

# FAST AND NON-APPROXIMATIVE LANGUAGE MODEL PREFIXQUERIES FOR WORD PREDICTION USING TOP-K JOINING TECHNIQUES

BACHELORARBEIT

zur Erlangung des Grades eines Bachelor of Science (B. Sc.)
im Studiengang Informatik

vorgelegt von

LUKAS SCHMELZEISEN
lukas@uni-koblenz.de

| | |
|---|---|
| Erstgutachter: | Prof. Dr. Steffen Staab<br>Institute for Web Science and Technologies |
| Zweitgutachter: | René Pickhardt<br>Institute for Web Science and Technologies |

Koblenz, im August 2015

## ABSTRACT

Write Abstract.

Next word prediction is the task of guessing the next word a user intends to type from the words they have already entered. Traditionally this problem is solved by calculating an argmax of language model probabilities for all words in a vocabulary. However this approach is slow and becomes linearly worse with increasing vocabulary size. This thesis proposes two independent optimizations. First, a novel approach is presented that allows to move a part of probability calculation into a precomputation step. Secondly it is shown how to apply top-k joining techniques to word prediction to avoid enumerating all words in the vocabulary. Using both optimizations sub-millisecond next word prediction time is achieved.

The concern of this thesis is solving noisy channel type queries that are based on statistical language model probabilities. Our novel approach is to formulate language models as weighted sum functions on occurrence counts. As these weighted sum functions are monotone, we can then employ various top-k joining techniques, to efficiently find those arguments that maximize the noisy channel's probability. This approach can be used for all noisy channel type queries, that can be expressed as monotone scoring functions. We will present our method by implementing next word prediction with it.

## ZUSAMMENFASSUNG

Übersetzte Abtract auf Deutsch.

License!

Dedication

Use etoolbox for ifthenelse in class.

Kill warnings.

Rewrite math package. Use left/right braces for set.

Use subequations more often?

Check for typographical errors before submission.

- Check if all ellipsis "..." are done using `\ldots`.
- Check if words ending in "s" require possessive apostrophe.

Keywords?

Section capitalization

# CONTENTS

# 1 INTRODUCTION

*Word prediction* or autocompletion is the art of inferring the next word (or words) a user intents to type from the previous words they have entered. The user can either select a completion, or in the case that it did not match, type the next indented character, with which more accurate completions can be given.

In the context of *natural language processing* the prediction requires understanding of the user's language. A successful general purpose word prediction will have knowledge about a language's grammar, word frequencies, idioms, and much more. This is in contrast to simpler solutions, like selecting the best matching word from a list of previously entered ones in a word processing software.

Word prediction has many benefits that become evident by its recent adoption in many smartphone keyboards[1]. Its most obvious utilization is to speed up text entry, where it is most beneficial to slow typists, or typists on a slow medium (Trnka et al. 2007). Furthermore word prediction has been shown to reduce spelling or grammar mistakes for users with spelling disorders (Newell et al. 1992). But it may also be of used as a method to start computing answers to a user query that they have not even finished typing [Citation needed].

For more benefits and citations read http://trace.wisc.edu/docs/wordprediction2001/.

Nowadays word prediction is a part of various software systems. Although word prediction has been primarily used to augment the communication of users with a difficulty to express themselves verbally (Swiffin et al. 1987; Trnka 2011), the benefits of a strong word prediction should be obvious to anyone typing large amounts of text on a computer: be it the list of possible queries, offered by popular search engines such as Google, or the next word suggestions that many modern mobile phones provide, to speed up the slow two-thumbs typing.

This thesis will address the problem of *next* word prediction. Our point of view is that we predict only one following word, although the techniques described in this thesis can also be applied to the prediction of whole sequences. Good interfaces usually suggest a list of next word predictions and allow the user to select from these. This is done to mitigate the error, should the top prediction turn out not to match the user's intend.

We formalize the task of next word prediction as follows:

$$\mathrm{NWP}_p^k(h) = \underset{\substack{w\in\Sigma,\\ p \text{ prefix of } w}}{\arg\max{}^\star_k}\; P\left(w \,|\, h\right) \tag{1.1}$$

Our task is such: given a history $h$ of entered words and an already entered prefix $p$ of the intended word, we predict $k$ completions. Of

1 For example the SwiftKey (http://swiftkey.com) or Swype (http://www.swype.com) keyboard applications (both visited on Aug. 1, 2015).

course $p$ may also be empty which corresponds to the case that the user has not yet typed any prefix. We subsume next word prediction with or without a prefix under the title of *prefix queries*, with a possibly empty prefix. We solve this task by finding the $k$ words $w$ in the vocabulary $\Sigma$, the set of all words, that maximize the probability $P(w \mid h)$. That probability models the likelihood of a word $w$ following the history $h$. $\arg\max_k^\star$ is a helper used to model the $k$ words that maximize the probability and an extension to the conventional $\arg\max$ function:

$$\arg\max_{x}{}_1 f(x) = \{x \mid \forall y : f(y) \leqslant f(x)\} \quad (1.2a)$$

$$\arg\max_{x}{}_k f(x) = \{x \mid \forall y \notin \arg\max_{z}{}_{k-1}^\star f(z) : f(y) \leqslant f(x)\} \quad (1.2b)$$

$$\arg\max_{x}{}_k^\star f(x) = \bigcup_{1 \leqslant i \leqslant k} \arg\max_{x}{}_i f(x) \quad (1.2c)$$

The quality of predictions directly correlates with the aptitude of the chosen probability measure for $P(w \mid h)$. Finding good ways to estimate these probabilities is the art of *statistical language models* (Section 1.1).

Computing next word predictions following Equation (1.1) is a very time expensive operation. Depending on the chosen probability estimate more or less complex calculations have to be performed, which usually require access to backing data structures. Furthermore that probability is calculated for all words in the vocabulary $\Sigma$, or at least for the set of all words one wants to use for prediction.

Multi-word prediction is typically implemented using *Viterbi-* or *beam search*-algorithms (Bickel et al. 2005; Jurafsky and Martin 2009). These algorithms are based on following most promising prediction paths and pruning the search space with some heuristic. They however are not applicable to the problem of predicting the next word only and at least Bickel et al. (2005) still employ the naïve method of querying all known words to find the next word in a multi-word prediction.

This thesis concerns itself with reducing the necessary query time of prefix queries. Fast word prediction is desirable because to help a user reduce keystrokes, predictions must be available to the user as soon as each keystroke is performed. But the ability to calculate word prediction in a manner faster than the reaction time of a human user might also important: for example in a client-server setting, where a central server has to calculate word prediction queries for a number of clients, the efficiency of the employed algorithm directly determines the necessary hardware of that server.

To this end two independent optimizations will be presented:

1. We will show a novel way to rewrite the recursive definitions for $P(w \mid h)$ of the two state-of-the-art language modeling techniques *Modified Kneser-Ney Smoothing* and the *Generalized Language Model* as weighted sums. In this representation the computation-costly weights will be independent of $w$ and can be calculated once in a precomputation step for each query. This

will significantly reduce the computation time of $P(w|h)$ for varying words $w$ and a fixed history $h$.

2. We will apply well known *top-$k$ joining techniques* (Section 1.2) to the problem of next word prediction. By utilizing a sophisticated data structure that allows retrieval of sequence-count-lists sorted by counts for arbitrary sequence prefixes, we will be able to considerably reduce execution time of next word prediction by avoiding to enumerate all words in the vocabulary.

These optimizations will allow us to give fast, *non-approximative* predictions. That is the predictions will be as good as the underlying language model allows: the system will always predict exactly these words which maximize the probability. No pruning of the word distribution's long tail is necessary.

The remainder of this chapter will introduce the two main subjects of this thesis: statistical language models (Section 1.1) and top-$k$ join queries (Section 1.2).

## 1.1 STATISTICAL LANGUAGE MODELS

*Statistical language models* assign a probability $P_{LM}(s)$ to a sequence of words $s = w_1^l = w_1 \ldots w_l$ by means of a probability distribution, created through statistical analysis of text corpora. Through their use we model knowledge of the user's language in the word prediction application.

A major problem in creating statistical language models from text corpora, is that even for very large corpora, most possible word sequences in a language will not be observed in training [Citation needed]. A common approximation is to make a $n$th-order *Markov assumption*, i.e. to assume that the probability of the word to be predicted $w_l$ only depends on the $n-1$ previous words $w_{l-n+1}^{l-1}$ rather than the full sequence $w_1^{l-1}$. Thus only sequences of length $n$ have to be considered. The resulting language models are called *$n$-gram models*. State-of-the-art language modeling uses $n$-grams with $n$ as large as 5 (Goodman 2001; Jurafsky and Martin 2009; Stolcke 2000).

Most language model probability formulas are given in recursive form, that only implicitly depend on the occurrence count of sequences in the corpus [Citation needed]. If these probabilities can be expressed as monotone scoring functions of occurrence counts, we can apply top-$k$ joining techniques in order to solve noisy channel type queries.

## 1.2 TOP-*k* JOIN QUERIES

Top-$k$ queries are queries whose results are limited to only the $k$ most important ones. These top-$k$ results are identified by scoring all answers with a *scoring function*, and selecting the $k$ highest scoring. *Top-$k$ join queries* are these top-$k$ queries that produce their results trough combination of multiple data sources (Ilyas et al. 2008).

One example might be the query to a web search engine. Search results are ranked by relevance, as computed by several criteria (matching keywords, links from other sites, popularity, ...). Users generally require more than just the one highest ranking search result, as it may not be exactly what they were looking for. On the other hand it would be infeasible to relevance-rank all sites in the search engine's database. Many search engines solve this by only displaying the top 10 sites, deemed most relevant to the user's query, and querying more if required.

Top-k joining techniques have widespread use and are of importance in many applications [Citation needed].

As it shows in the example, the naïve way to solve top-k join queries, i.e. to aggregate all possible join results, sort all of them by score, and then to discard non top-k results, is in most cases far to slow to be of any use. There are of plethora of different top-k processing techniques, but most of them achieve efficiency by requiring a *monotone scoring function* and then sorting data sources on each predicate of that function (Ilyas et al. 2008).

## 1.3 THESIS OVERVIEW

This thesis is structured as follows:

Chapter 2 will introduce the used notation and review the considered language models in this thesis: *Modified Kneser-Ney Smoothing* and the *Generalized Language Model*.

In Chapter 3 will describe a novel approach of how the traditional, recursive definitions of these language models can be rewritten as weighted sums. Additionally we will show that these language models are actually monotone scoring functions over occurrence counts.

Using this representation, Chapter 4 will discuss how top-k joining techniques can be applied to the problem of next word prediction. Two widely used techniques will be analyzed: the *Threshold Algorithm* and the No Random Access Algorithm.

In Chapter 5 we will evaluate the findings of this thesis and research how well they perform in practice.

Finally, Chapter 6 will conclude the thesis and discuss future work.

# 2 REVIEW OF CONSIDERED LANGUAGE MODELS

The currently most commonly used (Chelba et al. 2013; Jurafsky and Martin 2009) technique for estimating language models is *Modified Kneser-Ney Smoothing* by Chen and Goodman (1996, 1998, 1999), based on *Kneser-Ney Smoothing* by Kneser and Ney (1995). Bilmes and Kirchhoff (2003) introduced the concept of a *generalized parallel backoff*, which very recently Pickhardt et al. (2014) used to provide a generalization of Modified Kneser-Ney Smoothing, the *Generalized Language Model*.

Justify selection of MKN and GLM over recently popular LMs from Chelba et al. (2013).

Mention *Nerual Network Language Models* (Bengio et al. 2003; Mikolov 2012) (do we not use them because they can not be represented as weighted sums?), as they seem to be in favor of academic research recently.

This chapter will introduce the used notation (Section 2.1) and recall the language modeling techniques considered in this work, namely *Modified Kneser-Ney Smoothing* (Section 2.2) and the *Generalized Language Model* (Section 2.3). However we will only give and explain definitions for these language models. No motivation or justification of their genesis will be performed, as these are not necessary to understand the contribution of this thesis. Instead the interested reader is hereby referred to the primary sources.

## 2.1 NOTATION: COUNTS AND SKIPS

Let $c(w_1^n)$ denote the frequency count of a sequence's $w_1^n = w_1 \dots w_n$ occurrence in training data. $w_1^n$ may be any contiguous sequence of length $n$ in a text. So $w_1$ does not have to be the first word of a sentence or the like, it just designates the first word of the sequence.

By writing two words directly next each other, as in $w_1 w_2$, we mean to *concatenate* them to a sequence of words $w_1^2$. Sometimes for clarity concatenation is made explicit by writing $w_1 * w_2$. We also extend this notation to concatenations of sequences.

Let $\square$ be a wildcard (called *"skip"*), that can take the place of any word at its location. Thus, for example $c(\square w_1^n)$ is the frequency count of sequences in training data that start with any word $x \in \Sigma$ and where the remaining words are $w_1^n$:

$$c(\square w_1^n) = \sum_{x \in \Sigma} c(x\, w_1^n) \tag{2.1}$$

Where $\Sigma$ is the *vocabulary*, the set of all words in a language $\mathcal{L} \subseteq \Sigma^*$. It is easy to extend this definition to sequences with multiple skips:

Do we mention the difference of $c(\square w_1^n)$ to $c(w_1^n)$ in absence of Beginning of Sentence tags?

$$c(w_1 \square \square w_2) = \sum_{x,y \in \Sigma} c(w_1\, x\, y\, w_2) \tag{2.2}$$

Additionally we define *continuation* counts $N_k(\cdot)$ as the number of sequences from the training data that occur exactly $k$ times and can be constructed by replacing $\bullet$ (called *"continuation skip"*) with any word $x \in \Sigma$. For example $N_2(\bullet w_1^n)$ denotes the number of words that can precede $w_1^n$ in training data, and occur exactly 2 times:

$$N_2(\bullet w_1^n) = |\{x \in \Sigma \,|\, c(x\, w_1^n) = 2\}| \tag{2.3}$$

Further let $N_{k+}(\cdot)$ count sequences that occur $k$ or more times:

$$N_{k+}(\bullet w_1^n) = |\{x \in \Sigma \,|\, c(x\, w_1^n) \geqslant k\}| \tag{2.4}$$

In continuation counts regular skips and continuation skips can be mixed, for example:

Check if equation actually corresponds with implementation.

$$N_{1+}(\bullet w \square \bullet) = |\{x, y \in \Sigma \,|\, c(x\, w \square y) \geqslant 1\}| \tag{2.5}$$

To make a clear distinction between "normal" counts $c(\cdot)$ and continuation counts $N_\bullet(\cdot)$, we sometimes call the former *absolute* counts.

This notation mirrors that of previous publications in the field by Chen and Goodman (1996, 1998, 1999), Goodman (2001), and Pickhardt et al. (2014). With the exception of the new skip $\square$ wildcard, which is primarily based on yet unpublished work by RENÉ PICKHARDT.

Examples of counting with small text corpus?

## 2.2 MODIFIED KNESER-NEY SMOOTHING

Our definition of Modified Kneser-Ney Smoothing is that of an *interpolated* language model. It directly follows Chen and Goodman (1999), who report interpolated techniques to be superior to the previous work of *backoff* language models by Kneser and Ney (1995).

Therefore Modified Kneser-Ney smoothed language models are calculated as an interpolation of higher and lower order $n$-gram language models (Pickhardt et al. 2014). The highest order model $P_{\text{MKN}}$ is interpolated with lower orders $\hat{P}_{\text{MKN}}$ as follows:

$$P_{\text{MKN}}(w_n \,|\, w_1^{n-1}) = \frac{c^d(w_1^n) + \gamma(w_1^{n-1})\, \hat{P}_{\text{MKN}}(w_n \,|\, w_2^{n-1})}{c(w_1^{n-1} \square)} \tag{2.6}$$

With the *discounted* count $c^d(w_1^n) = \max\{c(w_1^n) - D(c(w_1^n)), 0\}$.

The exact definitions of $D(\cdot)$ and $\gamma(w_1^{n-1})$ are not important in our context, however for the sake of completeness they are given in Section 2.4.

We call the word $w_n$ for which we calculate the probability the *argument*, and the conditional sequence $w_1^{n-1}$ the *history*.

Equation (2.6) is only defined if the denominator $c(w_1^{n-1} \square) > 0$, that is if the history $w_1^{n-1} \square$ was seen in training. If $w_1^{n-1} \square$ was not seen, we perform a so called *"back-off"* and set:

$$P_{\text{MKN}}(w_n \,|\, w_1^{n-1}) = P_{\text{MKN}}(w_n \,|\, w_2^{n-1}) \quad \text{(if } c(w_1^{n-1} \square) = 0\text{)} \tag{2.7}$$

It is common to perform multiple back-offs until the history was actually seen.

Thus, interpolation with lower order models correspond to leaving out the first word of the history, on which our probabilities are conditioned. Lower order models are computed differently, and in turn interpolated with further lower orders:

$$\hat{P}_{\text{MKN}}(w_n \mid w_1^{n-1}) = \frac{N^d_{1+}(\bullet\, w_1^n) + \gamma(w_1^{n-1})\, \hat{P}_{\text{MKN}}(w_n \mid w_2^{n-1})}{N_{1+}(\bullet\, w_1^{n-1}\, \bullet)} \quad (2.8)$$

With $N^d_{1+}(\bullet\, w_1^n) = \max\{N_{1+}(\bullet\, w_1^n) - D(c(w_1^n)), 0\}$. the *discounted* continuation count.

Backing-off to compute lower orders is never necessary, because if history $w_1^{n-1}$ was seen, $w_2^{n-1}$ was seen as well.

This should actually not be true, because we increase n-gram length for first lower order. How do we handle this in current implementation?

