Bloom Filter and Lossy Dictionary Based Language Models

Abby D. Levenberg

Master of Science

Cognitive Science and Natural Language Processing

School of Informatics

University of Edinburgh

2007

Abstract

Language models are probability distributions over a set of unilingual natural language text used in many natural language processing tasks such as statistical machine translation, information retrieval, and speech processing. Since more well-formed training data means a better model and the increased availability of text via the Internet, the size of language modelling n-gram data sets have grown exponentially the past few years. The latest data sets available can no longer fit on a single computer. A recent investigation reported first known use of a probabilistic data structure to create a randomised language model capable of storing probability information for massive n-gram sets in a fraction of the space normally needed. We report and compare the properties of lossy language models using two probabilistic data structures: the Bloom filter and lossy dictionary. The Bloom filter has exceptional space requirements and only one-sided, false positive error returns but it is computationally slow in scale which is a potential drawback for a structure being queried millions of times per sentence. Lossy dictionaries have low space requirements and are very fast but with two-sided error that returns both false positives and false negatives. We also investigate combining the properties of both the Bloom filter and lossy dictionary and find this can be done to create a fast lossy LM with low one-sided error.

Acknowledgements

My advisor Dr. Miles Osborne for his valuable advice, fantastic ideas and for getting me enthusiastic (and a bit hyper) about this project during our weekly meetings.
David Talbot for his willingness to help though I was clueless.
To my family for believing in me long-distance.
To David Matthews—touché, Gregor Stewart—for stimulating conversations over coffee, and Joe Paloweski—for the veggie curries and insisting I occasionally relax.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Abby D. Levenberg*)

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

A language model (LM) is an integral part of many NLP tasks such as machine translation and speech processing. Over the past years though LM data sets have grown to the point they no longer fit in memories on even robust computers using gigabytes of RAM. This makes the increase in information the bigger data sets provide inaccessible without severely pruning the size of the data which in turn defeats the purpose of creating larger sets of data in the first place.

Since we can't use standard methods of modelling the data we have to find ways of representing the LM data with lossy encoding. Recent work (Talbot and Osborne, 2007a,b) has accomplished this completely for the first time. The LM is represented using a Bloom filter, a randomised data structure designed to test set membership with a small probability of error returns. The vanilla BF requires a large number of hashes to support its fantastic space savings, however, which makes the final Bloom filter language model slower than desirable. We recreate the Bloom filter language model in this investigation and report and compare it with first results on implementing language models using another randomised data structure–the lossy dictionary. Lossy dictionaries are fast and have low space requirements but they discard a small portion of the events in the set they encode making the probability of error in a language model that uses one higher than the Bloom filter language model. We attempt to bridge the gap between these two models and after extensive testing successfully create a fast lossy LM that still maintains low error probability.

The rest of this chapter gives a brief overview of the subject matter of the projects and outlines the rest of the report.

1.2 Language Models

Language Models (LM) are models of a conditional probability distribution over a collection of unilingual text. With a large enough sample of grammatical, well formed text and the right model of probability, the phrase “the big house”, for example, can be known to be more probable than the grammatical but odd “a mammoth dwelling” and decidedly more probable than the ungrammatical “house big the”.

The model can be discovered using *Maximum Likelihood Estimation* (MLE) over the text, but a LM can only be a finite sample of a language and as such must deal with data sparseness. Using only MLE as an estimate would return a null probability for any previously unseen words or phrases but there are always infinitely more well formed phrases the LM has not encountered so a zero probability for these unseen events is undesirable. Instead of MLE, LMs are built using *smoothing* techniques that reserve some of the known conditional probability mass for new events.

Since more language data produces more accurate and robust probability models, LM sizes have seen an steady increase in size from the millions of words to billions and, now, trillions. Ways of taking advantage of and learning from this increase in data size via lossy language models is a new and interesting area of research.

LMs and the methods used to build the probability models they use are discussed in Chapter 2.

1.3 Probabilistic Data structures

A *probabilistic data structure* (PDS) is a data structure that supports algorithms that inherently use a certain amount of randomness for their operations. Without a measure of randomness, the data these data structures encode would use a prohibitive amount of memory and/or the algorithms they support would be very slow. As a trade off for their space, speed, and computational efficiency, many PDSs allow for a small margin of error returns. We examine the theoretic properties of the Bloom filter and the lossy dictionary which aim to show upper and lower bounds of the PDSs when implemented.

Lossy encoding reduces the needed space for a set of data by changing the encoding characteristics of the data. The encoding is termed “lossy” as it usually requires discarding information so as to efficiently encode the *lossless* data in as small a storage space possible. Intuitively there is a balance between the quality of the lossy data (how well it reproduces the lossless content) and how much space is used to encode it. Current research aims to balance this space savings with retaining as much of the original data's information as possible.

One efficient way to transform string data is by a using a hash function which generates a reproducible *signature* of each piece of the data. The signature of each item is able to be stored in a much smaller space than that of the original data passed through the hash function. As such a hash function maps elements from a universal set $S \subseteq U$ of size w to a domain of a smaller size b. If the set S contains many elements and $b \ll w$, there will be elements of S that try to inhabit the same sub-space in the new domain. This is called a *collision*. To minimize collisions it's important a hash function's outputs uniformly distributed the values of S into the smaller domain.

In chapter 3 we examine the properties of hash functions and the two PDSs we use for lossy LM encoding - the *Bloom filter* (BF) (Bloom, 1970) and the *Lossy Dictionary* (LD) (Pagh and Rodler, 2001b). Both data structures are used to query for set membership but only the LD supports key/value pairs natively. The probability of errors for both are adjustable and are primarily dependent on space constraints. The BF maps outputs into a single array of shared bits while the LD uses a RAM based bucket lookup scheme. The LD also allows storing a select subset of the set assuming weighted information associated with the set's keys. The LD query time is constant while the BF query time is proportional to the space bounds and number of items being inserted into it.

1.4 Previous Work

In this chapter we report on the work done previously that this investigation is based on. As lossy language models is a new field of investigation, however, and the only previous work reported was done by Talbot and Osborne (2007a,b). A BF was used and a lossy language model was derived with optimal space bounds and a lower error rate. The research shows that when used by an SMT system, the lossy LM performs competitively with a lossless LM while only using a minor fraction of the space.

1.5 Testing Framework

In Chapter 5 we report on the structure and development of our testing framework. We implement both the BF-LM from (Talbot and Osborne, 2007a,b) as well as create a LM using the LD as the encoding PDS. Since lossy data does not guarantee a standard probability distribution our evaluation method was to derive the mean-squared error of the per sentence probability of the lossy LMs compared to lossless LMs created with the Stanford Research Institute Language Modelling toolkit (Stolcke, 2002).

1.6 Experiments

Details of the experiments we conducted are reported in Chapter 6. The goal of our experiments was to find which data structure best encodes a LM and why and we report our findings on how well each follows their proposed theoretic bounds for this task. We recreate the recent lossy BF-LM research and compare theits accuracy, size, and speed against the first known results for using a LD to store a LM. We evaluate each lossy LM structure against the lossless output and discover that the BF has lower MSE overall but the LD may be better suited for some applications because of its simplicity and quickness. We also investigate combining the positive properties of the space efficient BF and fast LD and discover the *Bloom Dictionary*, a data structure with constant access time and bit sharing that retains low error probability.

We conclude with a summary of our findings and suggested directions of research onward.

Chapter 2

Language Models

A *Language Model* is a statistical estimator for determining the most probable next word given the previous *n* words of context. A LM is in essence a function that takes as input a sequence of words and returns a measure of how likely that sequence of words in order would be produced by a native speaker of the language. As such, a good LM should prefer fluent, grammatical word ordering over randomly generated phrases. For example we desire a LM to show:

$$\Pr(\text{“the big house is green”}) > \Pr(\text{“big house the green is”}).$$

2.1 N-grams

The most common way in practice to model words in a LMs is via *n-grams*. With the n-gram model, we view the text as a contiguous string of words, $w_1, w_2, w_3, \ldots, w_{N-1}, w_N$, and we decompose the probability of the string so the joint probability

$$\begin{aligned}\Pr(w_1, w_2, w_3, w_4, \ldots, w_{N-1}, w_N) = \\ &\Pr(w_1)\Pr(w_2|w_1)\Pr(w_3|w_1,w_2)\Pr(w_4|w_1,w_2,w_3)\ldots \\ &\ldots\Pr(w_{N-1}|w_1,w_2,w_3,\ldots,w_{N-2})\Pr(w_N|w_1,w_2,w_3,\ldots,w_{N-2},w_{N-1}) \\ &= \prod_{i=1}^{N}\Pr(w_i|w_1^{i-1})\end{aligned}$$

and each word is conditionally related to the entire history proceeding it.

However, considering the entire history of the document for each next word is infeasible for two reasons. First, this method is computationally unfeasible and would require far too much resources for each prediction once we were well into the text. Second, most likely after the first few words of history we will find we're dealing with a sentence we have never come across before. Therefore the probability of the next

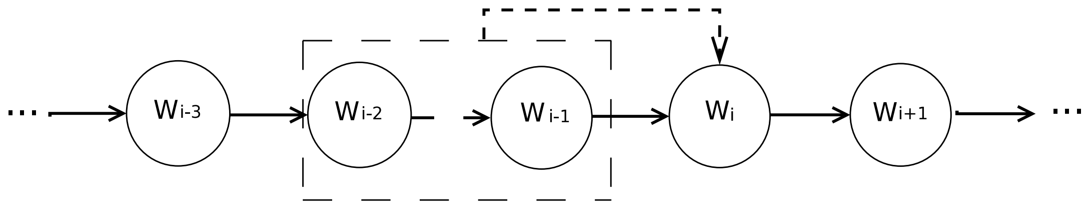

Figure 2.1: *A 3^{rd} order Markov Model. Instead of the full conditional probability history only a constrained window of $n-1$ events is used to predict w_i.*

word given the entire history would be unknown. Instead, LM's generally make an approximation of the probability based on the *Markov assumption* that only the last few events (or words) of prior local context affects the next word (Manning and Schütze, 1999, chap. 6). A *Markov model* makes the assumption only the previous $n-1$ words have context relations to the word being predicated. How many previous words the LM considers is known as the *order n* of the LM and is referred to as unigram, bigram, trigram, and fourgram for one, two, three and four order Markov models respectively. As an example, if we want to estimate the trigram probability of word w_i with a Markov model of order $n=3$ then we examine the $n-1$ previous words so now

$$\Pr(w_1, w_2, ..., w_i) \approx \Pr(w_i | w_{i-2}, w_{i-1}).$$

Figure 2.1 gives an illustration of the trigram model, the most commonly used model order in practice.

In general let w_{i-n+1}^{i} notate an n-gram of context length n ending with w_i. (Note: $i-(n-1) = i-n+1$.) Then to estimate the n-gram probabilities for a string of length N we have

$$\Pr(w_1^N) \approx \prod_{i=1}^{N} \Pr(w_i | w_{i-n+1}^{i-1}).$$

2.1.1 Google's Trillion Word N-gram Set

Adhering to the adage "There's no data like more data", Google has used their massive distributed computing and storage capabilities to develop the Trillion Word N-gram set[1]. This is the largest data set of n-grams available and was made available for public use in late 2006. The statistics of the Trillion Word N-gram set is shown in Table 2.1. The released data set is a pruned collection from even more text stripped from crawling

[1]`http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html`

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

Table 2.1: Google's Trillion Word N-gram Statistics.

a large portion of the available Internet. The n-grams take up 36 GB of spaced compressed and Google makes use of the data using their high power distributed computing environment.

2.2 Training

To train a LM we need a unilingual corpus of a substantial size and a statistical estimator that accurately models the probability mass of the corpus. In different ways all the estimators for LMs use the frequencies of the n-grams, so we need to count the occurrences of each n-gram from unigrams to the order of the model we're training.

2.2.1 Maximum Likelihood Estimation (MLE)

To begin, we can use the counts to build a MLE model with the *relative frequency* of n-grams in the text. With MLE the probability of a unigram appearing next is simply the number of times it appears in the corpus.

$$\Pr_{MLE}(w_i) = \frac{c(w_i)}{\sum_{i'} c(w_{i'})}$$

where $c(\ldots) = |w_i|$ and is a frequency function. The MLE method is so called because it maximizes its output parameters to model the training data. As an example (from (Jurafsky and Martin, 2000)), in the Brown Corpus [2], a corpus of a million words, the word "Chinese" occurs 400 times. The probability a LM built with MLE will assign to the unigram "Chinese" is $\Pr_{MLE}(Chinese) = \frac{400}{1000000}$ or .0004.

[2] http://icame.uib.no/brown/bcm.html

The conditional probability then for a n-gram of $order > 1$ is

$$\Pr_{MLE}(w_i|w_{i-n+1}^{i-1}) = \frac{c(w_{i-n+1}^{i})}{c(w_{i-n+1}^{i-1})}$$

or the count of how often a word occurs after a certain history of order $n-1$. Continuing with the above example, the probability for the bigram “Chinese food” occurring is the number of times “food” has “Chinese” as its history divided by $c(Chinese) = 400$. In the Brown corpus, “Chinese food” appears 120 times, so the raw

$$\Pr_{MLE}(food|Chinese) = 0.3.$$

2.2.2 Smoothing

While MLE is a straight forward way of obtaining the probability model, the problem of sparse data quickly arises in practice. Since there are an infinite number of grammatical n-grams and a LM can only contain a finite sample of these, the LM would assign many acceptable n-grams a zero probability. To allay this problem, we step away from MLE and use more sophisticated statistical estimators to derive the parameters. This is called *smoothing* or *discounting* as we discount a fraction of the probability mass from each event in the model and reserve the leftover mass for unseen events. What and how much should be discounted is an active research field with a number of smoothing algorithms available [3]. For example, the oldest and most straightforward is *Add-One smoothing* (de Laplace, 1996) where a constant, $l \leq 1$, is added to all vocabulary counts.

Two key concepts of smoothed LMs that were used are *backoff* and *interpolation*. When the LM encounters an unknown n-gram, it may be useful to see if any of the words in the n-gram are known events and to retrieve statistical information about them if so. This concept is called back-off and is defined as

$$\Pr_{BO}(w_i|w_{i-n+1}^{i-1}) = \begin{cases} \delta(w_{i-n+1}^{i-1})\Pr_{LM}(w_i|w_{i-n+1}^{i-1}) & \text{if } c(w_{i-n+1}^{i}) > 0 \\ \alpha\Pr_{BO}(w_{i-n+2}^{i}) & \text{otherwise} \end{cases}$$

where δ is a discounting factor based on the frequency of the history and α is a back-off penalty parameter. If we have a big enough vocabulary we may want to assign a zero probability for *out of vocabulary* (OOV) words when we reach the unigram level model. For example, we may have a domain specific vocabulary we care about or

[3]For an exhaustive list of smoothing algorithms and their derivations Chen and Goodman (1999, see).

not want to remove known probability mass for OOVs such as proper nouns or real numbers. Alternatively, we could smooth our LM so we have a small probability mass available for OOV words and assign this to the word once we reach the unigram level.

Interpolation linearly combines the statistics of the n-gram through all the orders of the LM. If the order of the LM is *n* the formula is defined recursively as

$$\mathrm{Pr}_I^n(w_i|w_{i-n+1}^{i-1}) = \lambda \mathrm{Pr}_{LM}(w_i|w_{i-n+1}^{i-1}) + (1-\lambda)\mathrm{Pr}_I^{n-1}(w_i|w_{i-n+2}^{i-1}).$$

The λ parameter is a weight which we use to set which order of the LM we'd like to impact the final outcome more. For example, we maybe trust trigram predictions over unigram ones. Or we can base λ on the frequency of the history. Note that the difference between these is that interpolation uses the information from lower order models regardless if the count is zero or not, whereas back-off does not.

For this project two smoothing algorithms were used; *Modified Kneser-Ney* (Chen and Goodman, 1999) and Google's *Stupid Backoff* (Brants et al., 2007).

2.2.2.1 Kneser-Ney

Modified Kneser-Ney (MKN) was derived from Kneser-Ney (KN) smoothing (Kneser and Ney, 1995). In the KN algorithm, the probability of a unigram is not proportional to the frequency of the word, but to the number of different histories the unigram follows.

A practical example best illustrates this concept. The bigram “San Francisco” may be an extremely common bigram in a corpus gathered from, say, the San Francisco Chronicle. If the bigram frequency is high, so too is the frequency of the words “San” and “Francisco” and each word will have a relatively high unigram probability if we estimated probability solely from counts. However, this intuitively should not be the case as the actual $\mathrm{Pr}(Francisco)$ is extremely small - almost zero perhaps - except when it follows “San”. As the lower order models are often used for back-off probabilities from the higher order models, we want to reserve the mass that would be wasted on events like “Francisco” for more likely events.

First we define the count of histories of a single word as

$$N_{1+}(\bullet w_i) = |\{w_{i-1} : c(w_i w_i) > 0\}|.$$

The term N_{1+} means the number of words that have one or more counts and the $\bullet$ means a free variable. Instead of relative frequency counts as with the MLE estimate, here the raw frequencies of words are replaced by a frequency dependent on the unique histories proceeding the words. The KN probability for a unigram is

$$\Pr_{KN}(w_i) = \frac{N_{1+}(\bullet w_i)}{\sum_{i'} N_{1+}(\bullet w_{i'})}$$

or the count of the unique histories of w_i divided by the total number of unique histories of unigrams in the corpus.

Generalizing the above for higher order models we have:

$$\Pr_{KN}(w_i|w_{i-n+2}^{i-1}) = \frac{N_{1+}(\bullet w_{i-n+2}^{i})}{\sum_{i'} N_{1+}(\bullet w_{i'-n+2}^{i'})}$$

where the numerator

$$N_{1+}(\bullet w_{i-n+2}^{i}) = |\{w_{i-n+1} : c(w_{i-n+1}^{i}) > 0\}|$$

and the denominator is the sum of the count of unique histories of all n-grams the same length of w_{i-n+2}^{i}. The full model of the KN algorithm is interpolated and has the form

$$\Pr_{KN}(w_i|w_{i-n+1}^{i-1}) = \frac{max\{c(w_{i-n+1}^{i} - D, 0\}}{\sum_{i'} c(w_{i'-n+1}^{i'})} + \frac{D}{\sum_{i'} c(w_{i'-n+1}^{i'})} N_{1+}(w_{i-n+1}^{i-1}\bullet)\Pr_{KN}(w_{i-n+2}^{i})$$

where

$$N_{1+}(w_{i-n+1}^{i-1}\bullet) = |\{w_i : c(w_{i-n+1}^{i-1}w_i) > 0\}|$$

and is the number of unique suffixes that follow w_{i-n+1}^{i-1}.

