Stream-based Statistical Machine Translation

Abby D. Levenberg

Doctor of Philosophy

School of Informatics

University of Edinburgh

2011

Abstract

We investigate a new approach for SMT system training within the *streaming* model of computation. We develop and test incrementally retrainable models which, given an incoming stream of new data, can efficiently incorporate the stream data online. A naive approach using a stream would use an unbounded amount of space. Instead, our online SMT system can incorporate information from unbounded incoming streams and maintain constant space and time. Crucially, we are able to match (or even exceed) translation performance of comparable systems which are batch retrained and use unbounded space. Our approach is particularly suited for situations when there is arbitrarily large amounts of new training material and we wish to incorporate it efficiently and in small space.

The novel contributions of this thesis are:

1. An online, randomised language model that can model unbounded input streams in constant space and time.

2. An incrementally retrainable translation model for both phrase-based and grammar-based systems. The model presented is efficient enough to incorporate novel parallel text at the single sentence level.

3. Strategies for updating our stream-based language model and translation model which demonstrate how such components can be successfully used in a streaming translation setting. This operates both within a single streaming environment and also in the novel situation of having to translate multiple streams.

4. Demonstration that recent data from the stream is beneficial to translation performance.

Our stream-based SMT system is efficient for tackling massive volumes of new training data and offers-up new ways of thinking about translating web data and dealing with other natural language streams.

Acknowledgements

First I'd like to thank my family for their belief in me.

A big thanks to the Statistical Machine Translation group at the School of Informatics, University of Edinburgh for their support. Special thanks to Philipp Koehn, Phil Blunsom, Trevor Cohn, Hieu Huang, Abhishek Arun, Alexandra Birch, David Talbot, Barry Haddow, and David Matthews for their advice and assistance.

Most of all I thank my advisor, Miles Osborne, for his fantastic direction and encouragement throughout. Without his valuable advice this thesis would have been significantly more difficult. As he showed me during my time under his tutelage, "It's all in the game."

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

List of Tables

List of Algorithms

Chapter 1

Introduction

Statistical Machine Translation (SMT) is driven by unsupervised learning from unlabelled data. Currently there is already a lot of data freely available for training SMT systems and since, as the saying goes, “there’s no data like more data”, there is a constantly increasing amount of it being made available by the SMT community. In many translation scenarios, new training data is available regularly and needs to be incorporated into an existing translation system. The standard algorithms for training the models of a SMT system, however, are slow, require a lot of memory, and not amenable to quickly incorporating new data into an existing system. This thesis proposes new training algorithms that have the ability to learn quickly from unbounded amounts of novel data while operating within bounded memory.

1.1 The Problem

Current training algorithms for SMT models are notoriously slow in practice and use memory that grows with the amount of training data. Training typically takes days of CPU processing time for a single language pair using standard data sets provided for SMT research competitions. A key feature of these SMT systems is their batch nature; once the model has been trained no new data can be added to it without fully retraining the model from scratch. While a plethora of research has been published developing new state-of-the-art models, there has been little research in optimizing existing techniques to allow for efficient (re)training.

This is surprising consider the amount of training data already being used and which is constantly increasing. We have entered what has been coined *The Petabye*

Age as the pool of data available for research is growing at exponential rates. [1] [2] In the natural language domain tens of thousands of websites continuously publish news stories in more than 40 languages, every day many millions of multilingual blog postings are posted, and there are over 30 billion e-mails sent daily and social networking sites, including services such as Twitter, generate staggering amounts of textual data in real time. All of this data provides the SMT community with a potentially extremely useful resource to learn from but it also brings with it nontrivial computational challenges of scalability and information extraction. The proliferation of data available means that while there is a lot more of it to use, not all of it may be suitable for what is being translated at the moment.

Besides all the data generated by the Web there are many situations where novel domain-specific training data is continually becoming available. For any organisation that is translating documents from some domain, we can view the incoming document collection as a *data stream* of source text that is chronologically ordered and implicitly timestamped. Each distinct domain comprises its own stream. As the recent part of the source stream is translated it could easily (from a performance perspective) become training data to aid in the translation for the next bit of the stream. This model of SMT use is widespread but due to their batch nature the learning algorithms for SMT are ill-suited for handling this type of streaming translation system effectively. Instead of being able to add just the new data to the SMT system the full system must be entirely retrained using all the old and new data combined.

In this thesis we address this problem by presenting novel streaming algorithms for efficient retraining of SMT systems. These algorithms have the ability to add new data to previously trained models while using space independent of the stream size. We list the specific contributions of the thesis below.

1.2 Thesis Contributions

The ideas presented in this work follow from the theory of data stream algorithms in the literature (Muthukrishnan, 2003). In the streaming translation scenario a given statistical model in the SMT pipeline is incrementally retrained by updating it arbitrarily often with previously unseen training data from a known source. Since the source pro-

[1] Wired Magazine, June 2008, Issue 16.07. At `http://www.wired.com/science/discoveries/magazine/16-07/pb_intro`

[2] The Economist, February 2010, Volume 394 Number 8671. At `http://www.economist.com/node/15557443`.

vides new data continuously (if at intervals) it is intuitively referred to as a *stream*. A stream can aggregate smaller streams within it and any one stream may bring in data at a very high rate. A given stream may be *unbounded* and may continue to provide data indefinitely.

These properties of streams, high rate of throughput and unboundedness, provide the computational challenges tackled in this thesis. An online SMT system must be able to produce translations quickly so retraining algorithms must be efficient. Of more concern is building models whose space complexity is independent of the size of the incoming stream. Using unbounded memory to handle unbounded streams is unsatisfactory. In this work we tackle each of the major SMT models individually and show how they can adapt quickly to an incoming stream within bounded space.

The major contributions of this thesis are as follows:

- We introduce a novel randomised language model (LM) which has the ability to adapt to an unbounded input stream in constant space and time whilst maintaining a constant error probability. We analyze the error rate and runtime of the new LM. Our experiments using the stream-based LM in a full SMT setup show that not only can we update the model with new data efficiently but that using recent *n*-grams from an in-domain stream improves performance when translating test points from the stream. This work was published in the proceedings of EMNLP 2009 (Levenberg and Osborne, 2009).

- We show how to model multiple incoming streams when they are drawn from variable domains and their throughput differs greatly. To do this we use simple adaptation heuristics using the decoding history of prior test points to combine the various streams into a single model in small space. This is the first online randomised LM that can use unbounded input. Our associated adaptation schemes are also novel. This work was previously reported in Levenberg et al. (2011).

- We present an incrementally retrainable translation model (TM) that has the ability to very quickly incorporate new parallel sentences. We describe application of an online EM algorithm for word alignments and show how this can work in conjunction with dynamic suffix arrays to produce an online TM for either phrase-based or grammar-based translation. The algorithm is efficient enough to allow incorporating new data at the single sentence level. This work was presented at NAACL 2010 (Levenberg et al., 2010).

- We put the stream-based LM and TM together to show how gains achieved by each are additive. We describe a completely online SMT system that adapts efficiently and in constant space to unbounded amounts of incoming data. This not only provides the ability to add new data to existing models without requiring the expense of full batch retraining but also improves translations for test documents which are drawn from a stream in chronological order which is the most common use case of SMT. This work was also presented in Levenberg et al. (2010).

- Demonstration that recency effects inherent in the stream improve translation quality for that domain.

1.3 Outline

We first review background material and preliminary experiments in Chapter 2 including an overview of SMT and its pipeline of models. We also review randomised LMs, introduce data stream theory and report on initial natural language streaming experiments. In Chapter 3 we present the novel stream-based LM with associated single stream experiments. We extend these experiments further in Chapter 4 by describing how we can adapt the stream-based LM to multiple incoming streams regardless of domain and stream size.

In Chapter 5 we introduce the incremental, stream-based TM with the online EM algorithm and describe the dynamic suffix arrays. We also report on using the full online setup with both the online LM and TM together in Chapter 5. Finally we conclude and offer directions for future work.

Chapter 2

Background

The work in this thesis draws on various fields in computer science including natural language processing, randomised algorithms, and the theory of data streams. In this chapter we review the background material necessary for understanding and relevant to the thesis contributions. In particular we review the fields of SMT, randomised LMs, and data streams. As well we discuss preliminary background experiments that motivate the streaming setting for SMT.

2.1 Statistical Machine Translation

In this section we present a high level overview of SMT and the various models that are touched upon in this thesis. This includes short reviews of LMs and TMs. An in-depth review of SMT is beyond the scope of this thesis and there are a number of comprehensive examinations of the field in the literature. See, for example, Lopez (2008a) and Koehn (2010) for a complete treatment.

2.2 Overview

In SMT natural language translation between two language pairs is treated as a data driven machine learning problem. Instead of using linguistically motivated rules of language production to translate between a source (foreign) sentence **f** to a target (English) sentence **e**, SMT uses statistical rules, learnt in an unsupervised manner from unlabelled parallel corpora, to find the target sentence translation **e** that has the highest model probability given the source sentence **f**. Learning word translations is cast as the problem of finding the hidden alignments between source and target sentences. Since

word ordering differs greatly between languages, the words in the source sentence may need to be *reordered* in the target language sentence to create an understandable translation.

SMT has been the dominant methodology in machine translation research since the seminal work of Brown et al. (1993) introduced the so-called 'IBM Models'. The IBM models are *word-based* models of translation. That is, each word in the source sentence is aligned to one (IBM Model 1) or more (IBM Model 3) target words. Reordering is handled based on absolute sentence position (IBM Model 2) or relatively based on previous word translations in the source sentence (IBM Models 4 and 5). The *HMM-based* alignment model is another word-based model where a translation is based only on the previous word's translation (Vogel et al., 1996).

Phrase-based models were the next big step in SMT research and, instead of restricting translations to single source words, moved to using consecutive sequences of words as the primary translation unit (Koehn et al., 2003). Using phrases as the basic unit of translation improves translation performance since local reorderings are now implicitly accounted for. As well, phrase translation handles one-to-many word mappings, idioms and expressions, word insertions and deletions, and other language specific nuances that can be learnt from bitext phrases but may break down if translated at the single word level. *Grammar* and *syntax-based* models (see Lopez (2008a)) for SMT were developed as a mechanism to better handle long distance reordering. In these models a translation equivalence is drawn not only between text on the source and target side but also the grammatical structure that each sentence pair is comprised of. These models use synchronous context free grammars (SCFGs), grammars that produce two output strings that represent terminals and nonterminals on both source and target sides, of varying complexity that further improve translation reordering.

Model parametrization was initially formulated using the noisy-channel model (Brown et al., 1990). Denoting the latent alignments as $\mathbf{a}$ and using Bayes' decomposition we have

$$\Pr(\mathbf{e}, \mathbf{a}|\mathbf{f}) := p(\mathbf{f}, \mathbf{a}|\mathbf{e})p(\mathbf{e}) \tag{2.1}$$

and we can ignore the denominator $p(\mathbf{f})$ since the source sentence stays constant for all choices of the target $p(\mathbf{e})$. Here, $p(\mathbf{f}, \mathbf{a}|\mathbf{e})$ are the translation/alignment choices learnt for the TM and $p(\mathbf{e})$ is a measure of fluency on the target output according to a LM. This basic noisy-channel model was extended into a more flexible linear model that allows for arbitrary feature functions that provide additional information to the

translation system (Och and Ney, 2001). We can write

$$\Pr(\mathbf{e},\mathbf{a}|\mathbf{f}) = \frac{\exp(\sum_{m=1}^{M}\lambda_m h_m(\mathbf{e},\mathbf{a},\mathbf{f}))}{\mathbf{Z}(f)} \tag{2.2}$$

where there are a total of M feature functions $h_m(\mathbf{e},\mathbf{a},\mathbf{f})$ each contributing to a translation's score by a weighting of λ_m with $\mathbf{Z}(\mathbf{f})$ the normalization term. In this setting the TM $p(\mathbf{f},\mathbf{a}|\mathbf{e})$ and the LM $p(\mathbf{e})$ are feature functions $h_m(\mathbf{e},\mathbf{a},\mathbf{f})$ that can be interpolated with other dependencies such as the distortion or reordering, generation, lexicalised translation, and part of speech and factored models.

In this thesis we deal primarily with the two main feature functions of any SMT system: the TM and the LM. Below we give brief reviews of both.

2.3 Translation Model

As described above, the TM is the parametrisation of $p(\mathbf{f},\mathbf{a}|\mathbf{e})$ which specifies the translation probabilities between words or phrases in a training corpus. Here we limit our descriptions to the specific algorithms and TM models used in this thesis.

2.3.1 Batch EM Algorithm

Phrases and grammar rules are traditionally extracted using heuristics over the learnt word alignments between the source and target text. Traditionally the batch Expectation-Maximisation (EM) algorithm is used to learn the latent alignment variable vector **a**. We first review the batch EM algorithm and then explain how it is applied to inducing the word alignments for SMT.

The general EM algorithm is a common way of inducing latent structure from unlabeled data in an unsupervised manner (Dempster et al., 1977). Given a set of unlabeled examples and an initial, often uniform guess at a probability distribution over the latent variables, the EM algorithm maximizes the marginal log-likelihood of the examples by repeatedly computing the expectation of the conditional probability of the latent data with respect to the current distribution, and then maximizing these expectations over the observations into a new distribution used in the next iteration.

Computing an expectation for the conditional probabilities requires collecting the *sufficient statistics* **S** over the set of *n* unlabeled examples. In the case of a multinomial distribution, **S** is comprised of the counts over each conditional observation occurring in the *n* examples. In traditional *batch* EM, we collect the counts over the entire dataset

Algorithm 1: Batch EM for Word Alignments. The full parallel corpus is iterated over T times and the sufficient statistics S are cleared after each iteration.

Input: $\{(f,e)\}$ set of (source,target) sentence-pairs

Output: MLE $\hat{\theta}_T$ over alignments **a**

$\hat{\theta}_0 \leftarrow$MLE initialization;

for *iteration* $t = 0,\ldots,T$ **do**

  $S \leftarrow 0$; `// reset counts`

  foreach $(f,e) \in \{(f,e)\}$ **do** `// E-step`

    $S \leftarrow S + \sum_{a' \in \mathbf{a}} \Pr(f,a'|e;\hat{\theta}_t)$;

  end

  $\hat{\theta}_{t+1} \leftarrow \bar{\theta}_t(S)$; `// M-step`

end

of n unlabeled training examples via the current 'best-guess' probability model $\hat{\theta}_t$ at iteration t (E-step) before normalizing the counts into probabilities $\bar{\theta}(\mathbf{S})$ (M-step). As the M-step can be computed in closed form we designate it in this work as $\bar{\theta}(\mathbf{S})$. After each iteration all the counts in the sufficient statistics vector **S** are cleared and the count collection begins anew using the new distribution $\hat{\theta}_{t+1}$.