The lowest order (the probability of a single word) can not be further interpolated and we use the relative continuation frequency of that word. Similarly if we only want to compute the probability of a single word on the highest order, we compute it as the relative frequency of that word:

$$P_{\text{MKN}}(w) = \frac{c(w)}{c(\square)} \quad (2.9a)$$

$$\hat{P}_{\text{MKN}}(w) = \frac{N_{1+}(\bullet\, w)}{N_{1+}(\bullet\,\bullet)} \quad (2.9b)$$

Note that this definition of lowest order probabilities deviates from Chen and Goodman (1999). They instead smooth the lowest order with the uniform distribution of $\frac{1}{|\Sigma|}$.

Why don't we do this?

## 2.3 GENERALIZED LANGUAGE MODEL

The equations of this section match the ones from Pickhardt et al. (2014), but are given in a slightly modified own notation that enables easier proceeding in Chapter 3.

The Generalized Language Model is a natural extension to Modified Kneser-Ney Smoothing. It is again based on interpolation, though it differs in the way lower order models are interpolated. The highest order is computed as:

$$P_{\text{GLM}}(w_n \mid w_1^{n-1}) = \frac{c^d(w_1^n) + \frac{\gamma(w_1^{n-1})}{\#_\partial w_1^{n-1}} \sum_{j=1}^{\#_\partial w_1^{n-1}} \hat{P}_{\text{GLM}}(w_n \mid \partial_j w_1^{n-1})}{c(w_1^{n-1}\,\square)} \quad (2.10)$$

Auxiliary definitions $c^d$, $N^d_{1+}$, $\gamma$ are the same as for Modified Kneser-Ney Smoothing and given in Section 2.4.

The skip operator $\partial_j w_1^{n-1}$ can be any mapping that relates $w_1^{n-1}$ to exactly $\#_\partial w_1^{n-1}$ many derived sequences. It is easy to see that for $\#_\partial w_1^{n-1} = 1$ and $\partial_1 w_1^{n-1} = w_2^{n-1}$ Equation (2.10) is an instantiation of Equation (2.6) (Modified Kneser-Ney Smoothing), and thus a true generalization.

In practice only one definition for $\partial$ is used though. That is $\#_\partial w_1^{n-1}$ is set to the number of non-skip words in $w_1^{n-1}$ and $\partial_j w_1^{n-1}$ replaces the jth non-skip word with a skip. For instance: $\partial_2 w_1^3 = w_1 \square w_3$.

Again as with Modified Kneser-Ney Smoothing, Equation (2.10) is not defined if $w_1^{n-1}$ is unseen and thus our denominator would be zero. The back-off step is then performed as:

$$P_{\text{GLM}}(w_n \mid w_1^{n-1}) = \frac{1}{\#_\partial w_1^{n-1}} \sum_{j=1}^{\#_\partial w_1^{n-1}} P_{\text{GLM}}(w_n \mid \partial_j w_1^{n-1}) \qquad (\text{if } c(w_1^{n-1}\,\square) = 0) \tag{2.11}$$

Replace $\square$ with $\bullet$ in $N_{1+}$ in GLM?

$\bullet$ at end of $N_{1+}$ correct?

Subsequently lower order models are computed as:

$$\hat{P}_{\text{GLM}}(w_n \mid w_1^{n-1}) = \frac{N^d_{1+}(\bullet\, w_1^n) + \frac{\gamma(w_1^{n-1})}{\#_\partial w_1^{n-1}} \sum_{j=1}^{\#_\partial w_1^{n-1}} \hat{P}_{\text{GLM}}(w_n \mid \partial_j w_1^{n-1})}{N_{1+}(\bullet\, w_1^{n-1}\, \bullet)} \tag{2.12}$$

The case of the lowest order occurs when $\#_\partial w_1^{n-1} = 0$ and is handled similarly to Modified Kneser-Ney Smoothing. In practice this is the case when $w_1^{n-1}$ only consists of skips.

$$P_{\text{GLM}}(w_n \mid w_1^{n-1}) = \frac{c(w_1^n)}{c(w_1^{n-1}\,\square)} \qquad (\text{if } \#_\partial w_1^{n-1} = 0) \tag{2.13a}$$

$$\hat{P}_{\text{MKN}}(w_n \mid w_1^{n-1}) = \frac{N_{1+}(\bullet\, w_1^n)}{N_{1+}(\bullet\, w_1^{n-1}\, \bullet)} \qquad (\text{if } \#_\partial w_1^{n-1} = 0) \tag{2.13b}$$

## 2.4 DISCOUNTING AND INTERPOLATION WEIGHTS

The purpose of the discounted count functions $c^d(\cdot)$ and $N^d_{1+}(\cdot)$ is to reduce the absolute values of higher order probability of the language model. This is done in order to have "left over probability mass" to assign to lower orders (Ney and Essen 1991; Ney et al. 1994).

These discounted counts are defined as:

$$c^d(w_1^n) = \max\{c(w_1^n) - D(c(w_1^n)), 0\} \tag{2.14a}$$

$$N^d_{1+}(\bullet\, w_1^n) = \max\{N_{1+}(\bullet\, w_1^n) - D(c(w_1^n)), 0\} \tag{2.14b}$$

Discounting is done by subtracting a value $D(c)$ from each count. The higher the count value $c$ the higher it is discounted (Chen and Goodman 1999):

$$D(c) = \begin{cases} 0 & \text{if } c = 0 \\ D_1 & \text{if } c = 1 \\ D_2 & \text{if } c = 2 \\ D_{3+} & \text{if } c \geqslant 3 \end{cases} \tag{2.15}$$

With the the discounting values $D_\bullet$ defined as (Chen and Goodman 1999):

$$D_1 = 1 - 2Y\frac{n_2}{n_1} \tag{2.16a}$$

$$D_2 = 2 - 3Y\frac{n_3}{n_2} \tag{2.16b}$$

$$D_{3+} = 3 - 4Y\frac{n_4}{n_3} \tag{2.16c}$$

Where $Y = \frac{n_1}{n_1+n_2}$, and where $n_i$ denotes the total number of of $n$-grams which appear exactly $i$ times in the training corpus (Chen and Goodman 1999).

This doesn't really fully explain how $n_i$ works does it?

Based on this we can define the interpolation weight $\gamma(\cdot)$ that determines how much lower order probabilities factor into those of higher orders (Chen and Goodman 1999):

$$\gamma(w_1^{n-1}) = D_1\, N_1(w_1^{n-1}\bullet) + D_2\, N_2(w_1^{n-1}\bullet) + D_{3+}\, N_{3+}(w_1^{n-1}\bullet) \tag{2.17}$$

For the given discounted count functions $c^d$ and $N^d_{1+}$ the interpolation weights $\gamma(\cdot)$ have to be chosen like this to ensure true probabilities.

# 3 FORMULATING LANGUAGE MODELS AS WEIGHTED SUMS

In this chapter we will present a novel way to represent the probabilities $P_{MKN}(w_n | w_1^{n-1})$ and $P_{GLM}(w_n | w_1^{n-1})$ as weighted sums of terms depending on $w_n$, for a fixed history $h = w_1^{n-1}$ and an arbitrary probability event $w = w_n$. In other words we will express the formulas given in Chapter 2 as equations of the following form:

$$P(w | h) = \sum_{i=1}^{N} \lambda_i^h \cdot \alpha_i^h(w) \tag{3.1}$$

To the authors knowledge, there is no other work on expressing existing language models as weighted sums. However, Jelinek and Mercer (1980) already experimented with trying to learn weighted sum parameters automatically for language modeling.

Being able to accurately represent probabilities in a form like this is highly beneficial for word prediction. As we can see in Equation (1.1) the computation for next word prediction is based on calculating multiple probabilities $P(w | h)$ with a fixed history $h$ but varying arguments $w$. As has been specified in Chapter 2, probability calculation is based on complex recursive formulas. Using our weighted sum representation this computation can be speed up by a huge factor, since the computation of the sum weights $\lambda_i^h$ can be performed once beforehand. For each final probability we then only have to perform the $N$ remaining lookups $\alpha_i^h(w)$ that actually depend on $w$.

Another handy benefit is that by this representation we have all terms that change with $w$ explicitly available, which is a prerequisite for top-$k$ joining and will be explained in Chapter 4. Lastly our algorithms for calculating sum weights turn out to be performing faster then the direct recursive implementations, which we measure in Section 5.3.

We will now present the basic idea of how to transform a probability $P(w_n | w_1^{n-1})$ into a weighted sum, that applies to both Modified Kneser-Ney Smoothing and the Generalized Language Model:

Note that in all Equations (2.6), (2.8), (2.10) and (2.12) the probability event $w_n$ only appears in the nominator of fractions, specifically in the argument terms of $c^d$ or $N_{1+}^d$. Additionally $w_n$ occurs as arguments for $c$ and $N_{1+}$ in the lowest order Equations (2.9) and (2.13). Our general idea is thus, to first expand recursive calls to $P(\cdot)$ or $\hat{P}(\cdot)$, and second to factor out those terms that do not depend on $w_n$ by the distributive property.

Section 3.1 explains our aprroach for Modified Kneser-Ney Smoothing, while Section 3.2 concerns itself with the Generalized Language Model.

## 3.1 MODIFIED KNESER-NEY SMOOTHING

Modified Kneser-Ney Smoothed probabilities $P_{\text{MKN}}(w \mid h)$ are calculated as follows (Section 2.2):

1. Backing-off: Leave out words at the beginning of the history $h$, until $h\,\square$ is seen for the first time (Equation (2.7)).
2. Highest order: Look-up frequency counts $c^{d}(h\,w)$ and $c(h\,\square)$ (Equation (2.6)) and shorten the history by one word.
3. Lower orders: Look-up continuation counts $N^{d}_{1+}(\bullet\,h\,w)$ and $N_{1+}(\bullet\,h\,\bullet)$ (Equation (2.8)) and shorten the history further until it is empty.
4. Lowest order: Look-up continuation counts $N_{1+}(\bullet\,w)$ and $N_{1+}(\bullet\,\bullet)$ (Equation (2.9)).

### 3.1.1 *Example*

As an example we will give the full formula expansion necessary to calculate $P_{\text{MKN}}(w_3 \mid w_1 w_2)$ for any sequence $w_1 w_2 w_3$, assuming sequence $w_1 w_2\,\square$ was seen:

$$P_{\text{MKN}}(w_3 \mid w_1 w_2) = \frac{c^{d}(w_1 w_2 w_3) + \gamma(w_1 w_2)\,\hat{P}_{\text{MKN}}(w_3 \mid w_2)}{c(w_1 w_2\,\square)} \tag{3.2a}$$

$$\hat{P}_{\text{MKN}}(w_3 \mid w_2) = \frac{N^{d}_{1+}(\bullet\,w_2 w_3) + \gamma(w_2)\,\hat{P}_{\text{MKN}}(w_3)}{N_{1+}(\bullet\,w_2\,\bullet)} \tag{3.2b}$$

$$\hat{P}_{\text{MKN}}(w_3) = \frac{N_{1+}(\bullet\,w_3)}{N_{1+}(\bullet\,\bullet)} \tag{3.2c}$$

Inserting the lower orders results in:

$$P_{\text{MKN}}(w_3 \mid w_1 w_2) = \frac{c^{d}(w_1 w_2 w_3) + \gamma(w_1 w_2)\frac{N^{d}_{1+}(\bullet\,w_2 w_3) + \gamma(w_2)\frac{N_{1+}(\bullet\,w_3)}{N_{1+}(\bullet\,\bullet)}}{N_{1+}(\bullet\,w_2\,\bullet)}}{c(w_1 w_2\,\square)} \tag{3.3}$$

By using the distributive property we obtain:

$$\begin{aligned} P_{\text{GLM}}(w_3 \mid w_1 w_2) = \quad & c^{d}(w_1 w_2 w_3) \cdot \frac{1}{c(w_1 w_2\,\square)} \\ + \quad & N^{d}_{1+}(\bullet\,w_3 w_3) \cdot \frac{\gamma(w_1 w_2)}{c(w_1 w_2\,\square)\,N_{1+}(\bullet\,w_2\,\bullet)} \\ + \quad & N_{1+}(\bullet\,w_3) \cdot \frac{\gamma(w_1 w_2)\,\gamma(w_2)}{c(w_1 w_2\,\square)\,N_{1+}(\bullet\,w_2\,\bullet)\,N_{1+}(\bullet\,\bullet)} \end{aligned} \tag{3.4}$$

In the final equation Equation (3.4) we can clearly see those terms that do depend on $w_3$ and those that do not. Note that $w_3$ never occurs on the right side of the multiplication dot. Per line the left side of the multiplication dot will form the terms $\alpha^{h}_{i}(w_3)$, while the ride side will form the sum weights $\lambda^{h}_{i}$.

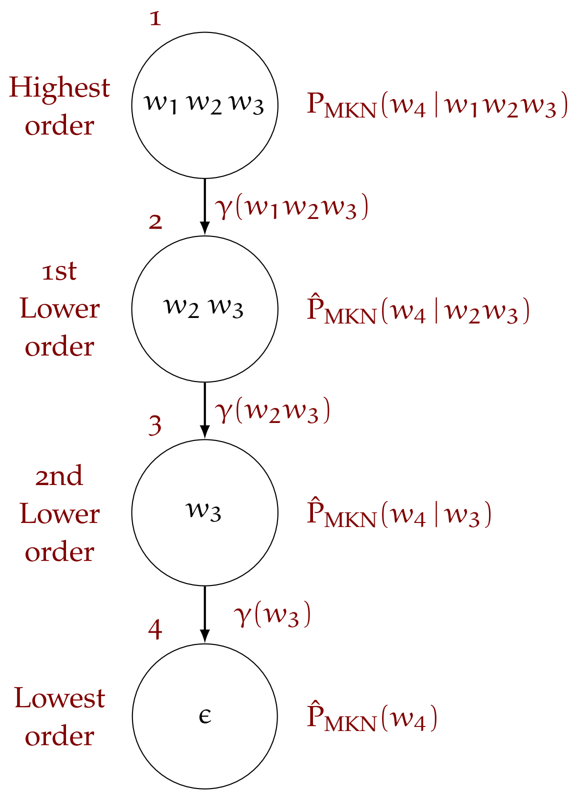


**Figure 3.1:** The *backoff graphs* shows the interpolation of different orders during computation of $P_{MKN}(w_4\,|\,w_1w_2w_3)$. Centered are the histories for each probability. It is assumed that the history $w_1w_2w_3$ was seen during training, so that no backing-off is necessary. Orders are combined with with interpolation weights $\gamma(\cdot)$.

But note that this is only valid if the sequence $w_1w_2w_3$ was seen in the training corpus, that is if $c(w_1w_2w_3) \geqslant 1$. If this is not the case, backoff steps have to be performed, and a different resulting formula is obtained.

### 3.1.2 *Sum Terms*

All counts except those of the lowest order are discounted using $c^d$ and $N^d_{1+}$ to interpolate that order with the following by the weight $\gamma(h)$. This process of interpolating different orders is visualized in Figure 3.1 for the case of a history of length three. Building similiar backoff graphs for other history lengths is straightforward.

Each order with a history $h'$ is interpolated with the following order where the first word of $h'$ is removed. Thus a total ordering on the histories of the orders can be specified. In Figure 3.1 we have thus assigned each history a number from 1 to 4. Let $\hat{h}_i$ denote the history of that graph with number $i$.

The first step in expressing $P_{MKN}(w\,|\,h)$ as a weighted sum is finding the number $N$ of sum weights. In the case of Modified Kneser-Ney Smoothing this is exactly the number of interpolation orders. Let $\hat{h}_s$ be the first seen history that is the result of backing off given history $h$ exaclty $s$ many times. The number $N$ of sum weights is then the number of words of that history plus one for the empty history.

$$N = |\hat{h}_s| + 1 \tag{3.5}$$

Where $|w_1^n| = n$ is the length of the $n$-gram $w_1^n$.

The next step is to find the actual terms $\alpha_i^h(w)$ that shall be weightedly added. When looking at the equations that define Modified Kneser-Ney we note the fact that only the count terms in the numerators depend on the probability event $w$. Exactly these terms compose our sum arguments:

$$\alpha_i^h(w) = \begin{cases} c(w) & \text{if } N = 1 \\ c^d(\hat{h}_s w) & \text{if } N \neq 1 \wedge i = 1 \\ N^d_{1+}(\bullet\, \hat{h}_{s+i-1} w) & \text{if } N \neq 1 \wedge 1 < i < N \\ N_{1+}(\bullet\, w) & \text{if } N \neq 1 \wedge i = N \end{cases} \tag{3.6}$$

Lastly we need to define the actual sum weights $\lambda_i^h$. We note that — because each order is interpolated with the weight $\gamma$ of the previous order — each order is in total weighted by the product of the $\gamma$s of all previous orders. Additionally lower order models occur in the numerators of fractions, because of that they are weighted by all previous denominator counts as well. Building on this we can define:

$$\lambda_i^h = \frac{1}{c(\hat{h}_s\, \square\,)} \prod_{j=2}^{i} \frac{\gamma(\hat{h}_{s+i-2})}{N_{1+}(\bullet\, \hat{h}_{s+i-1} \bullet\,)} \tag{3.7}$$

Specifying an algorithm to compute interpolation weights $\lambda_i^h$ that minimizes frequency count lookup (using the iterative definition of $\lambda_i^h$) is straightforward and given in Algorithm 3.1. In Line 9 the lower order sum weights are assigned. It takes advantage of the fact that Equation (3.7) can easily be written in a recursive manner. This is done to avoid repeated count lookups and multiplications.