2.2.2.2 Modified Kneser-Ney

The KN algorithm uses an *absolute discounting* method where a single value, $0 < D < 1$, is subtracted for each nonzero count. MKN enhances the performance of KN by using different discount parameters depending on the count of the n-gram. The equation for MKN is

$$\Pr_{MKN}(w_i|w_{i-n+1}^{i-1}) = \frac{c(w_{i-n+1}^{i}) - D(c(w_{i-n+1}^{i}))}{\sum_{i'} c(w_{i'-n+1}^{i'})} + \gamma(w_{i-n+1}^{i-1})\Pr_{MKN}(w_i|w_{i-n+2}^{i-1})$$

where

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

and

$$\begin{aligned} Y &= \frac{n_1}{n_1 + 2n_2} \\ D_1 &= 1 - 2Y\frac{n_2}{n_1} \\ D_2 &= 2 - 3Y\frac{n_3}{n_2} \\ D_{3+} &= 3 - 4Y\frac{n_4}{n_3} \end{aligned}$$

where n_i is the total number of n-grams with i counts of the higher order model n being interpolated. To ensure the distribution sums to one we have

$$\gamma(w_{i-n+1}^{i-1}) = \frac{\sum_{i \in \{1,2,3+\}} D_i N_i(w_{i-n+1}^{i-1} \bullet)}{\sum_{i'} c(w_{i'-n+1}^{i'})}$$

where N_2 and N_{3+} means the number of events that have two and three or more counts respectively.

MKN has been consistently shown to have the best results of all the available smoothing algorithms (Chen and Goodman, 1999; James, 2000).

2.2.2.3 Stupid Backoff

Google uses a simple smoothing technique, nicknamed *Stupid Backoff*, in their distributed LM environment. The algorithm uses the relative frequencies of n-grams directly and is

$$S(w_i|w_{i-n+1}^{i-1}) = \begin{cases} \frac{c(w_{i-n+1}^{i})}{c(w_{i-n+1}^{i-1})} & \text{if } c(w_{i_n+1}^{i}) > 0 \\ \alpha S(w_{i-n+2}^{i}) & \text{otherwise} \end{cases}$$

where α is a penalty parameter and is set to the constant $\alpha = 0.4$. The recursion ends once we've reached the unigram level probability which is just

$$S(w_i) = \frac{w_i}{N}$$

where N is the size of the training corpus. Brants et al. (2007) claims the quality of Stupid Backoff approaches that of MKN smoothing for large amounts of data. Note that S is used instead of P to indicate that the method returns a relative score instead of a normalized probability.

2.3 Testing

Once a LM has been trained, we need to evaluate the quality of the model by seeing whether the LM gives a high probability to well formed English, for example. One metric for this is *perplexity* which is the transformation of the *cross entropy* of the model.

The cross entropy is an upper bound on *entropy*. Entropy is fundamental in *information theory* and quantifies the information content of data by measuring the uncertainty associated with it (Manning and Schütze, 1999, chap. 1). If a random variable x exists over the range χ of information being evaluated with a probability distribution p, entropy for x is defined as

$$H(x) = -\sum_{x\in\chi} \Pr(x) log_2 \Pr(x)$$

. For example the outcome of a coin toss, where x can only be heads or tails and $\Pr(x) = 0.5$ for both, has a lower entropy then the outcome of throwing a dice, where x ranges over more values and $\Pr(x) = \frac{1}{6}$. As a throw of the dice has a higher measure of uncertainty associated with it than a coin toss the entropy is higher. The log in the formula can be in any base, but if we use base two then we're measuring the information content in *bits* which is an advantage when dealing with RAM models.

The cross entropy of a model, as the names suggests, is a measure of information between two probability distributions (de Boer et al., 2005). For some distribution q that models an actual distribution p that we don't know, the cross entropy is defined as

$$H(p,q) = -\sum_{x\in\chi} \Pr(x) log_2 q(x)$$

. The Shannon-McMillan-Breiman Theorem (Algoet and Cover, 1988) states that for both entropy and the cross entropy we can discard the term $\Pr(x)$ if the sequence of x is long enough. If we want the per-word cross entropy we divide by the total number of words so

$$H(p,q) = -\frac{1}{n}\sum_{x} log_2 q(x) = -\frac{1}{n} log_2 q(x_n^1).$$

Perplexity is defined as $PP = 2^{H(p,q)}$. Since the cross entropy is an upper bound on entropy, $H(p,q) \geq H(p)$, we can never underestimate the true entropy by using the cross entropy as a measure of the perplexity of our model. We can evaluate the perplexity on testing sets of data held out from the training. As an example, the lowest perplexity published of the Brown Corpus is $2^{7.95} = 247$ (Brown et al., 1992) which means the model has 247 possible uniform and independent choices for each word. In Shannon (1951); Cover and King (1978); Cover and Thomas (1990) the entropy for the English language is computed using various guessing and gambling techniques on what the next letter or word will be. All reports estimate English language entropy at around 1.3 bits per character.

The nature of a randomised LM does not guarantee correct probabilities as it represents the original data in some lossy way which skews the normal distribution. This means we cannot use information theoretic measures such as perplexity to judge the goodness of our lossy LMs. However, we can use a LM that does guarantee a normal distribution and measure how far the randomised LM's distribution deviates via both of the LMs outputs. We can do this using the MSE between a lossless and lossy LM over the same training and testing data. We can find the MSE unbiased estimator using the formula

$$MSE = \frac{1}{n-1}\sum_{i}^{n}(X_i - X_i')^2$$

where X is the lossless LM probability of event i and X' is the lossy probability for the same event.

2.4 LMs Case Study

N-gram language models are used in many NLP tasks such as speech processing, information retrieval, and optical character recognition (Manning and Schütze, 1999, chap. 6). We illustrate briefly how a language model is used in an oversimplified phrase-based statistical machine translation (SMT) (Koehn et al., 2003) system.

In SMT, given a source foreign language f and a target translation language e, we try to maximize $\Pr(e|f)$ for the best probable translation. Since there is never only one true translation of a sentence the objective is to find the most probable sentence out of all possible target language sentences that best represents the meaning of the foreign sentence. We use the noisy channel approach and have

$$argmax_e \Pr(e|f) = \frac{argmax_e \Pr(f|e)\Pr(e)}{\Pr(f)}$$

where $\Pr(f|e)$ is the conditional probability, $\Pr(e)$ is the prior. $\Pr(f)$ is the marginal probability but since it is the foreign sentence independent of e we can disregard it.

In phrase-based SMT the noisy channel formulation is turned into a log-linear model in practice and weighted factors are assigned to each feature of the system (Och and Ney, 2001). We use a *decoder* to decompose the source string into arbitrary phrases and generate a very large space of possible target language translations for each phrase from the *phrase table*. The three major components of probability in

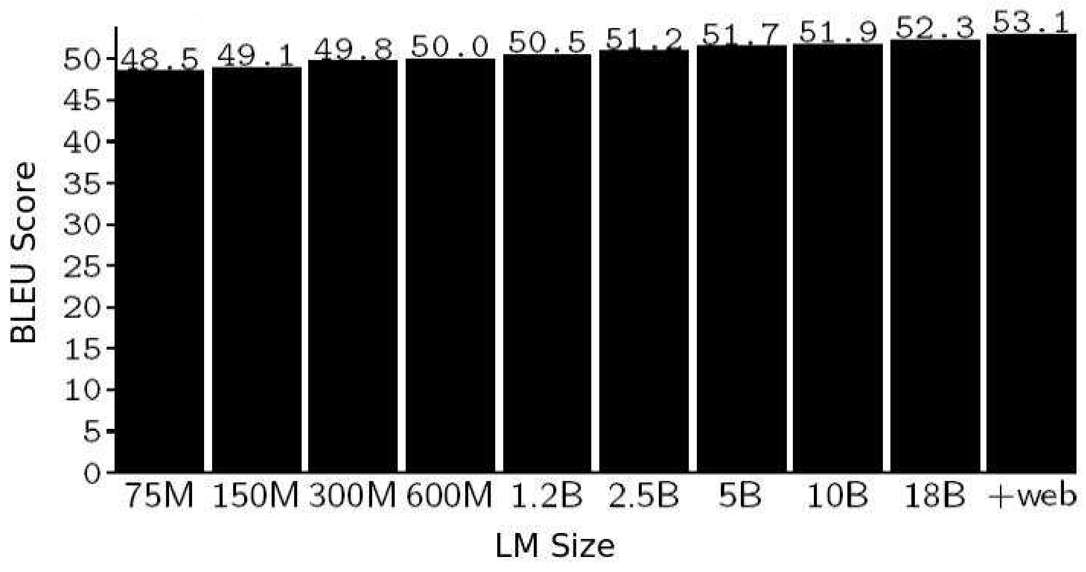

Figure 2.2: *LM size vs. BLEU score. Training the LM with more data increases the overall BLEU score on a log-scale.*

a phrase-based SMT system are the *phrase translation model*, $\phi(f|e)$, the *reordering model*, $\omega^{length(e)}$, and the LM, $\Pr_{LM}(e)$. In general we have

$$argmax_e \Pr(e|f) = argmax_e \ \phi(f|e)^{\lambda_\phi} \ \Pr_{LM}(e)^{\lambda_{LM}} \ \omega^{length(e)^{\lambda_\omega}}$$

where $\phi(f|e)$ is found using the probability distribution over a parallel corpus of phrase pairs weighted with other factors such as the length of the phrase. The reordering model $\omega^{length(e)}$ constrains the distance words are allowed to travel from their original index in the foreign sentence. $\Pr(e) = \Pr_{LM}(e)$ is the LM prior the decoder uses to score the “goodness” of the generated phrases in the target language, returning low probabilities for unlikely n-grams and higher probabilities for likely ones.

Often the LM is assigned a higher weight than other system features because, for example, the amount of unilingual data available to train the model is much greater than the amount of aligned multilingual text available and so can be “trusted” more. Och (2005) showed that increasing the size of the LM results in log-scale improvements in the BLEU score - a metric by which machine translated texts are scored. Figure 2.4, taken from (Koehn, 2007), shows the improvements gained by this increase in LM size.

Chapter 3

Probabilistic Data Structures

A data set like Google's Trillion Word N-gram is enormous and, at 24 GB when compressed, too large to fit in any standard machine RAM. For a system that accesses the LM hundreds of thousands of times per sentence, storing the set on disk memory will result in costly IO operations that will debilitate the system's performance. This requires us to find alternative ways of representing the n-grams in the LM if we want to gain performance from the data sets fantastic size without the traditional methods of handling massive data sets available such as mainframes or large-scale distributed environments such as Google's (Brants et al., 2007).

In this chapter we explore the properties of two data structures, the *Bloom filter* (BF) and *Lossy Dictionary* (LD), designed specifically for *lossy* representation of a set. Lossy encoding (also referred to as lossy compression) is a technique where some amount of the data being encoded is lost but the new representation retains enough of the original, *lossless* information to be useful when decoded. Lossy compression techniques are common for all types of multimedia storage with JPEG for images, MP3 for audio, and MPEG for video being well-known compression methods.

For the probabilistic data structures (PDS) we investigate, however, the data is transformed into a lossy representation using hash functions. These data structures are known as *probabilistic* as the algorithms they support inherently use a degree of randomness. Executing the algorithms without this measure of randomness, or deterministically, would consume too much time and/or memory depending on the problem being addressed. Instead the algorithms allow for a small probability of error as a trade off for their speed and space savings. The algorithms can be proven to have good performance on average with the worse case runtime being extremely unlikely to occur. Probabilistic algorithms and their formal verification is an active research area in computer science with corollaries in many academic fields. For example see Ravichandran et al. (2005); Hurd (2003) or the background of this project Talbot and Osborne (2007a,b).

Both the BF and LD allow queries of the sort, “Is $x \in S$?”. We call a positive return of the query a *hit* and a negative return a *miss*. There are two types of error returns when querying for set membership that we must concern ourselves with: false positives and false negatives. A *false positive* occurs when the queried element is not a member of the key set, $x \notin S$, but is reported as a hit and an associated value is returned in error. A *false negative* is the complement to the false positive, so an element $x \in S$ is reported as a miss and no associated information is returned. A PDS that allows for only one of these types of errors is said to have a *one-sided* error while one that allows for both types returns *two-sided* error.

However, a LM needs to support more than just set membership queries. Frequency values associated with keys that are in the set must be returned from the model too. The LD supports encoding key/value pairs already, but the BF does not. However, Talbot and Osborne (2007a) used some clever techniques to enable the support of key/value pairs in a BF-LM. This is discussed in the next chapter.

3.1 Hash Functions

A *hash function* is a function that maps data from a bit vector from one domain into another (usually smaller) domain such that

$$h : U \times \{0,1\}^w \rightarrow \{0,1\}^b.$$

In our discussion w is a integer that represents the bit length of a word on a *unit-cost RAM model* (Hagerup, 1998). The element to be hashed comes from a set S of size n in a universe U such that $S \subseteq U$ and $U = \{0,1\}^w$. The element's representation in the domain b is called a *signature* or *fingerprint* of the data. The hash function *chops* and *mixes* the elements of S deterministically to produce the signatures. A hash function must be reproducible so that if the same data passes through some hash function it has the same fingerprint every time. It must also be deterministic; if the outputs from some hash function for two data elements are different then we know the two pieces of data were not equal. Figure 3.1 shows an example of a hash function.

A hash function used with its counterpart data structure, the *hash table*, is a specialised type of *dictionary* or associative array that links *key-value* pairs when a queried

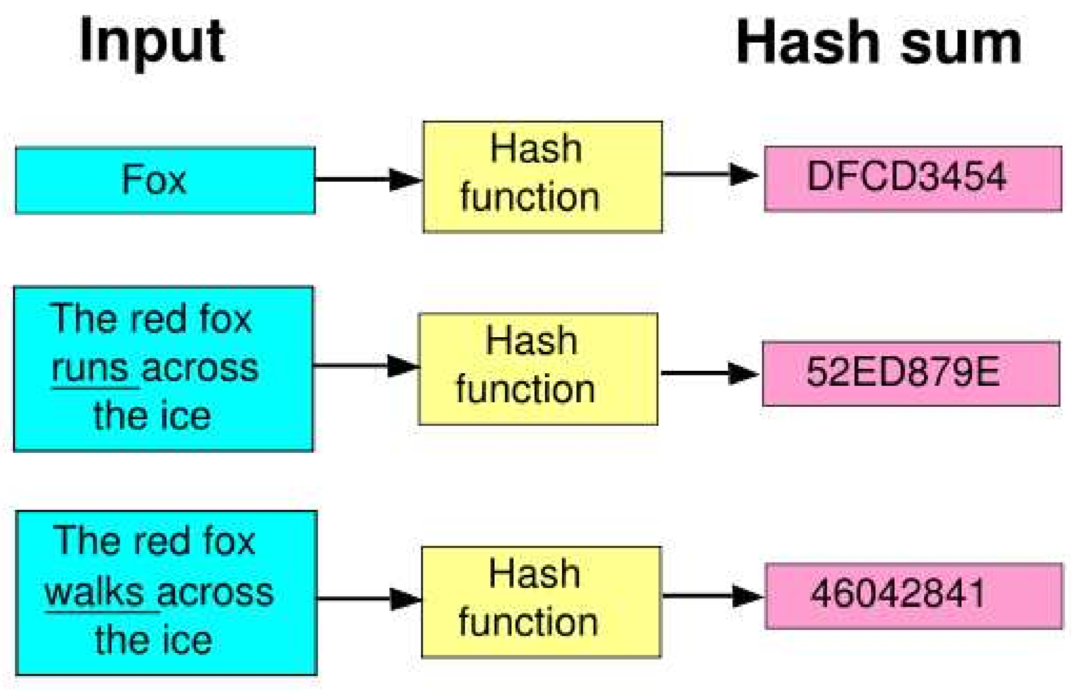

Figure 3.1: *An example hash function in action. The strings are transformed into hexadecimal signatures by the hash function.*

item is found to be a member of the set encoded in the table. A key k_i is generated by the hash function such that $h(k_i)$ maps to a value in the range of the table size, $\{0,1,\ldots,m-1\}$, and the associated value a_i is stored in the cell $h(k_i)$. A very attractive property of a vanilla hash table is its constant look-up time in the table regardless of the size of the data set in the hash table. Figure 3.2 illustrates a hash table scheme.

An essential property of a good hash function is *uniform distribution* of its outputs. Since we are using a binary base in the RAM model the number of unique values that can be encoded into b-bits is 2^b. If S is large and $b \ll w$, then some of the elements in S will collide when mapped into the smaller space b from w. *Collisions* are minimized by choosing a hash function whose outputs are uniformly distributed over b. We can view each possible value in b as a *bucket* that the hash function can “dump” its value into. If the hash functions outputs are not uniformly distributed they will cluster into a few buckets while many other buckets remain empty. We can employ statistical tests that measure the degree of randomness in the hash function output’s distribution. If the elements in b seem randomly distributed we know our hash function is not biasing outputs to certain buckets and so more likely to be uniformly distributed.