2.3.2 Word-based TM

Batch EM is used in word-based SMT systems to estimate word alignment probabilities between parallel sentences. From these word alignments more complex bilingual rules such as phrase pairs or grammar rules can be extracted. Given a set of parallel sentence pairs, $\{(\mathbf{f},\mathbf{e})_s\}_{s\in\{1,\ldots,n\}}$, where n is the total number of corresponding sentence pairs $(\mathbf{f},\mathbf{e})_s$ with **f** the source sentence and **e** the target sentence, we want to find the latent alignments **a** for a sentence pair $(\mathbf{f},\mathbf{e})_s$ that defines the most probable correspondence between words f_j and e_i such that $a_j = i$ in sentence s. (We omit the set subscript $(\mathbf{f},\mathbf{e})_s$ when it is clear from the context sentences **f** and **e** correspond.) We can induce these alignments using an *HMM-based* alignment model where the probability of a word alignment a_j is dependent only on the previous alignment at a_{j-1} (Vogel et al., 1996). We can write

$$\Pr(\mathbf{f},\mathbf{a} \mid \mathbf{e}) = \sum_{a' \in \mathbf{a}} \prod_{j=1}^{|\mathbf{f}|} p(a_j \mid a_{j-1},|\mathbf{e}|) \cdot p(f_j \mid e_{a_j}) \tag{2.3}$$

where we assume a first-order dependence on previously aligned positions.

To find the most likely parameter weights for the translation and alignment probabilities for the HMM-based alignments, we employ the EM algorithm via dynamic programming. Since HMMs have multiple local minima, we seed the HMM-based model probabilities with a better than random guess using IBM Model 1 (Brown et al., 1993) as is standard. IBM Model 1 is of the same form as the HMM-based model except it uses a uniform distribution instead of a first-order dependency. With $J = |\mathbf{f}|$ the length of the source sentence and the length of the target $I = |\mathbf{e}|$, IBM Model1 is

$$\Pr(\mathbf{f}, \mathbf{a}|\mathbf{e}) = \frac{p(J \mid I)}{(I+1)^J} \cdot \prod_{j=1}^{J} p(f_j|e_{a_j}) \tag{2.4}$$

and we have zero-dependency on word order. Although a series of more complex models are defined, IBM Models 2 to Model 6 (Brown et al., 1993; Och and Ney, 2003), researchers typically find that extracting phrase pairs or translation grammar rules using Model 1 and the HMM-based alignments results in equivalently high translation quality. In this thesis we only use the IBM Model 1 and the HMM-based alignment models. Nevertheless, there is nothing in our approach which limits us to using just Model 1 and the HMM model.

A high-level overview of the standard, batch EM algorithm applied to HMM-based word alignment model is shown in Algorithm 1.

2.3.3 Phrase-based TM

For the reasons mentioned in the overview above, phrase-based models for SMT translate using sequences of one or more words at a time. They produce better translation performance generally than the IBM word-based models and are used widely in research and industry.

In phrase-based SMT the word ‘phrase’ has no specific linguistic sense and it is left up to the learning algorithms to discern what constitutes a phrase. From Lopez (2008a), the translation process of phrase-based SMT takes the following steps:

1. The source sentence is split into phrases of various length (from length one to the maximum phrase length allowed by the model).

2. Each source phrase is translated separately into a set of candidate target phrases.

3. The target phrases are permuted into their final ordering to form a set of potential target sentences.

4. The target sentence with the highest model score is output as the single-best translation.

The phrases are (usually) extracted using various sets of heuristics that require the phrases to be consistent with the previously acquired word alignments between sentence pairs learnt using word-based models. Phrase probability is then estimated based on normalised relative frequency.

2.3.4 Hierarchical Phrase-based TM

An extended variant of the phrase model that allows for longer distance reordering is the hierarchical phrase model that incorporates both phrase-based translation along with a simplified SCFG grammar (Chiang, 2007). As with the phrases from the phrase-based model, the grammar is extracted via heuristics from word aligned sentence pairs and does not make use of linguistically motivated syntactic rules. The grammar specifies only a single nonterminal symbol for all productions. Further, the right-hand side of each rule is restricted to a small number of nonterminal variables. Each rule is a tuple of source to target translations with a mix of terminals and nonterminals. This gives the model its hierarchical nature since the nonterminal symbols act as gaps in the phrases that, when encountered by the decoder, are translated recursively. The ordering of nonterminals between the source and target productions can be exchanged so arbitrary long distance reordering is possible during translation.

For sizable corpora the number of phrase pairs or grammar rules extracted quickly becomes unwieldy in size. Below we discuss one of the methods developed for dealing with this problem which we extend later in this thesis.

2.4 Suffix Array Phrase Tables

Extracted phrases and their probabilities are stored in a *phrase table* data structure to be queried during test time by the decoder. For phrases extracted from moderately sized training corpora the size of the resulting phrase table is often too big to fit in any computer's memory. Callison-Burch et al. (2005) and Lopez (2008b) show how to bypass this problem and extract phrases or hierarchical syntax rules over large corpora for SMT during decoding by directly storing the bitext corpora along with the word alignments in memory via suffix arrays. *Suffix arrays* (Manber and Myers, 1990) are space-efficient data structures for fast searching over large text strings. Treating the

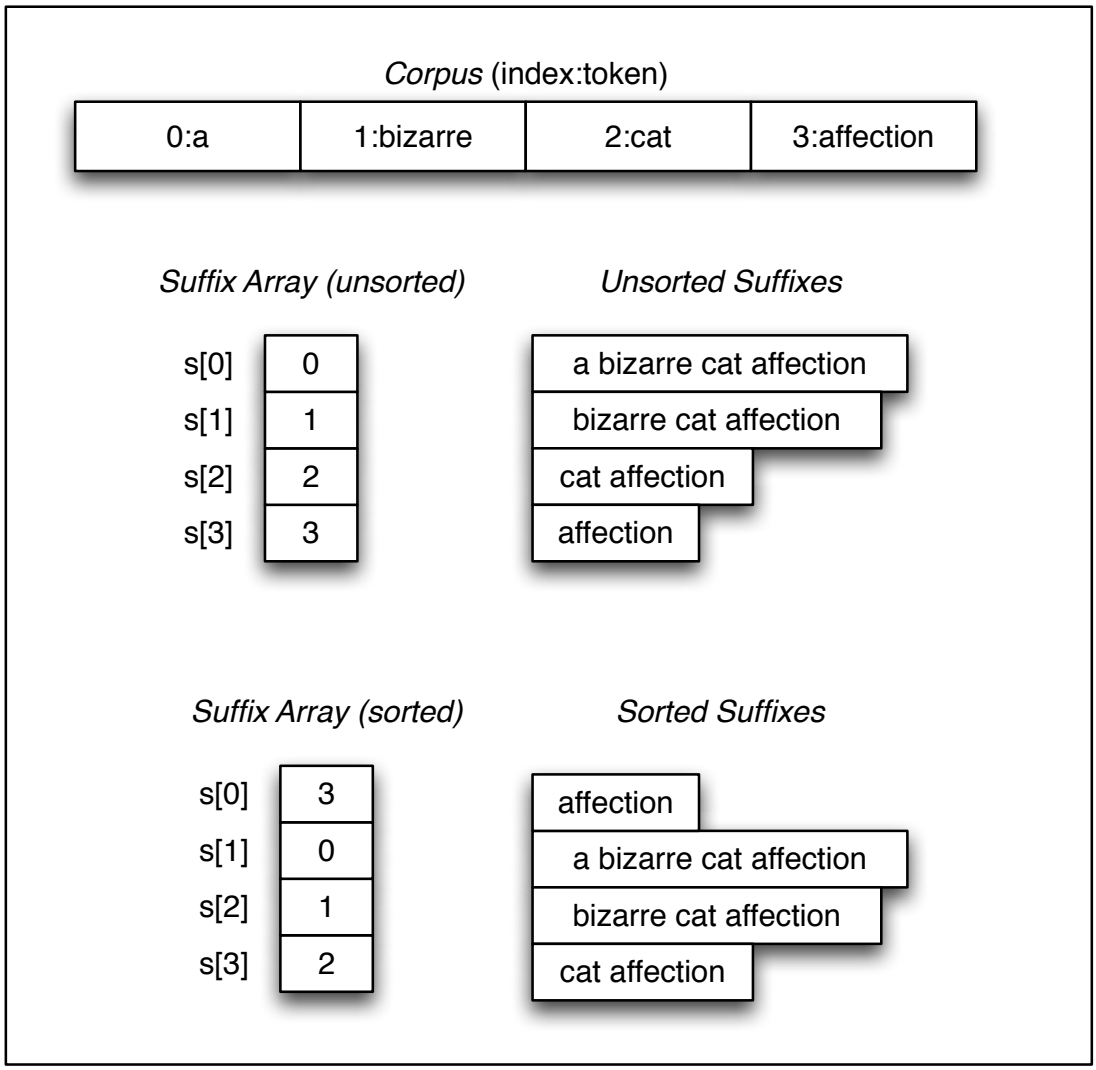


Figure 2.1: A toy example of a suffix array. The indexes of the lexically ordered corpora are stored in the final suffix array. Suffix arrays can be used to encode phrase tables for SMT.

entire corpus as a single string, a suffix array holds in lexicographical order (only) the starting index of each suffix of the string. After construction, since the corpus is now ordered, we can query the sorted index quickly using binary search to efficiently find all occurrences of a particular token or sequence of tokens. Then we can easily compute the statistics required such as translation probabilities for a given source phrase. When a phrase or rule is needed by the decoder, the corpora is first searched, all counts are accumulated, and the rule probability is computed on the fly. Suffix arrays can also be compressed, which make them highly attractive structures for representing potentially massive phrase tables.

Later in the thesis (Section 5) we show how we can incrementally add new phrases

to the TM by word aligning parallel sentences online and using a dynamic version of the suffix array phrase tables to encode the phrase table.

2.5 Language Model

An *n*-gram LM is a statistical estimator that, given a sequence of words, returns a measure of how probable those words with that ordering are based on the training data. The true probability of a word in a sentence should be conditioned on all prior words. Given the words $w_1, \ldots, w_i, \ldots, w_N$, which we also write as w_1^N, then its probability is

$$\Pr(w_1^N) = \prod_{i=1}^{N} p(w_i | w_1^{i-1}). \tag{2.5}$$

However, to handle sparsity, *n*-gram LMs use the Markov assumption that any word w_i is conditioned only on some short history of $n-1$ words where n is called the *order* of the *n*-gram. The above equation becomes

$$\Pr(w_1^N) = \prod_{i=1}^{N} p(w_i | w_{i-n+1}^{i-1}) \tag{2.6}$$

and the full sequence of words is broken up into a product of shorter sequences. An order of three to five is most commonly used in practice in SMT.

To parametrize the LM we could use maximum likelihood estimation (MLE) and score each *n*-gram based on the relative frequency of its occurrence in the training corpus. With $c(.)$ a function that returns the frequency of an *n*-gram, the probability for an *n*-gram (of order > 1) is

$$p(w_i | w_{i-n+1}^{i-1}) = \frac{c(w_{i-n+1}^{i})}{c(w_{i-n+1}^{i-1})} \tag{2.7}$$

which is the relative frequency of how often a word appears after a specific history of $n-1$ words.

However, the power-law distribution of natural language means only a tiny percentage of grammatical *n*-grams in any language will appear in a given training set. Using naive MLE in practice would assign many valid but unobserved *n*-grams a zero probability which is undesirable. To allay this sparsity problem many sophisticated *smoothing* algorithms have been described in the LM literature that ensure some of the distribution mass of the training data is reserved for unseen events. Key to LM smoothing are the concepts of *backoff* and *interpolation*. In backoff smoothing we make use

of the less sparse lower-order *n*-gram information to help smooth the probabilities of higher-order *n*-grams which may be unreliable. Formally we can write

$$p(w_i|w_{i-n+1}^{i-1}) = \begin{cases} \delta p(w_i|w_{i-n+1}^{i-1}) & \text{if } c(w_{i-n+1}^{i}) > 0 \\ \alpha(w_{i-n+1}^{i-1}) p(w_i|w_{i-n+2}^{i-1}) & \text{otherwise} \end{cases} \tag{2.8}$$

where δ is a discounting factor often based on history frequency and $\alpha(.)$ is a back-off penalty parameter that ensures normalisation. Similarly, interpolation linearly combines the statistics of the *n*-gram through all the orders of the LM. The formula is defined recursively as

$$p(w_i|w_{i-n+1}^{i-1}) = \lambda p(w_i|w_{i-n+1}^{i-1}) + (1-\lambda) p(w_i|w_{i-n+2}^{i-1}) \tag{2.9}$$

and the final score for an *n*-gram is the interpolated probability of all higher and lower-order grams in the LM. Below we describe two smoothing algorithms used in this thesis. For a complete overview of LM smoothing see Chen and Goodman (1999).

2.5.0.1 Modified 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 Pr(*Francisco*) maybe should be 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

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

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}.

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 formula 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

$$Y = \frac{n_1}{n_1 + 2n_2}$$

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

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).

2.5.0.2 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 recommended to be 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 $\Pr(.)$ to indicate that the method returns a relative score instead of a normalized probability.

2.6 Randomised Language Models

2.6.1 Overview

All else being equal, increasing the amount of in-domain training data often improves performance of NLP tasks since sparsity is reduced and greater coverage of the target

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 Corpus Statistics. The compressed data set is 24GB on disk.

domain is achieved. This is especially true for LMs for SMT. It is well known that increasing the amount of LM training data can improve the quality of machine translation (Och, 2005). With large corpora being released and the general availability of huge amounts of textual data, training a LM with a large amount of data is a necessary requirement for state-of-the-art SMT systems. This, in turn, strains or exhausts computational resources as processing these large data sets is time and memory intensive. A number of data sets are already available that are far too large for any average computer RAM—Table 2.1 shows the statistics for one of the large data sets released by Google—and the trend toward more data is likely only beginning.

Building LMs efficiently (or at all) using such large corpora is a major challenge researchers have been actively tackling. To reduce memory requirements, various *lossless* representations of the data have been employed such as using a trie (Stolcke, 2002), the space efficient prefix tree structure, entropy-based pruning (Stolcke, 1998), or block encoding (Brants et al., 2007). If we allow for a small measure of error in our model, however, we can gain significantly greater space savings by *lossy* representation and encoding of the data. Making use of the improved coverage and estimation ability of massive data sets motivated the research into randomised LMs (RLMs).

The *Bloom filter* (Bloom, 1970) and other randomised data structures based on it support approximate representation of a set S drawn from some universe U and enable queries of the sort "Is an item $x \in S$?". For very large data sets, concise storage is enabled by hash functions mapping between domains $h : U \rightarrow [0, 2^w - 1]$ where $2^w \ll |U|$, the domain size of the underlying universe. The trade-off for the spectacular space savings afforded by these data structures is a tractable measure of error obtained when queried.

In this section we review the work for RLMs. Since all RLMs use hashing as a fundamental operation we begin by reviewing hash functions.

2.6.2 Hash Functions

A *hash function* maps data from a bit vector from one domain into another domain such that

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

with $w \ll b$ commonly. In our discussion w is a integer that represents the bit length of a word on a *unit-cost RAM model* (Hagerup, 1998). The keys to be hashed comes from a set S of size n in a universe U such that $S \subseteq U$ and $U = \{0,1\}^b$. The key's representation in the smaller domain of w-bits is called a *signature* or *fingerprint* of the data. The hash function *chops* and *mixes* the keys of S deterministically to produce its output. A hash function must be deterministic and reproducible: for equivalent keys it must generate matching fingerprints and if the output of the hash function for two keys differs then we know the keys are not equal.