**Algorithm 3.1** Computing Modified Kneser-Ney sum weights

```
Input: $h$                                  ▷ history for which to determinate sum weights
Output: $\lambda_1^h, \dots, \lambda_N^h$                   ▷ list of sum weights
1: ▷ find first seen history
2: $s \leftarrow 1$
3: while $c(\hat{h}_s\, \square\,) = 0$ do                    ▷ while history is unseen
4:     $s \leftarrow s + 1$

5: ▷ compute sum weights
6: $N \leftarrow |\hat{h}_s| + 1$                          ▷ number of sum weights
7: $\lambda_1^h \leftarrow \frac{1}{c(\hat{h}_s\, \square\,)}$                       ▷ set weight of highest order
8: for $i$ from $2$ to $N$ do
9:     $\lambda_i^h \leftarrow \lambda_{i-1}^h \frac{\gamma(\hat{h}_{s+i-2})}{N_{1+}(\bullet\, \hat{h}_{s+i-1} \bullet\,)}$          ▷ weight of lower orders
```

## 3.2 GENERALIZED LANGUAGE MODEL

The difference between the Modified Kneser-Ney smoothed language model and the Generalized Language Model is the way in which orders are interpolated. Instead of just one probability — that of shortening the history by one — being factored in, in the Generalized Language Model the average of multiple probabilities of the next lower order is factored into the higher order probability.

This section will not consider the general case were one probability $P_{\text{GLM}}(w \,|\, h)$ incorporates exactly $\#_\partial\, h$ probabilities $\hat{P}_{\text{GLM}}(w \,|\, \partial_j h)$ of the next lower order. Instead only the special case described by Pickhardt et al. (2014), that is actually used in practice, is examined. That is the number of lower order probabilities incorporated $\#_\partial\, h$ is set to the number of non-skip words in $h$, and $\partial_j h$ is defined as replacing the $j$th non-skip word in $h$ with a skip $\square$.

### 3.2.1 *Example*

Examplatory the full formula expansion for $P_{\text{GLM}}(w_3 \,|\, w_1 w_2)$, assuming the sequence $w_1 w_2 \square$ was seen:

$$P_{\text{GLM}}(w_3 \,|\, w_1 w_2) = \frac{c^d(w_1 w_2 w_3) + \frac{\gamma(w_1 w_2)}{2}\left(\hat{P}_{\text{GLM}}(w_3 \,|\, \square w_2) + \hat{P}_{\text{GLM}}(w_3 \,|\, w_1 \square)\right)}{c(w_1 w_2 \square)} \tag{3.8a}$$

$$\hat{P}_{\text{GLM}}(w_3 \,|\, \square w_2) = \frac{N^d_{1+}(\bullet \square w_2 w_3) + \frac{\gamma(\square w_2)}{1} \hat{P}_{\text{GLM}}(w_3 \,|\, \square \square)}{N_{1+}(\bullet \square w_2 \bullet)} \tag{3.8b}$$

$$\hat{P}_{\text{GLM}}(w_3 \,|\, w_1 \square) = \frac{N^d_{1+}(\bullet w_1 \square w_3) + \frac{\gamma(w_1 \square)}{1} \hat{P}_{\text{GLM}}(w_3 \,|\, \square \square)}{N_{1+}(\bullet w_1 \square \bullet)} \tag{3.8c}$$

$$\hat{P}_{\text{GLM}}(w_3 \,|\, \square \square) = \frac{N_{1+}(\bullet \square \square w_3)}{N_{1+}(\bullet \square \square \bullet)} \tag{3.8d}$$

After insertion of lower order probabilities:

$$P_{\text{GLM}}(w_3 \,|\, w_1 w_2) = \tag{3.9}$$
$$\frac{c^d(w_1 w_2 w_3) + \frac{\gamma(w_1 w_2)}{2}\left(\frac{N^d_{1+}(\bullet \square w_2 w_3) + \frac{\gamma(\square w_2)}{1}\frac{N_{1+}(\bullet \square \square w_3)}{N_{1+}(\bullet \square \square \bullet)}}{N_{1+}(\bullet \square w_2 \bullet)} + \frac{N^d_{1+}(\bullet w_1 \square w_3) + \frac{\gamma(w_1 \square)}{1}\frac{N_{1+}(\bullet \square \square w_3)}{N_{1+}(\bullet \square \square \bullet)}}{N_{1+}(\bullet w_1 \square \bullet)}\right)}{c(w_1 w_2 \square)}$$

Finally, by using the distributive property, we arrive at the weighted sum representation: $\alpha^h_i(w)$ left of the multiplication dot, $\lambda^h_i$ right.

$$\begin{aligned} P_{\text{GLM}}(w_3 \,|\, w_1 w_2) = \quad & c^d(w_1 w_2 w_3) \cdot \frac{1}{c(w_1 w_2 \square)} \\ + \; & N^d_{1+}(\bullet \square w_2 w_3) \cdot \frac{\gamma(w_1 w_2)}{2\,c(w_1 w_2 \square)\, N_{1+}(\bullet \square w_2 \bullet)} \\ + \; & N^d_{1+}(\bullet w_1 \square w_3) \cdot \frac{\gamma(w_1 w_2)}{2\,c(w_1 w_2 \square)\, N_{1+}(\bullet w_1 \square \bullet)} \\ + \; & N_{1+}(\bullet \square \square w_3) \cdot \frac{\gamma(w_1 w_2)}{2 \cdot 1\, c(w_1 w_2 \square)}\left(\frac{\gamma(\square w_2)}{N_{1+}(\bullet \square w_2 \bullet)\, N_{1+}(\bullet \square \square \bullet)} + \right. \\ & \left. \frac{\gamma(w_1 \square)}{N_{1+}(\bullet w_1 \square \bullet)\, N_{1+}(\bullet \square \square \bullet)}\right) \end{aligned} \tag{3.10}$$

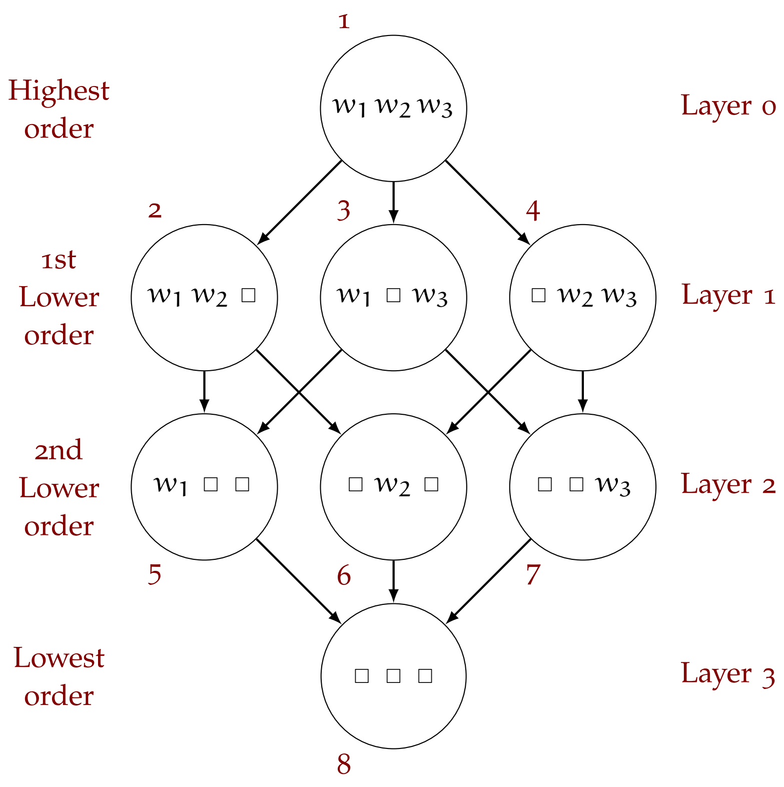


**Figure 3.2:** $P_{\text{GLM}}$ backoff graph of order 3 for history $h = w_1 w_2 w_3$.

### 3.2.2 *Backoff Graph*

Figure 3.2 shows the *backoff graph* introduced by Bilmes and Kirchhoff (2003). It visualizes the lower order probabilities that factor into those of higher orders, using that definition of $\partial_j h$. The nodes represent the histories $h = w_1 w_2 w_3$ of a probability $P_{\text{GLM}}(w_4 \,|\, w_1 w_2 w_3)$. The directed edges represent an immediate dependency, for example the probabilities that occur in the definition of $\hat{P}_{\text{GLM}}(w_4 \,|\, w_1 w_2 \square)$ are $\hat{P}_{\text{GLM}}(w_4 \,|\, w_1 \square \square)$ and $\hat{P}_{\text{GLM}}(w_4 \,|\, \square w_2 \square)$. In order theory, graphs that generalize this idea are also known as *hasse diagrams*.

We also call a graph that represents the inter-dependencies of probabilities during the calculation of $P_{\text{GLM}}(w \,|\, h)$ the *binomial diamond* of order $n = |h|$, because of its diamond-like shape, and a number of properties:

1. Unlike the case of Modified Kneser-Ney Smoothing there is no longer one clear way to define a total order of all the histories in the graph. Instead if we partitioned the graph into layers, where layer $l$ contains the histories that have exactly $l$ skip words, the graph has $n + 1$ layers.
2. Layer $l$ contains exactly $\binom{n}{l}$ nodes.
3. Nodes in layer $l$ have exactly $n - l$ outgoing and $l$ incoming edges.
4. There are $2^n$ nodes in the graph.

Item 1 is due to the fact that layer $l$ includes exactly those histories that are part of the $k$th highest order of probability.

To explain Item 2, it is helpful to imagine replacing words in a history as laying a mask of either a skip or a non-skip on top of each word. Then the number of nodes in each layer can be understood as permuting the skip and non-skip masks on top of the history. It is a well known fact that there are exactly $\binom{n}{l}$ permutations of $l$ tokens of type A (skips) and $k$ tokens of type B (non-skips) when $n = l + k$.

Item 3 directly follows from the fact, that each non-skip word in a history can be replaced by a skip, and that each skip is the result of those replacements.

Lastly, we can explain Item 4 because every history is just the result of laying a mask that is a combination of skips or non-skips on top of the original history. All possible masks are then all permutations of two elements (skips or non-skips) with repetition of length $n$, which there are $2^n$ arrangements of. We can also infer this from Item 2 via the binomial theorem $(x+y)^n = \sum_{l=0}^{n} \binom{n}{l} x^{n-l} y^l$, if we set $x = y = 1$, we obtain $\sum_{k=0}^{n} \binom{n}{l} = 2^n$.

Following the model of the backoff graph will be a major aid in understanding the weighted sum representation of the Generalized Language Model.

In order to have a notation avaialbe for all derived histories, we impose an arbitrary total ordering one those histories. One such ordering is given in Figure 3.2 and assigns each history a number between 1 and 8. Let $\hat{h}_i$ than be the $i$th history of an arbitrary ordering.

### 3.2.3 *Number of Sum Weights*

In the previous subsection, it was reasoned that, in the Generalized Language Model, for a history of length $n$ there are exactly $2^n$ possible derivative histories. Following this, one might make the assumption, that to evaluate $P_{\text{GLM}}(w \mid h)$ it is necessary to evaluate at most $2^n - 1$ different probabilities $\hat{P}_{\text{GLM}}(w \mid \partial^* h)$ where $\partial^* h$ is any of the $2^n - 1$ derived histories of $h$ excluding $h$ itself. This assumption however is wrong as it can be necessary that one history $h^*$ is both part of a highest order probability $P_{\text{GLM}}(w \mid h^*)$ and a lower order probability $\hat{P}_{\text{GLM}}(w \mid h^*)$ in the presence of unseen histories, as will shown in the next paragraph.

Let us for example calculate the probability $P_{\text{GLM}}(w_3 \mid w_1 w_2)$ on hypothetical training data. That hypothetical training data shall be different from Section 3.2.1 and now $w_1 w_2$ shall be unseen, that is $c(w_1 w_2) = 0$. Because of this the average of back-off probabilities following Equation (2.11) has to be formed:

$$P_{\text{GLM}}(w_3 \mid w_1 w_2) = \frac{1}{2} \left( P_{\text{GLM}}(w_3 \mid \square w_2) + P_{\text{GLM}}(w_3 \mid w_1 \square) \right) \quad (3.11a)$$

Note that these back-off probabilities for histories $\square w_2$ and $w_1 \square$ are still of the highest order $P_{\text{GLM}}$. We now further assume history $\square w_2$

to also not occur in training, so that further back-off is necessary. History $w_1 \,\square$ however is treated as seen:

$$P_{\text{GLM}}(w_3 \,|\, \square\, w_2) = P_{\text{GLM}}(w_3 \,|\, \square\,\square) \tag{3.11b}$$

$$P_{\text{GLM}}(w_3 \,|\, w_1 \square) = \frac{c(w_1 \square w_3) + \gamma(w_1 \square)\,\hat{P}_{\text{GLM}}(w_3 \,|\, \square\,\square)}{c(w_2 \square\,\square)} \tag{3.11c}$$

We can see, that $P_{\text{GLM}}(w_3 \,|\, \square\,\square)$ (Equation (3.11b)) and $\hat{P}_{\text{GLM}}(w_3 \,|\, \square\,\square)$ (Equation (3.11c)) both factor into probability $P_{\text{GLM}}(w_3 \,|\, w_1 w_2)$. They would be evaluated as:

$$P_{\text{GLM}}(w_3 \,|\, \square\,\square) = \frac{c(\square\,\square\, w_3)}{c(\square\,\square\,\square)} \tag{3.11d}$$

$$\hat{P}_{\text{GLM}}(w_3 \,|\, \square\,\square) = \frac{N_{1+}(\bullet\, \square\,\square\, w_3)}{N_{1+}(\bullet\, \square\,\square\, \bullet)} \tag{3.11e}$$

Because of this, it is possible that one history $\hat{h}_\bullet$ might both occur as highest order probability $P_{\text{GLM}}(w \,|\, \hat{h}_\bullet)$ and lower order probability $\hat{P}_{\text{GLM}}(w \,|\, \hat{h}_\bullet)$. It is non-trivial to decide exactly how many derived histories occur both as highest and lower order probabilities, and to fit algorithms to this finding. Consequently we only employ an upper bound of sum weights $N$ and set all unnecessary sum weights to zero. An upper bound for the number of sum weights is the number of derived histories times two, as histories can occur both in highest and lower order probabilities:

$$N = 2 \cdot 2^{|h|} \tag{3.12}$$

### 3.2.4 *Sum Terms*

Based on this we can define the sum terms for $1 \leqslant i \leqslant N$.

$$\alpha^h_{2i-1}(w) = \begin{cases} c^d(\hat{h}_i w) & \text{if } \#_\partial \hat{h}_i > 0 \\ c(\hat{h}_i w) & \text{if } \#_\partial \hat{h}_i = 0 \end{cases} \tag{3.13a}$$

$$\alpha^h_{2i}(w) = \begin{cases} N^d_{1+}(\bullet\, \hat{h}_i w) & \text{if } \#_\partial \hat{h}_i > 0 \\ N_{1+}(\bullet\, \hat{h}_i w) & \text{if } \#_\partial \hat{h}_i = 0 \end{cases} \tag{3.13b}$$

The terms with an odd index (Equation (3.13a)) correspond to $\hat{h}_i$ occuring as the highest order of probability, the terms with even index (Equation (3.13b)) to lower orders. The case of $\#_\partial \hat{h}_i = 0$ occurs when $\hat{h}_i$ consists only of skips, it thus can no longer be backoffed, and therefore presents the unigram case.

Finding the sum weights $\lambda^h_i$ is more complex. We will first reason about how those weights have to be structured and then give a more rigorous definition. The sum weights $\lambda^h_{2i-1}$, that are multiplied with $\alpha^h_{2i-1}(w)$ which compose the highest order weight, will be discussed first. Our solution is oriented along the backoff graph. The weight of a history is exactly then zero, if no backoff step of Equation (2.11) reaches that history. This occurs if for that history's node all histories of immediate parent nodes are seen. Conversly this means that the sum weights $\lambda^h_{2i-1}$ are proportional to the number of paths from the root note to the current node, for which the current node's history is seen but the histories of all others nodes in the path are not.

What also has to be considered is that in Equation (2.11) we divide the sum of all backoff probabilities by their count. The more backoff steps that have to be performed to reach a node, the lower its influence is. However in turn it also can be reached via more paths as seen in the backoff graph of Figure 3.2. In total there exist two paths from the root node to any node in layer 2. But there exists a total of six paths from the root node to the node in layer 3. We define a coefficient $\mu = \frac{(|h|-l_i)!}{|h|!}$, where $l_i$ is the layer in the backoff graph in which the node of history $\hat{h}_i$ lies. Sum weights $\lambda^h_i$ are also proportional to that coefficient $\mu$.

The final influence on $\lambda^h_{2i-1}$ stems from the fact that we defined Equation (3.13a) to only consist of the numerator of Equation (2.10). Because of this we also have to divide the sum weight by the denominator of that equation.

Sum weights $\lambda^h_{2i}$ for terms that occurs in lower orders are calculated in an analogous way. For the same argument as above they are also influenced by the coefficient $\mu$ and a denominator, but this time that of Equation (2.12). However for lower order weights they are also influenced by the denominators of equations that included them. This fact is especially visible by the nesting of fractions in Equation (3.9). From this equation we can conclude that a sum weight $\lambda^h_{2i}$ of a node is as sum of recursive calls from the root probability $P(w \mid h)$ to the current node. The chains of recursive calls that can reach a history $\hat{h}_i$ are exactly represented by the paths to it in the backoff graph. Each node contributes a $\frac{\gamma}{\text{denominator}}$ weight. Lower order sum weights thus consists of sums of multiplications, where the factors are based on the nodes along a path to the current node.

Additional explanation of $\mathtt{factor} \cdot \mathtt{sum}$ property.

Because of this complexity of terms that compose the sum weights we will not try to find a mathematical expression that defines these sum weights. Instead we will define them by specifying the algorithm that computes them: Sum weights $\lambda^h_i$ are calculated by Algorithm 3.2 which uses the helper functions of Algorithm 3.3.

The algorithm works with the notion of a backoff graph as shown in Figure 3.2. That graph is unique for every history $h$ and consist of $2^{|h|}$ nodes. Each node is assigned a derived history. A directed edge from node $A$ to node $B$ is drawn if $B$'s history can be created by replacing one word in $A$'s history with a skip $\square$.