Perfect hashing is a technique that, when the keys of set are static and all known

1

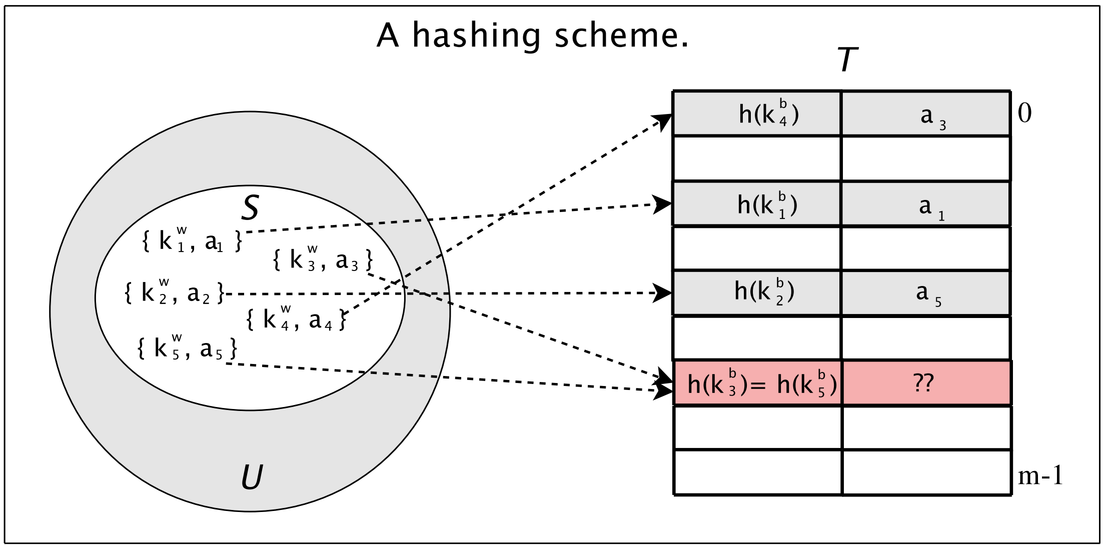

Figure 3.2: *The pairs of keys k_i and the associated information a_i of the set S are mapping via the hash function h into cells of the table T. The keys k_i are w-bit lengths in S and b-bit lengths in T. There is a collision between the elements k_2 and k_4 so the value of a_i for that cell in T is unknown.*

and with a large enough number of buckets (usually of the order n^2), theoretically allows for no collisions in the hash table with probability $1/2$ (Cormen et al., 2001). However, given the size of Google's data set, employing a perfect hashing scheme for the LM we're implementing is not feasible as the number of buckets that would be necessary would exceed a single machine's memory. There are other techniques to resolve collisions, including *chaining*, *open addressing*, and *Cuckoo hashing* (Pagh and Rodler, 2001a). A variant of the last technique is used in the Lossy Dictionary described further below.

3.1.1 Universal Hashing

One way we can minimize collisions in our hashing scheme is by choosing a special class of hash functions H independent of the keys that are being stored. The hash function parameters are chosen at random so the performance of the hash functions differ with each execution but we can show good performance on average. Specifically, if H is a collection of finite, randomly generated hash functions where each hash function $h_a \in H$ maps elements of S into the range $\{0, 1, \ldots, 2^b - 1\}$, H is said to be *universal* if, for all distinct keys $x, y \in S$, the number of hash functions for which

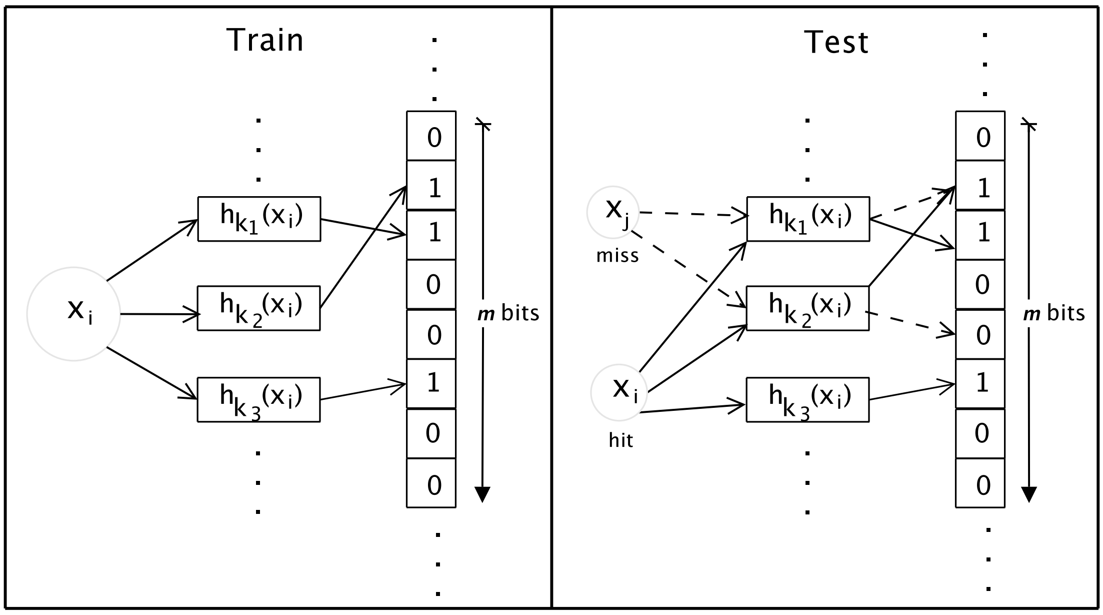

Figure 3.3: *Training and testing a Bloom filter. To train, we "turn on" k bits in the array for each key x_i. A test for membership fails if any index in the array the hash functions map to is zero. Else we have a hit and assume membership.*

$h(x) = h(y) \leq |H|/2^b$. The corollary to this is that for a randomly chosen hash function $h_a \in H$ we have the $P(h_a(x) = h_a(y)) \leq 1/2^b$ for distinct keys x and y. Using a bit of theory from numerical analysis we can quite easily generate a class of universal hash functions for which the above is provable. (A detailed proof can be found in Cormen et al. (2001).)

3.2 Bloom filter

The *Bloom filter* (BF) (Bloom, 1970) is a PDSs that supports queries for set membership. The BF has a unique encoding algorithm gives it rather spectacular space savings at the cost of a tractable, one-sided error rate.

Before training, the BF is an array of m bits initialized to zero. To train the BF we need k independent hash functions (such as a family of universal hash function described in the previous section) such that each hash function maps its output to one of the m bits in the array, $h_k(x) \rightarrow \{0, 1, \ldots, m-1\}$. Each element x in the set S of size n that we're representing with the BF is passed through each of the k hash functions and the output bit in the array is set to one. So for each $x \in S$, k bits of the array are "turned on". If the bit for a certain h_k is already set to one from a previous element then it stays on. This is the source of the fantastic space advantage the BF has over most other data structures that can encode a set - the fact that it shares the bit buckets between the elements of S.

To test an element for membership in the set encoded in the BF we pass it through the same k hash functions and check the output bit of each hash function to see if it is turned on. If any of the bits are zero then we know for certain the element is not a member of the set, but if each position in the bit array is set to one for all the hash functions, then we have a hit and we assume the element is a member. However, there is a possibility with a hit that we actually have a false positive with the same probability that a random selection of k bits in the array are set to one.

If we assume that the hash functions are uniformly distributed and will select each index in the m-bit array with equal probability then an arbitrary bit is set to one by a certain hash function with probability $1/m$. The probability any bit is not one after execution of a single hash is

$$1 - \frac{1}{m}.$$

The probability that a bit is still zero after a single element has passed through all k hash functions is

$$\left(1 - \frac{1}{m}\right)^k.$$

Adapting the above for all n elements gives

$$\left(1 - \frac{1}{m}\right)^{kn}$$

that a bit is zero. The probability, then, that a bit is one is

$$1 - \left(1 - \frac{1}{m}\right)^{kn}.$$

If an element were not a member of S, the probability that it would return a one for each of the k hash function, and therefore return a false positive, is

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k.$$

Rewriting using the limit of the base of the natural log gives

$$\left(1 - \exp\left(-\frac{kn}{m}\right)\right)^k$$

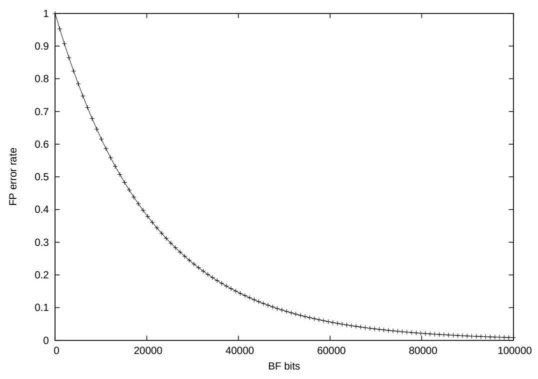

Figure 3.4: *BF error vs. space*

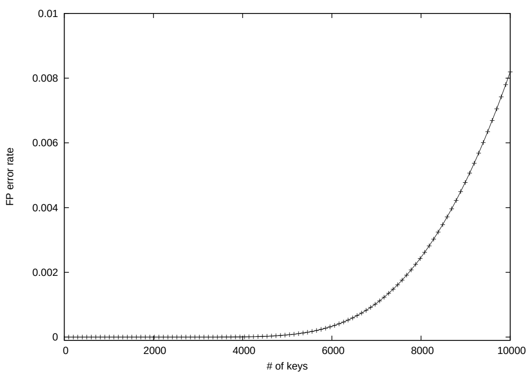

Figure 3.5: *BF error vs. # keys*

and taking the derivative, setting to zero to minimize the error probability and solving for k gives the optimal number of hash functions

$$k = \ln 2 \left(\frac{m}{n}\right)$$

which implies half the bits of the BF array should be set to one. This gives a false positive probability of

$$\left(\frac{1}{2}\right)^k \approx 0.6185^{m/n}$$

For a given n the probability of false positives decreases as m increases and more space is used. For a static m the error rate increases as n increases. This trade-off between error rates versus space and the number of keys is shown in Figures 3.4 and 3.5 [1].

BF's are one of the most widely used PDSs in industry with many practical applications such as spell-checkers, database applications (Costa et al., 2006) and network routing (Broder and Mitzenmacher, 2002). The nature of the encoding of the BF makes it impossible to retrieve the original data which is considered a feature when used in security sensitive domains such as IP address caching. It is also impossible to remove a key from a BF without corrupting other elements in the set [2]. There have been various alternatives suggested to BF such as Pagh et al. (2005), but the simplicity and overall performance of the original BF has made it the baseline by which most other PDSs are compared.

[1]The graphs in figures 3.4 and 3.5 are shown for the same m (bits) and n (# keys).

[2]This can be achieved using *Counting filters* (Fan et al., 2000) where another bit array is used to encode the elements removed from the original BF. However, space usage is up to four times greater and we must deal with false positives of the deleted items which then become false negatives overall which aren't permitted in the BF scheme.

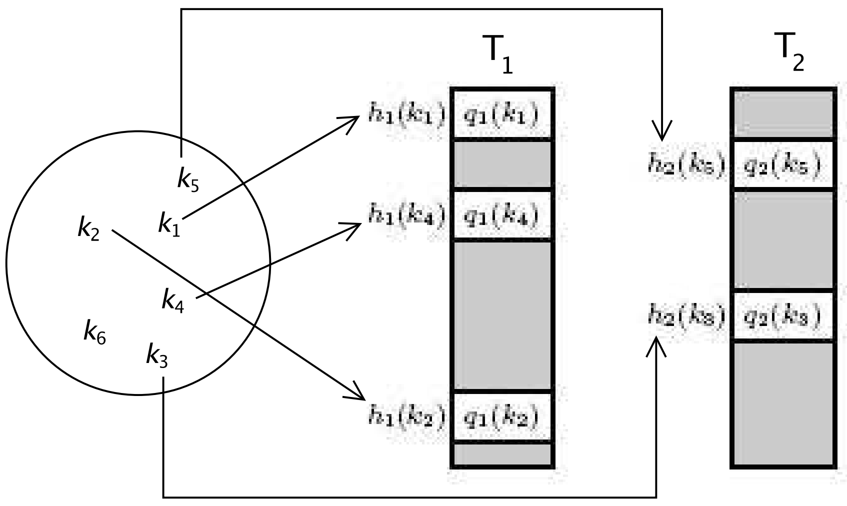

Figure 3.6: *The Lossy Dictionary has two hash tables T_1 and T_2 each with their own hash and quotient functions. Quotient representations $q_1(k_i)$ or $q_2(k_i)$ of elements of the set are stored in $h_1(k_i)$ of T_1 or $h_2(k_i)$ of T_2. k_6 is not stored and becomes a false negative.*

3.3 Lossy Dictionary

The BF addresses only the membership problem [3] and does not implement an associative array where, if a key is a member of the set, we can retrieve an associated value of the key. The *Lossy Dictionary* (LD) (Pagh and Rodler, 2001b) is a memory efficient, simple PDS that allows storage and very fast retrieval of *key-value* pairs at the cost of two-sided error. We can still get a measure of false positives as with the BF, but we also get false negatives as some of the data of S is thrown away when constructing the LD. That is, we only store a subset S' of S such that $S' \subseteq S$. If the associated key values are weighted in some arbitrary way so some of the keys are more valued than others, the LD tries to maximize the total sum of weights included in the LD under a given space constraint and for a specific error parameter ε.

We restate the general theorem of the LD below:

> Let a sequence of keys $k_1, \ldots, k_n \in \{0,1\}^w$, associated information $a_1, \ldots, a_n \in \{0,1\}^l$, and weights $v_1 \geq \cdots \geq v_n \geq 0$ be given. Let $r > 0$ be an even integer, and $b > 0$ an integer. Suppose we have oracle access to random hash

[3] *Bloomier filters* (Chazelle et al., 2004), a generalization of Bloom filters, are able to encode key/-value pairs. They use a pair of bit arrays per bit possible in the result. For each pair, child pairs of arrays are trained to handle the case when a key in one bit array of the pair causes a false positive in its counterpart array. If the possible results have high entropy they can be hashed to smaller values before implementing the filter to keep the number of array pairs small.

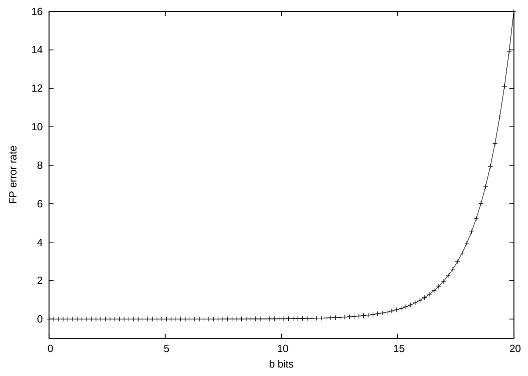

Figure 3.7: *FP rate of LD.*

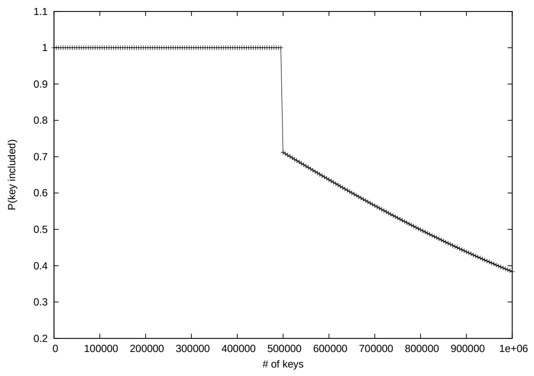

Figure 3.8: *Probability k_i is included in LD.*

functions $h_1, h_2 : \{0,1\}^w \to \{1,\ldots,r/2\}$ and corresponding quotient functions $q_1, q_2 : \{0,1\}^w \to \{0,1\}^s \setminus 0^s$. There is a lossy dictionary with the following properties:

1. The space usage is $r(s-b+l)$ bits (two tables with $r/2$ cells of $s-b+l$ bits).
2. The fraction of false positives is bounded by $\varepsilon \leq (2^b-1)r/2^w$.
3. The expected weight of the keys in the set stored is $\sum_{i=1}^{n} p_{r,i} v_i$ where

$$p_{r,i} \geq \begin{cases} 1 - 52r^{-1}/\left(\frac{r/2}{i} - 1\right), & \text{for } i < r/2 \\ 2(1-2/r)^{i-1} - (1-2/r)^{2(i-1)}, & \text{for } i \geq r/2 \end{cases}$$

   is the probability that k_i is included in the set.
4. Lookups can be done using at most two (independent) accesses to the tables.
5. It can be constructed in time $O(n \log^* n + rl/w)$.

The *quotient functions* $q(k)$ in the theorem above are quotients of hash functions such that "when storing a key k in cell $h(k)$ of a hash table, it is sufficient to store $q(k)$ to uniquely identify k among all other elements hashing to $h(k)$" (Pagh and Rodler, 2001b) in fewer bits than storing $h(k)$ directly. The space used to store the quotient function output is the first $s-b$ bits, but note that if we set $b=0$ we obtain a LD with theoretically no false positives (assuming perfect universal hashing). Since there can be at most 2^b elements of $U = \{0,1\}^w$ that share the same quotient function value and we disallow quotient function outputs to be zero, false positives are bounded from above by 2^b-1 for each cell. The associated information is stored in the remaining l bits of each cell. The total number of cells in the table is r. With two hash tables T_1 and T_2 we get $r/2$ cells per table and $m = r(s-b+l)$ bits used in general. We can choose optimal r given a predetermined maximum error ε and space usage m. Figure 3.6 (in part from Pagh and Rodler (2001b)) shows the overall structure of the LD.

To train the LD, we use the following greedy algorithm:

1. Initialize a union-find data structure for the cells of the hash tables.

2. For each equivalence class, set a "saturated" flag to false.

3. For $i = 1, \dots, n$:

   (a) Find the equivalence class c_b of each cell $h_b(k_i)$ in T_b, for $b = 1, 2$.

   (b) If c_1 or c_2 is not saturated:

      i. Include k_i in the solution.

      ii. Join c_1 and c_2 to form an equivalence class c.

      iii. Set the saturated flag of c if $c_1 = c_2$ of if the flag is set for c_1 or c_2.

In practice, we simply hash an element k_i into the cell $h_1(k_i)$ of T_1. If the cell $T_1[h_1(k_i)]$ is empty we store $q_1(k_i)$ and the associated value v_i in it and we're finished. Else we repeat the process for $h_2(k_i)$ of T_2. If the cell in T_2 is not empty, k_i is discarded and included in the subset of false negatives. Figure 3.8 shows the probability of a key being included in the tables from a set of size one million. For $i < r/2$ we keep almost all the keys with almost certain probability, but for $i \geq r/2$ the probability of a key being included drops sharply throughout. We examine this property empirically in chapter four.

During testing a similar process is repeated. We hash a test element k_i with h_1 and get the quotient value. If the value in $T_1[h_1(k_i)] = q_1(k_i)$ we assume a hit and retrieve the associated value v_i. If it doesn't match we hash k_i through h_2 and q_2 and check for a match in T_2. If the equality fails, we report a miss which may or may not be a false negative.