A hash function used with its counterpart data structure, the *hash table*, is a specialised type of *dictionary* or associative array that stores key/value pairs. In the simplest case, a key x is stored in the hash table at the index generated by the hash function h such that the output $h(x) \rightarrow \{0,1,\ldots,m-1\}$ maps to a value in the range of an array of size m. An 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 2.2 illustrates a simple 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 w-bits is 2^w. If S is large and $w \ll b$, then some of the elements in S will collide when mapped into the smaller space. *Collisions* are minimized by choosing a hash function whose outputs are uniformly distributed over the space $[0, w-1]$. We can view each possible value in $[0, w-1]$ 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. This will lead to a large number of collisions and poor performance.

A large body of research has been written on algorithms to reduce the probability of collisions in hash tables. *Perfect hashing* is a technique that theoretically allows no collisions in a hash table that can be created with probability $1/2$ (Cormen et al.,

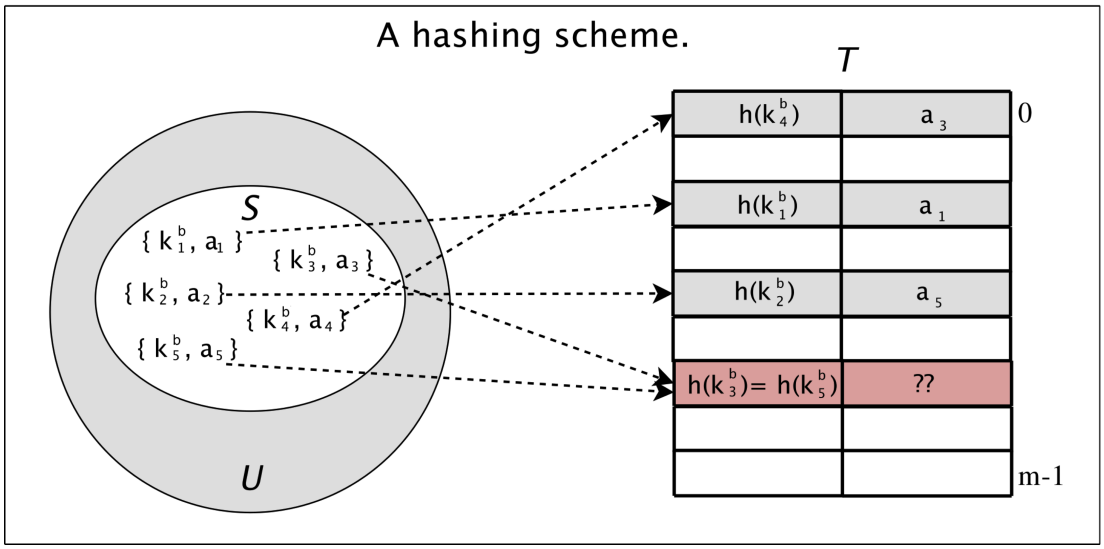


Figure 2.2: The pairs of keys k_i and the associated information a_i of the set S are mapped via the hash function h into cells of the table T. The keys k_i are b-bit lengths in S and w-bit lengths in T. There is a collision between the elements k_2 and k_4 so the value of a_i for that cell in T is unknown.

2001). We give an example of a perfect hash function in the Bloomier filter LM description below. Other well known solutions for collision avoidance include *chaining*, *open addressing*, and *Cuckoo hashing* (Pagh and Rodler, 2001).

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 instantiation but show good performance on average. Specifically, if H is a collection of finite, randomly generated hash functions where each hash function $h_i \in H$ maps elements of S into the range $[0, 2^w - 1]$, H is said to be *universal* if, for all distinct keys $x, y \in S$, the number of hash functions for which $h(x) = h(y) \leq |H|/2^w$. That is, for a randomly chosen hash function $h_i \in H$ we have the probability for a collision $P(h_i(x) = h_i(y)) \leq 1/2^w$ for distinct keys $x \neq y$. Using theory from numerical analysis we can 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).)

Such a family of universal hash functions with these properties is defined in Carter and Wegman (1977) so the ith hash function is of the form

$$\mathcal{H}_i := h_{a[i],b[i]}(x) = (a[i]x + b[i]) \bmod P,$$

where P is a prime chosen so $P > n$, and a, b are integer arrays whose values are randomly drawn from the range $0,\ldots,P-1$. All RLMs described and analyzed below use universal hash function families of this type. Since throughout this thesis we are only concerned with the hashing of n-grams so x in the expression for $\mathcal{H}_i$ above is actually a sequence of words of length $|x|$. Hence we use a specific instantiation of a universal hash function family such that for the ith hash function we have

$$\mathcal{H}_i := h_{a[i],b[i]}(x) = \sum_{j=1}^{|x|} (a[i][j] * x[j] + b[i][j]) \bmod P$$

where $x[j]$ is the jth word of the n-gram being hashed and all numbers $a[i][j]$, $b[i][j]$ in the doubly indexed arrays are randomly generated integers from the range $[0,P-1]$.

2.6.3 Bloom Filter LM

The Bloom filter (BF) is a randomised data structure that supports queries for set membership that are widely used in industry. Applications include database applications, network routing, and spell-checkers (Costa et al., 2006; Broder and Mitzenmacher, 2002). The nature of the encoding of the filter makes the original data irretrievable which is a positive feature when used in security sensitive domains such as IP address caching. It is also impossible to remove a key from a BF without the chance of corrupting other elements in the set. There have been more space efficient alternatives proposed in the literature (Pagh et al., 2005) but the simplicity and overall performance of the original Bloom filter has made it the essential randomised data structure by which most others are compared.

Bloom Filter:

The BF has a unique encoding algorithm which gives it spectacular space savings at the cost of a tractable, one-sided error rate. In the first complete model of its kind, Talbot and Osborne (2007b) used a variation of the BF to encode a smoothed language model which matched baseline translation performance using a fraction of the baseline model's memory. First we describe the basic data structure.

At the start the BF is an array of m bits initialized to zero. To populate a BF we need k independent hash functions drawn, for example, from a family of universal hash functions described above. Each hash function maps its output to one of the m bits in the array, $h(x) \rightarrow \{0,1,\ldots,m-1\}$. Each element x in the support set S is passed through each of the k hash functions, $h_1,\ldots,h_k$, and the resulting target bits in the array

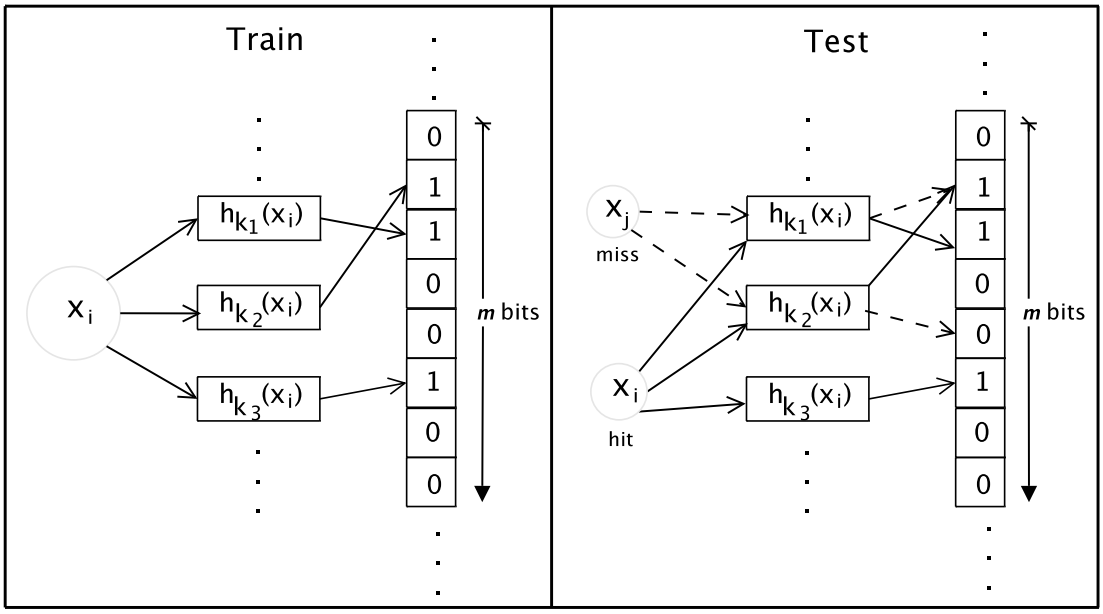


Figure 2.3: Populating and testing a Bloom filter. To populate we "turn on" k bits in the array for each item $x \in S$. A test for membership fails if any index queried is zero. Else assume membership.

are set to one. In other words k bits of the array are "turned on" for each item $x \in S$ in the support. While inserting a new item, any target bit already on from any previous items stays so. The source of the fantastic space advantage the BF has over most other data structures is the bit sharing between elements of the support 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. Conversely, if each position in the bit array is set to one for all k hash functions then we have a hit and assume the element is a member. However, there is a chance for a test item $x' \notin S$ not in the support that all k hash functions will target random bits turned on for other elements $x \in S$. This a false positive error and is obtained with the same probability that a random selection of k bits in the array are set to one.

Assuming the hash functions are uniformly distributed, each index in the m-bit array is targeted with equal probability of $1/m$. Given $n = |S|$ as the size of the support S, 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 all n elements in the support set have passed

through the k hash functions is

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

so the probability, then, that a bit is one is therefore

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

If an element were not a member of S, the probability that it would return a one for each of the k hash functions, 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

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

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.

Log-frequency Bloom filter LM:

The work in Talbot and Osborne (2007a) and Talbot and Osborne (2007b) first reported a complete randomised LM using a BF encoding. The BF supports only member queries, but a LM requires storing key/value pairs where the key is the n-gram and the value its count or smoothed probability.

To use the BF as a dictionary, a *log-frequency* encoding scheme was used. The n-gram counts were first quantized using a logarithmic codebook so the true count $c(x)$ was represented as a quantized count q such that

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

with b the base of the log. The quantised frequency value of each n-gram was "appended" to it and the composite event was entered into the BF. The accuracy of the codebook relies on the Zipf distribution of the n-grams. Few events occur frequently and most occur only a small number of times. For the high-end of the Zipf distribution the counts in the log-frequency Bloom filter LM are exponentially decayed using this scheme. However, the large gap in distribution of these events means that a ratio of the likelihood of the model is preserved in the log-frequency encoding.

To allow retrieval of its value, each n-gram was inserted into the filter q times. That is, for each level q', $q' \in [1,q]$, is appended to the n-gram and k random bits are turned on for that event. Each quantisation level $q' \in [1,q]$ has a unique hash function associated with it so in total qk bits are set for each n-gram. During testing the process is repeated until a query encounters a zero bit or the maximum count is reached. The highest value returned is used as the log-frequency of the test event. Since the majority of n-grams in a distribution have low counts, for most queries this process is repeated only 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

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

where here q represents the quantised count returned by the BF and b is the base of the logarithm used for the logarithmic codebook. *Run-time smoothing* is then performed to retrieve the smoothed probability. The statistics needed for the smoothing algorithm are also encoded in the BF bit array with the exception of singleton events. In this case, the proxy that an event was a singleton was the event itself.

The Bloom filter 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 7% fraction of the 924MB lossless LM space, the BLEU scores for translations done using the Bloom filter LM matched those for translations done with a lossless LM. However, because of the log-frequency encoding and therefore the necessity of multiple queries to the bit array for each test n-gram, the Bloom filter LM is slow in practice. A faster variant of the Bloom filter LM is described below. Note also the bit sharing of the BF means that deleting any n-gram from a populated filter may potentially corrupt other n-grams. And, given an incoming stream of data, continual insertion of n-grams into the BF without deletions would result in increasing higher error as more and more bits are set. This is shown in Figure 2.4

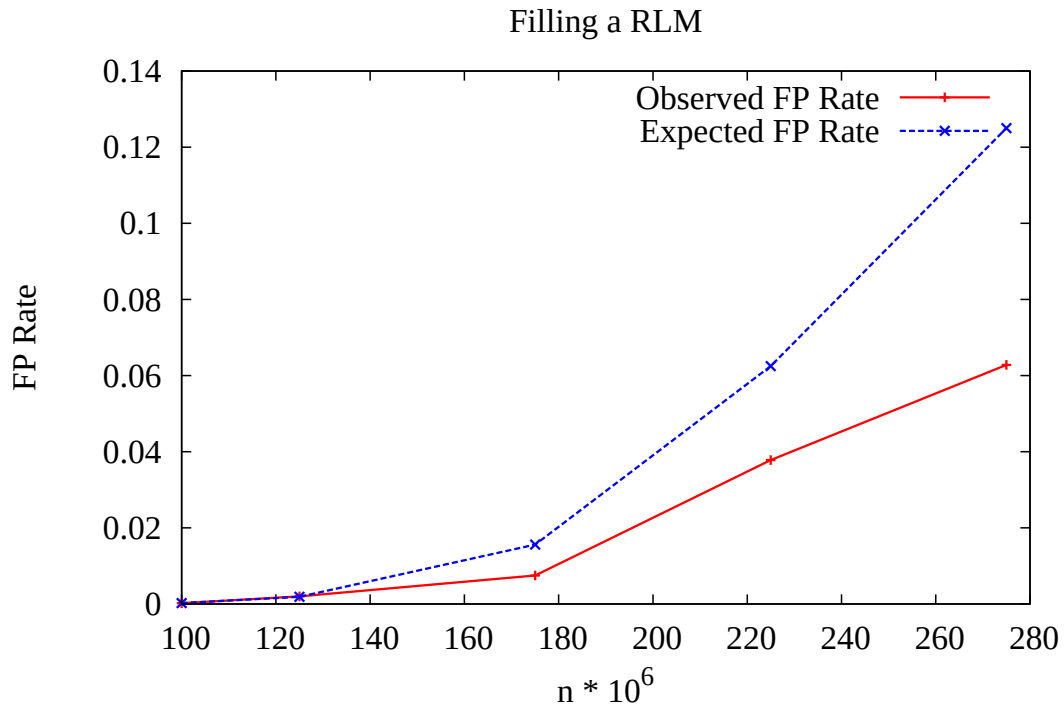


Figure 2.4: For RLMs with equal memory usage, error rates rise with the number of events they encode. Shown are the observed and expected (theoretical) false positive error rates.

2.6.4 Bloomier Filter LM

The BF can only encode and test set membership. The *Bloomier* filter from Chazelle et al. (2004), a dictionary extension of the BF, was used in Talbot and Brants (2008) to encode a RLM of a different strain. It uses *perfect hashing* to map each element $x \in S$ to an index in a large associative array A to encode key/value pairs $(x, v(x))$ where the value $v(x) \leq V$, the largest value encountered in the support.

Using a universal hash function $\mathcal{H} := \{h_i : i \in [1,k]\}$, k possible locations are chosen in A but only one location $h_i(x)$ is used to store $(x, v(x))$. Before encoding the n-grams in the array A, a greedy randomised algorithm first finds an ordered matching of n-grams $x \in S$ to locations $A[h_i(x)]$. The ordered matching means that we have a perfect hash of the support S in the Bloomier filter LM. However, due to the random hash functions, the perfect hash algorithm is not guaranteed to find an ordered matching after one attempt and hence may need to be repeated many times until a suitable ordering is reached.