*Parents* of a node $A$ are all nodes $B$ that have a directed edge from $B$ to $A$. *Childrens* of a node $A$ are all the nodes, that $A$ is a parent of. *Ancestors* of a node $A$ are all nodes that can be reached from $A$ by repeated application of the parent relation. A *path* exists to node $A_1$ from node $A_m$, if there is a set of nodes $\{A_1, \ldots, A_m\}$ such that it holds for all nodes $A_i$ in that set, that $A_{i+1}$ is a parent of $A_i$ for $1 \leqslant i < m$. The *root* of a graph is the node that has no parents and is an ancestor to all other nodes.

**Algorithm 3.2** Computing Generalized Language Model sum weights

```
Input: $h$                                   ▷ history for which to determinate sum weights
Output: $\lambda^h_1, \dots, \lambda^h_N$                      ▷ list of sum weights
 1: $N \leftarrow 2 \cdot 2^{|h|}$                                ▷ number of sum factors
 2: for each node in graph do
 3:     ▷ data lookup
 4:     node.count $\leftarrow c(\hat{h}_{\text{node.index}} \square)$
 5:     node.contCount $\leftarrow N_{1+}(\bullet \hat{h}_{\text{node.index}} \bullet)$
 6:     node.gamma $\leftarrow \gamma(\hat{h}_{\text{node.index}})$
 7:     $\mu \leftarrow \frac{(|h| - \text{node.layer})!}{|h|!}$

 8:     ▷ weight for highest order
 9:     if node.count $= 0$ then
10:         $\lambda^h_{2 \cdot \text{node.index} - 1} \leftarrow 0$
11:     else if node $=$ root then
12:         $\lambda^h_{2 \cdot \text{node.index} - 1} \leftarrow \frac{1}{\text{node.count}}$
13:     else
14:         $\lambda^h_{2 \cdot \text{node.index} - 1} \leftarrow \frac{\mu \cdot \textsc{HighestOrderWeight}(\text{root}, \text{node})}{\text{node.count}}$

15:     ▷ weight for lower orders
16:     if node.contCount $= 0$ or node $=$ root then
17:         $\lambda^h_{2 \cdot \text{node.index}} \leftarrow 0$
18:     else
19:         $\lambda^h_{2 \cdot \text{node.index}} \leftarrow \frac{\mu \cdot \textsc{LowerOrdersWeight}(\text{root}, \text{node}, \textbf{true})}{\text{node.contCount}}$
```

To assign sum weights $\lambda^h_{2i-1}$ and $\lambda^h_{2i}$ to every history $\hat{h}_i$ of the backoff graph we iterate over all nodes in the graph in Line 2. First some required data is prefetched (Lines 4 to 7), then the highest order weight is computed (Lines 9 to 14), and lastly that for lower orders (Lines 16 to 19).

*node.index* is simply the position of the total ordering of the backoff graph the node got assigned. The actual ordering does not matter, it however has to be consistent such that $\hat{h}_{\text{node.index}}$ is the history that belongs to that node. Accessing a count value like $c(\hat{h}_{\text{node.index}} \square)$ in Line 4 is a potentially expensive operation, as a separate data structure has to be queried. *node.count*, *node.contCount*, and *node.gamma* shall simply be floating point variables that are stored local to each node and thus fastly accessible. A possible implementation that supports this is showcased in Section 3.2.5. Line 7 is listed under data lookup because the computation of $\mu$ can simply be replaced by a lookup to a precomputed table with two-dimensional keys. *node.layer* is the layer of the node in the backoff graph.

The properties *node.count*, *node.contCount*, and *node.gamma* are not only needed for calculations of the current node, but also those of all its descendants (for example Line 21). Because of this an implementation has to ensure that the for-each loop in Line 2 only iterates over a node in layer $l$ if all nodes of $l-1$ have been previously iterated.

**Algorithm 3.3** Helper functions for GLM sum weight computation

```
20: function HIGHESTORDERWEIGHT(ancestor, node)
21:     if ancestor.count ≠ 0 then                    ▷ if ancestor is seen
22:         return 0  ▷ this path does not contribute to highest order

23:     if ancestor.isParentOf(node) then                  ▷ end of path?
24:         return 1

25:     sum ← 0
26:     for each child in ancestor.childs() do
27:         if child.isAncestorOf(node) then
28:             sum ← sum + HIGHESTORDERWEIGHT(child, node)
29:     return sum
```

```
30: function LOWERORDERSWEIGHT(ancestor, node, firstSeen)
31:     ▷ calculate mult
32:     if firstSeen and ancestor.count ≠ 0 then
33:         factor ← $\frac{\text{ancestor.gamma}}{\text{ancestor.count}}$
34:         firstSeen ← false
35:     else if ancestor.contCount ≠ 0 then
36:         factor ← $\frac{\text{ancestor.gamma}}{\text{ancestor.contCount}}$
37:     else
38:         factor ← 1

39:     if ancestor.isParentOf(node) then                  ▷ end of path?
40:         if firstSeen then
41:             return 0     ▷ node occurs as highest order for this path
42:         return mult

43:     sum ← 0
44:     for each child in ancestor.childs() do
45:         if child.isAncestorOf(node) then
46:             sum ←
47:               sum + LOWERORDERSWEIGHT(child, node, firstSeen)
48:     return factor · sum
```

Next the highest order weight $\lambda^h_{2i-1}$ is calculated. That weight is zero if the node's history is unseen (Lines 9 and 10), as of Equation (2.11). If the current node is the root node, we can specify the weight directly (Lines 11 and 12), as of Equation (2.10). Else we will have to apply the reasoning we outlined above and define the sum weight as the fraction of coefficient $\mu$, number of paths, and denominator (Lines 13 and 14). HIGHESTORDERWEIGHT(*ancestor*, *node*) is a helper function that just counts the number of paths that only spans unseen nodes from an *ancestor* node to the current *node* in a recursive manner.

Lastly the weight $\lambda^h_{2i}$ for lower orders is calculated. Again if the term that would occur in the denominator *node.contCount* is zero, that term is assigned a weight of zero. This also happens for the root note, as it can never occur in a lower order term (Lines 16 and 17). Else a

**Listing 3.1** C data structures to implement the backoff graph

```
struct node {
  int count;
  int contCount;
  double gamma;
};

struct graph {
  char∗ history[n];
  struct node nodes[n ∗ n]
};
```

similar term to the highest order weight is computed, however with a different helper function (Lines 18 and 19).

That helper function LOWERORDERSWEIGHT(*ancestor*, *node*, *firstSeen*) forms the core of the algorithm. We have reasoned above that the weight for lower orders is proportional to a sum of multiplication, where each multiplication is built from one path from the root to the current node and each node contributes a $\frac{\gamma}{\text{denominator}}$ factor. This however depends on the node: if a node is the first seen one on the path it contributes a $\frac{\gamma(\hat{h}_\bullet)}{c(\hat{h}_\bullet \square)}$ factor (Lines 32 and 33) as of Equation (2.10). Following nodes on the path contribute a $\frac{\gamma(\hat{h}_\bullet)}{N(\bullet\,\hat{h}_\bullet\,\bullet)}$ factor (Lines 35 and 36) as of Equation (2.12). Nodes that are unseen contribute the neutral element of multiplication, i.e. they don't change the weight (Lines 37 and 38) as of Equation (2.11).

Also we don't just compute sums of multiplications but instead apply the distributive property were possible to reduce computational cost. Each recursive step of the helper function thus returns only a *factor* times *sum* product, where the sum consists of incarnations of further recursion depth (Lines 43 to 48).

### 3.2.5 *Implementation*

This subsection will explain our approach of implementing the backoff graph that both consumes as little space as possible while being fast to traverse. Our representation fits into a small contiguous amount of memory on the stack, which will prevent cache misses, and thus result in access time improvements.

We reasoned in Section 3.2.2 that a backoff graph, for a history $h$ with length $|h|$, has $2^{|h|}$ nodes. Our core idea is to now store the graph into an array of $2^{|h|}$ `structs`.

We will interpret each node's history as a binary number, where each set word in the history becomes a zero and each skip becomes a one. That is the node of history $w_1 \square w_3 w_4 \square$ for a graph of order 5 is stored at the index $\texttt{01001}_2 = 9$. That number is used as the index for each node in the array. This representation makes it very fast to compute the layer, all parents, or all children of a node. These tasks can be specified entirely as bit operations, which should be available

**Listing 3.2** C code to compute the next iteration index

```
if (index == 0)
    index = 1:
    return;

int x = numberOfSetBits(index);

// compute lexicographically next bit permutation
// with x bits set
int t = (index | (index - 1)) + 1;
index = t | ((((t & -t) / (index & -index)) >> 1) - 1);

// highest number with x bits set
int m = ((1 << x) - 1) << (n - x);
if (index > m)
    // next is smallest number with (x+1) bits
    index = (1 << (x + 1)) - 1;
if (index > (1 << n) - 1)
    // restart, if next number would be more than n bits
    index = 0;
```

as direct machine instructions for most architectures. No materialized lists of all parents or all children are necessary.

The C code of Listing 3.1 exhaustively lists necessary data structures to implement Algorithms 3.2 and 3.3. `n` has to be a predefined constant that specifies the (maximal) length of histories. The `struct node` consists of exactly those data members that are used for data lookup storage in the algorithm. `history` stores the $n$-gram $h$, and `nodes` bundles all nodes of a graph in one instance.

We will now explain how all necessary data can be retrieved from that representation:

- The history $\hat{h}_i$ of a node at index $i$ is simple the original `history`, where a skip $\square$ is placed at each position, where the node's index has a set bit.
- A node is the top root node if its index has no set bits, and in turn a node is the bottom leaf node if all bits in the index are set.
- The layer of a node is the number of set bits in its index.
- As explained in Section 3.2.2 a node at layer $l$ has exactly $l$ parents and $|h| - l$ children.
- The index of the ith parent (child) of a node is found by unsetting (setting) the ith set (unset) bit.
- A node `A` is an ancestor of another node `B` if it the bitwise and operation on their indices returns `A`'s index respectively it is an descendant if the same holds for the bitwise or operation:

$$\texttt{A} \text{ ancestor of } \texttt{B} \Leftrightarrow \texttt{A.index \& B.index == A.index}$$
$$\texttt{A} \text{ descendant of } \texttt{B} \Leftrightarrow \texttt{A.index | B.index == B.index}$$

with & the bitwise-and, and | the bitwise-or.

The only non-trivial problem that follows from that representation is finding the proper ordering of indices in which they should be accessed during an in-order loop over the graph. An in-order loop over the backoff graph iterates all nodes, such that a node of layer $l$ is only accessed once all nodes of layer $l-1$ have been processed.

Listing 3.2 gives an algorithm in C code that generates the next in-order index[1]. As input it requires the previous `index` and the backoff graph's order $\texttt{n} = |h|$. To ensure the fastest possible execution time the algorithm aims to be a minimal composition of bit-level operations.

## 3.3 MONOTONY

We now want to prove that the probability definitions for Modified Kneser-Ney Smoothing (Section 3.1) and the Generalized Language Model (Section 3.2) are monotone functions on the $\alpha_i^h(w)$. This means that higher $\alpha_i^h(w)$ values correspond to higher probabilities. We can formalize that property as:

$$\left(\forall i \in \{1,\ldots,N\}\colon\ \alpha_i^h(w) \geqslant \alpha_i^h(w')\right) \implies P(w\,|\,h) \geqslant P(w'\,|\,h) \quad (3.14)$$

This is an important property because it allows the application of top-$k$ joining techniques in Chapter 4.

Because we have defined these probabilities as sums of weights $\lambda_i^h$ multiplied with terms $\alpha_i^h(w)$ (Equation (3.1)), and since addition and multiplication are trivially monotone, it suffice to show that the $\lambda_i^h$ are always non-negative to prove that our probability definitions are monotone.

To show that $\lambda_i^h$ are non-negative we take at look at their definitions (Equation (3.7), Algorithms 3.2 and 3.3), and observe that they are defined as a combination (multiplication and division) of occurrence counts $c(\cdot)$ and $N_{1+}(\cdot)$, numbers of paths, and interpolation weights $\gamma(\cdot)$. Since by their nature occurrence counts and numbers of paths are natural numbers, we only need to show that the interpolation weights $\gamma(\cdot)$ are also non-negative. These interpolation weights are defined as sums of discount values $D_\bullet$ multiplied with continuation counts $N_\bullet(\cdot)$ (Equation (2.17)). Since continuation counts again are defined as natural numbers, it remains to prove that the discount values are non-negative.

A non-formal argument is the fact that the name and usage of discounting terms would be nonsensical if negative values would be permitted. Additionally we can derive this fact from the definition of the discounting values $D_1, D_2, D_{3+}$ (Equation (2.16)). It is easy to see that these discounting values increase as the number of n-grams $n_\bullet$ in the corpus increases. The factors $Y\frac{n_2}{n_1}$, $Y\frac{n_3}{n_2}$, $Y\frac{n_4}{n_3}$ reach the limit of $\frac{1}{2}$ for an infinitely large corpus, and thus infinitely large number of

1 The code to compute the lexicographically next bit permutation is courtesy of `http://graphics.stanford.edu/~seander/bithacks.html#NextBitPermutation` (visited on July 14, 2015). It requires an architecture that stores negative integers in two's complement format.

n-grams $n_\bullet$. As a result of the fact that these factors are limited, none of the discount values can be non-negative. Thereby we can see that our probability definitions are indeed monotone.

# 4 FAST PREFIX QUERIES USING TOP-$k$ JOINS

Update math notation once it has been fixed in Chapter 3.

As motivated in the introduction, we want to solve prefix queries based on statistical language models. Our objective is to reduce the execution time of a high number of next word prediction queries for arbitrary histories and prefixes. Equation (1.1) shows that next word prediction queries $\mathrm{NWP}_p^k(h)$ are an argmax of probabilities. To optimize query time we follow a two-pronged approach: First we identify calculations shared by multiple probabilities, and perform them only once up front. Secondly we try to avoid enumerating the whole argument space of the argmax. Instead our goal is to always compute the probability for the next most promising argument, and to terminate as soon as we are sure to have found the $k$ words that maximize the probability.

As specified in Chapter 2 each language model probability is calculated from occurrence counts of sequences in a corpus. In order to avoid having to analyze the corpus on each query for its relevant sequences, the number of occurrence for all contained sequences in a corpus is counted up front in a learning phase. We store those counts in a data structure that is optimized for necessary retrieval. We will explain that data structure and define what necessary retrieval means in Section 4.2. Subsequently we may persist that data, so that for each requested corpus the necessary analyzation only has to be performed once.

To find the answer for a query $\mathrm{NWP}_p^k(h)$, probabilities $P(w \mid h)$ have to be calculated for a fixed history $h$ but many different $w$. We have shown in Chapter 3 that we can represent each probability as a weighted sum that resembles Equation (3.1). The weights $\lambda_i^h$ of that sum are composed of exactly these counts that are independent of the argument $w$. This enables us to calculate the $\lambda_i^h$ once beforehand and reuse them for every probability calculation of a query. The remaining factors $\alpha_i^h(w)$ depend on $w$. This means we can find a function $s$ that expresses our probabilities[1]:

It is not visible from the notation that $s$ depends on the $\lambda_i^h$.

$$P(w \mid h) = s\left(\alpha_1^h(w), \ldots, \alpha_N^h(w)\right) \tag{4.1}$$

## 4.1 TOP-$k$ JOINING TECHNIQUES

The problem we are trying to solve, namely finding the $k$ words $w$ that maximize our probability $P(w \mid h)$, is actually well researched under the topic of *top-$k$ joining techniques*. The result of a join operation is the set of all combinations of tuples from multiple data sources,

1 The definition for $s$ is just Equation (3.1). This new representation is only introduced to make it clear, that given the $\lambda_i^h$ the probability can be calculated from the $\alpha_i^h(w)$.

where each tuple combination satisfies the *join condition*. Top-k joins are then these that only return the k highest scoring according to some *scoring function*.

We can easily see that our prefix queries fit the scheme of top-k join queries. To calculate the probability of any word $w$, we have to look up all $\alpha_i^h(w)$ that belong to that word. Finding all combinations $\left(\alpha_1^h(w), ..., \alpha_N^h(w)\right)$ for each word $w$ is actually a join operation with the join condition that the $w$ is constant over the $\alpha_i^h(w)$ of a single combination. As defined, for a query $\mathrm{NWP}_p^k(w)$ we only want to return the k best predictions. The k best predictions are those that yield the k highest values according to a scoring function, which in our case is the probability. Each join combination gives the arguments for the scoring function, as shown in Equation (4.1).

Almost all top-k joining techniques that try to be more efficient than enumerating all join combinations, require *monotone* scoring functions. Monotone in our context means that higher $\alpha_i^h(w)$ values correspond to higher probabilities. We have shown in Section 3.3 that our probability definitions are monotone on the $\alpha_i^h(w)$. This fact can be utilized to find the top-k predictions by always calculating the probability for the next promising $w$ with the highest $\alpha_i^h(w)$.

Finding the $w$ with the highest $\alpha_i^h(w)$ requires sorting. Since join combinations $\left(\alpha_1^h(w), ..., \alpha_N^h(w)\right)$ are N-dimensional there is no meaningful order defined on them, besides sorting by the scoring function. However that requires calculating the probability for each join combination, which is precisely what we are trying to avoid. For this reason we instead maintain N data relations, one for each $\alpha_i^h(w)$, and sort them independently in the learning phase. During query time, we can then perform *sorted access* on these relations, that is for a given history h we want to retrieve words $w$ paired with and sorted by their $\alpha_i^h(w)$. Some top-k joining techniques additionally require *random access*, that is for a given history h and a given word $w$ we want to find $\alpha_i^h(w)$. The data structure we use for this will be explained in Section 4.2.