3.4 Comparison

The LD uses the RAM approach for storing and accessing information which processes bits in groups of *words*. The theoretic word model uses units of computation in binary base $\{0,1\}^w$ which simulates a processor where bits are compared in word sizes w that range from 2^3 (8-bit) up to 2^7 (128-bit) normally. The BF, on the other hand, uses a boolean model which requires more computing time to compare the same number of bits that an LD can do with one bucket comparison.

As stated, the BF doesn't support associated key/value pairs as each bit is accessed in isolation and is either on and a member of the set or off and not a member. There are no other values possible in a boolean setting. When we use the unit-cost RAM model we group a pair of bit-buckets or divide a bucket into various sub-buckets that can easily represent associated key/value data. This is the underlying cause of the space advantage the BF has between the two PDSs as well since the word-based RAM models *must* use a certain minimum number of bits for each item in the data set. The RAM model can't share bits among the keys it is encoding like the BF does as once a word bucket is setup to represent a key from the set, adding or changing the bits in it will destroy the information for the originally contained key. Once the BF is setup, it also can't change bits arbitrarily without potentially destroying information related to multiple keys. In this way the BF can be envisioned as a single giant bucket which is trained to represent the whole space between the bits and it derives it's space advantage over the LD from this.

The probability of error for each structure depends specifically on the data set and free variable parameter settings such as the space bounds or encoding scheme used for the buckets in the LD. But the BF's one-sided error is an advantage compared with the two-sided error of the LD which can return both false negatives and false positives so we can say that all things being equal the BF has lower error returns than the LD.

Chapter 4

Previous Work

The previous work and basis for this project was the groundbreaking work published by Talbot and Osborne (2007a) and Talbot and Osborne (2007b). The research reported in these papers was the first time a randomised data structure such as a BF was used to encode a LM and some very clever ideas were used to minimize the false positive returns and enable the BF to support key-value pairs rather than just set membership queries.

To train the BF as a data structure that supports key-value pairs, where the key is the n-gram and the value is the count of that n-gram in the corpus, the value was appended to the key before entering this composite event into the BF. To minimise the number of bits needed for the frequency value a *log-frequency* encoding scheme was used. The n-gram counts were first quantized by using a logarithmic codebook so the actual count $c(x)$ of the n-gram was represented as a quantized count $qc(x)$ such that

$$qc(x) = 1 + \lfloor \log_b c(x) \rfloor.$$

The accuracy of this codebook relies on the Zipf distribution of the n-grams where a few events occur frequently but most events occur only a small number of times. For the high-end of the Zipf distribution the counts in the BF-LM are exponentially decayed using the log-frequency scheme. However, the large gap in distribution of these events means that a ratio of the likelihood of the model was preserved in the BF-LM.

To keep it tractable as well as minimize the false positive probability the composite events of n-grams and their quantised counts were entered into the BF using *sub-sequence filtering*. With sub-sequence filtering, each n-gram was entered into the BF with attached counts from one up to the value of $qc(x)$ for that n-gram. During

```
Algorithm 1 Training frequency BF
  Input: $\mathcal{S}_{train}$, $\{h_1, ...h_k\}$ and $\mathcal{BF} = \emptyset$
  Output: $\mathcal{BF}$
  for all $x \in \mathcal{S}_{train}$ do
    $c(x) \leftarrow$ frequency of $n$-gram $x$ in $\mathcal{S}_{train}$
    $qc(x) \leftarrow$ quantisation of $c(x)$ (Eq. 1)
    for $j = 1$ to $qc(x)$ do
      for $i = 1$ to $k$ do
        $h_i(x) \leftarrow$ hash of event $\{x, j\}$ under $h_i$
        $\mathcal{BF}[h_i(x)] \leftarrow 1$
      end for
    end for
  end for
  return $\mathcal{BF}$
```

```
Algorithm 2 Test frequency BF
  Input: $x$, $MAXQCOUNT$, $\{h_1, ...h_k\}$ and $\mathcal{BF}$
  Output: Upper bound on $c(x) \in \mathcal{S}_{train}$
  for $j = 1$ to $MAXQCOUNT$ do
    for $i = 1$ to $k$ do
      $h_i(x) \leftarrow$ hash of event $\{x, j\}$ under $h_i$
      if $\mathcal{BF}[h_i(x)] = 0$ then
        return $E[c(x)|qc(x) = j - 1]$ (Eq. 2)
      end if
    end for
  end for
```

Figure 4.1: *The algorithms (from Talbot and Osborne, 2007a) for training and testing the n-gram counts in a BF.*

testing the frequency for a n-gram being queried is found by appending counts to the n-gram starting from one. The BF is then checked for membership for that composite n-gram/count event with the standard BF testing algorithm. If each of the *k* hash function index values are set to one in the BF array, the value of the n-gram is incremented by one. The process is repeated until one of the hash outputs indexes a zero bit in the array or a maximum counter is reached and the number of iterations minus 1 is returned as the count in the BF-LM. For the majority of the n-grams this process is only repeated once or twice because of the log-frequency scheme. The one-sided error rate of the BF ensures the returned frequency is never underestimated.

The original count is then approximated by the formula

$$E(c(x)) \approx \frac{b^{j-1} + b^j - 1}{2}.$$

and *run-time smoothing* is then performed to retrieve the smoothed probability in Talbot and Osborne (2007b). The statistics needed for the smoothing algorithm used are also encoded in a lossy format in a bit array with the exception of singleton events. In this case, the proxy that an event was a singleton was the event itself.

The BF-LM was tested in an application setting via a SMT system with the Moses decoder (Koehn and Hoang, 2007) trained on the Europarl corpus (Koehn, 2003). Using a 8% fraction of the 924 MB lossless LM space, the BLEU scores for translations done using the BF-LM matched those for translations done with a lossless LM. Other tests compared the BLEU score versus various other properties of the BF-LM such as the false positive probability, the MSE between a lossless and lossy model, and the bits used per n-gram. Though the space savings were quite spectacular, the BF-LM is a slow data structure due to the number of hashes used to support the bit-sharing as well as the added hashes for the sub-sequencing to support key/value pairs. In average the translation time took between 2–5 times longer when a BF-LM was used.

This work provided the first complete framework for using randomised data structure LMs in NLP and was the basis of investigation for this project.

Chapter 5

Testing Framework

This chapter describes the testing framework we used to test our lossy LMs. An overview of the framework's structure is shown in figure 5.1.

5.1 Overview

We implemented lossy language models using the BF-LM from Talbot and Osborne (2007a) and a lossy dictionary LM (LD-LM) using various encoding schemes. We can't use perplexity or cross-entropy to measure the goodness of a lossy LM as the error returns may skew the distribution and report better than actual results. Instead, since we can be sure that the reported perplexity and probabilities from the lossless LM are correct, we used SRILM Stolcke (2002) to train a lossless LM and used its output as a gold standard to measure the performance of our lossy LMs. Specifically, we used the mean squared error (MSE) of the per sentence difference between the log-probabilities of our lossy LMs and SRILM's output. We aim to minimize the MSE throughout our experiments.

5.2 Training Data

For training the same corpus was always used for both lossy and lossless LMs. After initial setup and testing with just a few sentences, a medium sized corpus of 3.5 million words and 1.8 million unique n-grams that comes with the SRILM source was used for training during the bulk of the experiments. Final tests were done with the 110 million word British National Corpus [1].

[1] `http://www.natcorp.ox.ac.uk/`

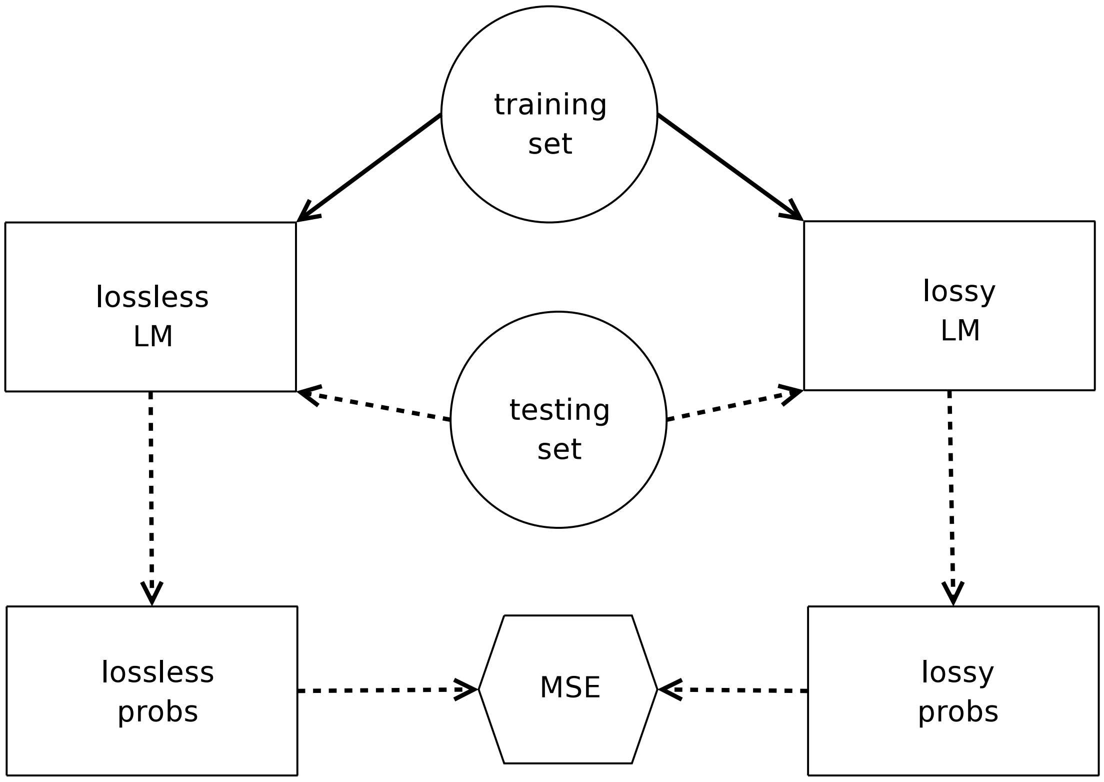

Figure 5.1: *Our testing framework. The same training and testing data was used for both lossy and lossless LMs and the MSE of the probability differences was used as a evaluation metric.*

5.2.1 Smoothing

The lossless LM was trained with MKN smoothing for orders 1–3. We also implemented MKN smoothing for our lossy LM. This means we need to collect the count of counts of all order n-grams of the model that occur from one to four separately for the discount parameters $D_{i\in\{1,2,3+\}}$, the counts of the unique histories for all the n-grams of each order of the model for the probability $P_{MKN}(w_{i-n+1}^{i})$, and the count of how many unique words follow each history for the $\gamma(w_{i-n+1}^{i-1})$ function which ensures a normalized distribution. As in Talbot and Osborne (2007b), we don't need to explicitly store the values of singleton counts since we can use the event itself as a *proxy* for one. This means we can store the parameters of the smoothing algorithm in separate lossy data structure of a much smaller size than the one that stores the n-gram events and counts. Pseudocode for the algorithm to find the count of unigram contexts is listed in Appendix A of this report.

Each order of the LM for which we use MKN smoothing requires a transversal of the data. For large data sets and an higher order model of fivegrams, this is very time and resource consuming to do repeatedly. During our experiments, since we were purely testing the information content and properties of the data structures encoding the LM and not interested in the precise probabilities of the events themselves, we used Google's Stupid Backoff smoothing algorithm for the lossy LMs because of its speed and simplicity. An extremely small part of the difference in the outputs between the lossless and lossy LMs stemmed from the different smoothing techniques used between the gold standard SRILM output and our lossy LMs, but as this was a constant factor between the lossy data structures we could safely ignore it.

To use the smoothed probabilities during testing we split each test sentence into tokens and surrounded the first and last of them with begin (<s>) and end (</s>) sentence markers respectively. To extract the n-grams in the sentence we iterate through each word index of the sentence adding the histories of the word starting from the highest order length of the model and working towards the word we're finding the conditional probability for. For Stupid Backoff we'd pickup backoff probabilities as we built the n-gram and clear them if the full order n-gram was found in the LM since it isn't an interpolated algorithm. For MKN, we found the backoff and recursed until we reached the unigram level at which point we'd pick up the full n-gram probabilities, add them to the backoffs and return through the stack. The algorithms for smoothed probability retrieval are listed in full in Appendix A.

5.3 Testing Data

To test our lossy LMs for perplexity we used clean text of 5290 more or less grammatical sentences. However, in an application environment such as SMT most of the n-grams being passed to the LM by the decoder are rubbish. To simulate this we generated multiple completely randomised text files of varying length and tested the LMs with these as well. For all the results of the experiments shown in this chapter we have taken the average of the results of multiple tests on both the perplexity data file and arbitrary randomized n-gram files unless specified otherwise. As well, the tests that compare the PDS LMs were always executed with the same instantiation of the hash function family.

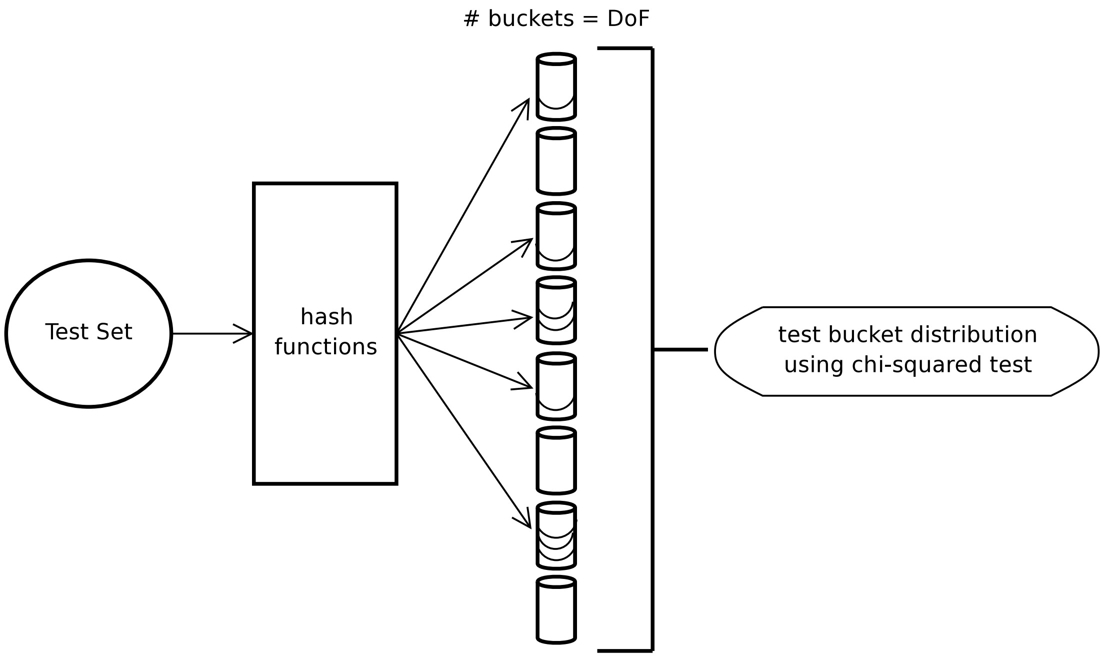

Figure 5.2: *Uniform distribution test for hash function outputs.*

5.4 Hash Functions.

We used a class of randomised, universal hash functions described in Carter and Wegman (1977). Given a key k of string data, we break it up into $r+1$ segments $k = \langle k_0, k_1, \ldots, k_r \rangle$ which corresponds naturally to each ASCII character of the string. We then hash each element of the decomposed key and define one member of the hash function family as

$$h(k) = \sum_{i=0}^{r} (a_i k_i + b_i) \bmod m$$

where $a = \langle a_1, a_2, \ldots, a_r \rangle$ and $b = \langle b_1, b_2, \ldots, b_r \rangle$ and all a_i and b_i are chosen randomly from $\{0, 1, \ldots, m-1\}$ where m is a prime number guaranteed to be larger than any of the hash outputs.

To test the distribution of the outputs of the hash function family, we used the *Chi-squared test*. The chi-squared test is a measure of the randomness of a distribution and the chi-squared value χ^2 is obtained by summing the squared difference between the all obtained results o and the expected, uniform result e divided by the expected result:

$$\chi^2 = \sum_i \left[\frac{(o_i - e)^2}{e} \right]$$

The value χ^2 is then compared to chi-squared distribution tables and the critical p-value is obtained as a function of the degrees of freedom of the distribution. We maintained randomness of the distribution if our critical value $p > 0.05$.

Our hash function outputs were designed to range between 16-bit outputs (2^{16}) to arbitrarily high values (though ranging beyond 2^64 values is superfluous even for the trillion word data set). We split the test up between the lower and upper halves of the output bits and measured the distribution of each half. Each element of the test data was hashed by a randomly chosen member of our universal hash family and the hash value bit-masked and right-shifted (for the upper half) to split the upper and lower half of the bits. Then the index of the hash values in a representative data structure was incremented and, after all elements had been hashed and bucketed, the chi-squared distribution was performed for both halves. If both the upper and lower half distributions were randomly distributed separately then they were random together as well but if many outputs were written to the same subset of the buckets the chi-squared test would show the hash outputs as not being randomised and we could expect high collisions for the chosen hash family (Mulvey, 2006).

We ran a total of 500 tests over 3 million elements for 16, 32, and 64-bit outputs, concentrating mostly on the higher bit outputs. Only twice was the critical p-value $<$ 0.05 which was the expected result that, with near certainty, the universal class of hash function's outputs were random and uniformly distributed.

Chapter 6

Experiments

In this chapter we focus on the experiments conducted over the course of the research and the empirical results of the experiments. This chapter also reports the findings of the first known use of a Lossy Dictionary to implement a language model. Since the LD has many free variables associated with it, there are a lot of parameters we can fiddle with and a number of variations of LD-LM we examine.

6.1 Baseline LD-LM

Our LD-LM baseline was the first attempt at using a LD for a LM and we tried a novel approach. Instead of encoding just the n-gram events and their associated counts, a *sub-bucket* scheme was used to try and include a smoothed LM representation in the LD tables with no run-time smoothing. Our hope was to create a proper ratio of a lossless LM in the LD-LM using this scheme so no runtime smoothing would be needed.