Once each key has a unique cell associated with it each n-gram x is associated with

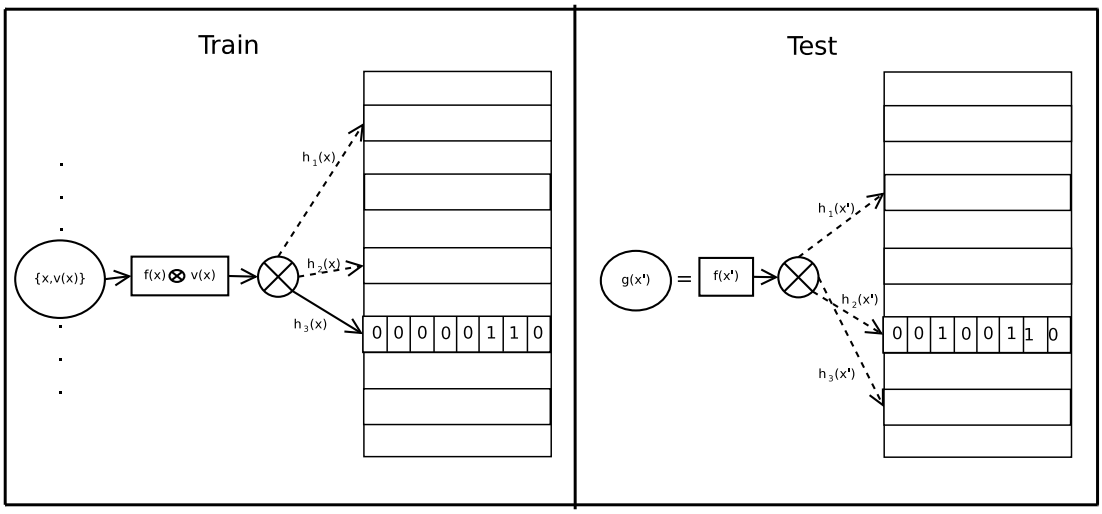


Figure 2.5: Populating and testing a Bloomier filter with $k = 3$ hash functions. To insert an item $x \in S, v(x) \leq V$, XOR the value $v(x)$, a fingerprint $f(x)$, and the values of $k-1$ indexes in the array $A[h_i(x)]\ \forall\{i|i \in [1,k] \wedge i \neq j\}$ where $A[h_j(x)]$ is the pre-matched index for x. To test item x', XOR the fingerprint $f(x')$ and the k values $A[h_i(x')]\ \forall\{i|i \in [1,k]\}$. If the value $g(x') \leq V$ we assume a hit.

a fingerprint generated by another hash function $f(x) : x \in S \rightarrow [0, 2^w - 1]$ where w is the number of bits allocated to each cell of A. To set an index for an n-gram x is

$$A[h_i(x)] = v(x) \otimes f(x) \otimes \left(\bigotimes_{j=1 \cap j \neq i}^{k} A[h_j(x)] \right)$$

where $\otimes$ represents the binary exclusive-bitwise OR (XOR) operator. When testing, the value returned for a test n-gram x' is

$$g(x') = f(x') \otimes \left(\bigotimes_{j=1}^{k} A[h_j(x')] \right).$$

A good property of the Bloomier filter LM is the perfect hash guarantees for any $x \in S$ there is no error returned for the corresponding value $v(x)$ during test time. (This is unlike the BF LM which can return an over-estimate for events in the support.) However there is still a chance for false positives. An error occurs when a test item $x' \notin S$ is assigned a value $g(x') \in [0,V]$. Since $g(.)$ is distributed uniformly at random the probability of a false positive error is

$$\Pr(g(x') \leq V | x' \notin S) = V/2^r.$$

Despite this, in large-scale SMT tests a major saving in space was achieved with negligible loss to performance.

The Bloomier filter trades off space for time compared against the Bloom filter. The Bloomier filter is word-based (operations are performed over multiple bits) so it uses more space than the Bloom filter which operates on only a single bit per operation. The native support for key/value pairs means there is no need to encode n-grams multiple times in the filter. This makes it faster to query since the number of queries k to the Bloomier filter is constant and independent of the counts associated with the n-grams. However, finding the ordered matching necessary for the encoding of the LM is both resource and time consuming. This makes the Bloomier filter LM ill-suited for stream-based translation since frequent retraining is required.

2.6.5 RLM Conclusion

We have shown how RLMs achieve significant space savings compared to their lossless counterparts. By exploiting hash functions and sharing the bits of the data structure between members of the support during encoding, Bloom and Bloomier filter LMs achieve matching translation quality for SMT while using little memory. The trade-off for the memory saved is the false positives rates that were analyzed.

However, neither the Bloom or Bloomier filter allow for online adaptation of the n-grams in the model. This means that to add any new n-grams to an existing model the LM must be fully retrained. Frequent batch retraining incurs a high computational footprint in both resources and time and is not optimal for a stream-based SMT system. Later in this thesis we investigate an online version of the batch Bloomier filter LM that supports incremental retraining suitable for stream-based processing. In the next section we describe the theory behind the data streams approach to computation.

2.7 Data Streams

Research on data streams evolved from the need to analyse massive throughput of dynamic data on systems with limited resources. Data streams are defined in Muthukrishnan (2003) as

> "input data that comes at a very high rate [which] means it stresses communication and computing infrastructure so it may be hard to transmit...compute...and store."

Application domains such as network and router monitoring, financial, and database systems can generate terabytes of data each day with millions or billions of updates

hourly. Data streaming algorithms and their randomised data structures, called *sketches*, provide theoretically well founded mechanisms to glean statistics from huge amounts of data while keeping computation and space bounds tractable.

It is of interest to natural language tasks to note the prohibitive use of memory a raw data stream demands stems not primarily from the sheer amount of incoming data but rather the domain size of the universe the underlying signal of the data stream represents. For example, suppose we received masses of tuples representing all peoples on Earth and their current weight in kilos and we wished to calculate some distribution over the set of weights. Although the number of updates would be large it would be fairly trivial to manage a stream of this nature in memory as we have a fairly constrained signal size of, say, one to 500. Compare this to the universe size of IP addresses, which is potentially 2^{64}, or the set of unique n-grams we may encounter from an unbounded text web stream, which may be unbounded. Unlimited RAM is infeasible and storing and processing this amount and type of data on disk is not feasible for real time applications as disk I/O is expensive. Data streams provide a way to efficiently select and incorporate new, relevant data into our models online without the requirement for full batch retraining.

In the rest of this section we examine the basic model of data streams and review the relevant literature.

2.7.1 Basic Model

As described in Cormode and Muthukrishnan (2005), the basic data stream model begins with a n dimension vector $\vec{a}$ initialized at time $t = 0$ to $\mathbf{0}$, the zero vector. At a given time t we have a current state of $\vec{a}(t) = [a_1(t), \ldots, a_i(t), \ldots, a_n(t)]$ that describes some underlying space or signal at that moment. Updates to entries in $\vec{a}$ are received as a stream of m pairs (i_t, c_t) where i_t is an index of $\vec{a}$ at time t with data c_t. There are three models of updates to $\vec{a}$ listed below from least to most general:

- *Time Series*. Each update (i_t, c_t) replaces the value in a_i so $a_{i_t} = c_t$.

- *Cash Register*. Updates are strictly positive increments, $a_{i_t} \geq 0$ so $\vec{a}$ is monotonically increasing.

- *Turnstile*. Updates can be either positive or negative so the value in any $a[i]$ can both increase or decrease. There are two variations on this model. A *strict*

Turnstile model has $\forall t, a_{i_t} \in \mathbb{R}^+$. A *non-strict* model has entries $a_{i_t} \in \mathbb{R}$ so values can be negative.

To place these models in context, imagine the input signal was a stream of text that represented document statistics over which different NLP tasks were to be performed. If the task at hand was single-document topic detection, it would make sense to use a variant of the Time Series model and for each new document, replace the entries in the underlying vector. For language modeling the Cash Register model is more appropriate as we want to account for the full distribution of the input over time. The tuples in the stream would be frequencies of words in the text and we would increase each n-gram count in the vector as it was encountered. The Turnstile model with decrements is not one that would usually be encountered in an NLP setting as most models of language are based on word or context counts. The occurrence of words, patterns, and other information is either present and counted or, if it is absent, has a zero count and therefore isn't reported. One possible scenario in which a Turnstile model would be employed is if the stream statistics were comparisons to a previously known oracle distribution. In this case the counts would indicate how much the events differed in value from the oracle's frequency distribution.

For the rest of this section we analyse and describe only the more general Cash Register and Turnstile models. For these models the t^{th} update means:

$$\begin{aligned} a_{i_t} &= a_{i_t}(t-1) + c_{i_t} \\ a_{i'} &= a_{i'}(t-1) \qquad\qquad \forall i' \neq i_t \end{aligned}$$

At a given time t we would like to compute functions of interest over the signal $\vec{a}$ but the stream input brings in masses of updates quickly and the size and domain of $\vec{a}$ is potentially very large. Classic storage and compute solutions which use trade-offs of space or time linear to the data are infeasible; space and time linear to the input is still prohibitive for practical use. Data stream algorithms have the desiderata that the space used for storage is sublinear, preferably poly-logarithmic, in n and what would be needed to store $\vec{a}$ explicitly, and the per item processing time and overall compute functions are fast, preferably (but not necessarily) of the same order as the storage.

We can use space sublinear in input size by making linear projections of the input stream into structures that hold only summaries or a *sketch* of the data and approximating the functions desired. We approximate the correct answer, however, with some guarantee to the accuracy of the result. Typically the accuracy depends on two adjustable parameters, ε and δ. We say the approximation returned is within a $(1 \pm \varepsilon)$

factor of the true result with probability δ of failure where δ is the level of randomness allowed in the query functions. The compute time and storage required will commonly depend on ε and δ. When both parameters are used in an algorithm the resulting query result is called a (ε, δ)-approximation (Cormode, 2008).

There are three statistical formulations commonly used to prove the (ε, δ)-approximation guarantees hold. Let X be a random variable with expectation $E(X) := \mu$ and variance σ^2 and $k > 0$ be a positive real. Then *Markov's inequality* states:

$$\Pr(|X| \geq k) \leq \frac{\mu}{k}.$$

Chebyshev's inequality holds that

$$\Pr(|X - \mu| \geq k\sigma^2) \leq \frac{1}{k^2}.$$

A *Chernoff bound* applies the above Markov inequality to the moment generating function of X to get tighter bounds on tail probabilities (Motwani and Raghavan, 1995). If the variable $X = \sum_{i=1}^{n} X_i$ represents the summation over the i.i.d random variables $X_i \in [0, 1]$, a (restricted) *Chernoff bound* has

$$\Pr(X - \mu \geq k) \leq e^{-2k^2/n}.$$

2.8 Text Streams: Utilizing Recency

Suppose we have an incoming stream of today's newswire stories in a foreign language that we would like to translate into English. It is natural to think that having up-to-date English news available during the translation task will ease and possibly improve the final translation. The stories between language pairs will overlap and the translator may get a better idea of how to express the foreign sentences in English. We can extend the same concept to a streaming (online) translation system which is constantly employed to translate current news and other documents. Intuitively using relevant data will still be beneficial to such a system. The question then is which of the massive amount of available data is potentially relevant and how can we find and use it efficiently.

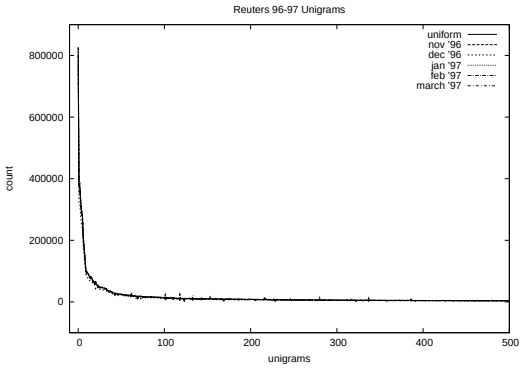
In this section we motivate how we utilized NL streams in the SMT experiments reported in this thesis. We describe our initial background experiments and demonstrate that, given an incoming stream with its implicit timeline, *recency*, a well known property of natural language, can be used to improve performance of test points within the stream.

To motivate this work we initially investigated the time-dependent variability of NL distributions. Our experiments were primarily done using both the Europarl corpus (Koehn, 2003), a collection of parallel sentences in many European languages containing the spoken proceedings of the European Parliament spanning over a decade, and the RCV1 corpus (Rose et al., 2002), a time-stamped, chronological corpus of one year's worth of multilingual newswire data from Reuters. For example, Figure 2.6 shows how the distribution, shown via raw counts here, of the most frequent *n*-grams in the RCV1 corpus varies monthly compared against the normalized distribution for the full year. As is evident, the *n*-gram counts are bursty even for documents within the same domain. The bursty nature of natural language was shown too in Curran and Osborne (2002).

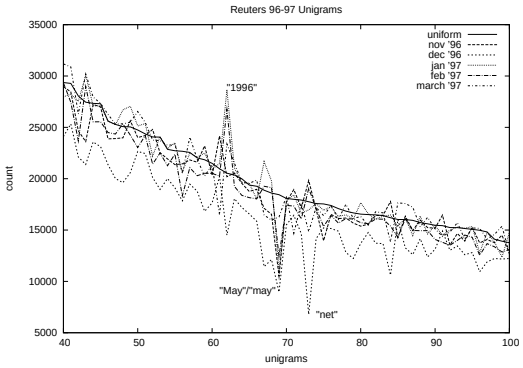
Given this well known phenomenon and adding to it the fact that there will always be more training data available than we are able to store in its entirety moved us to investigate how, given a restriction on available memory, we could choose a better than random subset of the full set of training data for a given task. One fairly obvious initial approach for training data selection is to use recency, a well established property of natural language where chronologically more recent language has higher relevance to the present. Recency is both explicitly and implicitly prevalent in NLP from HMMs conditioning on only a short history of prior events (those that are recent to the current point being analyzed) to the effect of priming in spoken conversation. We ran a number of preliminary experiments with an enforced memory constraint and, given a time-stamped test document, used a "sliding window" of chronological, overlapping subsets of the training data. We tested using various tasks and associated metrics which we describe in the sections below.