Among other things, top-k joining techniques can be classified by the type of join operation they support (Ilyas et al. 2008). The join operation for our problem is whether for two values $\alpha_i^h(w_x)$ and $\alpha_j^h(w_y)$ the $w_x$ and $w_y$ are equal. This is actually the simplest and most common type of join operation and is called an *equi-join*. They have the nice property that for each $\alpha_i^h(w)$ there is exactly one corresponding $\alpha_j^h(w)$ for all j that are different from i, which allows easy indexing in a hash table by the key $w$. Top-k joining techniques that operate on equi-joins are also called *top-*k *selections* (Ilyas et al. 2008).

Two fundamental algorithms were discovered by Fagin et al. (2001) and other authors independently: the *threshold algorithm*, which requires sorted and random access, and the *no random access algorithm*, which only requires sorted access. We will apply these two algorithms to the problem of next word prediction and compare them against each other and the simple argmax approach. Fagin et al. (2001) prove that both these algorithms are *instance optimal* solutions to top-$k$ join queries: there is no possible algorithm that provides a better solution over all possible database contents.

However algorithms which outsource some computation into a precompution step, and thus achieve a faster query response are still possible: there exists a plethora of such more sophisticated top-$k$ joining techniques, of which Ilyas et al. (2008) surveys a wide number. For example *onion indices* (Chang et al. 2000) and *rank join indices* (Tsaparas et al. 2003) feature an extended learning phase, in which optimized data representation are created over all arguments to the scoring function. These allow faster top-$k$ joins for arbitrary scoring functions over the same arguments. However optimizations like these are not applicable to our prefix query problem, as the history $h$ changes with each query $\mathrm{NWP}_p^k(h)$. Because of this the whole argument space $\alpha_i^h(w)$ changes, which in turn means we would have to built indices with each query, obviously defeating the purpose. The *Rank Join operator* (Ilyas et al. 2004) is another more recent algorithm, designed for fast solution in the presence of arbitrary join conditions. However in the precense of an equi-join it practically reduces to the threshold algorithm.

## 4.2 CORPUS DATA STRUCTURE: COMPLETION TRIE

For our needs we require a data structure that can return sorted pairs $(w, \alpha_i^h(w))$ for each $i$ for a given history $h$. Additionally as we are solving prefix queries, it has to be possible to specify a prefix $p$ with which all $w$ have to start.

On top of that the threshold algorithm requires random access to the data structure, that is for a given $w$ we want to find $\alpha_i^h(w)$. Random access might yield additional information to an algorithm and thus allow it to terminate earlier. Also our definitions to calculate the sum weights $\lambda_i^h$ given in Chapter 3 require this form of random access, so it is desirable for our data structure to support it. The alternative would be to maintain a separate data store that is optimized for random access — for example a hash map — which would necessitate a huge memory usage.

Another desirable characteristic of data storage is data compression. Since the analyzation of larger corpora yields to more accurate language models [Citation needed], a higher degree of data compression allows to perform more accurate word prediction on the same hardware. Obviously a balance between data compression and access time has to be found, since fast prefix queries are the goal of this thesis.

The data structure which satisfies all these requirements is the *completion trie* described by Hsu and Ottaviano (2013). A completion trie is a compact prefix tree that allows storage of arbitrary pairs of strings and integer-scores. A prefix tree is an ordered tree data structure where each node stores a character sequence. Retrieving a full string from the trie is done by concatenating all character sequences along a path from the root to a leaf node. In a completion trie each node also stores the highest score of all its leaf nodes. This enables very fast retrieval of all string-score-pairs sorted by score, where the string start with a requested prefix, hence the name *completion* trie.

Hsu and Ottaviano (2013) devise a method for storing only a minimal number of tree edges, while maintaining data locality. Further employing a variable-byte encoding they report an average compression ratio of their data structure of 51% compared to raw uncompressed text files. In the same publication Hsu and Ottaviano (2013) present two alternative data structures for the same task that achieve compression ratios of 29% respectively 30%. However they report access times which are at least twice as high as those of the completion trie. Since speed is the primary concern of this thesis these alternatives were not further considered.

For $1 \leqslant i \leqslant N$ we maintain $N$ many completion trie instances and store pairs $(h * w, \alpha_i^h(w))$ in each trie for all possible histories $h$ and words $w$, where $h * w$ is the concatenation of $h$ and $w$. This enables all our desired accessing schemes:

Actually $\hat{h}$.

Querying i-th trie is only said implicitly.

- Sorted access for some history $h$ is done by querying for completions of $h$. Results are then pairs where the string starts with the requested history. The remaining part of the string is the word $w$. Results are sorted on the score $\alpha_i^h(w)$.
- A prefix $p$ with which $w$ has to start is easily included. Querying for completions of $h * p$ returns the same pairs as above, but only those where $p$ is a prefix in $w$.
- Random access for a fixed history $h$ and fixed word $w$ is achieved by walking the path along the trie that is described by $h * w$. If that path ends in a leaf node, this node's score is $\alpha_i^h(w)$. If that path does not exists, or the found node is not a leaf node, $\alpha_i^h(w)$ is zero.

## 4.3 THRESHOLD ALGORITHM

The *Threshold Algorithm*, applied to our problem $\mathrm{NWP}_p^k(w)$, roughly works as follows (Fagin et al. 2001):

1. We perform sorted access to data sources (in our case completion tries are queried with prefix $h * p$) in any order. One strategy might be a round robin approach, that is each source is accessed once in turn.
2. Every time a pair $(w, \alpha_i^h(w))$ is a encountered with a new word $w$, we look up all other $\alpha_j^h(w)$ for $j \neq i$ with random access. With these the word probability $P(w|h) = s(\alpha_1^h, \dots, \alpha_N^h)$ can be calculated. We keep track of the $k$ pairs $(w, P(w|h))$ with the highest probabilities.
3. Let $\overline{\alpha}_i$ denote the last count $\alpha_i^h(w)$ that was encountered during sorted access to the ith data source. Then the *threshold* $t = s(\overline{\alpha}_1, \dots, \overline{\alpha}_N)$ is an upper bound for the probability of all words that have not yet been seen.

4. Once we have encountered $k$ probabilities greater than the current threshold $t$ the algorithm terminates and the $k$ best pairs $(w, P(w \mid h))$ are returned.

Our implementation of the *Threshold Algorithm* is given in Algorithm 4.1.

As input we require $N$ completion tries from which we can extract $(w, \alpha_i^h(w))$ pairs in sorted or random order as described in Section 4.2. Further, as we are finding answers to an $\mathrm{NWP}_p^k(h)$ query, those parameters are necessary. Both *history* and *prefix* are strings that may be empty; *limit* has to be an integer greater zero. The output *predictions* will be a list composed of at most *limit* $(w, P(w \mid h))$ pairs, sorted descendingly by probability.

The calculation of probabilities depends on sum weights $\lambda_1, \ldots, \lambda_N$. They only depend on the fixed *history* and can thus be computed once at the start of each query (Line 1). To calculate the *threshold*, the upper bound for all of remaining probabilities, the factors $\overline{\alpha}_1, \ldots, \overline{\alpha}_N$ are required. Those are the last accessed counts $\alpha_i(w)$ of each trie. Before performing any trie-access we initialize them to the highest count in the trie (Lines 2 and 3). *Peeking* a data source is the operation that returns the next entry in it, without advancing its iterator.

*seen* is a set that keeps track of all processed words thus far (Line 4). It's purpose is to avoid random access for the calculation of probabilities that have been performed before. It is an optimization that enables a faster execution time at the cost of a higher space complexity. HashSet

*queue* keeps track of the word-probability-pairs with the highest probabilities that are candidates for prediction (Line 5). At any point in time it contains at most *limit* $+ 1$ entries. It is implemented as a priority queue, which we expect to be ordered on the probabilities of each entry. Thus efficient querying (Line 28) or removal (Line 22) of the entry with the lowest probability is possible.

After initialization is done we query words until we have found the *limit* best predictions. The NEXTTRIE() function in Line 7 returns the number $i$ of the trie to be queried next. It is thus stateful and needs to keep track of previous return values. In our implementation we perform a round-robin approach, e.g. tries are returned in the order $1, 2, \ldots, N, 1, 2, \ldots$ Once a trie is depleted it is omitted.

For each iteration we perform sorted access and retrieve the next word-count-pair from the current trie, that starts with the requested *history* and *prefix* (Line 8). The NEXT($\cdot$) method of a trie returns the current pair under the iterator and advances it in sorted order. If the word has been encountered in a previous iteration from a different trie we continue with the next iteration step (Lines 9 and 10). If the word is new, we add it to the list of seen words (Line 11).

In the following the probability $P(w \mid h)$ for the word is calculated (Line 18). The factor $\alpha_i$ is known from sorted access (Line 15), all other $\alpha_j$ with $j \neq i$ are retrieved by random access (Line 17). With the word's probability we update our record of the *limit* best predictions thus far (Lines 20 to 22). The sorting of the priority queue happens implicit with the PUSH($\cdot$) operation.

**Algorithm 4.1** *Threshold Algorithm* to solve $\mathrm{NWP}_p^k(h)$

```
Input: $C_1, \dots, C_N$                    ▷ corpus data stored in Completion Tries
Input: history, prefix, limit
              ▷ history $h$, prefix $p$ and limit $k$ for prefix query $\mathrm{NWP}_p^k(h)$
Output: predictions                          ▷ list of word-probability-pairs
 1: $\lambda_1, \dots, \lambda_N \leftarrow$ CALCSUMWEIGHTS(history)          ▷ see Chapter 3
 2: for $i$ from $1$ to $N$ do
 3:     $\overline{\alpha}_i \leftarrow C_i$.PEEKCOUNT(history $*$ prefix)       ▷ threshold factors
 4: seen $\leftarrow$ new SET()                                  ▷ set of seen words
 5: queue $\leftarrow$ new PRIORITYQUEUE()                ▷ prediction candidates
 6: while $i \leftarrow$ NEXTTRIE() do
 7:     ▷ perform sorted access, track seen words
 8:     word, count $\leftarrow C_i$.NEXT(history $*$ prefix)
 9:     if seen.CONTAINS(word) then
10:         continue  ▷ skip words already seen in previous iterations
11:     seen.ADD(word)
12:     ▷ compute word probability
13:     for $j$ from $1$ to $N$ do
14:         if $j == i$ then
15:             $\alpha_j \leftarrow$ count
16:         else
17:             $\alpha_j \leftarrow C_j$.GET(history $+$ word)          ▷ random access
18:     probability $\leftarrow \sum_{j=0}^{N} \lambda_j \alpha_j$
19:     ▷ keep limit-many best predictions
20:     queue.PUSH((word, probability))
21:     if queue.SIZE() $>$ limit then
22:         queue.REMOVELOWESTPROBABILITYENTRY()
23:     ▷ compute threshold
24:     $\overline{\alpha}_i \leftarrow \alpha_i$
25:     threshold $\leftarrow \sum_{j=0}^{N} \lambda_j \overline{\alpha}_j$
26:     ▷ stop if better predictions are impossible
27:     if queue.SIZE() $=$ limit and
28:       threshold $<$ queue.GETLOWESTPROBABILITY() then
29:         break
30: predictions $\leftarrow$ queue.TOLIST()
```

Afterwards the last access sorted count of the current trie is update (Line 24) and with it the *threshold* calculated (Line 25). If we already found *limit* predictions and the probability of all those predictions is higher than our upper bound of all remaining ones, the algorithm terminates (Lines 27 to 29). The resulting *predictions* are formed by the conversion of the *queue* to a list (Line 30), where the sorting of the *queue* is respected.

## 4.4 NO RANDOM ACCESS ALGORITHM

The *No Random Access Algorithm*, applied to our problem $\mathrm{NWP}_p^k(w)$, roughly works as follows (Fagin et al. 2001):

1. We perform sorted access to data sources (in our case completion tries are queried with prefix $h * p$) in any order. One strategy might be a round robin approach, that is each source is accessed once in turn.

2. We maintain a mapping that initially maps all words to *unkown* counts $\alpha_i^h(w)$. For every word-count pair queried, we set that count in the mapping to its now know value.

3. Using this mapping we calculate upper and lower bounds for all seen words. Let $\overline{\alpha}_i$ denote the last count $\alpha_i^h(w)$ that was encountered during sorted access to the $i$th data source. The upper bound for a word probability is calculated by replacing unkown counts with their highest possible value $\overline{\alpha}_i$. Lower bounds are calculated by replaced unkown counts with zeros.

4. Once $k$ words have been found, whose lower bound of probability is higher than all other upper bounds, the algorithm terminates and returns these $k$ best predictions.

Algorithm 4.2 shows our implementation of the *No Random Access Algorithm*.

The input requirements are the same as for the *Threshold Algorithm* described in Section 4.3. The output is similar, however we only return a list of predicted words without their probabilities, as the algorithm doesn't compute the exact probabilities. Also the initialization of sum weights $\lambda_1, \ldots, \lambda_N$ and last seen counts $\overline{\alpha}_1, \ldots, \overline{\alpha}_N$ is the same as before (Lines 1 to 3).

*wordCounts* is the mapping from words $w$ to $N$-tuples that contain the counts $\alpha_i^h(w)$ (Line 4). Unset values are interpreted as having *unkown* counts.

*queue* is a priority queue that stores (word, upper bound, lower bound) tuples (Line 5). As opposed to the *Threshold Algorithm* the queue needs to keep track of bounds for all encountered words, not just *limit* many. Entries in the queue are sorted by lower bounds and for objects with equal lower bounds sorted by their upper bounds.

*predictions* is the list of all possible word predictions (Line 6). Words in the list are sorted by probability descending without actually calculating the probability. It contains at most *limit* items.

**Algorithm 4.2** *No Random Access Algorithm* to solve $\mathrm{NWP}_p^k(h)$

```
Input: $C_1, \ldots, C_N$                ▷ corpus data stored in Completion Tries
Input: history, prefix, limit
        ▷ history $h$, prefix $p$ and limit $k$ for prefix query $\mathrm{NWP}_p^k(h)$
Output: predictions                                          ▷ list of words
 1: $\lambda_1, \ldots, \lambda_N \leftarrow$ CALCSUMWEIGHTS(history)      ▷ see Chapter 3
 2: for $i$ from $1$ to $N$ do
 3:     $\overline{\alpha}_i \leftarrow C_i$.PEEKCOUNT(history $*$ prefix)
 4: wordCounts $\leftarrow$ new MAP()       ▷ map from words to their counts
 5: queue $\leftarrow$ new PRIORITYQUEUE()                  ▷ all predictions
 6: predictions $\leftarrow$ new LIST()
 7: while $i \leftarrow$ NEXTTRIE() do
 8:     ▷ perform sorted access, track known counts per word
 9:     word, count $\leftarrow C_i$.NEXT(history $*$ prefix)
10:     if not wordCounts.CONTAINS(word) then
11:         $\alpha_1, \ldots, \alpha_N \leftarrow$ unkown, . . . , unkown
12:     else
13:         $\alpha_1, \ldots, \alpha_N \leftarrow$ wordCounts.GET(word)
14:         queue.REMOVE(word)
15:     $\alpha_i \leftarrow$ count
16:     wordCounts.SET(word, $(\alpha_1, \ldots, \alpha_N)$)
17:     ▷ compute word probability upper- and lower-bound
18:     $\overline{\alpha}_i \leftarrow \alpha_i$
19:     upperBound $\leftarrow 0$
20:     lowerBound $\leftarrow 0$
21:     for $j$ from $1$ to $N$ do
22:         if $\alpha_j =$ unknown then
23:             upperBround $\leftarrow$ upperBound $+\lambda_j \overline{\alpha}_j$
24:         else
25:             upperBound $\leftarrow$ upperBound $+\lambda_j \alpha_j$
26:             lowerBound $\leftarrow$ lowerBound $+\lambda_j \alpha_j$
27:     ▷ predict words with lower bound $\geqslant$ all other upper bounds
28:     queue.PUSH((word, upperBound, lowerBound))
29:     while true do
30:         word, upperBound, lowerBound $\leftarrow$ queue.POP()
31:         if lowerBound $\geqslant$ queue.GETLARGESTUPPERBOUND() then
32:             predictions.ADD(word)
33:         else
34:             queue.PUSH((word, upperBound, lowerBound))
35:             break
36:         if predictions.SIZE() $=$ limit then
37:             exit
38: for $i$ from predictions.SIZE() to limit do
39:     word, upperBound, lowerBound $\leftarrow$ queue.POP()
40:     if upperBound $= 0$ then
41:         break
42:     predictions.ADD(word)
```

The selection of the next trie with NEXTTRIE() (Line 7) and the following sorted access to that trie (Line 9) are equal to those of the *Threshold Algorithm*. Each new word-count-pair is stored in *wordCounts*. If the word has not been seen before (Line 10) we initialize all counts $\alpha_1, \ldots, \alpha_N$ as *unkown* values. Otherwise we load previously stored counts from the mapping (Line 13). Additionally as we know the word to receive updated bounds its entry from the *queue* (Line 14). Afterwards we update the one $\alpha_i$ we have retrieved the value for (Line 15) and write the resulting count-tuple back into the map (Line 16).

With the update counts we can compute a better upper and lower bound for the word probability. First we update the last accessed count of the current trie (Line 18). Bounds are then calculated as the weighted sums of $\alpha_j$ (Lines 25 and 26). In case the the value of any $\alpha_j$ is *unkown*, it is substituted with the highest possible value $\overline{\alpha}_j$ for the calculation of the upper bound (Line 23) or in the case of the lower bound it is not factored in at all.

The newly calculated bounds of the word are then added to the *queue* (Line 28). Afterwards all words that have a higher lower bound than the largest upper bound of all other words in the queue are added as final *predictions*. First the next word with the highest lower bound is retrieved from the queue with the POP() operation (Line 30). If that word's lower bound is greater or equal to the largest remaining upper bound in the queue (Line 31) we can add it to the list of *predictions* (Line 32) and repeat the process. Otherwise we add the word back into the queue and continue with new sorted retrieval. The algorithm terminates once *limit* predictions have been found (Lines 36 and 37). This procedure ensures that predictions are added in order of decreasing probabilities.