The sub-bucket encoding scheme is shown in figure 6.1. Our LD stored all the information for a n-gram in just one cell unlike a traditional hash table which uses one cell for the hash signature and a conceptually adjacent cell for the associated information. Using just once cell per bucket saves a considerable amount of space and makes the division of a pre-defined space parameter easier to manipulate. For our baseline each table cell was divided in 3 sub-buckets. For example, if a table cell was 16-bits, we use a 8-4-4 sub-bucket scheme where the first sub-bucket of the 8 most significant bits (MSB) would be used for the quotient signature [1] of the n-gram, the next 4 bits to encode a quantized value of the probability and the least significant 4 bits remaining

[1]The quotient functions were another family of hash functions with the added stipulation that no output could be zero.

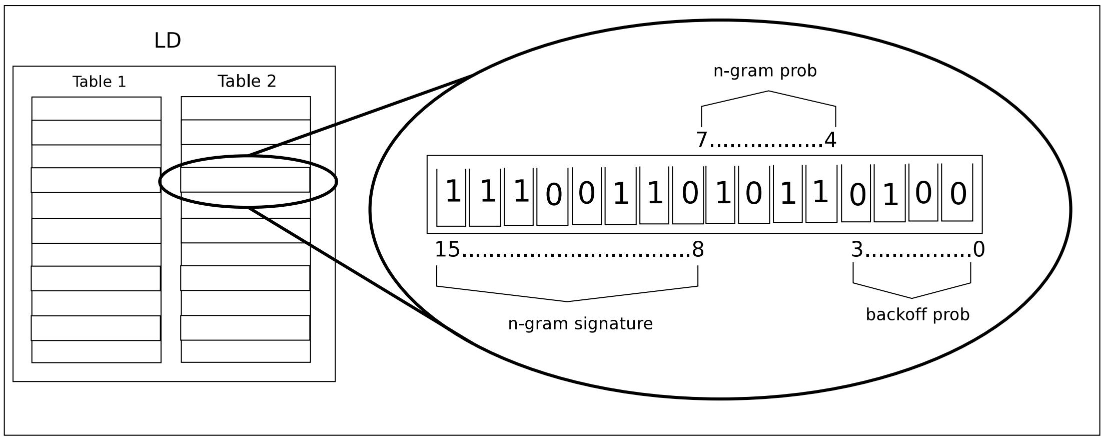

Figure 6.1: *An 8-4-4 LD-LM sub-bucket scheme. Each cell is divided into sub-buckets where the 8 MSBs represent the n-gram signature, the 4 middle bits the main n-gram probability, and the LSB bucket the back-off probability.*

were used for the back-off probability.

The n-gram and back-off probabilities are small floating point numbers, so we used a quantizing function $qc_{prb}(k) = \lceil \log_b(\Pr(k) \times 10^d) \rceil$ where $d = 3$ for the back-off probabilities and $d = 1$ for the n-gram probabilities. The LD-BF was trained using a series of left bit-shifts and additions to combine the n-gram quotient signature and the probabilities to create a composite value for each cells. The MSB of the probability sub-buckets was used as a positive or negative sign marker [2] and if the absolute value of the probability was too large to fit in the remaining allocated bits, all of them would be set to 1 for the maximum value. The index of the tables was found by hashing the n-gram and taking the modulo of the hash output with the table size. If the first table cell was empty, the composite event was inserted in it. Otherwise the n-gram was passed through another member of the hash function family to get an index for table 2. If this was empty the composite value was inserted there; if not the n-gram and the probabilities were discarded.

During testing, the test n-gram would be hashed for an index and checked against the n-gram signatures in the table. If they matched it would be considered a hit and the quantized probabilities and backoff probabilities would be extrapolated using bitwise *AND* with a mask for each sub-bucket and right bit-shifts. The expected probabilities

[2] The +/- sign marker was to be consistent with the SRILM output which had both.

were recovered using the formula $\Pr(k) = b^j/10^d$ where j was the number retrieved from the encoded value in the cell. Example pseudocode for the algorithms to insert a value into a cell of the LD and retrieve the probabilities during testing are listed in Appendix A.

Figure 6.2 shows the percentage of the original data set that was contained in the LD as a function of the space parameter in megabytes (MB) for 8 and 16-bit cell sizes. With 16-bit cells and 1 MB of space we keep only 30% of the data, but at 10 MB 97% of the data is retained in the LD. If we use 8-bit cells, since we have double the number of table cells we keep a higher percentage of the data for a smaller table size. Obviously, we also lose half the information storage per event. In comparison, the lossless LM for our small training set used 34.99 MBs to store the whole set.

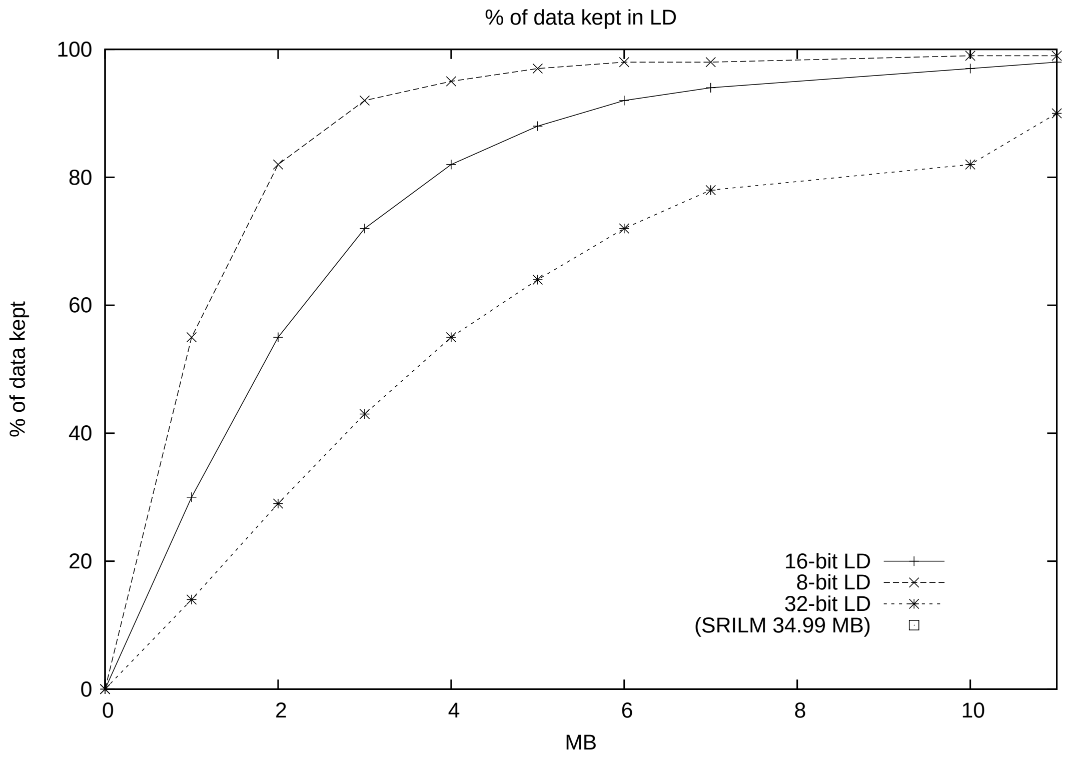

Figure 6.2: *The percentage of the original data set kept in the LD-LM as the size increases in MB. The lossless LM for this set of training data is stored at 35 MB.*

The graph in figure 6.3 shows the inverse relation to the percentage of the data kept which is the number of collisions for different space settings. As more space is given the number of collisions decrease until, for an amount of space that is far beyond what is necessary for the items in the space, we have a limit of zero collisions. We can tell from this that most collisions don't come from a large number of n-grams being mapped to the same output buckets by the hash functions, but rather by how much we limit the available range of outputs by the number of table cells available. Figure

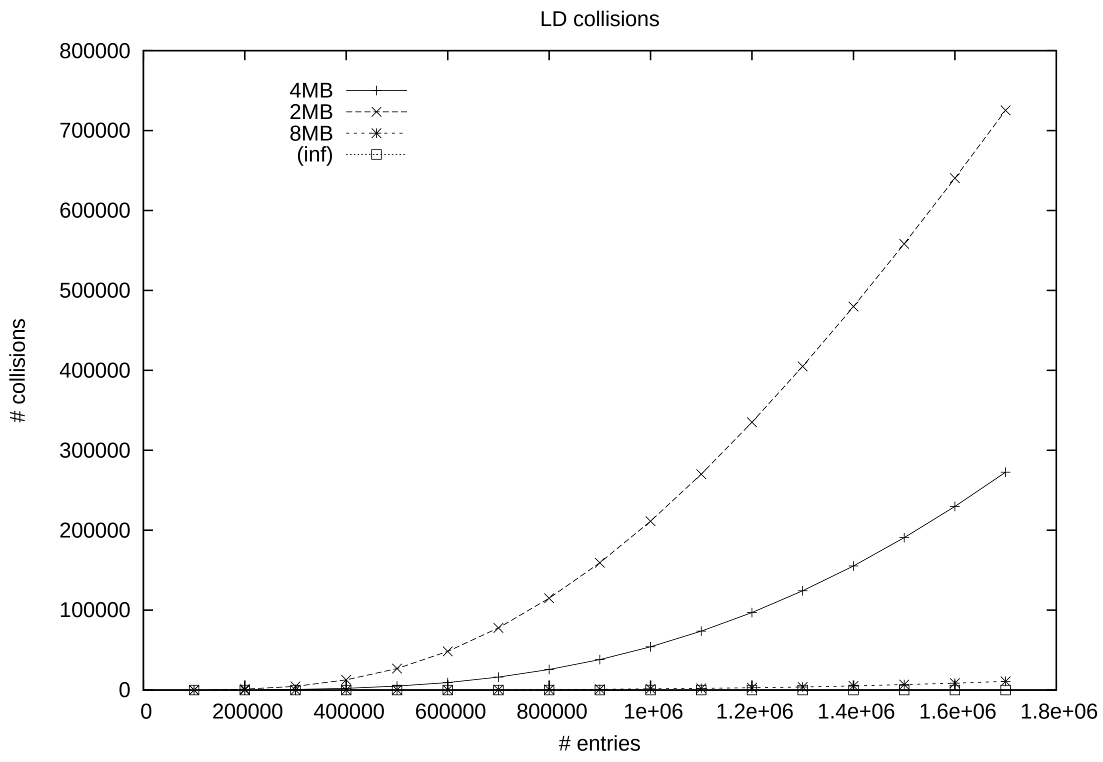

Figure 6.3: *The collisions of the LD for different table sizes. The subset of collisions become the set of false negatives for the lossy LM.*

6.4 shows the test results for different cell sizes and parameter settings of the sub-bucket model. The sub-bucket encoding scheme used for within table cells discarded a large amount of the probability information resulting in a high per sentence MSE. Interestingly, in a recent discussion Google's Director of Research, Peter Norvig, said Google had empirically found that only 16 levels (4 bits) of probability difference are necessary for highest BLEU scores in their SMT system (Norvig, 2007). Our tests correlate with this to the extent that the 16 bit 8-4-4 scheme has much the same error results as a 32 bit 21-7-4 scheme with twice the memory. The 24-4-4 outperforms the 21-7-4 scheme too implying the extra bits are better used to decrease the false postive rate. Using more than 4 bits for the probability doesn't seem to increase the lossy LM accuracy, but using less bits for either the probability or back-off results in higher errors as with the 16 bit 10-3-3 or 9-4-3 schemes. It seems worthwhile to do further investigation to see if the ratio of the sub-bucket LM is valid in practice, i.e., does it work and give higher probabilities for well-formed n-grams regardless of the MSE when compared with a lossless LM.

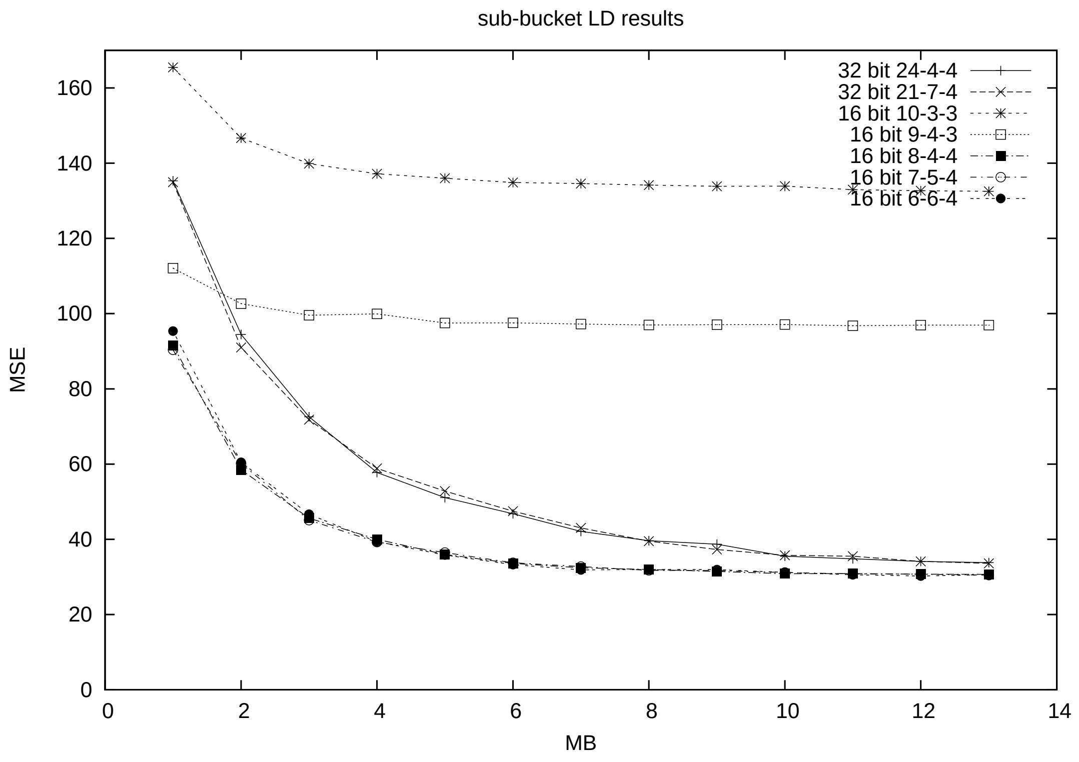

Figure 6.4: *The MSE error for the sub-bucket encoding scheme.*

6.2 Choosing a Weighted Subset

Until know we've been inserting events into the LD-LM at random from the n-gram set, but one of the features of the LD is being able to choose the subset of the associated values so as to maximize the sum of the weights contained in the LD. In practice, this is done by a simply ordering the keys in *non-increasing* order. In the case of the n-grams we order the weights by decreasing counts or the most probable n-grams. Google's n-gram set is in alphabetical order so we need to reorder the set by decreasing counts. However, we can't use a traditional sorting algorithm such as mergesort for this problem as the data set is too large to fit in memory and the sorting algorithms require the data to be held in a in-RAM array. Further, we must maintain the connection between the weight and the key and therefore can't separate the weights entirely from the n-grams to sort them.

We used an algorithm that depends in part on the Zipf distribution of the n-gram set to fit all but the lowest weighted data in memory which requires runtime $O(2n)$.

On the first pass we *prefix* count the number of each unique weight associated with the keys in the set, i.e. every unique weight has the count of the number of events with its same weight plus the sum of the keys with weight greater than its own. We store this number in a sorted dictionary structure called a *map* in C++. The keys of the map are the weights and the values are the prefix counts of all events in the n-gram set with that weight. On the second pass we store n-grams with counts >1 in an array in RAM memory with n-gram at the index in this array of the prefix count of the n-gram's weight. Then we subtract one from that weights prefix count in the map for the index of the next key in the set with the same weight. In this way we fill in the slots of the array so finally we end up with an array of n-grams in decreasing order. We write all events with the lowest weights out to static memory until all other prefix counts are zero. After, when we're training the LM, we first use the non-increasing array in RAM memory and then add the singleton events of lower weight in arbitrary order from the singleton file. This sorting algorithm's pseudocode is listed in Appendix A.

Once the weights are ordered in non-increasing order, the subset that is included in the LD will be the maximum subset able to be included given the hash function set and the size of the tables. Figure 6.5 show how ordering the data effects the results for the MSE of the sub-bucket scheme. Note that we use the smoothed probabilities as our weight metric for the most valued n-grams in the sub-bucket scheme. Since the higher order n-grams of the model usually have a greater smoothed probability mass associated with them compared to lower order n-grams, they're are at the beginning of the ordered set and therefore are included in the LD-LM with very high probability. However, these n-grams don't have backoff weights associated with them as no higher order model exists that will backoff to them since they are the highest order of the LM. Without ordering, the unigrams and bigrams with lower probability mass but also backoff mass are entered first and most of them end up in the LD-LM. This trade-off is a draw back of the sub-bucket scheme that with tight space bounds we lose much of the smoothed backoff probability mass in the ordered LD when including the highest weighted subset. We examine the effect of choosing the highest weighted subset on the MSE using the n-grams frequencies in experiment 4.