2.8.1 Effect of Recency on Out of Vocabulary Rates

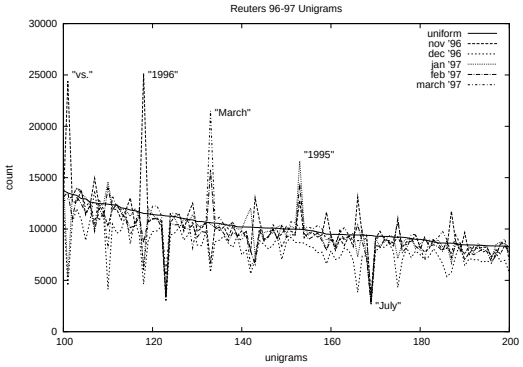
An example of the effect of recency on the out of vocabulary (OOV) rate is shown in the plots in Figure 2.7. The OOV rate is the percentage of words, or *n*-grams, in a test set that do not appear in the training data. For Figure 2.7 we used a sliding window along the chronologically ordered data's timeline to create multiple *epochs* of sentences from the French section of the Europarl corpus and tested the OOV rate against a document held-out from the end of the Europarl timeline. Europarl is a parallel corpus and any French phrases in the test document that do not occur in the training data means no translation is available for them. Clearly a higher OOV rate



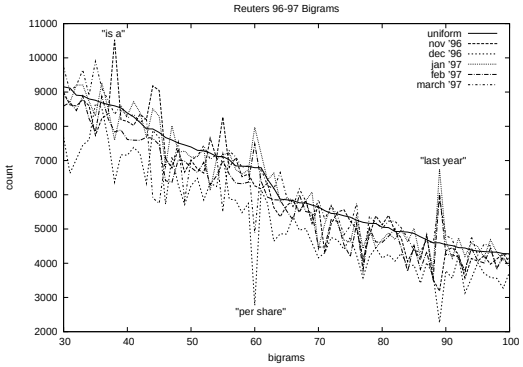
(a) Unigram Zipf distribution



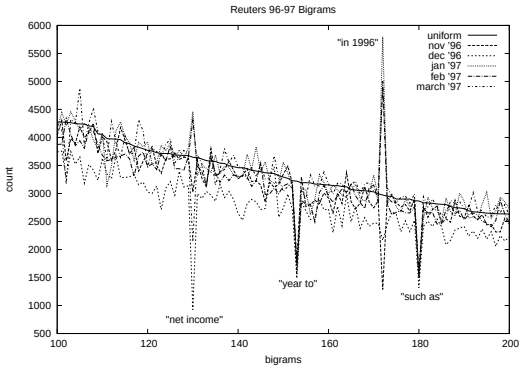
(b) Top 30-100 frequent unigrams include words: *billion, 1996, 1997, shares, bank, against*



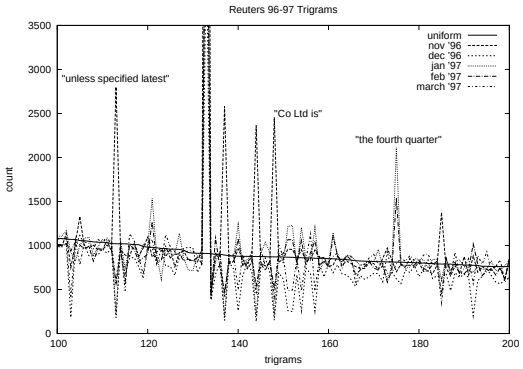
(c) Top 100-200 frequent unigrams include words: *stock, president, 1995, yen, London, dollar*



(d) Top 30-100 frequent bigrams include words: *on Friday, New York, Prime Minister, per share*

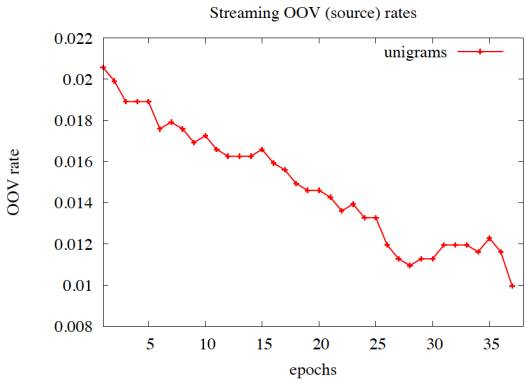


(e) Top 100-200 frequent bigrams includes: *the dollar, I think, he added, chief executive*

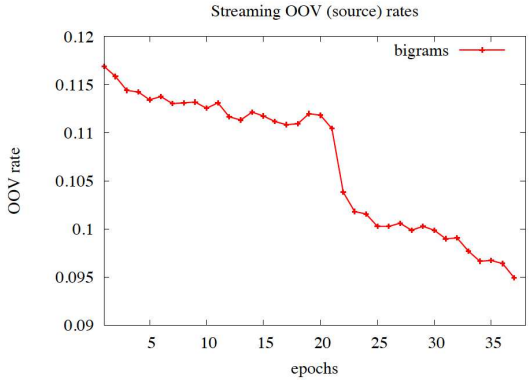


(f) Top 100-200 frequent trigrams includes: *the European Union, a spokesman for, the fourth quarter*

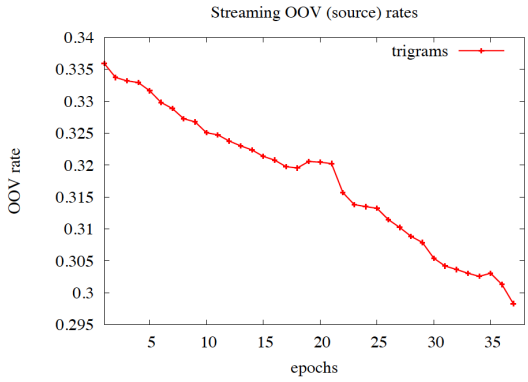
Figure 2.6: A look at the distribution of the most frequent items in the RCV1 corpus for different orders of n-grams. The numbers along the x-axis represent the ith most frequent n-gram in the corpus. The solid line represents the normalised distribution. For many n-grams the variance is large.



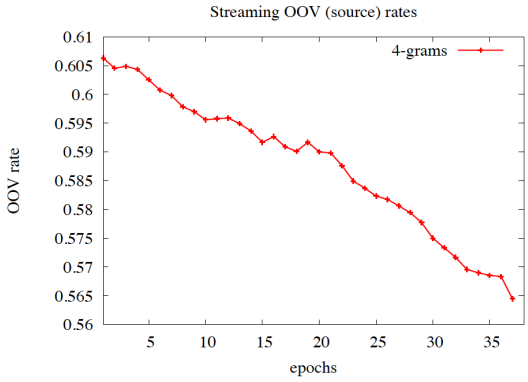
(a) Example Unigram OOV Rates



(b) Example Bigram OOV Rates



(c) Example Trigram OOV Rates



(d) Example 4-gram OOV Rates

Figure 2.7: For a given test document and a memory restriction on the number of sentences allowed per epoch, the OOV rates for phrases of length 1-4 are reduced by using sentences drawn from training data that is chronologically more recent to the test date.

impacts performance translation. As is clearly seen from the plots in Figure 2.7, for *n*-grams of orders one to four there is a consistent downward trend in the OOV rate as the sentences are drawn from times closer to the held-out test point with the epoch closest to the test point having the lowest OOV rate amongst all epochs for each *n*-gram order.

2.8.2 Effect of Recency on Perplexity

We also ran preliminary investigations using perplexity as a metric. Perplexity is a transformation of the entropy or cross entropy of a distribution and is the primary evaluation metric for LMs in the literature. In evaluating LMs, with n test states for a random variable x drawn from an unknown distribution, the per-word perplexity of the

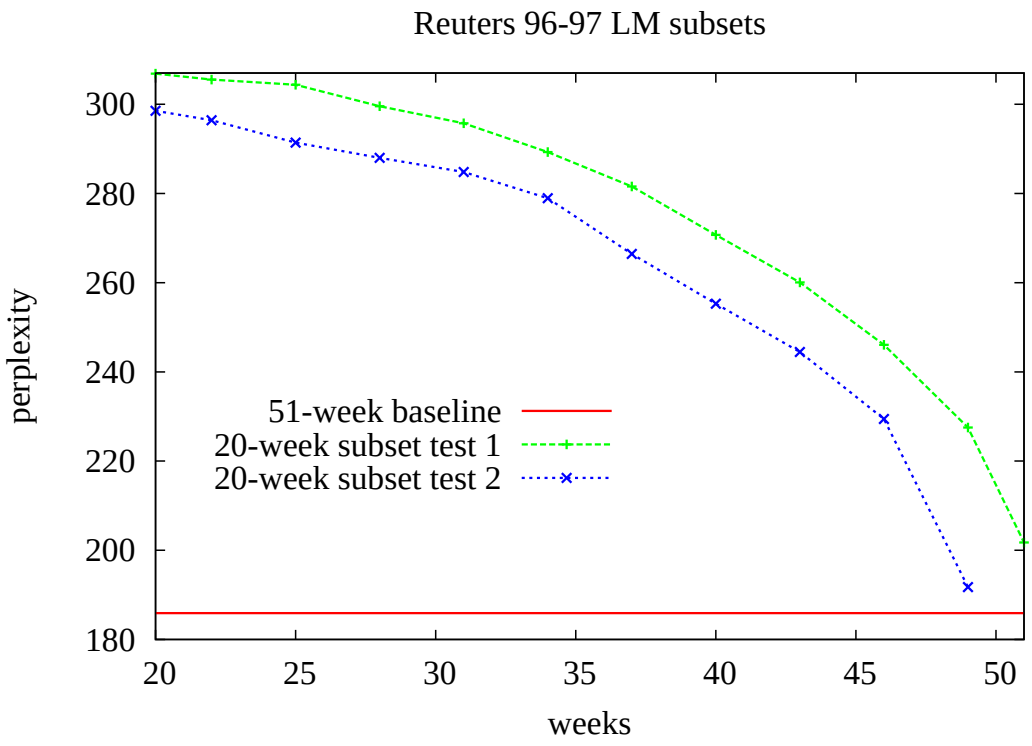


Figure 2.8: Perplexity decreases as the training data is drawn from times closer to the test data. The points on the slope indicate the last week of training data for each subset LM.

proposed model q is

$$2^{-\sum_{i=1}^{n} \frac{1}{n} \log q(x_i)}.$$

The lower the per-word perplexity returned the better the proposed model accounts for the test data. Using the entire English RCV1 as a baseline, we again imposed a memory bound within which we build the models and test subsets of the data. The strict memory bound mimics our inability to store unlimited data. We treat the full RCV1 corpus as "all" the data and then test using subsets of the training data. Our goal is to find that subset by which we get closest to the performance of the baseline.

For these experiments we made use of the sliding windows approach over the RCV1 training data timeline. Since the documents are timestamped and chronologically ordered we can mimic stream-based processing for the RCV1 corpus. As an oracle baseline a LM was built using data from the full timeline that spanned 51 weeks. We then trained multiple LMs of much smaller sizes, coined *subset LMs*, to simulate memory constraints. For a given date in the RCV1 stream, these subset LMs were trained using a fixed window of previously seen documents up to that data. For each

	$LM^{240MB}_{10-02-97}$	$LM^{465MB}_{10-03-97}$	$LM^{240MB}_{14-04-97}$	$LM^{303MB}_{11-08-97}$
11-02-97	**194.121**	N/A	N/A	200.494
15-04-97	235.297	218.221	**185.059**	N/A
12-08-97	263.937	236.634	251.6	**169.434**

Table 2.2: Comparison of perplexity on different test dates. Column title subscripts are dates corresponding to the last day of the training data for that LM. Column title superscripts are the size of each LM in RAM as reported by SRILM. Cells with "N/A" are so because the LM training data includes the test data for that day.

subset LM we slid the window forward by three weeks and then obtained perplexity results for each subset LM against a static held out test set.

Figure 2.8 shows an example. For this experiment subset LMs were trained using a sliding window of 20 weeks with the window advancing over a period of three weeks each time. The two arcs correspond to two different test sets drawn from different days. The arcs show that recency still has a clear effect: populating LMs using material closer to the test data date produces improved perplexity performance. The LM chronologically closest to a given test set has perplexity closest to the results of the significantly larger baseline LM which uses all the stream. As expected, using all of the in-domain data yields the lowest perplexity.

It happens that the RCV1 corpus has more data on average per day towards the final months of the year it spans. A subset LM from the beginning of the year will contain less data than one built from an equal time span at the end of the year. To test whether the improvements came from a temporal relation to the test data or because the later models contained more training data we crossed checked subset LMs of various sizes with test data occurring directly after each. The results of this experiment are shown in table 2.2. We see that LMs of considerably larger size still do not match the performance of smaller LMs that are trained from data directly prior to the test date.

We note that this is a robust finding, since we also observe it in other domains. For example, we conducted the same tests over a stream of 18 billion tokens drawn from 80 million time-stamped blog posts downloaded from the web with matching results. The effect of recency on perplexity has also been observed elsewhere (see, for example, Rosenfeld (1995) and Whittaker (2001)).

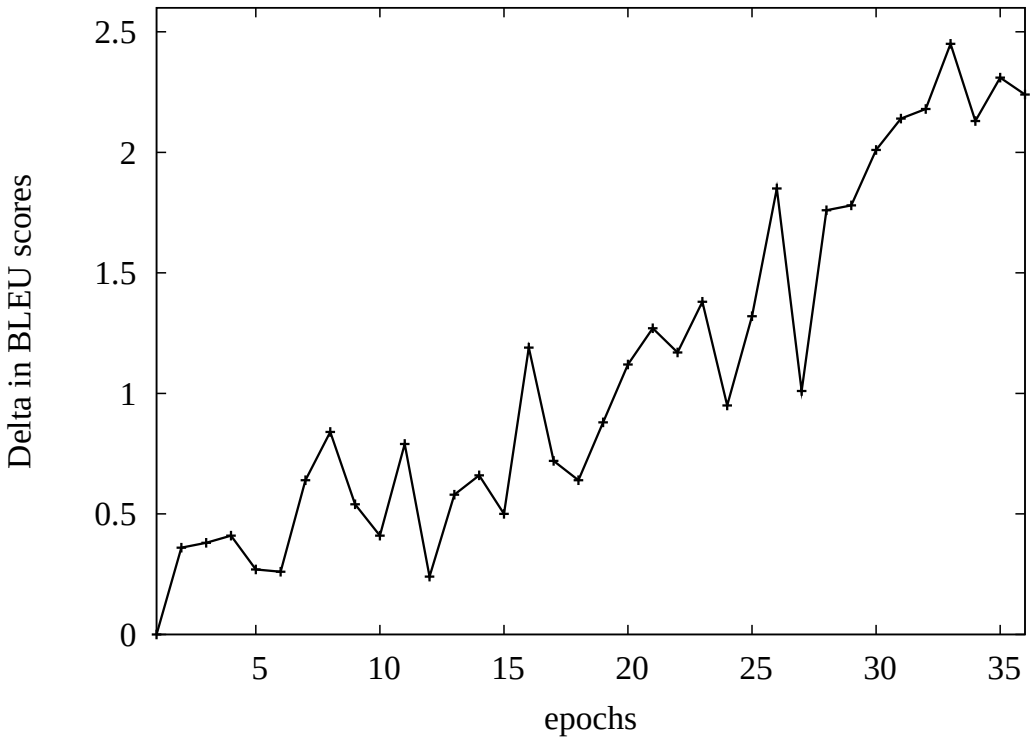


Figure 2.9: Recency effects on SMT performance. Depicted are the differences in BLEU scores for multiple test points decoded by a static baseline system and a system batch retrained on a fixed sized window prior to the test point in question. The results are accentuated at the end of the timeline when more time has passed confirming that recent data impacts translation performance.

2.8.3 Effect of Recency on Machine Translation Performance

Given an incoming stream of parallel text, we gauged the extent to which incorporating recent sentences into a TM affects translation quality as measured by BLEU. BLEU is an automatic evaluation metric that measures the precision of the translation by calculating matches of *n*-grams of length 1 to *n* between sentences of a given machine translated document and (possibly multiple) reference translations (Papineni et al., 2001).

We used the Europarl corpus with the French-English language pair using French as source and English as target. Europarl is released in the format of a daily parliamentary session per time-stamped file. The actual dates of the full corpus are interspersed unevenly (they do not convene daily) over a continuous timeline corresponding to the parliament sessions from April,1996 through October, 2006, but for conceptual simplicity we treated the corpus as a continual input stream over consecutive days.

As a baseline we aligned the first 500k sentence pairs from the beginning of the corpus timeline. We extracted a TM for and translated 36 held out test documents that were evenly spaced along the remainder of the Europarl timeline. As seen in Figure 2.9 these test documents effectively divided the remaining training data into epochs and we used a sliding window over the timeline to build 36 distinct, overlapping training sets of 500k sentences each.