Attention must be paid to the upper bounds of all objects in the *queue*. Technically whenever a $\overline{\alpha}_i$ value is updated, the upper bound of each item in the queue, for with $\alpha_i$ is unknown, will change. So in practice one would have to recalculate all upper bounds and resort the list every time a $\overline{\alpha}$ value is updated. This is not shown in the code in the algorithm. However one can take advantage of the fact that our probabilities are monotone, and the sorting of items thus remains constant for changing $\alpha_i$. This mean one only needs to recalculate the upper bound in places where the actual value is needed.

A case than can occur for small corpora, is that all tries may be depleted before *limit* predictions have been established. Because of this we fill the *predictions* with words from the *queue* in order until *limit* many have been reached (Lines 38 to 42). However if remaining words have an probability upper bound of zero (Line 40) we prefer to have less than *limit* predictions, as the remaining ones are guaranteed not to match.

## 4.5 IMPLEMENTATION

Not actually $h * p$ but $\hat{h} * p$.

Can't actually store $\alpha(\cdot)$ but only counts $c(\cdot)$. Discounts fuck everything up and sometimes require random access even when only doing sorted.

Faster threshold calucation.

Can't just do $C_i$.NEXT$(\cdot)$, iterators needed instead.

Optimized NEXTTRIE(i)mplementation possible with blah...

Set zero alpha bars.

## 4.6 DISCUSSION

Space requirements of algorithms.

NRA can give predictions.

NRA sorting may be bad for small corpora.

Random access may be far more expensive on distributed systems.

Corpus needs to be preanalyzed.

Access strategy: Mention Güntzer et al. (2000) and Güntzer et al. (2001) improvements to TA and NRA.

## 4.7 ALTERNATIVES

Why beam search is impossible.

Beam search is a path finding algorithm that always expands the most promising states, and terminates when a solution was found.

This however is not directly applicable to our problem, as finding a solution is typically very easy (get top node from sorted pattern counts and take it as solution).

How to interpret: expanding the most promising state?

1. Peek sorted counts, list with highest count is most promising.
2. Now how to determine if we want to do random access to missing pattern counts or rather fetch next sorted count.
3. Could do beam width many sorted accesses, and complete missing patterns with random acccess and call it a day.

For random access the cost of an edge is only known, once the edge is walked. For sorted access we can peek.

# 5 EVALUATION

We will now evaluate the benefit of the two independent optimizations we proposed to next word prediction: representing language model probabilities as weighted sums (Chapter 3) and calculating the word with the highest probability via top-k joining (Chapter 4). To this end we empirically compare their performance in a variety of usage scenarios.

This chapter is organized as follows: first the experimental setup used over all experiments is described in Section 5.1, then our form of data visualization is explained in Section 5.2, and finally weighted sums are evaluated in Section 5.3 and top-k joining in Section 5.4.

## 5.1 EXPERIMENTAL SETUP

As explained in Chapter 2, the calculations of the considered language models' probabilities depend on occurrence counts obtained through statistical analysis of text corpora. For this evaluation the *Open American National Corpus (OANC)* by Ide and Suderman (2007) was used. It is a free collection of written text and transcripts of spoken data, featuring both historical and contemporary American English of all genres. It contains around 600 thousand sentences which amount to 14 million words in total.

In our experiments only the written part of that corpus was used, because next word prediction has no application to speaking. The corpus comes tagged with semantic sentence boundaries which were used to split the data into sentences. No begin or end of sentence tags were inserted, as to not falsify the actual occurrence counts. Tokenization was performed by assuming word boundaries to occur at each space. To avoid punctuation marks directly following words being counted as only one word, all special characters were surrounded by spaces. This means for our analysis, punctuation marks count as regular words. This has the downside of text like `you're` becoming `you ' re`. Other than that, the data was left as is. In particular no case normalization, stemming or removing of names was performed.

That text was then randomly split into 80% training and 20% heldout data. To measure how the algorithms under examination behaved for different sizes of training data, we successively discarded 50% of sentences from training. Through this we obtained five datasets equaling 100%/50%/25%/12.5%/6.25% of the original training corpus size. Table 5.1 shows the different datasets. Distinct language models were then learned on each of these training sets.

From the heldout data 40,000 non-overlapping sequences of ten words were selected. For our experiments all but the last words of that sequence were considered to form the history, and the last word then is the word for which either a probability is calculated, or which

| Corpus | No. Sentences | No. Unique Words | Raw File Size |
|---|---|---|---|
| `OANC` | 608,277 | 196,345 | 71.0 MiB |
| `Training-100%` | 486,409 | 178,671 | 57.0 MiB |
| `Training-50%` | 243,204 | 133,038 | 29.0 MiB |
| `Training-25%` | 121,602 | 98,209 | 15.0 MiB |
| `Training-12.5%` | 60,801 | 70,988 | 7.1 MiB |
| `Training-6.25%` | 30,400 | 50,372 | 3.6 MiB |

**Table 5.1:** Overview of the different datasets used for training.

is expected to be predicted by next word prediction. For cases were shorter histories of words were needed, words were removed from the front of the sequences, so the words whose probability are calculated respectively which are predicted remain the same. To avoid the problem of how to handle unknown words in the evaluation, these sequences were chosen as to not contain any words, that do not occur in the smallest 6.25% training corpus. Because of this no pruning of words had to be performed, and an `UNK` token was not assigned an artificially occurrence count.

The evaluation was performed using the *Generalized Language Modeling Toolkit (GLMTK)*[1], a freely available toolkit in Java to compare different language modeling techniques, that I wrote as part of my work as a student research assistant. Initially it was able to perform the traditional, recursive calculations for both Modified Kneser-Ney Smoothing and the Generalized Language Model. For this thesis I implemented next word prediction using all optimizations described in Chapters 3 and 4 as well as all experiments from this chapter.

Tag bachelor commit and mention in footnote.

All experiments were run on a virtual machine that was assigned eight 3.0 GHz processor cores, each with a cache size of 4096 KiB. Although multiple cores were available, all experiments were run in a single thread, one after another, to avoid thread communication influencing the benchmark. We used Linux (3.2.0-75-virtual) as the operating system, and the OpenJDK with java version 1.7.0_79 as the implementation of the Java Platform. The Java Virtual Machine (JVM) was assigned 16 GiB of main memory, a limit that none of the experiments hit.

Java Virtual Machines are know to exhibit "warm-up" behavior [Citation needed]. That is it takes some time while running a program to find hotspot sections in the code and JIT compile these. Traditionally experimenters want to avoid that benchmarks are distorted by this warm-up behavior [Citation needed]. To circumvent this, the code under benchmark is usually executed multiple times, to ensure JIT compilation of the relevant sections, and the actual benchmark is only performed afterwards. To this end, we decided to run each experiment twice: the first execution is for warming-up the Java Virtual Machine, while actual tracking of performance metrics only occurs in the second run. As each experiment consists of executing 40,000 test

1 The *Generalized Language Modeling Toolkit (GLMTK)* is freely available under `https://github.com/renepickhardt/generalized-language-modeling-toolkit/`.

sequences, we are confident that the warm-up is completed before the second execution.

## 5.2 DATA VISUALIZATION: BOX PLOTS

Because of the large sample size of 40,000 testing sequences, showing all measured data points in diagrams is unfeasible. Instead statistical characteristics of the measured distributions of data have to be displayed. In our experiments, it turned out that finding fitting statistical characteristics is non-trivial. Natural choices for this are the mean and the standard deviation of distributions. However these are only suited to accurately describe measurements that are normal distributed, and are not suitable for our needs because of reasons:

- Some measured distributions exhibited a non-negligible amount of *right-skewness*. In other words the mass of the distribution is concentrated on the left. This case occurred when measuring the runtime of algorithms, and can be explained by increased communication of the operation system during the calculation of a tiny part of the test sequences. Such outliers should not unduly affect the statistical characteristics, which is not true for the mean and the standard deviation. On top of this the standard deviation is only capable of showcasing symmetric dispersion, so its use would be misleading for skewed distributions.

- Another observation is, that a few distributions are *multimodal*. That is multiple distinct peaks are visible in a density histogram of that distribution. Again this mostly occurred when measuring the runtime of algorithms, and can be explained by the fact that the 40,000 test sequences can be partitioned into several classes of different computational costs. For example calculating probabilities is vastly different depending on how many words of the history were seen in the training corpus. Both the mean and the standard deviation do not accurately describe multimodal distributions.

Instead, more *robust*, *non-parametric* statistical characteristics have to be used, that do not impose assumptions about the underlying measured distributions. One common form of visualizing data that fits these requirements is the use of *box plots*. Box plots visually describe the measured distribution through the combination of five characteristics. We follow the convention of Tukey (1977):

- As the name suggests the defining feature of box plots is their use of boxes. The lower/upper bar of the box gives the location of the first respectively third *quartile* of the data. So 50% of data measuring points fall into the plot's box.

- The band inside the box gives the location of the *median*. By its position relative to the quartile it is possible to visually estimate the skewness of the distribution.

- The box is surrounded by whiskers, who reach to the smallest/largest datum in the sample. But their length is limited to at most 1.5 *inter quartile ranges*. Every measuring points outside of that range is considered to be an outlier. Outliers are not shown in the plot.

## 5.3 PROBABILITY CALCULATION THROUGH WEIGHTED SUMS

We now want to evaluate how our method of calculating language model probabilities through weighted sums $P(w \mid h) = \sum_{i=1}^{N} \lambda_i^h \cdot \alpha_i^h(w)$, described in Chapter 3, fares against the traditional, recursive approach. The metric we will measure is *calculation time*. Comparison from a space complexity point of view is unnecessary, as both approaches only require a data structure that allows obtaining the occurrence counts of arbitrary sequences[2].

Our first experiment aims to compare the calculation time of one single probability for both approaches and both considered language models, independent of any application. To this end we tracks the calculation time that is necessary to calculate the conditional probability $P(w \mid h)$ of test sequences. The last word of a test sequence gives the word $w$ for which the probability is calculated, while the first to the second to last words of the sequence form the conditional history $h$. Probability calculation time depends mostly on the order of the employed Markov assumption, as this directly determines the length of the considered history and thus the recursion depth of the calculation. In the experiment we therefore track the calculation time for different $n$-gram lengths of the the sequence under testing. Lower length sequences are derived from higher length ones by removing the word at the beginning, so that the words $s$ under prediction remains the same over all $n$-gram lengths. The results are shown in Figure 5.1.

In Figure 5.1a for Modified Kneser-Ney Smoothing we observe a linear or better relation between $n$-gram-length and calculation time. In the range of 1- to 4-grams the calculation time of the traditional, recursive approach is slightly lower, while from 5-grams onwards the weighted-sum method comes of a bit faster. Without measuring any kind of significance, our conclusion from this figure is that both approaches perform probability calculation in approximately the same time. This can be explained by the fact that both techniques calculate results in exactly the same manner. The weighted-sum approach does not lead to any reduction in complexity, as it moves the recursive calculations into the generation of weights.

Figure 5.1b gives the same experiment for the Generalized Language Model. Because of the log-scale on the y-axis we conclude

2 One might argue that the weighted sum method also needs to store the sum weights $\lambda_i^h$. But as we only calculate these values at query time for a specific query, only $N$ weights have to be stored per query. As explained in Chapter 3 $N$ is usually a very small number. For example calculating probabilities under a 5th order Markov assumption — that is the length of the history $h$ is capped to 4 words — requires storing at most 5 (MKN, Equation (3.5)) respectively 32 (GLM, Equation (3.12)) floating point numbers. Such a small requirement of memory can be neglected

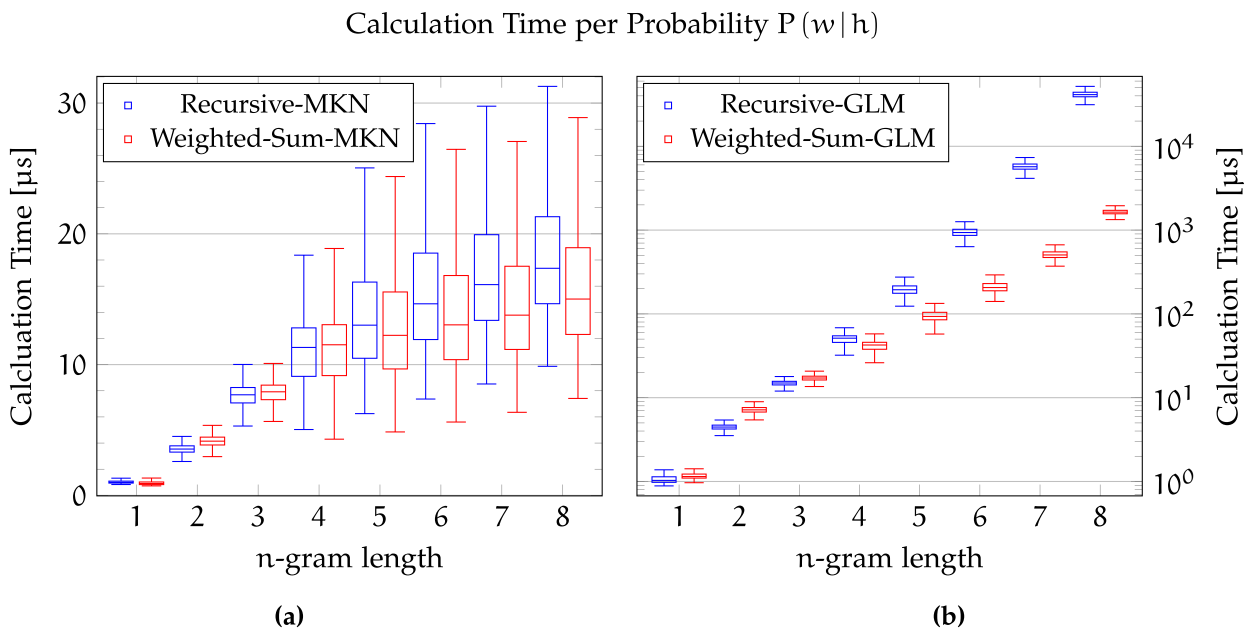


**Figure 5.1:** Time to calculate one probability for different n-gram lengths comparing the traditional, recursive approach (Chapter 2) as well as our novel weighted-sum approach (Chapter 3) to calculate both Modified Kneser-Ney Smoothing (MKN) and the Generalized Language Model (GLM).

a clear exponential relation between n-grams-length and calculation time for both approaches, Again for short n-grams lengths (1- to 3-grams) the traditional, recursive approach is slightly faster, while for longer n-grams (4-grams and onwards) the weighted-sum method gets increasingly better. A reduction in algorithmic complexity did not take place, as the relation is still exponential, but the weighted-sum method seems to be performing its calculations in a much more efficient manner. For 5-grams, the most widely used value in practice (Goodman 2001; Jurafsky and Martin 2009; Stolcke 2000), we observe that the weighted-sum approach is about double as fast.

But the main advantage of the weighted sum approach lies not in the fact that the calculation for a single, arbitrary probability is faster. Instead in the application of next word prediction, it allows to calculate the sum weights $\lambda_i^h$ only once in a pre-computation step of each query. This is possible because for a next word prediction query, all required probabilities depend on a constant history $h$, as explain in Chapter 3.

An interesting question is now: how much time of each probability calculation is actually used to calculate the weights, and can therefore be saved by off-loading its calculation to a pre-computation step? Figure 5.2 shows this proportion for the previous experiment. We can see that the amount of saved work increases with n-gram length. For the interesting value of 5-grams we note a work load reduction from about 55% for Modified Kneser-Ney Smoothing to 70% for the Generalized Language Model with an application like next word prediction.

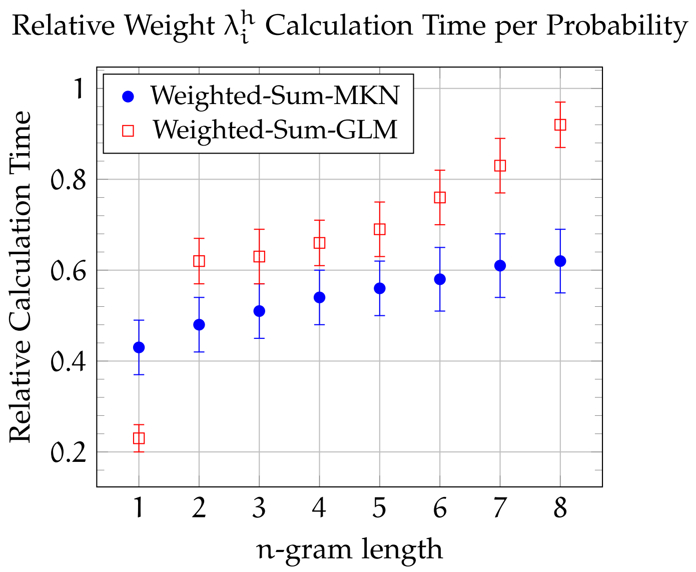


**Figure 5.2:** Proportion of the calculation time per probability that is used for the calculation of the weights $\lambda_i^h$. Given for the weighted-sum approach of both Modified Kneser-Ney Smoothing (MKN) and the Generalized Language Model (GLM). A value of 0.6 means that 60% of time was used to calculate the weights and the remaining 40% of time was used to calculate the final probability using these weights. Shown is the mean of the measured values with the errors bars displaying the standard deviation.

We conclude that there is no downside to the weighted-sum approach. For Modified Kneser-Ney Smoothing it is able to calculate probabilities in approximately the same time, while it is about twice as fast for the Generalized Language Model. Additionally, in the application of next word prediction, it to further reduce the calculation time by around 55% respectively 70% depending on choice of language model.

For different percentages of seen sequences? Number of weights per model.