6.3 BF-LM

We implemented the BF-LM from Talbot and Osborne (2007a) described in background Sect. 4 of this report for our next experiment. For training, the n-grams and

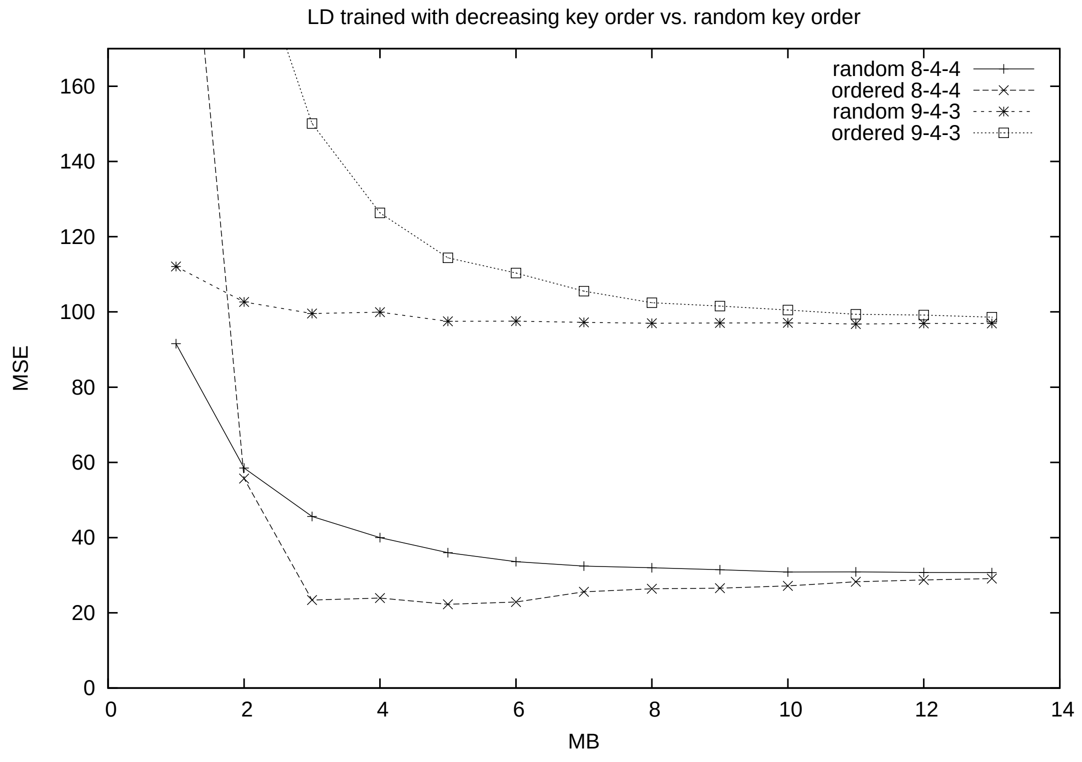

Figure 6.5: *Choosing the highest weighted subset of n-grams results in lowered error rates for the same space.*

their quantized frequencies were combined into composite events and were added to the BF with sub-sequence filtering. As stated previously, we used Google's Stupid Backoff for the smoothing algorithm and performed runtime smoothing during testing. The BF encodes the entire n-gram set so we don't have false positives to deal with. The errors come solely from false positives and how much the counts are degraded when quantized and the expected count is retrieved. The MSE is shown in figure 6.6 in contrast with the sub-bucket LD-LM MSE to highlight the much lower error rates of the BF-LM.

The graphs in figure 6.7 and 6.8 demonstrate the time differences between the LD and BF which we measure in the number of hash functions executed during during training and testing since the hash functions execute in constant time and are the same between the PDSs. The BF adjusts the number of hash functions to execute to an optimal setting as a function of the number of elements in the key set and bits in its array. As the error rate decreases with more space, the BF takes a longer time to execute. In contrast, the LD uses constant time regardless of the space. While the training time is not so important in that the LM will only be trained once, the time

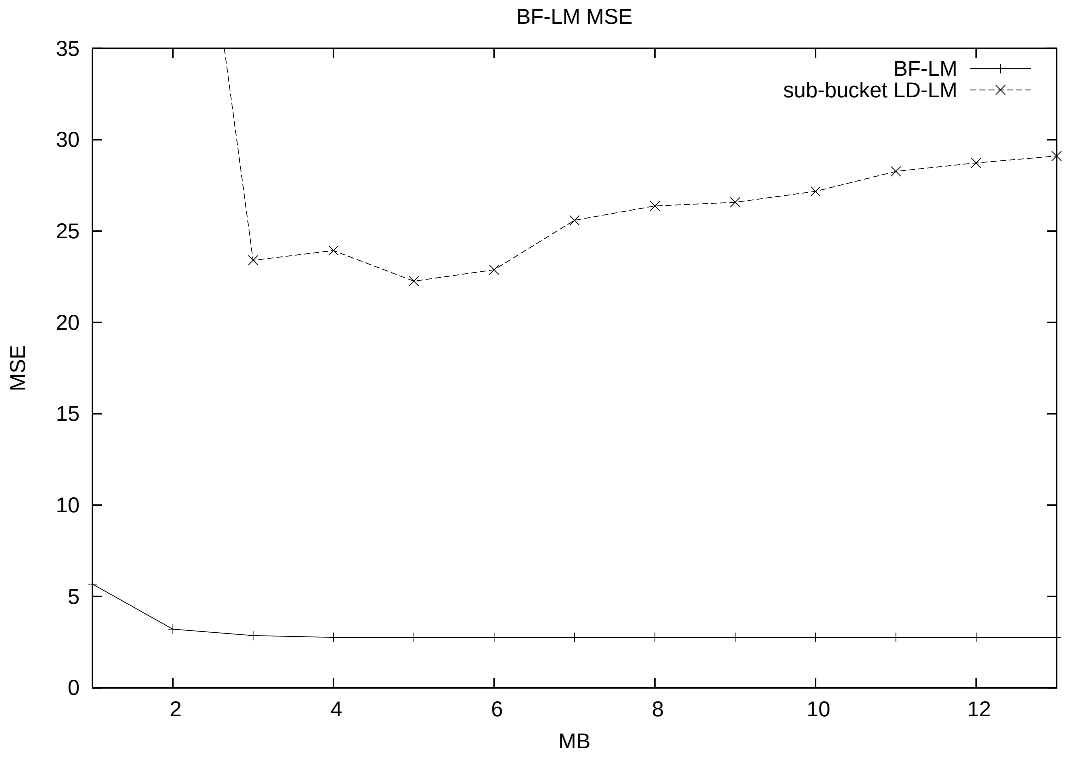

Figure 6.6: *MSE for the BF-LM.*

needed for testing a n-gram is a consideration for the data structure being chosen. A typical SMT system, for example, may query a LM millions of times per sentence and a very slow LM may have a substantially negative impact on the overall system performance. We wanted to find ways to narrow the time and error differences between the LD-LM, which has a high error rate but is very fast, and the BF-LM, which has minimal errors but is very slow.

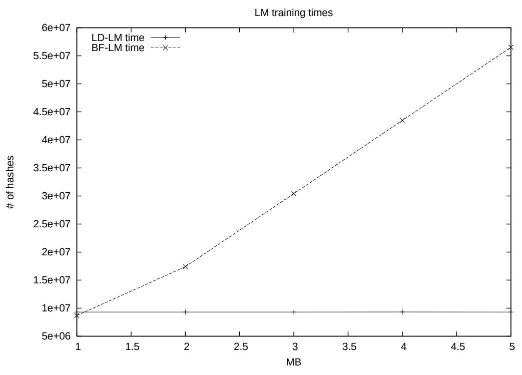

Figure 6.7: *Training times.*

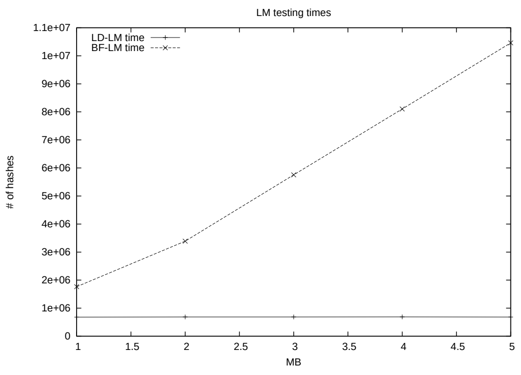

Figure 6.8: *Testing times.*

6.4 Composite LD-LM

We implemented a new scheme for the LD-LM that encoded n-grams and their quantized frequencies directly as composite events and, like the BF-LM, used runtime smoothing during testing. The frequencies were quantized and expected counts retrieved with the same formulas used in the BF-LM. We smoothed with Stupid Backoff. We still kept exact sub-buckets for the n-gram signature and the counts, but as we no longer needed a sub-bucket for a backoff probability there was more bits available for the n-gram signature. Also, because we know the highest frequency in our training corpus we can explicitly set the number of bits of this sub-bucket to be what we need. For example, using only 3 bits for the quantized counts would only give us a maximum of 749 for our retrieved expected count. However, using 4 bits gives a maximum of over 6 million.

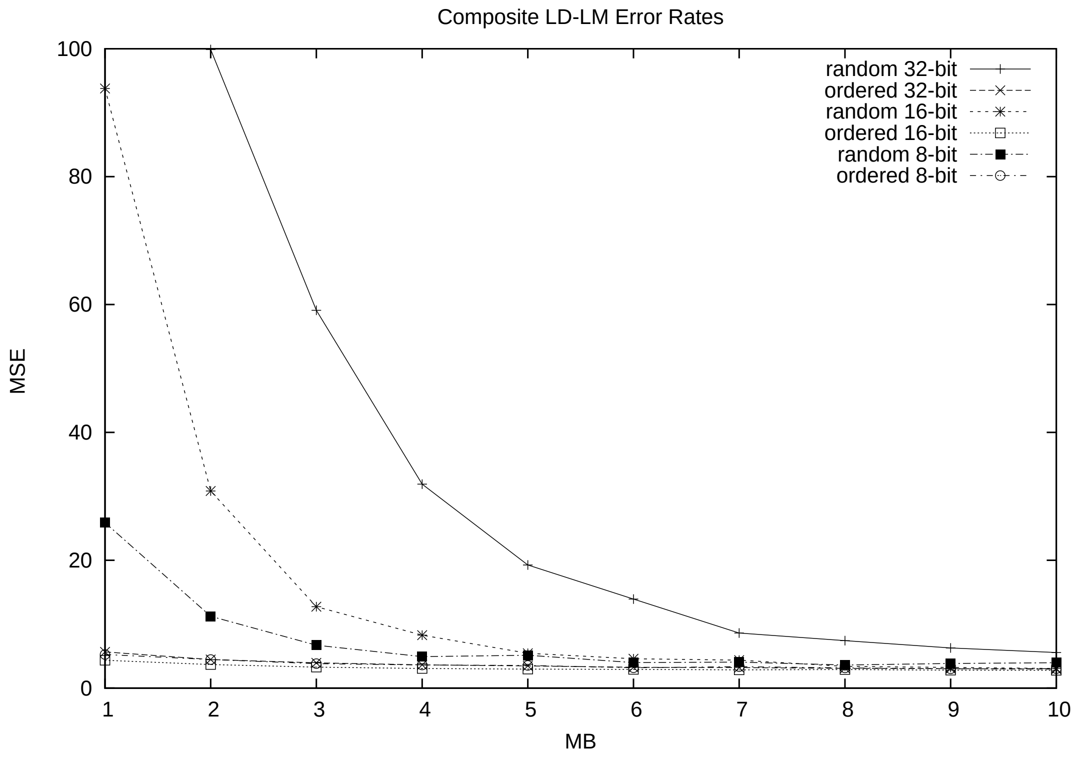

Figure 6.9: *The MSE for the unordered composite LD-LM for lower space bounds are not comparable to that of the ordered LD-LM.*

This scheme led to a lower false positive rate than the sub-bucket LD as we used more space for the signature of each event and, as we no longer discarded probability information for each event of the LM, a lower MSE too. The figure in 6.9 shows the MSE for the different space parameters and bit models of the composite LD both.

We also show the difference between the MSE for ordering to include the highest weighted subset of the data versus random insertions into the LD-LM. As the graph shows, because of the Zipf distribution the error difference for the lower space bounds is acutely in favor of the ordered LD-LM model. We take a more detailed look at the MSE for the ordered LD-LM in figure 6.10 and we compare those with the BF-LM MSE.

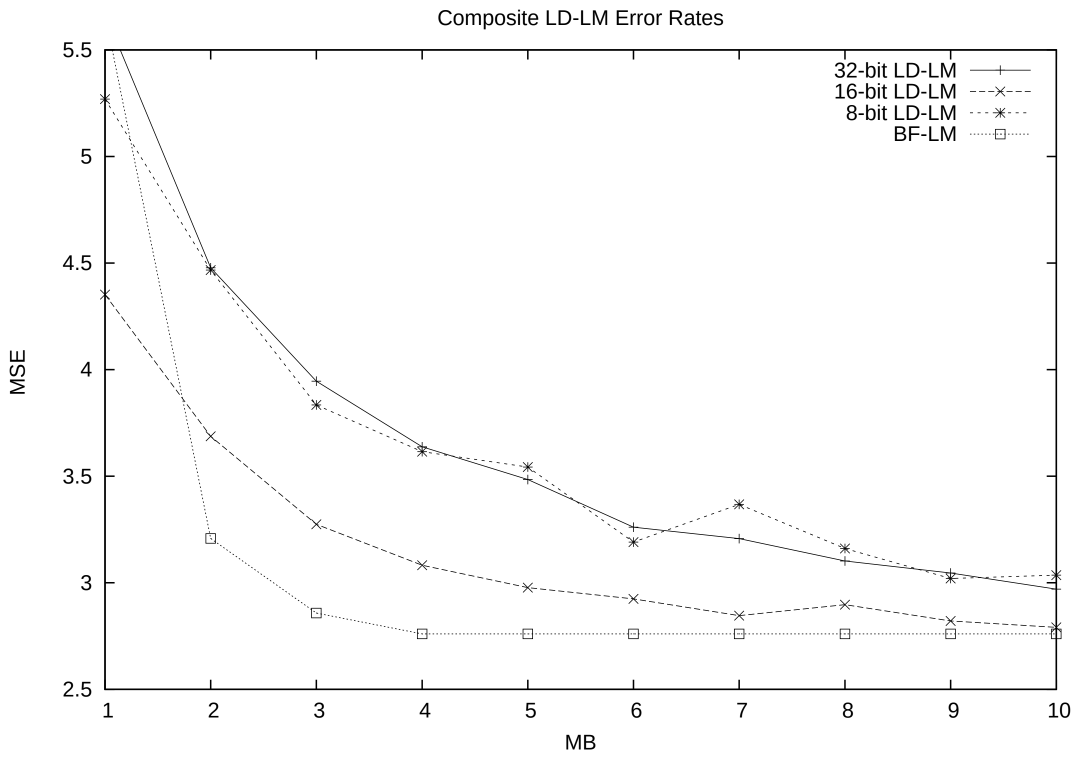

Figure 6.10: *A closer look at the perplexity based MSE rates for the ordered composite LD-LM compared to the BF-LM MSE.*

Figure 6.11 gives error rates for random data tests only and random test data and perplexity test data combined. We see that the two lossy LMs perform quite the same for these messy n-grams. It is counterintuitive, however, that the BF-LM seems to perform considerably better with less space than the higher bounds especially when the probability for false positives is taken into consideration. For example, with the training corpus size and 1 MB of space, where the BF-LM has the lowest MSE for both LMs, we have a false positive probability of over 15%. This can be explained by realising that the errors are primarily generated by the discarding information when encoding the n-gram counts in the lossy LMs. Most of the per sentence probabilities generated from the lossy LMs are substantially lower than the lossless LM's score. So, when we get a large amount of false positive hits and therefore illegitimate probability mass added to the per sentence score of the BF-LM we narrow the information gap between the gold standard and the test LMs as table 6.1 shows. This shows that while the MSE rate can give us a solid idea of the overall performance of our LM [3] we should still rely on output from systems that depend on a well formed LM for final analysis.

Lossless LM	1-MB BF-LM	4-MB BF-LM
-29.802800	-29.299274	-34.257257
-15.079100	-11.757704	-15.809171
-3.686740	-3.924542	-3.924542
-23.198100	-22.466118	-26.386337
-32.809300	-31.791770	-33.407531

Table 6.1: Example per-sentence log-prob output for different sizes of the BF compared to the lossless LM.

In general, we lowered the error rates of the LD-LM with the composite scheme considerably but haven’t quite reached the low error rate of the BF-LM.

6.5 Sub-sequence Filtered LD-LM and BF-LM with “False Negatives”

In the next experiment we implemented sub-sequence filtering in our LD-LM to test if the false positive rate was greater for the LD-LM than for the BF-LM. As before we quantized the count of each n-gram and during training we inserted composite events of an n-gram and an attached integer from 1 to the quantized count of that n-gram into the LD. Note that in this scheme we use an entire cell bucket for each item added. During testing, we hashed each test n-gram with an integer attached to it starting again from one. If that n-gram returned a hit in the LD, we incremented the integer and tested the LD again. Otherwise, if the n-gram with the attached integer returned a miss, we returned the integer attached minus one as the count in the LD. The algorithms for training and testing the sub-sequence LD-LM were much the same as the ones shown

[3]Machine translation experiments have shown the output of a BF-LM improves as the MSE is lowered (Talbot and Osborne, 2007b)

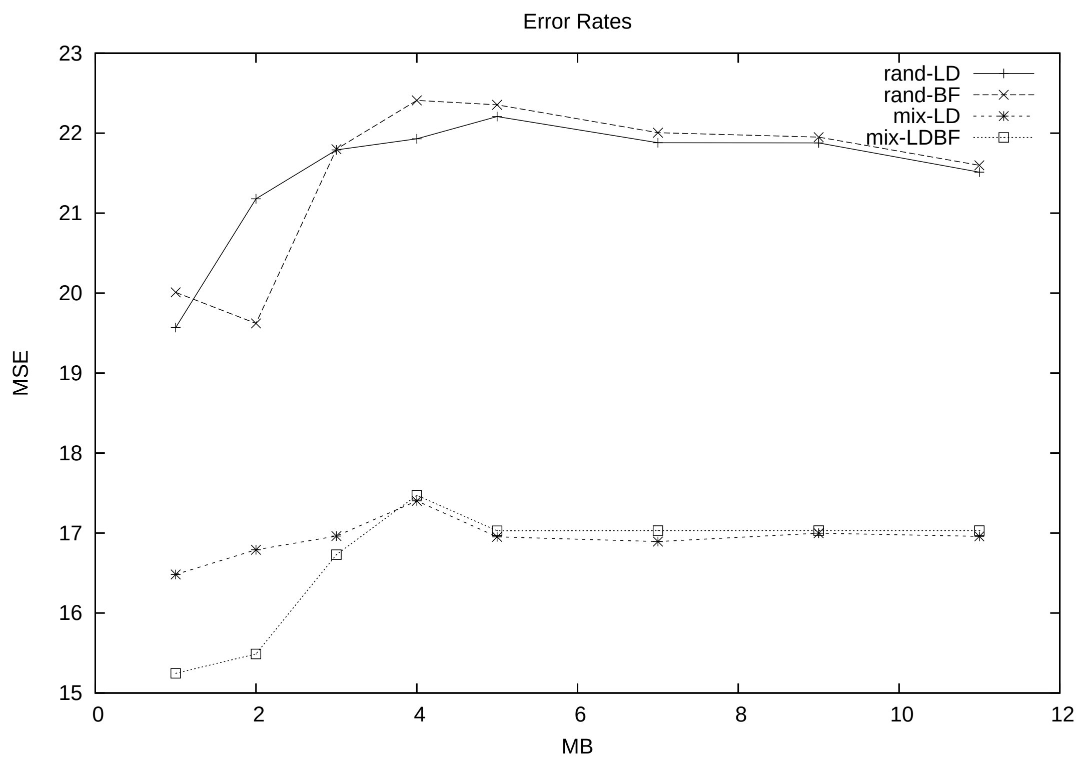

Figure 6.11: *When we test purely random data the MSE for the BF-LM and LD-LM are nearly equivalent or in favor of the LD-LM. The lower BF-LM MSE for the lowest space bounds are because of the extra probability mass from the false positives.*

in figure 4.1 on Sect. 4 except for the data structure used. The pseudocode for our implementation is listed in Appendix A.