We then translated all 36 test points again using a new set of grammar rules for each document extracted from only the sentences contained in the epoch that was before it. To explicitly test the effect of recency on the TM all other factors of the SMT pipeline remained constant including the language model and the feature weights. Hence, the only change from the static baseline to each epoch's performance was the TM data which was based on recency. Note that at this stage we did not use any incremental retraining.

Results are shown in Figure 2.9 as the differences in BLEU score between the baseline TM versus the translation models trained on material chronologically closer to the given test point. The consistently positive deltas in BLEU scores between the static model that is never retrained and the models that are retrained show that we achieve a higher translation performance when using more up-to-date TMs that incorporate recent sentence pairs. As the chronological distance between the initial, static model and the retrained models increases, we see ever-increasing differences in translation performance.

These preliminary experiments show that a possible way to tackle data selection in the streaming translation setting is to always focus the attention of the model being updated on the most recent part of an incoming stream of training data. To maintain constant space we must remove data from the model that came from the receding parts of the stream and replace it with the present. However, current algorithms for (re)training the models of an SMT system are computationally expensive in terms of both time and space and so not appropriate for this type of stream-based translation where we often update the system with small batches of novel training data.

2.9 Conclusion

In this chapter we reviewed the background theory, models, and experiments for our work. We described the basic elements of a SMT system and described in detail RLMs. We introduced data streams and described how we are able to use the recency inherent

in a text stream to improve our models. For the rest of this thesis we describe data structures and streaming algorithms that allow for efficient retraining of a SMT system. Our approaches allow for quick updates to the models of an SMT system in bounded space eliminating the need for batch retraining.

Chapter 3

A Stream-based Language Model

In this chapter we introduce a stream-based, randomised LM that operates within bounded memory. This is the first LM of its kind in the literature and constitutes one of the foremost contributions of the thesis. The online LM presented extends the Bloomier filter LM (Talbot and Brants, 2008) from Chapter 2.6. Like the batch Bloomier filter LM, the online model is randomised and stores the set of n-grams it encodes in small space. Instead of computing a time consuming perfect hash function offline, however, the stream-based LM uses a fast, *online perfect hashing* scheme. Due to these properties we refer to the LM as the *online randomised LM* (ORLM). The ORLM and its basic properties were described previously in Levenberg and Osborne (2009).

3.1 Overview

The stream-based translation scenario considered is as follows: we assume a source stream that each day brings any number of new documents that need translation. We also assume a separate stream of in-domain documents in the target language. Intuitively, since the concurrent streams are from the same domain, we can use the contexts provided in the target stream to aid in the translation of the source stream. This is shown pictorially in Figure 3.1.

The goal is to accurately model the stream's distribution at the current point in the timeline. Crucially, however, over time the stream's domain or its underlying distribution may change. Since the entirety of the unbounded stream cannot be represented in constant space whilst maintaining a constant error rate, we are forced to throw some old information away to free space in the model for new, currently pertinent text. As

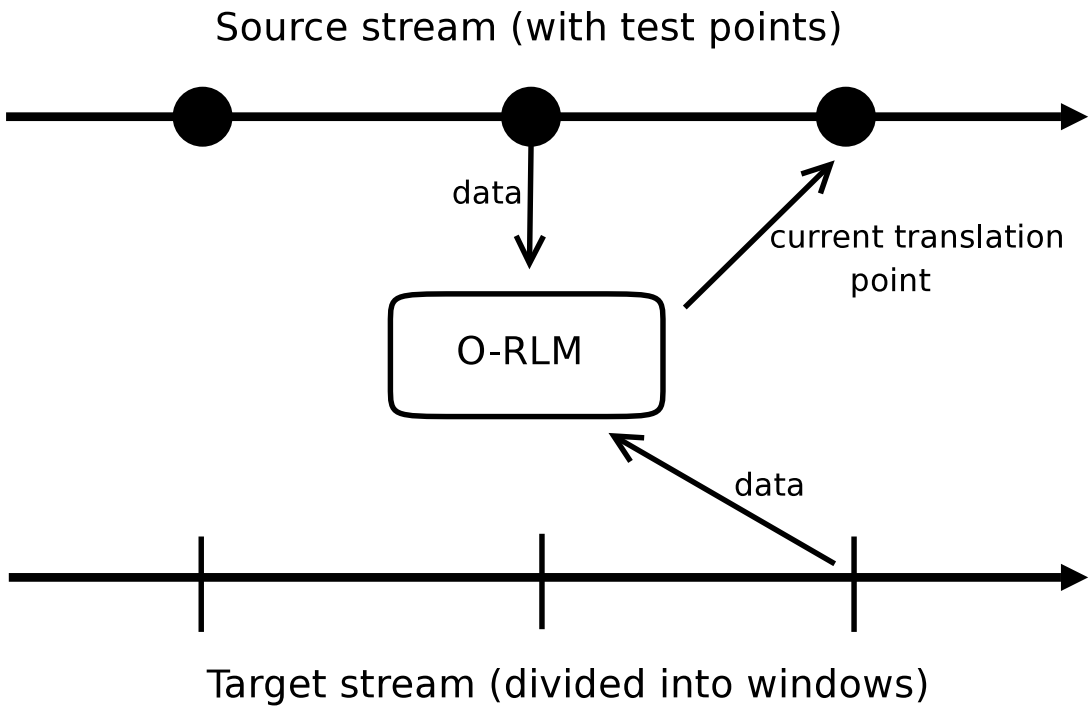


Figure 3.1: Stream-based translation. The ORLM uses data from the target stream and the last test point in the source stream for adaptation.

described previously in Section 2.6, deletions cannot be done using the current crop of RLMs from the literature due to their inherent bit sharing. To accomplish this we introduce a new data structure, the dynamic Bloomier filter, described in detail in the next sections.

3.2 Dynamic Bloomier Filter via Online Perfect Hashing

In this section we describe the online perfect hash function which is the core mechanism of the ORLM. The online perfect hash algorithm was first introduced and analyzed by Mortensen et al. (2005) in which the authors target problems that

> " are typical of data stream computation, where one needs to support a stream of updates and queries, but does not have space to hold the entire state [in] the data structure."

The original intention of their data structure was to achieve linear space for a dynamic range reporting task but the resulting dynamic Bloomier filter can be applied to any problem which can be formulated (implicitly or explicitly) as an incoming stream of key/value pairs. Specifically, given such a stream of incoming key/value pairs, the

dynamic Bloomier filter uses a family of universal hash functions to support online inserts and deletes. This is unlike the batch Bloomier filter which does not support online adaptation. Using deletions, the dynamic Bloomier filter can maintain *constant memory* throughout its usage. As with the batch Bloomier filter, the keys in the support set from the incoming stream are represented as random fingerprints generated from hash functions. As such, there is a small probability for generating false positives when a collision occurs in the hash space for the stream keys. This is discussed in detail later in this chapter. Here we describe the algorithm for the online perfect hash function.

Let $\mathbf{U}$ be a universe of arbitrary size and $\mathbf{S} \subset \mathbf{U}$ be an incoming stream of key/-value pairs drawn from $\mathbf{U}$ that we wish to model. The perfect hash function and data structures that comprise the dynamic Bloomier filter consists of the following parts:

1. A collection of r randomised dictionaries, $\{d\} = \{d_0, \ldots, d_{r-1}\}$, that hold a large subset $S \subseteq \mathbf{S}$ of the keys as fingerprints. The randomised dictionaries $\{d\}$ are the primary storage mechanism for the dynamic Bloomier filter and hold a large percent of the data from the stream.

2. An exact (lossless) dictionary $\bar{d}$ that holds a small subset $S' = S \setminus \mathbf{S}$ of the stream. It serves as an overflow dictionary for the set of keys in the stream $\mathbf{S}$ that collide in the randomised dictionaries.

3. A deterministic top level hash function $\phi : \phi(x) \rightarrow [0, r-1]$ where $x \in \mathbf{S}$ is a key from the stream. This hash function is used to distribute keys from the stream into the set $\{d\}$ of r randomised dictionaries.

4. A family of deterministic universal hash functions $\{h\} = \{h_0, \ldots, h_{r-1}\}$ where the hash function h_i, $0 \leq i < r$ is associated with the ith randomised dictionary d_i. This set of hash functions is used to generate random fingerprints for the keys in the stream.

Conceptually we can view the randomised dictionaries $\{d\}$ as a set of independent "buckets" that each store some unique subset of the stream (Figure 3.2). To *insert* a key x from the incoming data stream $\mathbf{S}$ the top level hash function $i = \phi(x)$ assigns a key x to one of the buckets $d_i \in \{d\}$, the set of randomised dictionaries. If the dictionary d_i is full then the key x is stored in the overflow dictionary $\bar{d}$. Else, if d_i is not full, the bucket d_i uses its associated hash function h_i to generate the fingerprint $h_i(x)$ for the key x. Then the bucket d_i is searched to determine if the fingerprint $h_i(x)$ already resides in it. If so, there is a collision and the key x is redirected to the overflow dictionary $\bar{d}$ where

it is stored exactly. Otherwise the fingerprint $h_i(x)$ is stored in d_i. Values attached to the keys are stored accordingly; either in the randomised buckets or the overflow dictionary.

To *query* the dynamic Bloomier filter for a test item x' and potentially retrieve its corresponding value, we first check whether the test item x' is in the overflow dictionary $\bar{d}$. If it is, the value is returned and we are finished. If it is not, the test element x' passes through the top level hash function $i = \phi(x')$ to get the target bucket d_i and the corresponding hash function to generate the fingerprint $h_i(x')$. We then search the bucket d_i for that fingerprint. If the fingerprint $h_i(x')$ is found then with high probability we have a match of the test item x'. If the fingerprint is not found we know with certainty that the element x' is not a member of the support.

Deletions are done in similar fashion. Given an item x' to delete, first the overflow dictionary is searched and, if x' is found, it is removed. If the key x' is not found in the overflow dictionary than the randomised dictionaries are consulted. If the key's fingerprint is found then we assume a match and the key is removed from the bucket d_i. Again there is a chance that we have a false positive and will therefore delete the wrong element.

A variation of deletion (and insertion) is an *update* where the count of an element that is already contained within the dynamic Bloomier filter is modified. When updating the model we essentially follow the algorithm for deletions in the paragraph above but adjust the associated value of the key if it is found instead of removing. (This is unlike inserts where we assume a collision if the element is already contained within the randomised dictionaries). There is still probability that an update will occur on the wrong element due to a false positive error.

3.3 Language Model Implementation

In the above section we described the dynamic Bloomier filter with no reference to language modeling. In this section we show how a dynamic Bloomier filter LM can use an unbounded input stream of pairs of n-grams and their counts in small space. (We use the terms dynamic Bloomier filter and ORLM interchangeably henceforth.)

Before any online updating is done, the (empty) ORLM is initialized with a stream of n-grams and counts extracted from a large corpus as with a traditional LM. This is called "seeding" the LM. When seeding the ORLM we follow the insertion procedure described in the proceeding section. Duplicate fingerprints in the randomised buckets

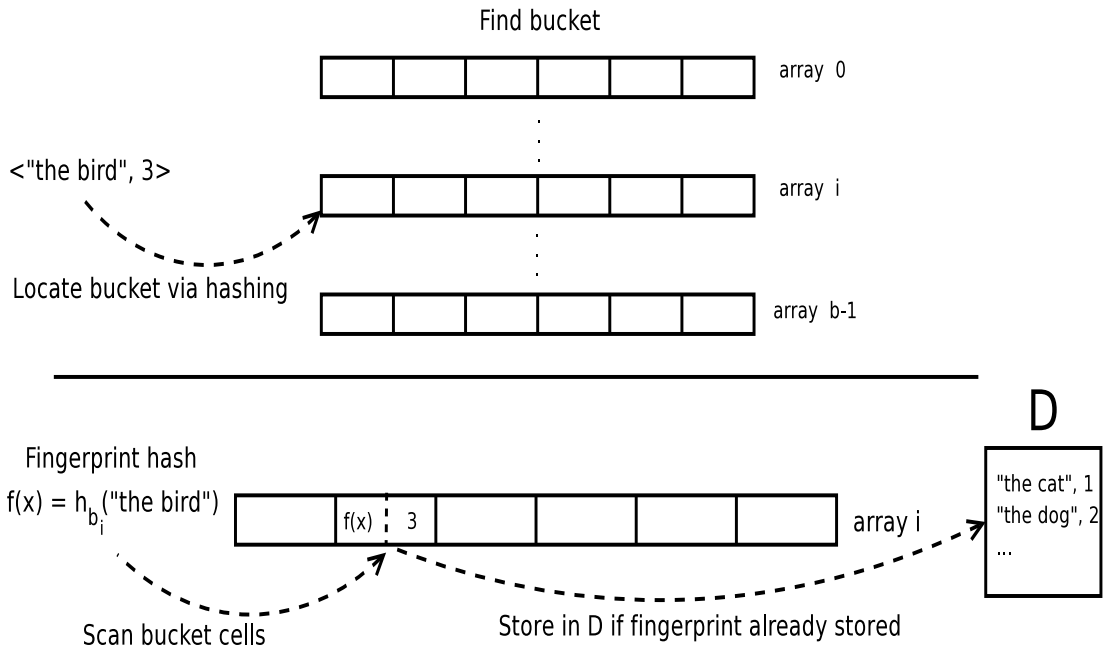


Figure 3.2: Inserting an *n*-gram into the dynamic Bloomier filter. Above: an *n*-gram is hashed to its target bucket. Below: the fingerprint for the *n*-gram is generated and the target bucket is scanned. If a collision occurs that *n*-gram is diverted to the overflow dictionary; otherwise the fingerprint is stored in the bucket.

are considered *collisions* and the *n*-gram causing the collision is passed along to the overflow dictionary. After seeding we have a perfect hash of the seed corpus and no errors can occur during test time for *n*-grams in the ORLM.

During seeding a perfect hash is possible since it is known a priori each *n*-gram in the initial stream is unique and all collisions are handled accordingly. Collisions, though, are produced by the same mechanism that generates the false positive errors—duplicate fingerprints generated by the hash functions for nonmatching *n*-grams. Hence when we have no uniqueness guarantees and update the ORLM with a batch of recent *n*-grams from the stream, identical fingerprints are taken to be *n*-gram matches instead of collisions. This means we may update the wrong *n*-gram. We describe in detail how we handle stream updates to the ORLM in Section 3.4. The remainder of this section deals with the base properties of the ORLM. Here we describe the implementation choice for the data structures used for the randomised and overflow dictionaries and how they are used.

For the randomised dictionaries $d_0, \ldots, d_{r-1}$ we simply used a large bit array that is divided into r sub-arrays. Each sub-array d_i, $i \in [0, r-1]$ forms a single bucket that

contains a constant and equal number of array cells. The size of each cell, the number of bits used for each n-gram fingerprint, we denote by w. w is the primary parameter for memory used by the ORLM and also determines the error rate as we will show. The hash functions $\{h\}$ and ϕ for fingerprint generation and target bucket allocation respectively are from random universal hash function (UHF) families described in Chapter 2.6.