## 5.4 NEXT WORD PREDICTION

### 5.4.1 *Prediction Quality*

First we will compare the achieved prediction quality of the employed language models. Note that for this experiment the choice of underlying algorithm does not matter. The Simple Argmax, the Threshold Algorithm, and the No-Random Access Algorithm always predict the same words, they just compute these results in different ways.

A standard metric to evaluate the quality of a word prediction system is the degree of *normalized keystroke savings* (Bickel et al. 2005; Swiffin et al. 1987; Trnka 2011). They measure the percentage of keystrokes that can be omitted by using next word prediction com-

| | Normalized Keystroke Savings | |
|---|---|---|
| n-gram length | MKN | GLM |
| 2 | 0.44 | 0.44 |
| 3 | 0.50 | 0.50 |
| 4 | 0.51 | 0.51 |
| 5 | 0.51 | 0.52 |

**Table 5.2:** Comparison of normalized keystroke savings for Modified Kneser-Ney Smoothing (MKN) and the Generalized Language Model (GLM) over varying $n$-gram lengths. A prediction was only considered valid if the intended word occurred as the top-1 prediction.

pared to letter-by-letter typing. Normalized keystroke savings (NKSS) are calculate as (Trnka 2011):

$$\text{NKSS} = \frac{\text{keystrokes}_{\text{normal}} - \text{keystrokes}_{\text{with prediction}}}{\text{keystrokes}_{\text{normal}}} \tag{5.1}$$

In the context of next word prediction we are only concerned with the keystroke savings of entering single, independent words. For our purpose $\text{keystrokes}_{\text{normal}}$ thus is the number of characters of a word, while $\text{keystrokes}_{\text{with prediction}}$ is the number of characters of a word one has to type until the system suggests the indented word as a completion. This however is not a clear definition: most next word prediction systems do not only offer a single prediction after each keystroke, but a list of $k$ predictions from which the user can choose. Measurements of normalized keystroke savings are thus inherently depend on the choice of $k$.

There also is no clear consensus on how keystrokes are counted (Trnka 2011). For this thesis, we count each character as a single keystroke, independent of case or accessibility from a normal keyboard. Word separating characters like spaces or newlines are not considered in our analysis, as they do not matter for next word prediction. On top of that we did not count the process of selecting the indented word from the list of predictions as a keystroke to enable keystroke savings to lie in the complete range from zero to one.

Table 5.2 shows our measurement of normalized keystroke savings with the testing sequences for both considered language models. As is apparent, the choice between Modified Kneser-Ney Smoothing and the Generalized Language Model is largely irrelevant from a keystroke savings point of view, at least for our experimental setup: for two significant digits, all values match, with the exception of the last row, which only differs slightly. The Generalized Language Model however comes with severely higher costs in both required space and runtime as we will see in Sections 5.4.2 and 5.4.3.

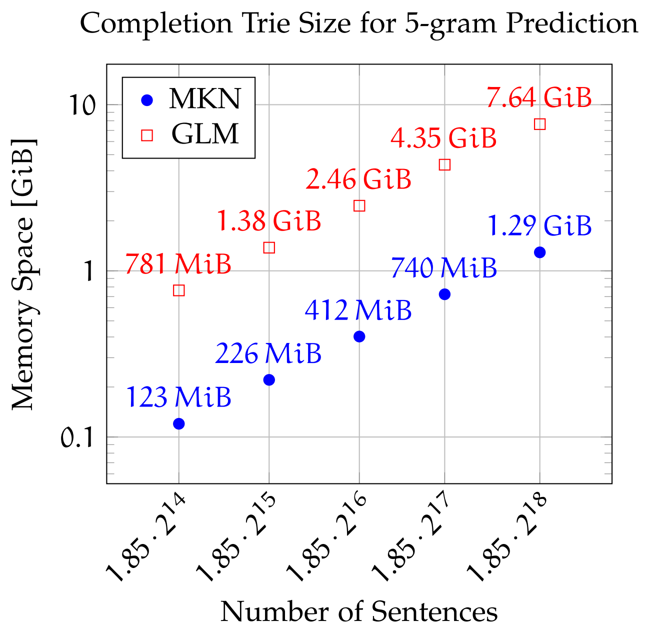


**Figure 5.3:** Memory sizes of the completion tries used to store the occurrence counts that are required to compute 5-gram next word prediction. Distinguished for Modified Kneser-Ney Smoothing (MKN) and the Generalized Language Model (GLM).

### 5.4.2 *Space Requirements*

Next we will measure the memory requirements of our next word prediction system. Again the choice of algorithm to calculate next word prediction is irrelevant. For all algorithms we use the same data structure to store occurrence counts of text corpora: the completion trie described in Section 4.2. This data structure is optimized for compact storage and supports all required access types. However the amount of necessary data varies with the employed language model. As the Generalized Language Model also needs to keep track of the occurrence of skipped $n$-grams, we expect it to always require more memory.

Figure 5.3 shows the necessary memory for each different corpus size and each language model. In the plot all measurement points of one language model technique lie on one pretty clear, straight line. Note that both axis feature logarithmic scales. Thus the relation between the number of sentences in a corpus and the required memory space is monomial in practice. As the slope of the observed line is lesser than one, the power of the monomial also has to be lass than one. We thus conclude that the relation between increasing corpus size to the required memory is better than linear.

For each measured corpus size, the Generalized Language Model takes roughly six times the amount of memory. Together with the fact that its prediction quality is only very slightly better, there is no reason for its use over Modified Kneser-Ney Smoothing. This argument will be reinforced by comparing their runtimes.

Explainable by six times as many patterns?

### 5.4.3 *Runtime Analysis*

Finally we want to compare the presented next word prediction algorithms by their runtime, as fast next word prediction algorithms were the main objective of this thesis. In this thesis we described three different solutions: (1) the Simple Argmax algorithm, which calculates the probability of all words in a vocabulary and then selects the ones with the highest probabilities, (2) the Threshold Algorithm, known from the field of top-$k$ joining, which utilizes sorted and random access to a sorted list of occurrence counts to find predictions faster, and (3) the No-Random-Access Algorithm which works similarly but only requires sorted access.

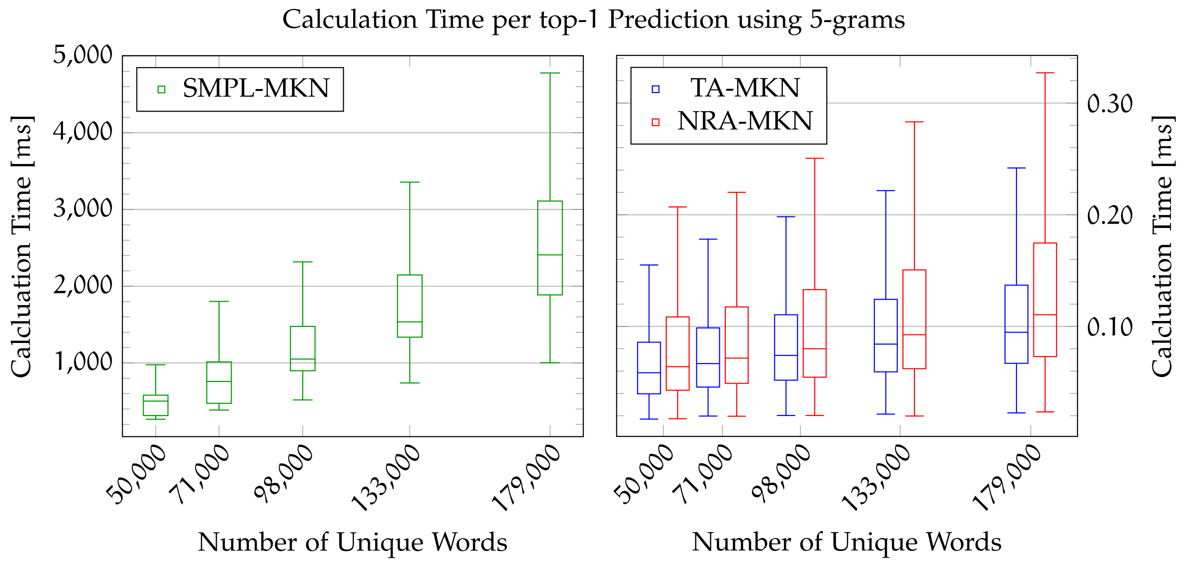
We studied the behavior of these algorithms under two different language models in a wide variety of settings. We measured the runtime of each algorithm (1) applied to Modified Kneser-Ney Smoothing and the Generalized Language Model, (2) with occurrence counts from five differently sized training corpora, (3) for $n$-gram lengths from two to five, (4) predicting one to five words. Figures 5.4 to 5.9 show selected results from these experiments.

In Figures 5.4 and 5.7 the relation between the number of unique words in a corpus and the runtime of each algorithms is shown. A linear relation can be observed for measurements of the Simple Argmax algorithm. This can easily be explained by the fact, that it calculate one probability for each unique word. The top-$k$ joining algorithms seem to grow with the logarithm of the number of unique words. However no clear statement can be made from these measurements, the relation could also be linear in practice, and the seemingly logarithmic relation could only be the result of variance. On the other hand a logarithmic relation would be well explainable, because top-$k$ algorithms precisely try to avoid enumerating all words. More importantly though, the top-$k$ joining algorithms are able to calculate the results multiple orders of magnitude faster for both language models. Calculations that ranged in the order of seconds were speed up to sub-millisecond performance.

Figures 5.5 and 5.8 attest similar runtime-improvements. They show the connection between employed $n$-gram length and calculation time. For Modified Kneser-Ney Smoothing a linear relation is observed, while that for the Generalized Language Model is exponential. It is clear that the relevant factor is calculation time of probabilities, as Figure 5.1 has shown identical correlations between $n$-gram lengths and calculation time. Together with Table 5.2 this figure gives a good overview of possible runtime / prediction quality trade-offs.

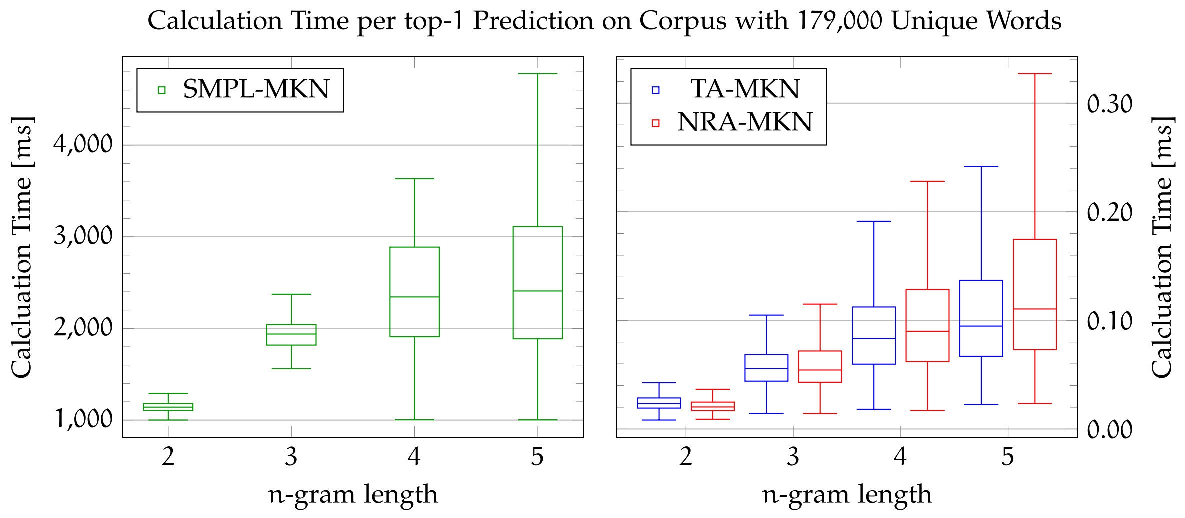
Does it make sense to take measurement of unique words vs trie depth?

Lastly Figures 5.6 and 5.9 show how these algorithms behave when more than one word is to predicted. Obviously for the Simple Argmax Algorithm the time remains constant, as it calculates probabilities for all words anyhow. Keeping track of the $k$ highest probabilities can be considered a no-op versus actually calculating the probabilities. On the other hand, the top-$k$ joining algorithms correlate linearly with the number of words to predict. The No Random Access seems to be



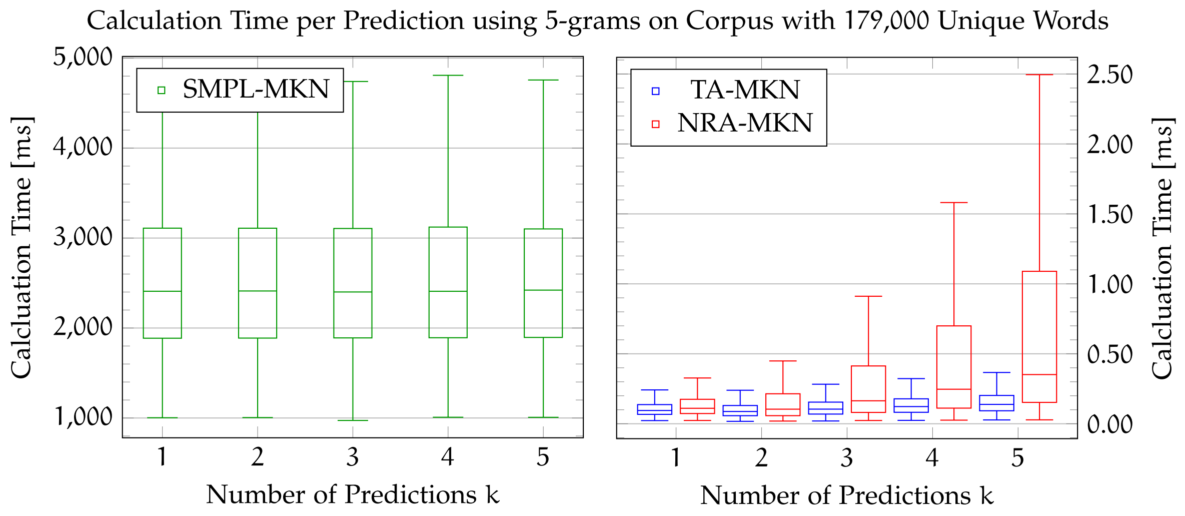
**(a)** **(b)**

**Figure 5.4:** Time to calculate one top-1 word prediction query using 5-grams over varying training corpus sizes for Modified Kneser-Ney Smoothing.



**(a)** **(b)**

**Figure 5.5:** Time to calculate one top-1 word prediction query with language model n-gram length varied from two to five for Modified Kneser-Ney Smoothing. Calculated on the `Training-100%` corpus.



**(a)** **(b)**

**Figure 5.6:** Time to calculate one top-k word prediction query using 5-grams for Modified Kneser-Ney Smoothing. Calculated on the `Training-100%` corpus

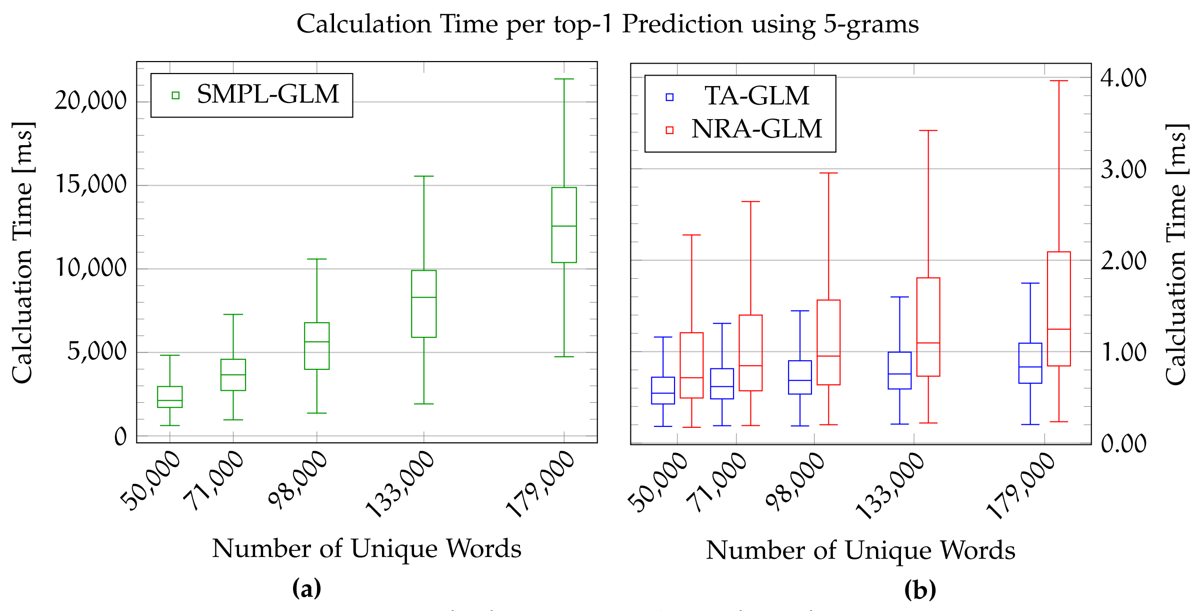


**Figure 5.7:** Time to calculate one top-1 word prediction query using 5-grams over varying training corpus sizes for the Generalized Language Model.