From the higher MSE numbers in graph 6.12 we can guess that it’s not the false positives in the LD-LM that increase the error rates but the amount of false negatives or information that is discarded from the set that negatively effect it. To verify this, we trained the BF-LM with only the subset of n-gram events and their counts that were entered into the LD-LM inducing the same false negative rate in both PDSs. As expected, the graph in figure 6.13 the MSE between the two lossy LMs are basically identical.

6.6 Bloom Dictionary LM

The reason the BF can maintain its one-sided error is because it shares the bit space between events in the set it encodes. This is impossible to do with a bucket scheme such as the LD where each event gets a distinct allocation of bits only to itself. In this

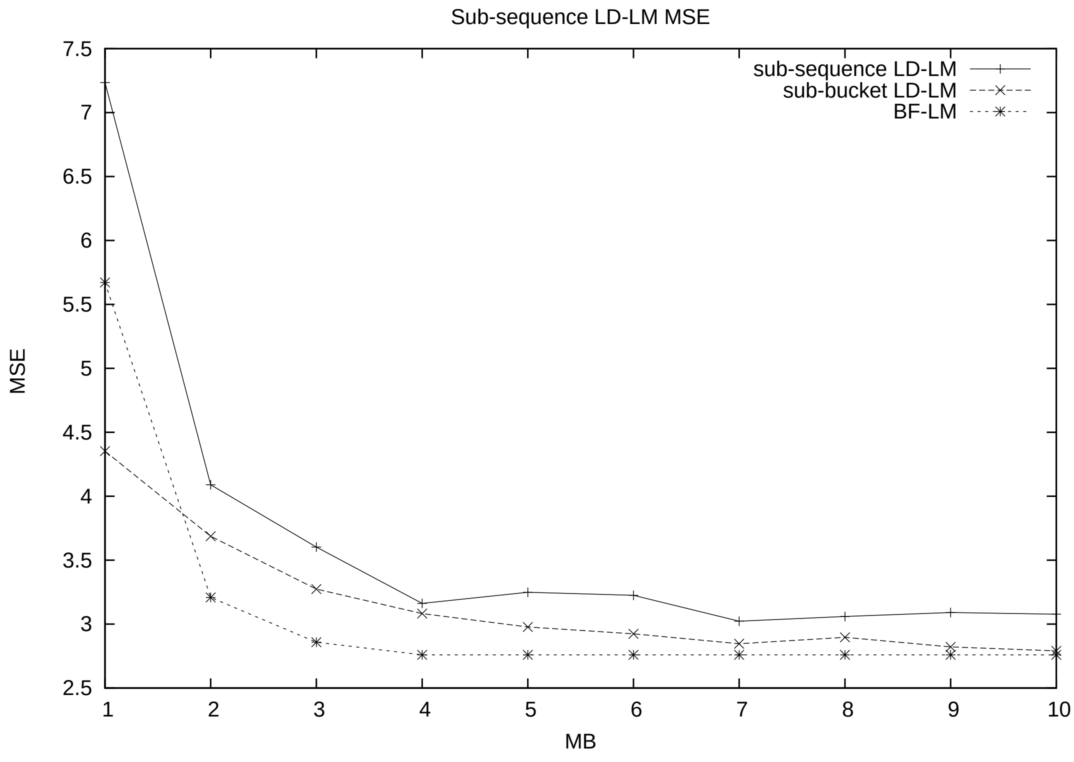

Figure 6.12: *The error rates of the LD-LM are not lowered by sub-sequencing the n-gram events meaning the higher MSE compared with the BF doesn't stem from false positives.*

experiment we tested sharing the bit space amongst the events in the tables to create a "*Bloom dictionary*" (BD). With sub-sequence filtering events added, we train the BF hashed the n-gram to get a cell index for table 1, then used the quotient function to select a single bit from that cell. Then we repeat for table 2, so each table had a bit turned on for each event. In this way we could encode the whole original set and remove false negatives from the LD-LM. This training technique is illustrated in figure 6.14.

To test a BD-LM for an element, we iterate through composite events of a n-gram and integer starting from 1. If any of the bits are not turned on, we know the element in not in the set just as with a BF. If all bits are on, we increment the counter by one and test the BD-LM again until we encounter a 0 or hit a maximum counter.

The original paper (Pagh and Rodler, 2001b) discussed using more than two tables for a LD and the author's analysis was that this would not be useful for a number of reasons. It would require another set of hashes and bit probes for starters and, for a given space parameter, the range would be smaller for the table index hash functions

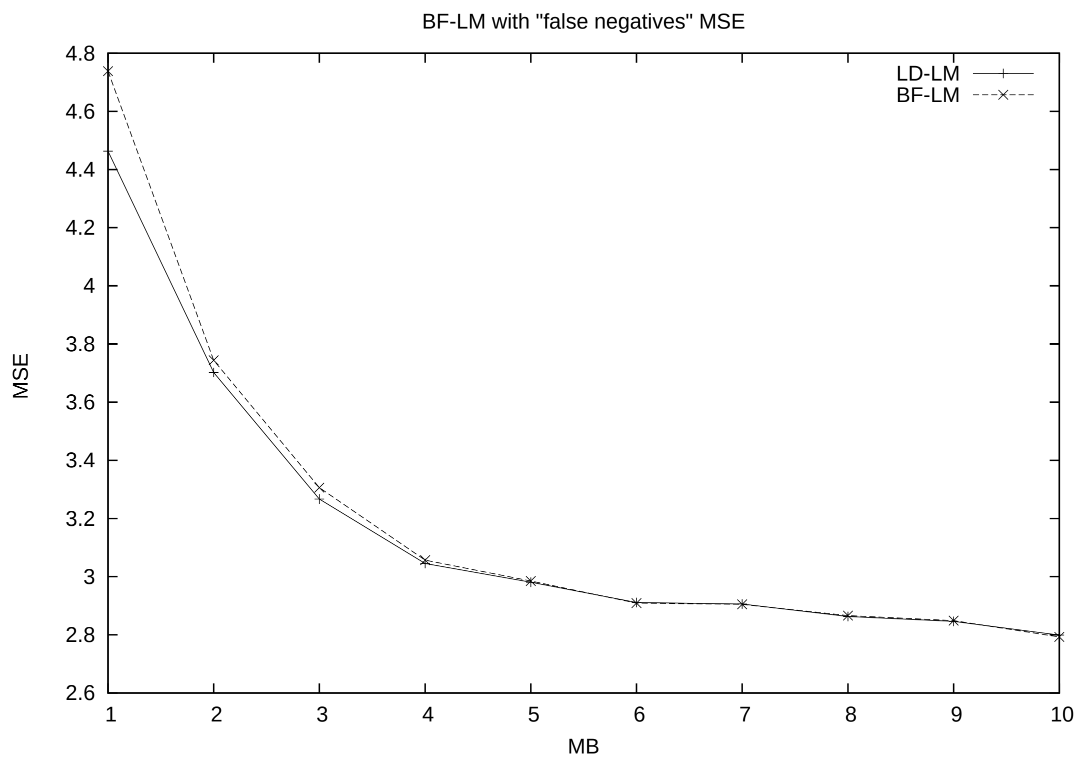

Figure 6.13: *Using the same subset of n-grams included in the LD-LM for the BF-LM (so the BF-LM imitates the false negatives of the LD-LM) results in nearly identical MSE between the two PDSs.*

so the quotient bits in each cell may need more space to reduce false positives. Also, it would increase the uncertainty of which table the data should be inserted into and retrieved from. For our purposes, however, executing another pair of hashes would still keep the LD-LM time constant which would still be a marked improvement over the BF-LM times. Further, with bit sharing it doesn't matter about the reduced index range of each table since now we use every table for each n-gram which also means there is no level of uncertainty about which table to use. So we tested using more than two tables as well with positive results as the graph in figure 6.15 shows. Also, because we're now only setting a bit a cell, we used only 8-bts per cell to increase the range of each table for the space bound set.

6.6.1 BD Properties

Encoding the whole set lowers the error to at least that of the BF-LD, so we have achieved a PDS encoded LM with constant time and low error rates. However, a closer look at the properties of the BD shows of course it's only a basic variant of a BF. From

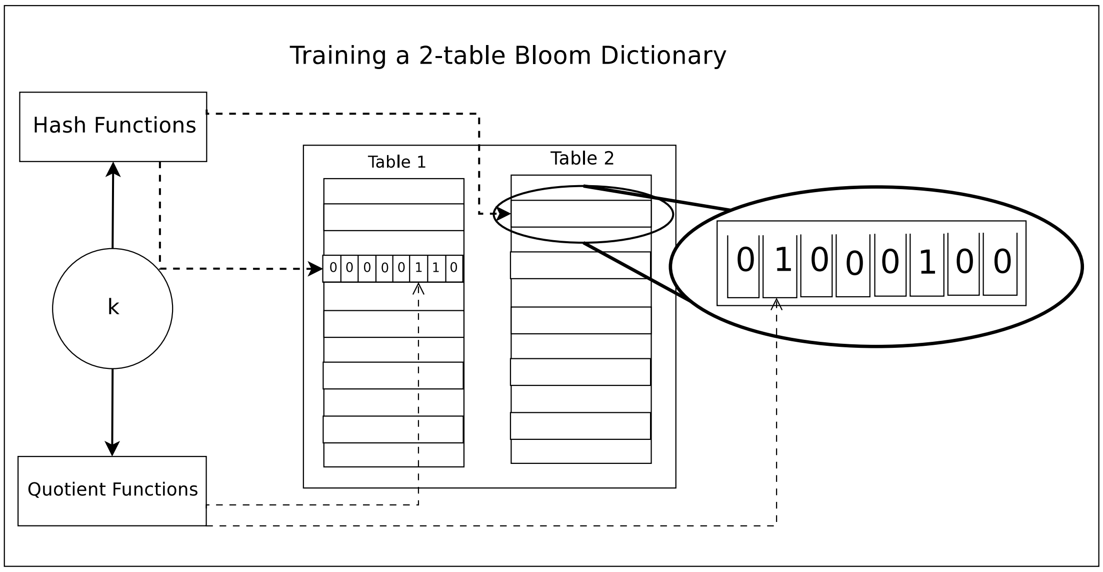

Figure 6.14: *We train a BD-LM with no false negatives using sub-sequence filtering by hashing an event to a cell index and then using the quotient functions to set a single bit in the cell.*

the LD theorem stated in Sect. 3.3 we have the total cell number over all tables as r with each cell of size is $c = b + l$. Let $m = r \times c$ be the total number of bits in our structure and an integer $t > 0$ be the number of tables in the LD. Assuming uniform distribution of our hash functions the probability of hashing event x to an index in table t is

$$\Pr(table[cell]|h_a(x)) = \frac{1}{r/t} = \frac{t}{r}$$

The probability of quotient bit in a cell being 1 is

$$\Pr(cell[bit] = 1|q_a(x)) = \frac{1}{c} \times \Pr(table[cell]|h_a(x)) = \frac{1}{c} \times \frac{t}{r} = \frac{t}{m}.$$

If we set $t = 1$ we have the same initial bit probability, one over the total number of bits in the structure, for which the BF false positive probability was derived in Sect. 3.2. With $t > 1$ the binomial probability that a randomly selected bit being set in each table is

$$\Pr(bits = 1) = \left(\frac{t}{m}\right)^t$$

and

$$\Pr(bits = 0) = \left(1 - \frac{t}{m}\right)^t.$$

Recall k is the number of times each event is hashed in a BF. If we choose the same number of hashes for both, $k = 2 \times t$, we get a lower probability for the BF by some

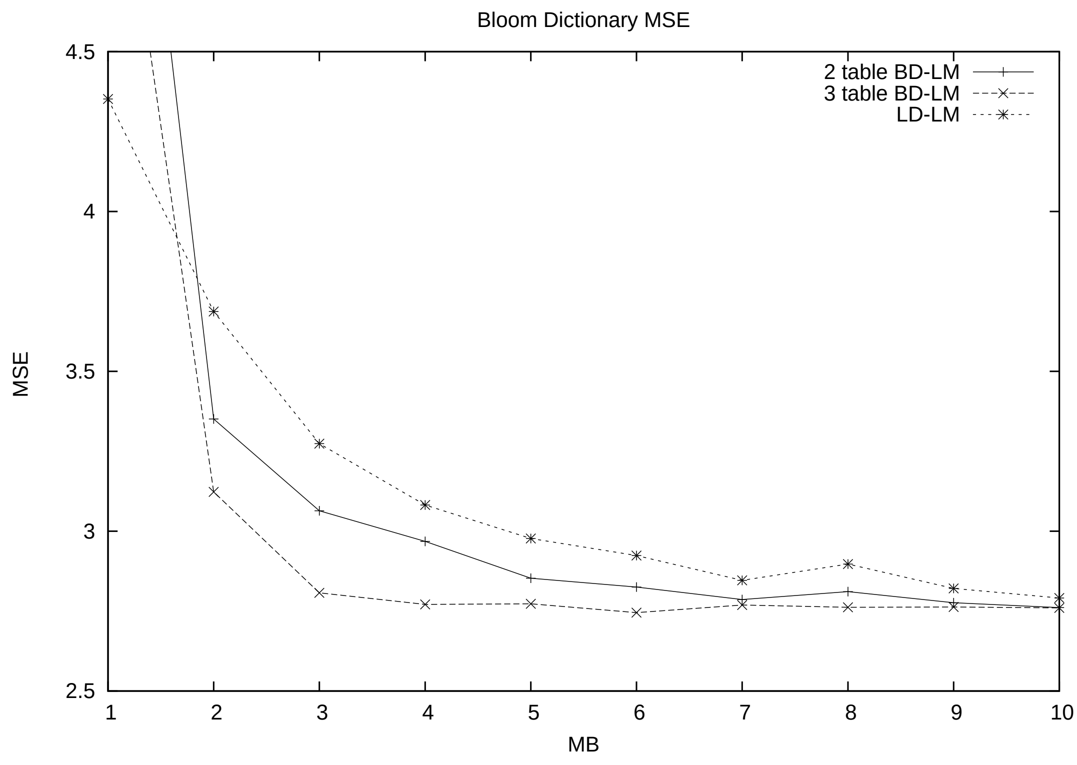

Figure 6.15: *The MSE of the BD-LM for 2 and 3 tables compared with the standard LD-LM.*

constant factor $t-1$ when $t>1$ in the BF. Hence the BD is a constant time BF with a marginally degraded false positive probability.

This above derivation is verified empirically in graph 6.16 which shows the MSE differences between the BF-LM and the BD-LM with the same constant number of hashes between them. It also shows the MSE results for the BF-LM and LD-LM for the same space.

6.7 Using More Data

We used the British National Corpus, a corpus of 110 million word vocabulary, to do some inquiries into using larger sized data sets for the lossy LMs since that's what they're designed for and also to verify results obtained on the smaller development set.

The pruned BNC n-gram set has around 30 million n-gram events and the SRILM MKN-smoothed trigram model uses 445 MB to encode the model. We were able to encode 98% of our smaller corpus using about 30% of the space with the 16-bit LD-LM, and with the larger BNC corpus we kept 97% of the data for the same space ratio.

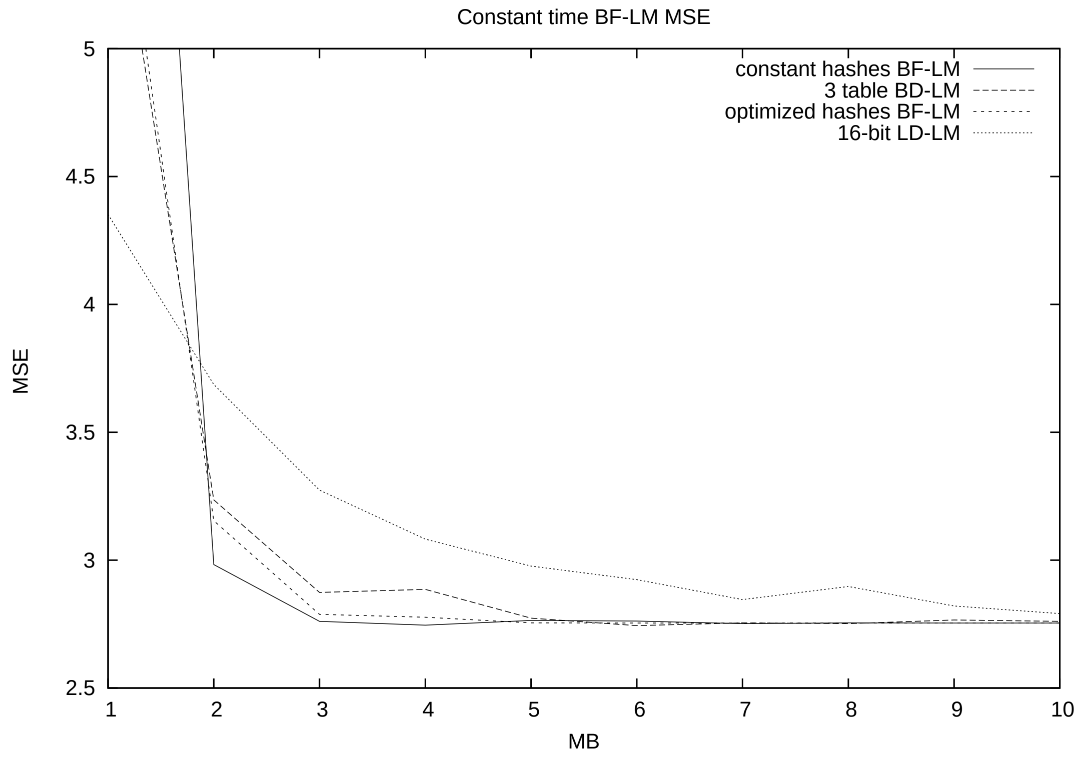

Figure 6.16: *The MSE of a constant time BF-LM converges with a regular BF-LM and is slightly better than a 3 table BD-LM.*

The Talbot and Osborne (2007b) analysis of the BF-LM showed that the upper bound of bits needed per n-gram [4] to approximate the results of the lossless LM in machine translation was 15 bits. This value held for our previous tests when the MSE was minimized when the BF-LM was between 3–4MBs or used a little less than 10% of the lossless LM space. We use this to set space bounds for both lossy LMs during the MSE tests conducted using the BNC n-grams as our training set and a test set of 5000 either randomly selected or randomly created sentences.