The details for inserting an n-gram into our array-based model are shown in pseudocode in Algorithm 2. As described above, initially we have the empty data structures $d = \{d_0, \ldots, d_{r-1}\}$, an empty array conceptually divided into r buckets, $\bar{d}$ for the overflow dictionary, V for the associated value array, and $BCnt$ is an array of size r that keeps track of the number of bucket cells in each d_i, $i \in [0, r-1]$. Each bucket d_i contains B cells. We instantiate random hash function families ϕ and $h = \{h_0, \ldots, h_{r-1}\}$. To insert an n-gram x into the ORLM, we first find the target bucket in d using the top level hash function $i \rightarrow \phi(x)$. Then the n-gram's fingerprint $fp \rightarrow h_i(x)$ is generated using the hash function h_i attached to the bucket d_i. This produces a fingerprint fp whose value is in the range $h_i(x) \rightarrow [0, 2^w - 1]$. If the bucket d_i is already full, which we can find by checking $BCnt[i]$, we store the n-gram x in the overflow dictionary $\bar{d}$. Otherwise we find the $firstRow$ and $lastRow$ of the target bucket d_i and linearly scan each cell. If the fingerprint $h_i(x)$ is already in the bucket d_i the original n-gram x is stored exactly in the overflow dictionary $\bar{d}$. After searching, if the bucket d_i does not already contain the fingerprint $h_i(x)$ it is inserted into into the first available empty cell found during the traversal of d_i. In Algorithm 2 we denote this cell using the $index$ variable. We insert the fingerprint fp into the empty cell at $d[index]$ and the corresponding value v in $V[index]$. Finally we increment the bucket counter array $BCnt[i]$.

Note that any dynamic data structure can be employed for the overflow dictionary $\bar{d}$. A good choice for this is a sparse hash map that can be queried in constant time and requires little space. [1]

The choice of an array implementation for the randomised dictionaries is intuitive and useful for a few reasons. An array is simple to implement, space efficient, uses constant memory, and operations over it are fast. This is important since, when inserting, we are required to scan all cells of a bucket to ensure no other n-gram encoded in that bucket has a matching fingerprint. Since the fingerprints are unordered, using an array requires a linear scan of each cell in the bucket. Searching over all the cells in

[1] We make use of an open source implementation of a space efficient hash table that can be found at `http://code.google.com/p/google-sparsehash/`.

Algorithm 2: ORLM Online Insert

input: S (stream with keys x, values v)
$d \leftarrow$ empty set of buckets;
$V \leftarrow$ empty values;
$\bar{d} \leftarrow$ empty overflow dictionary;
$\phi, \{h_0, \ldots, h_{r-1}\} \leftarrow$ universal hash functions;
$B \leftarrow$ bucket size;
$BCnt \leftarrow$ empty bucket counter array;
forall $(x, v) \in S$ **do**
  $i \leftarrow \phi(x)$; // bucket hash
  $fp \leftarrow h_i(x)$; // finger hash
  $index = -1$;
  if $BCnt[i] == B$ **then** // d_b full
    $\bar{d}[x] \leftarrow v$;
    return;
  end
  $firstRow = i \times B$;
  $lastRow = firstRow + B$;
  for $b \leftarrow firstRow$ **to** $lastRow$ **do**
    if $d[b] == fp$ **then**
      $\bar{d}[x] \leftarrow v$;
      return;
    end
    if $d[b] == 0$ *and* $index == -1$ **then**
      $index = b$;
    end
  end
  $d[index] = x$;
  $V[index] = v$;
  $BCnt[i] = BCnt[i] + 1$
end

a bucket takes worst-case constant time where the constant is measured by how many cells are allocated to each bucket. Using an array also mimics the implementation of the batch-based Bloomier filter LM which makes it feasible to directly compare the extra memory required for the overflow dictionary $\bar{d}$ (Figure 3.4).

LM	Expected	Observed	RAM
Lossless	0	0	7450MB
Bloom	0.0039	0.0038	390MB
Bloomier	0.0039	0.0033	640MB
ORLM	0.0039	0.0031	705MB

Table 3.1: Example false positive rates and corresponding memory usage for randomised LMs. The table compares expected versus observed false positive rates for the Bloom filter, Bloomier filter, and ORLM obtained by querying a model of approximately 280M events with 100K unseen n-grams. We see the bit-based Bloom filter uses significantly less memory than the cell-based alternatives and the ORLM consumes more memory than the batch Bloomier filter LM for the same expected error rate.

As with other RLMs, before being stored the counts associated with the n-grams are first quantized via a log-based quantization codebook. When queried, the original counts are recovered to their nearest binned value. Storing the quantized counts can be done by the overlaying technique used in the batch Bloomier filter LM (Section 2.6) However, as we will show in the analysis below, overlaying the quantised values on their keys results in significantly higher error than storing the keys and values separately. Hence we get better performance by allocating distinct bits for the n-gram values at the expense of using slightly more memory.

Simplified pseudocode for querying the dynamic Bloomier filter is shown in Algorithm 3. Given the structures initialized in Algorithm 2, we first check where a test n-gram x' is in the overflow dictionary $\bar{d}$. If so, we return the associated value and are finished. Else, we repeat the procedure from Algorithm 2 and select a target bucket $i \rightarrow \phi(x')$ using the top level hash function, generate a fingerprint using the hash function associated with the ith bucket, $fp \rightarrow h_i(x')$, and scan each cell of the bucket d_i from $firstRow$ to $lastRow$. If we find the fingerprint fp we return its corresponding value from the associated value array V. Else, when we reach the $lastRow$ entry of the bucket d_i we know that the n-gram x' is not currently encoded in the model.

In practice, as with the other RLMs described in Section 2.6 we use *subsequence* filtering to provide sanity checks and further reduce the error rate of the model. Essentially, this technique breaks the test n-gram into smaller parts and queries the model for each part separately. Once it is known independently which parts of the n-gram are encoded in the model the existence of the full n-gram is established. The more pieces

Algorithm 3: Querying for a *n*-gram *k* in the ORLM during test time.

input : key x'
output: value v or 0
have : $d, \bar{d}, V, \phi, h, BCnt$ (from Algorithm 2)
$v = 0$;
if $x' \in \bar{d}$ **then**
  $v \leftarrow$ *value associated with* $\bar{d}[k]$;
else
  $i \leftarrow \phi(x')$; // bucket hash
  $fp \leftarrow h_i(x')$; // finger hash
  $v = 0$;
  $firstRow = i \times B$;
  $lastRow = firstRow + B$;
  for $b \leftarrow firstRow$ **to** $lastRow$ **do**
    if $d[b] == fp$ **then**
      $v = V[b]$;
      break;
    end
  end
end
return v;

the original *n*-gram is split into the more strenuous the checks and, consequently, the lower the level of error. In the simplest case we know *a priori* that a trigram (x_0, x_1, x_2), for example, is in the model only if the bigram (x_0, x_1) and the unigram (x_0) exist in the model already. Typically smoothing algorithms start with the full *n*-gram and query for shorter histories only if the original *n*-gram is not found. However, by reversing this and querying the model starting with the unigram and building up to the full trigram we avoid unnecessary false positive errors.

3.3.1 Analysis

Here we analyze the performance and space requirements of our instantiation of the ORLM. We examine the false positive error rate as well as the memory requirements of the full model. Specifically, since the main difference in data structures between the batch Bloomier filter LM and the ORLM is the addition of the overflow dictionary

necessary for the online perfect hashing, we are especially interested in the trade-off between the space used by the array that comprises the randomised dictionaries and that used by the overflow dictionary.

False Positive Error Rate:

Suppose we create a new ORLM and provide parameters for the total memory for all randomised dictionaries $\{d\}$, the number of cells in each bucket $c = |d_i|$, and w, the number of bits each cell contains. Intuitively, the false positive error (and, subsequently, the collision) rates are a function of the parameters c and w since they are caused by duplicate fingerprints in a given bucket d_i. Given a fingerprint $h_i(x)$ of w bits, there are c cells in d_i which may potentially already encode the same fingerprint $h_i(y) = h_i(x), y \neq x$. As described in Section 2.6.2, the probability for a UHF generating a pairwise collision is well known to be 2^{-w}. To get the false positive error rate of the ORLM we must multiple that probability by the number of cells in each bucket c since each cell comparison is another independent chance for a fingerprint match. Hence, for n-grams x and y with $x \neq y$ we have the simple formulation for the false positive probability as

$$\Pr(h_i(x) = h_i(y)|c,w) = \min(\frac{c}{2^w}, 1). \tag{3.1}$$

The effect of the number of cells per bucket c on the false positive rate is shown in Figure 3.3. The tests in this figure were conducted over a stream of 1.25B n-grams from the Gigaword corpus (Graff, 2003). We set our space usage to match the 3.08 bytes per n-gram reported in Talbot and Brants (2008) and held out just over 1M unseen n-grams to test the error rates.

Memory Usage:

At the highest level, the perfect hash function succeeds online by associating each n-gram with only **one** cell in the randomised dictionaries rather than having it depend on cells (or bits) which may be shared by other n-grams as with other RLMs. Since each n-gram's encoding in the model uses its own set of distinct bits it is independent of all other events contained within the model. This means an n-gram cannot corrupt other n-grams when inserted, updated or deleted. This also means that we provably require more space for the ORLM than for the static Bloomier filter LM which shares its bits between elements (Chazelle et al., 2004; Mortensen et al., 2005). The *space* used by the dynamic Bloomier filter is comprised (primarily) of the space used for the randomised dictionaries to store n-grams and their values as well as the additional space

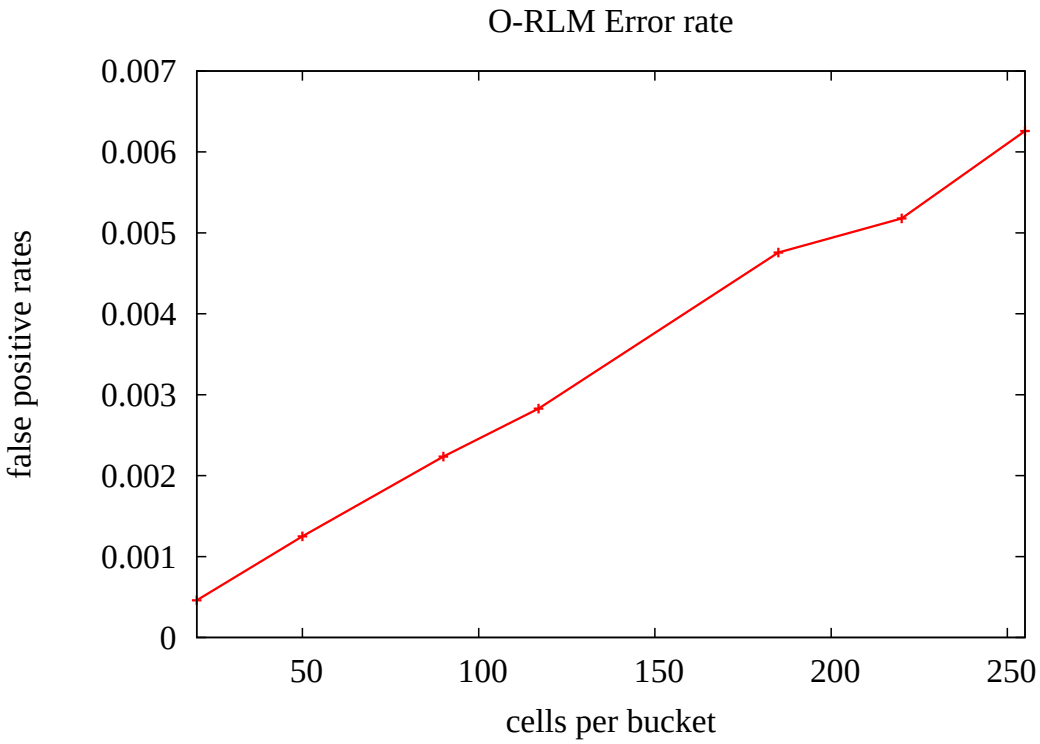


Figure 3.3: The ORLM error rises in correlation with the number of cells per bucket.

required by the overflow dictionary, the hash functions and the quantisation codebook.

There is a potentially a very large amount of memory needed for the hash functions when setting up the vanilla dynamic Bloomier filter. As stated, each bucket d_i, $i \in [0, r-1]$ in the r randomised dictionaries $d_0, \ldots, d_{r-1}$ requires its own associated hash function h_i to generate the fingerprints of the n-grams that reside in it. The number of hash functions required for the family $\{h\}$ is r if we naively set up the hash functions to be one-to-one with the randomised buckets. If r is reasonably large the hash functions will consume substantial memory.

To see this recall from Section 2.6 that the universal hash function formulation for n-grams is $h_i(x) = \sum_{j=1}^{n} a[i][j] * x[j] + b[i][j] \pmod{P}$ where j ranges over each word of the n-gram x, P is prime, and a and b are double arrays of randomly generated four byte integers. Clearly, given the maximum size of an n-gram is n, the space needed for the hash function family $\mathcal{H}$ is $\Omega(8nr)$ bytes since our space usage is linear in the bytes required for each hash function's arrays a and b, which must be able to handle n words each, and the number of buckets r. We can reduce the memory necessary for the hash functions by instantiating a much smaller number $r' \ll r$ of hash functions and associate hash function $h_{i'}$, $i' \in [0, r'-1]$ to bucket d_i where $i = \phi(x)$ is the assignment

from the bucket hash function and $i' = i \bmod r'$. This bit of engineering decreases the space usage for the hash functions by factor of $r - r'$. Since each bucket is independent of all others this has no effect on the analysis.

Overflow Dictionary:

The false positive/collision probability directly affects the size of the overflow dictionary $\bar{d}$. Recall that we store $S' = S \setminus \mathbf{S}$ in the overflow dictionary $\bar{d}$ where S' is the set of all n-grams S from the stream which generate fingerprint collisions with n-grams previously encoded in the randomised buckets. The probability for generating such a collision is the same as the probability for a false positive. To see this clearly we can view a false positive error as a type of collision with a test n-gram $x \in \mathbf{U} \setminus \mathbf{S}$ that is not in the support. We can bound the size of the overflow dictionary with high probability by bounding the expected number of collisions. This expectation is also the expected number of items in the overflow dictionary and an upper bound on the expected memory needed can be drawn. Assuming truly random hash functions, Mortensen et al. (2005) use a Chernoff bound for random variables with limited independence to show that the space required by the overflow dictionary is $O(s)$ bits with high probability where $s = |S|$ is the number of n-grams in the stream. Of course the exact memory used depends on the data structure used for the overflow dictionary but, to put this into perspective, if we were encoding an ORLM with one billion n-grams then we could expect to use a minumum of 120MB of extra space for the overflow dictionary.

Another practical factor that affects the size of the overflow dictionary is the relationship between the number of buckets r and the number of cells c per bucket. Given a constant number of total bits for all the randomised dictionaries there is an obvious tradeoff between r and c. Making the number of buckets r large means that the number of cells per bucket $c = |d_i|$ must be small and, since it is a multiplicative factor in our error probability, we would like to keep c as small as possible. However, due to the hash functions not being perfectly random in practice, Figure 3.4 shows what happens if we make the cells per bucket c too small. Many of the buckets fill up early on which diverts all other n-grams the bucket hash function ϕ sends to them to the overflow dictionary $\bar{d}$. This is less space efficient.