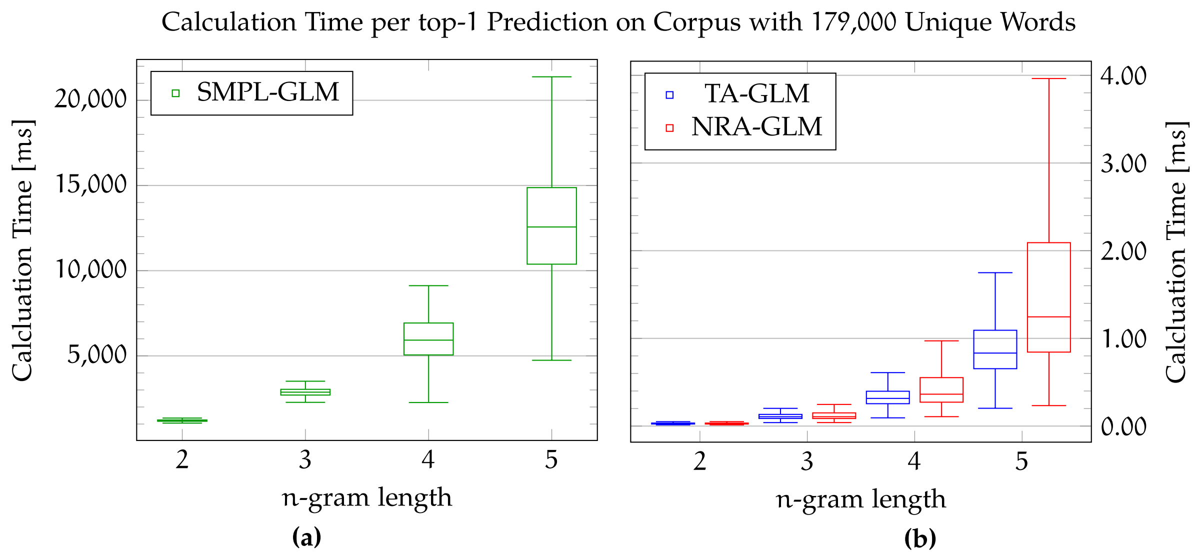


**Figure 5.8:** Time to calculate one top-1 word prediction query with language model n-gram length varied from two to five for the Generalized Language Model. Calculated on the `Training-100%` corpus.

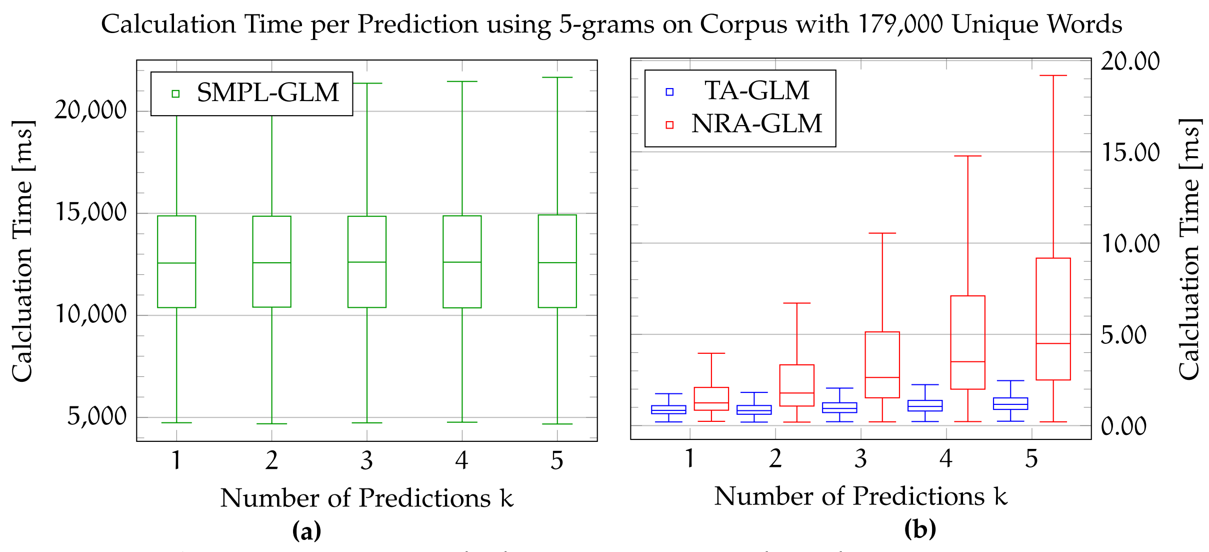


**Figure 5.9:** Time to calculate one top-k word prediction query using 5-grams for the Generalized Language Model. Calculated on the `Training-100%` corpus

performing significantly worse; the linear relation is almost impossible to make out in the plot for the Threshold Algorithm.

Over all experiments a clear improvement in runtime through top-k join algorithms is observable. At large the calculations using Modified Kneser-Ney Smoothing were around ten times faster than those of the Generalized Language Model. Additionally the Threshold Algorithm was able to outperform the No Random Access Algorithm in all but the most niche measurement[3].

To quantify that last point we will perform a comparison of both top-k join algorithms. To this end we summarize the previous experiments by dividing the No-Random-Access-Algorithm-runtimes through the Threshold-Algorithm-runtimes for each performed query. Figure 5.10 shows a histogram of these ratios. Note the logarithmic x-axis and the thus increasing bin sizes. By the mean we could say that the Threshold Algorithm is fifteen times faster for an average query. However the mean might not be a good measure because the data is not normal distributed. But even for a robust measure like the median the Threshold Algorithm is about one and a half times faster. We thus conclude that for our experiment setup and data structure access costs the Threshold Algorithm significantly outperforms. One should only use the No-Random-Access Algorithm because of structural inabilities to support random access. Note that this result is not surprising and directly follows from data theory.

Missing: space requirements of algorithms

Missing: experiments with prefix

3 The only measuring point for which the No Random Access Algorithm was faster were for 2-grams. However 2-grams are not used in any application in practice because they yield comparably bad prediction quality (Table 5.2).

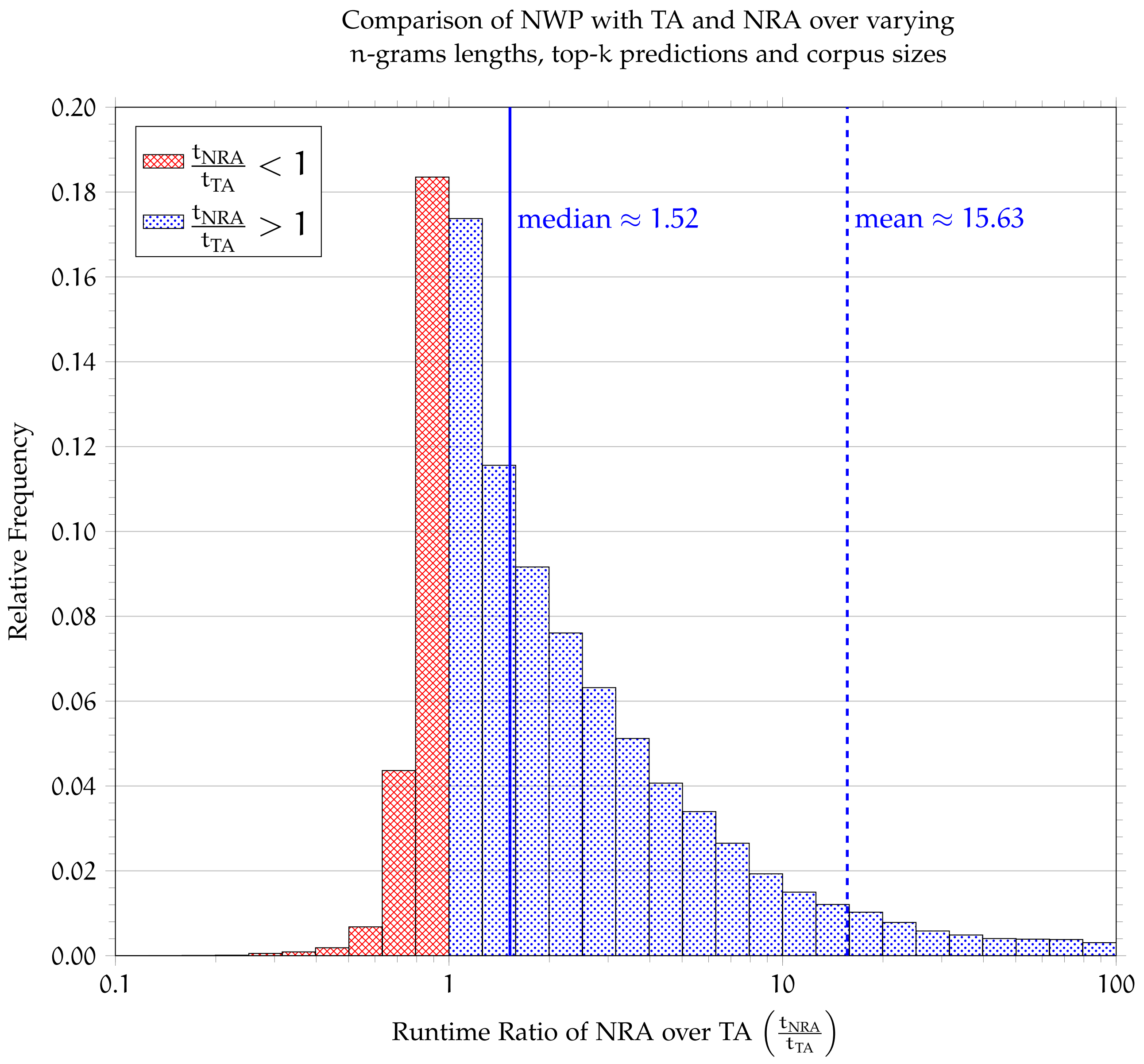


**Figure 5.10:** lol

# 6 CONCLUSION

Key-contribution is weighted sum representation of GLM.

Possibility to improve Argmax using Quick-Combine (Güntzer et al. 2000) and Stream-Combine (Güntzer et al. 2001).

Advanced top-k algorithms that optimized on a fixed history $h$ might be called for if one wants to learn sum weights $\lambda$.

Other Noisy Channel Tasks might not fit the equi-join and require more sophisticated Argmax algorithms (see (Ilyas et al. 2004)).

Research how far pruning of counts is possible without affecting scores.

# REFERENCES

Bengio, Yoshua, Ducharme, Réjean, Vincent, Pascal, and Janvin, Christian (2003). "A Neural Probabilistic Language Model." In: *J. Mach. Learn. Res.* 3, pp. 1137–1155 (cit. on p. 5).

Bickel, Steffen, Haider, Peter, and Scheffer, Tobias (2005). "Predicting Sentences Using N-gram Language Models." In: *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*. HLT '05. Vancouver, British Columbia, Canada: Association for Computational Linguistics, pp. 193–200 (cit. on pp. 2, 42).

Bilmes, Jeff A. and Kirchhoff, Katrin (2003). "Factored Language Models and Generalized Parallel Backoff." In: *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology: Companion Volume of the Proceedings of HLT-NAACL 2003–short Papers - Volume 2*. NAACL-Short '03. Edmonton, Canada: Association for Computational Linguistics, pp. 4–6 (cit. on pp. 5, 16).

Chang, Yuan-Chi, Bergman, Lawrence, Castelli, Vittorio, Li, Chung-Sheng, Lo, Ming-Ling, and Smith, John R. (2000). "The Onion Technique: Indexing for Linear Optimization Queries." In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. SIGMOD '00. Dallas, Texas, USA: ACM, pp. 391–402 (cit. on p. 29).

Chelba, Ciprian, Mikolov, Tomáš, Schuster, Mike, Ge, Qi, Brants, Thorsten, Koehn, Phillipp, and Robinson, Tony (2013). *One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling*. Tech. rep. Google (cit. on pp. 5, 59).

Chen, Stanley F. and Goodman, Joshua T. (1996). "An Empirical Study of Smoothing Techniques for Language Modeling." In: *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*. ACL '96. Santa Cruz, California: Association for Computational Linguistics, pp. 310–318 (cit. on pp. 5 sq.).

– (1998). *An Empirical Study of Smoothing Techniques for Language Modeling*. Tech. rep. Computer Science Group, Harvard University (cit. on pp. 5 sq.).

– (1999). "An empirical study of smoothing techniques for language modeling." In: *Computer Speech & Language* 13.4, pp. 359–393 (cit. on pp. 5–9).

Fagin, Ronald, Lotem, Amnon, and Naor, Moni (2001). "Optimal Aggregation Algorithms for Middleware." In: *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database*

*Systems*. PODS ’01. Santa Barbara, California, USA: ACM, pp. 102–113 (cit. on pp. 28, 30, 33).

Goodman, Joshua T. (2001). “A Bit of Progress in Language Modeling.” In: *Computer Speech & Language* 15 (4), pp. 403–434 (cit. on pp. 3, 6, 41).

Güntzer, Ulrich, Balke, Wolf-Tilo, and Kießling, Werner (2000). “Optimizing Multi-Feature Queries for Image Databases.” In: *Proceedings of the 26th International Conference on Very Large Data Bases*. VLDB ’00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 419–428 (cit. on pp. 36, 51).

– (2001). “Towards Efficient Multi-Feature Queries in Heterogeneous Environments.” In: *Proceedings of the International Conference on Information Technology: Coding and Computing*. ITCC ’01. Washington, DC, USA: IEEE Computer Society, pp. 622–628 (cit. on pp. 36, 51).

Hsu, Bo-June (Paul) and Ottaviano, Giuseppe (2013). “Space-efficient Data Structures for Top-k Completion.” In: *Proceedings of the 22Nd International Conference on World Wide Web*. WWW ’13. Rio de Janeiro, Brazil: International World Wide Web Conferences Steering Committee, pp. 583–594 (cit. on pp. 29 sq.).

Ide, Nancy and Suderman, Keith (2007). *The Open American National Corpus (OANC)*. URL: `http://www.anc.org/data/oanc/` (visited on July 5, 2015) (cit. on p. 37).

Ilyas, Ihab F., Aref, Walid G., and Elmagarmid, Ahmed K. (2004). “Supporting Top-k Join Queries in Relational Databases.” In: *The VLDB Journal* 13.3, pp. 207–221 (cit. on pp. 29, 51).

Ilyas, Ihab F., Beskales, George, and Soliman, Mohamed A. (2008). “A Survey of Top-k Query Processing Techniques in Relational Database Systems.” In: *ACM Comput. Surv.* 40.4, 11:1–11:58 (cit. on pp. 3 sq., 28 sq.).

Jelinek, Frederick and Mercer, Robert L. (1980). “Interpolated estimation of Markov source parameters from sparse data.” In: *In Proceedings of the Workshop on Pattern Recognition in Practice*. Amsterdam, The Netherlands: North-Holland, pp. 381–397 (cit. on p. 11).

Jurafsky, Daniel and Martin, James H. (2009). *Speech and Language Processing (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. (cit. on pp. 2 sq., 5, 41).

Kneser, Reinhard and Ney, Hermann (1995). “Improved backing-off for M-gram language modeling.” In: *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*. Vol. 1, 181–184 vol.1 (cit. on pp. 5 sq.).

Mikolov, Tomáš (2012). “Statistical Language Models Based on Neural Networks.” PhD thesis. Ph. D. thesis, Brno University of Technology (cit. on p. 5).

Newell, Alan, Arnott, John, Booth, Lynda, Beattie, William, Brophy, Bernadette, and Ricketts, Ian (1992). “Effect of the “PAL” word prediction system on the quality and quantity of text generation.” In: *Augmentative and Alternative Communication* 8.4, pp. 304–311. eprint: http://www.tandfonline.com/doi/pdf/10.1080/07434619212331276343 (cit. on p. 1).

Ney, Hermann and Essen, Ute (1991). “On smoothing techniques for bigram-based natural language modelling.” In: *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, 825–828 vol.2 (cit. on p. 8).

Ney, Hermann, Essen, Ute, and Kneser, Reinhard (1994). “On Structuring Probabilistic Dependencies in Stochastic Language Modelling.” In: *Computer Speech and Language* 8, pp. 1–38 (cit. on p. 8).

Pickhardt, René, Gottron, Thomas, Körner, Martin, Wagner, Paul Georg, Speicher, Till, and Staab, Steffen (2014). “A Generalized Language Model as the Combination of Skipped n-grams and Modified Kneser Ney Smoothing.” In: *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vol. abs/1404.3377. Baltimore, Maryland: Association for Computational Linguistics, pp. 1145–1154 (cit. on pp. 5 sqq., 15).

Stolcke, Andreas (2000). “Entropy-based Pruning of Backoff Language Models.” In: *CoRR* cs.CL/0006025 (cit. on pp. 3, 41).

Swiffin, Andrew, Arnott, John, Pickering, J. Adrian, and Newell, Alan (1987). “Adaptive and predictive techniques in a communication prosthesis.” In: *Augmentative and Alternative Communication* 3.4, pp. 181–191. eprint: http://dx.doi.org/10.1080/07434618712331274499 (cit. on pp. 1, 42).

Trnka, Keith (2011). “Word Prediction Techniques for User Adaptation and Sparse Data Mitigation.” AAI3443248. PhD thesis. Newark, DE, USA (cit. on pp. 1, 42 sq.).

Trnka, Keith, Yarrington, Debra, McCaw, John, McCoy, Kathleen F., and Pennington, Christopher (2007). “The Effects of Word Prediction on Communication Rate for AAC.” In: *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Companion Volume, Short Papers*. NAACL-Short ’07. Rochester, New York: Association for Computational Linguistics, pp. 173–176 (cit. on p. 1).

Tsaparas, P., Palpanas, T., Kotidis, Y., Koudas, N., and Srivastava, D. (2003). “Ranked join indices.” In: *Data Engineering, 2003. Proceedings. 19th International Conference on*. IEEE, pp. 277–288 (cit. on p. 29).

Tukey, John W. (1977). *Exploratory Data Analysis*. Addison-Wesley (cit. on p. 39).

# A TIMELINE

- Write chapters
  - Introduction **<done>**
  - Review of Considered language Models **<done>**
  - Formulating Language Models as Weighted Sums **<done>**
  - Fast Prefix Queries using Top-*k* Joins (1-2day)
  - Evaluation (3-4 days)
  - Conclusion (1-2 days)
- Run experiments **<almost done>**
- Finishing (3-4 days)
  - Fix latex
  - Reformulate shitty sentences
  - Check formatting
- Peer review, spelling correction (1-2 weeks)
- Make slides, prepare presentation **<almost done>**

# TODO LIST