Since the SRILM model was pruned to ignore singleton trigram events, we setup the lossy LMs to first select the set complement of the singleton trigrams in the corpus, then enter the events into their structures. For the LD we enter the set both unordered and ordered as we did for the previously to select the largest amount of probability information possible. The results of our tests are summarised in table 6.2. And as expected, the BF-LM had the lowest error rates for a given space parameter and exceptionally slow execution times for both training and testing.

In general we've verified on the larger data set that the properties discovered for the

[4]We used the raw count of the n-grams to compute the bit-space and not the added counts derived from the sub-sequence filtering

Table 6.2: Training and Testing with the BNC Corpus

LM Type	Size	MSE	% data kept
Lossless LM	445MB	N/A	100
BF-LM	50	3.669	100
Unsorted LD-LM	50	17.66	76.30
Sorted LD-LM	50	5.25	76.80
Sorted LD-LM	133	2.73	97.6
BD-LM	50	4.416	100

lossy LMs using the smaller training set hold over scale. This is important as Google's Trillion Word N-gram set, for example, is exponentially larger than data sets we've been working with.

Chapter 7

Conclusions and Further Work

We've reimplemented the lossy BF-LM work of Talbot and Osborne (2007a) and compared its properties with the first implementation using a LD as the encoding data structure for a LM. Much of the research time was spent trying and testing various ways to best represent the n-grams in the LD space. Also, as Lossy LMs are a new area of investigation in NLP, this is the first comparison between two PDSs suited for their use.

Our findings follow directly with the properties of these data structures. With equal space bounds between them the BF-LM's one-sided error rates are lower than the LD-LM's two-sided error. Without the ability to enter the whole n-gram set we can't get rid of false negatives so we can't bridge the difference in the error rates between the BF-LM and the LD-LM. Because of the bit-sharing, the BF-LM also has a significant space advantage over the LD which requires an exclusive bit-bucket per event in then n-gram set. The LD-LM relies on including the highest valued subset of the data to keep it's error rates low. The LD-LM is many times faster then the BF-LM and, with its native support of associated information with key/value pairs, simple to setup as a smoothed LM since there is no need for sub-sequence filtering. We also found with sub-sequence filtering we can combine the constant access time of the LD and the bit sharing of the BF to create a bit-sharing LM that is fast, represents the whole set of n-grams, and returns minimal errors.

Unfortunately time constraints did not enable us to do all the experiments we would have liked for this report. As we mainly concentrated on the properties of the lossy LMs in the abstract there is still more work to do testing the LD-LM in a real-world application such as an SMT system. In general this work should not be considered complete until this is done especially as our experiments showed that the MSE doesn't tell the entire story regarding the overall goodness of a lossy LM. The only previous work of this type, reported in Talbot and Osborne (2007b), tested the BF-LM in a SMT system where translation with the lossy BF-LM matched the BLEU scores of translations that used a lossless LM but it also took between 2 and 5 times longer for each translation experiment. It would be interesting to discover for what space bounds the LD-LM can match these same BLEU scores. This would also show what ratio of data in the huge n-gram sets is actually made use of for these tasks. It would also be interesting to see how the BD-LM performs in an application environment. If the results of this research hold, we should be able to match the BLEU scores of a lossless LM without extending translation time at all and still maintain a significant space advantage overall.

Appendix A

Lossy Dictionary LM Algorithm Examples

A.1 Calculate unigram contexts for MKN Smoothing Algorithm

```
 /*
  * We define two containers for unigrams that hold each unigrams token,
  * number of unique histories/extensions, and lists of history/exten-
  * sion tokens.  Then given a n-gram or sentence, we can store the prefix
  * and suffix counts along with a list of the words to compare new words
  * against for each unigram.  Note that a similar procedure must be done
  * for each order of the LM so the vocabulary may need to be mapped to
  * a numerical representation first.
  */
#define
  unigramHist[word, cnt, list[]]
  unigramExtn[word, cnt, list[]]
⋮
function get_unigram_histories(sentence[]) {
  for each idx in sentence[] {
    cur_word = sentence[idx];
    prev_word = sentence[idx - 1];
    next_word = sentence[idx + 1];
    if(cur_word in unigramHist[]){
      if(prev_word not in unigramHist[cur_word].list[]) {
        add prev_word to unigramHist[cur_word].list[];
        /* increment count of histories for unigram */
        unigramHist[cur_word].cnt ++;
      }
    }
    else {
      add cur_word to unigramHist[];
      add prev_word to unigramHist[cur_word].list[];
      /* new history so set history counter to 1 */
      unigramHist[cur_word].cnt = 1;
    }
    if(cur_word in unigramExtn[]){
      if(next_word not in unigramExtn[cur_word].list[]) {
        add next_word to unigramExtn[cur_word].list[];
        /* increment unique extension counter */
        unigramExtn[cur_word].cnt ++;
      }
    }
    else {
      add cur_word to unigramExtn[];
      add next_word to unigramExtn[cur_word].list[];
      unigramExtn[cur_word].cnt = 1;
    }
  }
}
```

A.2 MKN Probability Retrieval Algorithm

```
 /*
  * A straight-forward recursive interpolation algorithm that follows
  * the MKN smoothing algorithm to get the probabilities.  Recursion ends
  * when we reach the unigram order and we assume use of an OOV prob.  We
  * Here we assume oracle access to the counts of all n-gram frequencies,
  * histories and suffixes.
  */
function MKN_ngram_probs(ngram[], word_idx, context_length) {
  /* if we're using OOV probabilities then replace the unknown token*/
  if( ngram[word_idx] is OOV ) {
    ngram[word_idx] = ''<unk>'';
  }
  /* build history */
  for( idx = context_length; idx <= word_idx; idx++ ) {
    tokens += ngram[idx];
    /* get backoff probs and recurse through lower order models*/
    if( idx = word_idx - 1 ) {
      backoff = get_extension_param(ngram) * MKN_ngram_prob(ngram[], context++);
    }
    else if( idx = word_idx ) {
      freq = get_count(ngram);
      ngram_length = word_idx - context_length - 1;
      prb = (freq - getDiscount_param(freq)) / allCounts[ngram_length];
      /* found prob so exit */
      context = wrd_idx + 1;
    }
  } end for
  return( prb + backoff );
}
```

A.3 Sub-bucket LD-LM training and testing algorithms

```
 /* Our sub-bucket LD-LM training and testing algorithms used a series
  * of bit-shifts and bit-masks to place and retrieve values in a bit-
  * bucket.  The << and >> are left and right bit-shift operators
  * and & is the bitwise AND operator.
  */
#define
  /* sub-bucket sizes and bit-masks */
  NGRAM_BITS   8
  PRB_BITS     4
  BKOFF_BITS   4
  QUOT_MASK    ($2^{NGRAM\_BITS}$ - 1) << (PROB_BITS + BKOFF_BITS)
  PRB_MASK     ($2^{PRB\_BITS}$ - 1) << BKOFF_BITS
  BKOFF_MASK   $2^{BKOFF\_BITS}$ - 1
⋮
function train(ngram, prb, bkoff) {
  /* create composite value of [signature + prob + backoff] */
  cell_value = get_quotient_signature(ngram) << (PRB_BITS + BKOFF_BITS);
  cell_value = cell_value + (quantize(prb, d = 1) << BKOFF_BITS);
  cell_value = cell_value + quantize(bkoff, d = 3);
  /* get table 1 index and insert if empty */
  idx = hash_1(ngram) mod table_size;
  if( table_1[idx] is empty ) {
    table_1[idx] = cell_value;
  }
  else {
  /* get table 2 index and insert if empty */
    idx = hash_2(ngram) mod table_size;
    if( table_2[idx] is empty ) {
      table_2[idx] = cell_value;
    }
    else {
      false_negatives ++;
    }
  }
}

function test(ngram) {
  found = false;
  quotient_sig = (get_quotient_signature(ngram) << (PRB_BITS + BKOFF_BITS));
  idx = hash_1(ngram) mod table_size;
  /* retrieve cell value from table 1 if quotient signature matches */
  if( table_1[idx] & QUOT_MASK == quotient_sig ) {
    found = true;
    cell_value = table_1[idx];
  }
  else {
  /* retrieve cell value from table 2 if quotient signature matches */
    idx = hash_2(ngram) mod table_size;
    if( table_2[idx] & QUOT_MASK == quotient_siq ) {
      found = true;
      cell_value = table_2[idx];
    }
  }
  if( found ) {
  /* retrieve the quantized probabilites */
    prb = (cell_value & PRB_MASK) >> BKOFF_BITS;
    prb = retrieve_prb(prb, d = 1);
    bkoff = cell_value & BKOFF_MASK;
    bkoff = retrieve_prb(bkoff, d = 3);
    return(prb, bkoff);
  }
  else {
    return(false);
  }
}
```

A.4 Key/value pair sorting algorithm.

```
 /*
  * The algorithm used to sort the weights without losing
  * key/value pair relations or storing the whole set being
  * stored in RAM.
  */
function sort_descending(document) {
  /* First get count of each n-gram weight */
  for each n-gram in document {
    prb = n-gram weight;
    map_weights[prb]++;
  }
  /* Do prefix addition over map */
  prefix = 0;
  for each item i in map_weights {
    map_weights[i] = map_weights[i] + prefix;
    prefix = map_weights[i];
  }
  /* Sort the n-grams */
  for each n-gram in document {
    prb = n-gram.weight;
    if( prb > 1 ) {
      place = map_weights[prb];
      sorted_list[place] = n-gram;
      map_weights[prb]--;
    }
    else if( prb == 1 ) {
      write_to_file(ngram);
    }
  }
}
```

A.5 Stupid Backoff Algorithm

```
 /*
  * Google’s Stupid Backoff is not interpolated and directly uses the
  * frequencies of n-gram events so it is very simple to implement and
  * doesn’t require anymore additional information than the LM already
  * gives.
  */
#define
  LM_order 3
⋮
function stupid_backoff(sentence[], word_idx) {
  /* backoff penalty parameter*/
  α = 1.0, prob = 0;
  /* get context length from LM order */
  if( word_idx > (LM_order - 1) ) {
    context = word_idx - (LM_order - 1);
  }
  else {
    context = 0;
  }
  while( context <= word_idx ) {
    /* build history */
    ngrams = ‘’;
    for( idx = context; idx <= word_idx; idx++ ) {
      ngram = sentence[idx] + ‘ ’;
      /* get backoff probs */
      if( (idx = word_idx - 1) and in_LM(ngram) ) {
        sub_freq = get_frequency(ngram);
      }
      else if( (idx = word_idx) and in_LM(ngram) ) {
        freq = get_frequency(ngram);
        /* if not in unigram prob */
        if( (sub_freq > 0) ) {
          prob = freq / sub_freq;
          /* prob found so exit */
          context = word_idx + 1;
        }
        /* else if we are in unigram prob */
        else if(context == word_idx ) {
          prob = $\alpha\times$ (freq / corpus_size);
        }
      }
      else {
        $\alpha$ = $\alpha\times$ 0.4;
      }
    } end for
    context++;
  } end while
  return( prob );
}
```

A.6 BD-LM Training and Testing Algorithm

```
 /*
  * The algorithms for training and testing the BD-LM. We use
  * sub-sequence filtering and the log-frequency scheme.  The
  * operator $|=$ is binary OR and & is binary
  * AND.
  */
function train(ngram, count) {
  freq = quantize(count);
  if( MAX < freq ) {
    MAX = freq;
  }
  for( i = 1 to freq ) {
    for( j = to to table_number ) {
      idx = hash_j(ngram + i) mod table_size;
      qBit = getQuotient(ngram + i) mod cell_size;
      /* do this for each table */
      if( j mod table_number == 0 ) {
        table_0[idx] |= qBit;
        ⋮
      }
    }
  }
}
function test(ngram) {
  found = true;
  freq = 0;
  while( found = true AND freq < MAX ) {
    i = freq + 1;
    for( j = to to table_number ) {
      idx = hash_j(ngram + i) mod table_size;
      qBit = getQuotient(ngram + i) mod cell_size;
      /* do this for each table */
      if( j $\mod$ table_number == 0 ) {
        if( (table_0[idx] & qBit) not equal qBit ) {
          found = false;
          ⋮
        }
      }
    }
  }
  if( freq $>$ 0 ) {
    freq = retrieveExpectedCount(freq);
  }
  return( freq );
}
```

Bibliography

Algoet, P. H. and Cover, T. M. (1988). A sandwich proof of the shannon-mcmillan-breiman theorem. *The Annals of Probability*, 16(2):899–909.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426.

Brants, T., Popat, A. C., Xu, P., Och, F. J., and Dean, J. (2007). Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 858–867.

Broder, A. and Mitzenmacher, M. (2002). Network applications of bloom filters: A survey. In *In Proc. of Allerton Conference*.

Brown, P. F., Pietra, V. J. D., Mercer, R. L., Pietra, S. A. D., and Lai, J. C. (1992). An estimate of an upper bound for the entropy of english. *Comput. Linguist.*, 18(1):31–40.

Carter, J. L. and Wegman, M. N. (1977). Universal classes of hash functions (extended abstract). In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 106–112, New York, NY, USA. ACM Press.

Chazelle, B., Kilian, J., Rubinfeld, R., and Tal, A. (2004). The bloomier filter: an efficient data structure for static support lookup tables. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 30–39, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Chen, S. and Goodman, J. (October 1999). An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, 13:359–393(35).

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. The MIT Press, 2nd edition.

Costa, L. H. M. K., Fdida, S., and Duarte, O. C. M. B. (2006). Incremental service deployment using the hop-by-hop multicast routing protocol. *IEEE/ACM Trans. Netw.*, 14(3):543–556.

Cover, T. and King, R. (1978). A convergent gambling estimate of the entropy of English. *IEEE Transactions of Information Theory*, IT-24:413–421.

Cover, T. and Thomas, J. (1990). *Elements of Information Theory*. John Wiley, New York.

de Boer, P.-T., Droese, D., Mannor, S., and Rubinstein, R. (February 2005). A tutorial on the cross-entropy method. *Annals of Operations Research*, 134:19–67(49).

de Laplace, M. (1996). *A Philosophical Essay on Probabilities*. Dover Publications.

Fan, L., Cao, P., Almeida, J., and Broder, A. Z. (2000). Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293.

Hagerup, T. (1998). Sorting and searching on the word ram. In *STACS '98: Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, pages 366–398, London, UK. Springer-Verlag.

Hurd, J. (2003). Formal verification of probabilistic algorithms. Technical Report 566, University of Cambridge Computer Laboratory.

James, F. (2000). Modified kneser-ney smoothing of n-gram models. Technical report, Research Institute for Advanced Computer Science.

Jurafsky, D. and Martin, J. H. (2000). *Speech and Language Processing. An Introduction to Natural Language Processsing, Computational Linguistics, and Speech Recognition.* Prentice Hall. New Jersey.

Kneser, R. and Ney, H. (1995). Improved backing-off for m-gram language modelling. In *Proceedings of the IEEE Conference on Acoustics, Speech and Signal Processing*, volume 1, pages 181–184.

Koehn, P. (2003). Europarl: A multilingual corpus for evaluation of machine translation. Available at `http://people.csail.mit.edu/˜koehn/publications/europarl.ps`.

Koehn, P. (2007). Empirical Methods in Natural Language Processing. from course slides at `http://www.inf.ed.ac.uk/teaching/courses/emnlp/`.

Koehn, P. and Hoang, H. (2007). Factored translation models. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 868–876.

Koehn, P., Och, F. J., and Marcu, D. (2003). Statistical phrase-based translation. In *NAACL '03: Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology*, pages 48–54, Morristown, NJ, USA. Association for Computational Linguistics.

Manning, C. D. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. The M.I.T. Press. Massachusetts.

Mulvey, B. (2006). Hash functions. `http://bretm.home.comcast.net/hash/`.

Norvig, P. (2007). Theorizing from data: Avoiding the capital mistake. Available at `http://www.youtube.com/watch?v=nU8DcBF-qo4`.

Och, F. (2005). The google statistical machine translation system for the 2005 nist mt evaluation, oral preesntation at the 2005 nist mt evaluation workshop.

Och, F. J. and Ney, H. (2001). Discriminative training and maximum entropy models for statistical machine translation. In *ACL '02: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 295–302, Morristown, NJ, USA. Association for Computational Linguistics.

Pagh, A., Pagh, R., and Rao, S. S. (2005). An optimal bloom filter replacement. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 823–829, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Pagh, R. and Rodler, F. F. (2001a). Cuckoo hashing. *Lecture Notes in Computer Science*, 2161:121–129.

Pagh, R. and Rodler, F. F. (2001b). Lossy dictionaries. In *ESA '01: Proceedings of the 9th Annual European Symposium on Algorithms*, pages 300–311, London, UK. Springer-Verlag.

Ravichandran, D., Pantel, P., and Hovy, E. (2005). Randomized algorithms and nlp: using locality sensitive hash function for high speed noun clustering. In *ACL '05: Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 622–629, Morristown, NJ, USA. Association for Computational Linguistics.

Shannon, C. (1951). Prediction and entropy of printed English. *Bell Sys. Tech. Journal*, 30:50–64.

Stolcke, A. (2002). Srilm – an extensible language modeling toolkit. In *Proc. Intl. Conf. on Spoken Language Processing, 2002.*

Talbot, D. and Osborne, M. (2007a). Randomised language modelling for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 512–519, Prague, Czech Republic. Association for Computational Linguistics.

Talbot, D. and Osborne, M. (2007b). Smoothed Bloom filter language models: Tera-scale LMs on the cheap. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 468–476.