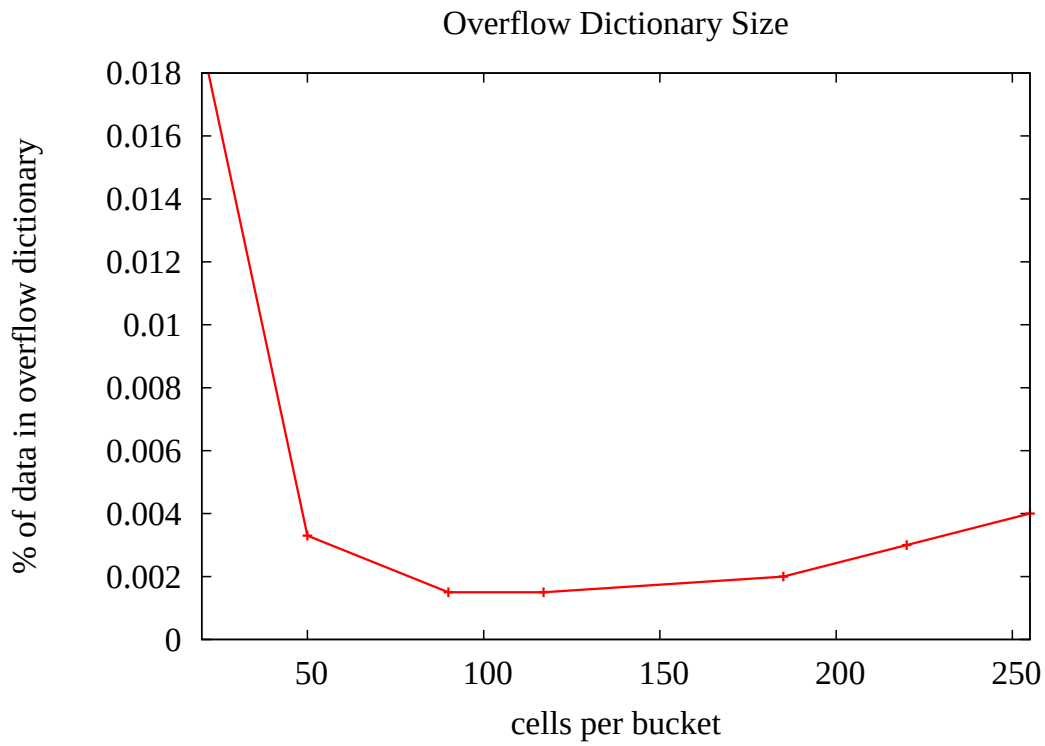


Figure 3.4: Too few cells per bucket causes a higher percentage of the data to be stored in the overflow dictionary due to full buckets. The tests in this figure were conducted with the same data as in Figure 3.3

3.3.2 Batch Bloomier Filter LM Comparison

Here we compare the ORLM with the batch Bloomier filter LM from Talbot and Brants (2008). As discussed, one of the main differences is that the ORLM requires more space than the batch Bloomier filter LM due to the space required by the overflow dictionary (Figure 3.3). However the batch Bloomier filter is not able to be incrementally retrained. This means to add any recent *n*-grams from the stream the batch Bloomier filter LM must be fully retrained. Obviously this is far slower than adding the new *n*-grams from the stream as they are encountered. Figure 3.5 compares the time needed to incrementally retrain the ORLM with varying amount of *n*-grams from the input stream compared to the time needed for batch retraining.

The false positive probability in Equation 3.1 is valid for the array-based ORLM setup when each *n*-gram fingerprint uses bits distinct from the bits used to store that *n*-gram's value. This is unlike the Bloomier filter where, as described in Section 2.6, each value is overlaid atop its attached *n*-gram by use of the XOR function. False positives occur in this setup when a test *n*-gram that was not encoded in the model returns a

random value that is less than the maximum encoded value V seen during training. We can use the same formulation for the ORLM which requires less space than using distinct bits to store the fingerprints and their corresponding values. This means for each test n-gram x we XOR the bucket-specific fingerprint $h_i(x)$ with each cell in d_i until we have a value less than V, the maximum value observed. This negatively effects the error rate/collision probability since, unlike the batch Bloomier filter, for each n-gram tested we query multiple times.

For each bucket d_i, given c and w as before and taking V as the maximum encoded value in the model, we have an error chance of

$$\Pr(h_i(x) \otimes \{d_i\} < V | c, w) = \min(\frac{cV}{2^w}, 1) \tag{3.2}$$

with $\{d_i\}$ denoting the set of cells contained in the bucket d_i. Since the maximum value V may be large, using this technique may seriously effect the overall error rate. Note we also introduce a new error type and lose the strict one-way error allowed by the model. As we scan a bucket we could encounter *mismatches* between n-grams that are members of the support when, for two random fingerprints, the value returned by the XOR operation returns a value $v \leq V$. The means we could potentially return the wrong values for existent n-grams in the model as well. Both of these reasons, higher error rates plus the introduction of mismatches, make the overlay technique used by the batch Bloomier filter for further space savings ill suited to the ORLM.

3.3.3 ORLM in a Batch SMT Setting

In this section we establish that the ORLM works as a traditional LM for SMT. We compare the translation performance of the ORLM against results from a lossless LM built using SRILM (Stolcke, 2002) and the batch Bloomier filter LM as described in Section 2.6.

For our experimental setup we use only publicly available resources. For decoding we used Moses (Koehn and Hoang, 2007). Our parallel data for the translation model was taken from the Spanish-English section of Europarl. For test material, we manually [2] translated 63 documents of roughly 800 sentences from three randomly selected dates (January 2nd, April 24, and August 19) spaced throughout the RCV1 corpus (Rose et al., 2002) timeline (Chapter 1). We held out 300 randomly selected

[2] As RCV1 is not a parallel corpus we translated the reference documents ourselves. This parallel corpus is available from the author.

Test Date	Lossless	Bloomier	ORLM
Jan	40.01	39.67	39.22
Apr	38.11	37.59	37.97
Aug	32.55	32.39	32.78
Avg	36.89	36.55	36.65

Table 3.2: Baseline comparison translation results (in BLEU) using data from the first stream epoch with a lossless LM (4.5GB RAM), the batch Bloomier filter LM and the ORLM (300MB RAM). All LMs are static.

sentences for minimum error rate training (Och, 2003) and optimised the parameters of the feature functions of the decoder for each experimental LM run.

Our initial tests consist of training the LMs using the data contained in the first stream epoch (Table 3.5) and then, mimicking batch LM behavior, translating all three test points in the stream with the static LMs. All the LMs were unpruned 5-gram models which used backoff interpolated smoothing. Modified Knesser-Ney smoothing (Chen and Goodman, 1999) was used for the lossless LM, which stores the *n*-gram probabilities and their backoff weights in memory, while both the Bloomier filter and ORLM, which store the quantised counts of the *n*-grams, used Stupid Backoff smoothing (Section 2.5.0.2). We set no space restriction on the lossless LM and the randomised LMs each had an error rate of $1/2^8$. Results are shown in Table 3.2. The translation results show the ORLM performs adequately as a batch LM since the results between all LMs are comparable. The lossless LM achieves slightly better performance overall due to the more sophisticated smoothing used.

So far in this section we analyzed the properties and compared the performance of the ORLM as a stand-alone LM. However, we have not as yet discussed how the ORLM is used for its designated purpose—to model and adapt online to an unbounded input stream. We survey our adaptation methods in the next section.

3.4 ORLM Adaptation

Now that we can adapt a LM in constant memory we discuss some simple adaptation approaches to adapting the ORLM with novel *n*-grams from the incoming data stream while keeping the probability and backoff models well-formed. There are many ways to do LM adaptation and we stress the methods we use in this section are only illustra-

Test Date	Severe	Random	Conservative
Jan	36.44	36.44	36.44
Apr	35.87	31.08	35.51
Aug	29.00	19.31	29.14
Avg	33.77	29.11	33.70

Table 3.3: Adaptation results measured in BLEU. Random deletions degrade performance when adapting a ORLM with error $= \frac{90}{2^{12}}$.

tive of what can be done using the ORLM.

In Chapter 1 we established that, given a memory constraint, using recent data is useful for language modeling in terms of perplexity when compared to a static model. However, we cannot hold the full, unbounded input stream in memory at any one time. We do not have access to the full stream at any given point and, as well, it is infeasible to allow unbounded memory for our model. Since we have the criteria of the ORLM operating within constant memory, to add new, potentially useful *n*-grams into the model from the incoming stream means we need a structured scheme for deleting old (preferably useless) *n*-grams to free space in the ORLM for *n*-grams from the incoming stream.

When processing the stream, we aggregate data for some consecutive portion, or *epoch*, of the input stream. We can vary the size of stream window. For example we might batch-up an hour or week's worth of material. Intuitively, smaller windows produce results that are sensitive to small variation in the stream's distribution while larger windows (corresponding to data over a longer time period) average out local spikes. Then we free bits in the ORLM for the most recent set of *n*-grams being considered for entry into the model. We considered the following update strategies:

1. **Conservative.** For each new *n*-gram encountered in the stream, insert it in the ORLM and remove one or more previously inserted *n*-gram which was never requested by the decoder. To preserve consistency we do not remove prefix grams: lower-order grams that are needed to estimate backoff probability for higher-order smoothing. If an *n*-gram from the incoming stream is already encoded in the ORLM, we keep the *n*-gram and add the new count to the previous one.

2. **Severe.** Differs from the conservative approach above only in that we delete *all* unused *n*-grams (i.e. all those not requested by the decoder in the previous

Epoch	Stream Window
1	20/08/1996 to 01/01/1997
2	02/01/1997 to 23/04/1997
3	24/04/1997 to 18/08/1997

Table 3.4: The RCV1 stream timeline is divided into the windowed epochs shown in this table for our translation experiments.

translation task) from the ORLM before adapting with data from the stream. This means the data structure is sparsely populated for all adaptation runs.

3. **Random.** Randomly sample the incoming stream and for each previously unseen *n*-gram encountered, insert it and remove some previously inserted *n*-gram. Deletion occurs randomly and irrespective of whether the old *n*-gram was ever requested by the decoder or is a prefix. This approach corrupts the well-formedness of the underlying model.

Table 3.3 shows translation results in BLEU for these strategies. (We will describe the stream-based translation experiment details further in the next section.) Clearly, by using the random sampling strategy we take a significant performance hit whereas both the conservative and severe approaches have approximately the same performance. We describe the novel techniques that allow for high performance LM adaptation below.

The above conservative and severe strategies rely heavily on keeping the ORLM well-formed by not removing lower order *n*-grams that comprise prefixes for higher-order *n*-grams. As the random deletions demonstrated, removing prefix *n*-grams indiscriminately eventually corrupts the model. In addition, since we employ subsequence filtering to lower the error rates, by removing prefixes of *n*-grams we may begin to return false negatives for good *n*-grams. Since the highest order grams comprise the bulk, by far, of the total number of elements in any LM, we can keep all *n*-gram prefixes and still remove enough *n*-grams to free space for new stream data.

To keep track of the prefixes of the model we add a bit array to the ORLM. Each bit of the prefix array has a one-to-one correspondence with a cell in the array of randomised dictionaries. When inserting into the ORLM whilst adapting, we verify that all prefixes of an *n*-gram are encoded in the model. If not, we add the full *n*-gram sequence and, for each prefix of the *n*-gram we set its corresponding bit in the prefix bit array. Then when deleting we do not remove any *n*-grams which have their prefix

bit set. By adding the prefix bit array we use slightly more memory but at one bit per *n*-gram it is trivial overall and we can keep the ORLM well-formed throughout adaptation.

While adapting with the conservative and severe methods we also do not delete any *n*-grams that have been requested by the decoder. This feedback helps ascertain what part of the incoming stream has value to each test points. To keep track of decoder requests we again employ a bit array that has a one-to-one correspondence with the cells of the randomised buckets. During test time, if the decoder requests an *n*-gram that is encoded in the model, we "turn on" the bit associated with that *n*-gram to track this. During the deletion stage while adapting we now have a small subset of the *n*-grams in the model marked for their usefulness. We do not delete these. After each adaptation period we clear the bit array and continue tracking decoder requests afresh. In this way we can keep the pertinent set of old *n*-grams that were useful in the last test point since they may be potentially useful in the next decoding run.

When a particular *n*-gram is targeted for an update its quantised value in the current model is retrieved, the old count updated with the new count, and then the new value quantised again before being encoded back into the ORLM. Since the quantisation scheme from Section 2.6.3 serves as bucketing function to group all values within some range, the interaction between the quantisation used for encoding the associated values of each *n*-gram and the stream-based updates may, at first, seem to counteract each other unless the new count from the stream for a *n*-gram is large. In fact this is not the case since we can specify our quantisation function to allow nearly exact representation of low counts and only to bucket ranges of higher counts. Hence, when we receive an update for a *n*-gram residing in the model with a low count it will always jump quantisation levels when updated and incremented by any value, even just a count of one. Only a tiny percentage of the *n*-grams in the model with very high counts (those at the head of the encoded Zipf distribution) will remain in the same quantisation level when updated with a new count that is small.

In summary, by using minimally more space (two bits per *n*-gram specifically) we can ensure the ORLM stays well-formed throughout adaptation and retains the set of *n*-grams shown to be useful to translating the source stream. The distribution of the incoming stream is taken into account despite the quantisation. We show now that these exceptionally simple techniques for adaptation produce surprisingly good results for stream-based translation.

Order	Full	Epoch 1	Epoch 3
1	1.25M	0.6M	0.7M
2	14.6 M	6.8M	7.0M
3	50.6 M	21.3M	21.7M
4	90.3 M	34.8M	35.4M
5	114.7M	41.8M	42.6M
Total	271.5M	105M	107.5M

Table 3.5: Distinct *n*-grams (in millions) encountered in the full stream and example epochs.

3.5 Stream-based Translation Experiments

In this section we demonstrate the usefulness of the ORLM for stream-based translation. Our target language text stream was generated from the RCV1 corpus and we use the same three test points from the initial translation experiments described in Section 3.3.3. These test dates effectively divided the stream into three *epochs* between them. The windowed timeline for these epochs is shown in Table 3.4.

After each test point in the stream we adapt the ORLM to the set of *n*-grams in the next epoch using the severe adaptation heuristic (Section 3.4). All *n*-grams from the stream and their counts are incorporated into the ORLM and then the next test point in the stream is translated. For comparison, for each test point we also batch retrained the Bloomier filter LM with the data contained in the epoch prior. Results are shown in Table 3.6.

As expected, since they use identical training data, performance is about equal between the Bloomier LM using batch retraining and the ORLM using the online perfect hash function. The key difference is that each time we batch retrain the Bloomier filter LM we must compute, offline, a perfect hash of the new training set. This is computationally demanding since the batch perfect hashing algorithm uses Monte Carlo randomisation which fails routinely and must be repeated often. To make the batch algorithm tractable the training data set must be divided into lexically sorted subsets as well which requires multiple extra passes over the data.

In contrast, the ORLM is incrementally retrained online and the order of the data is irrelevant. This makes it significantly more resource efficient since we find bits in the model for the *n*-grams dynamically without using more memory than we intially set.

	Batch Retrained Bloomier		ORLM	
Date	BLEU	Delta	BLEU	Delta
Jan	39.67	0	39.22	0
Apr	40.43	+2.84	40.63	+2.66
Aug	38.53	+6.14	38.26	+5.48
Avg	39.54	+2.99	39.37	+2.71

Table 3.6: Translation results for stream-based LMs in BLEU. The batch retrained Bloomier filter LM and the stream-based ORLM use 300MB each with equal error rates of $\frac{1}{2^8}$. The *Delta* columns show the differences in BLEU score between the static models from Table 3.2 and the streaming LMs.

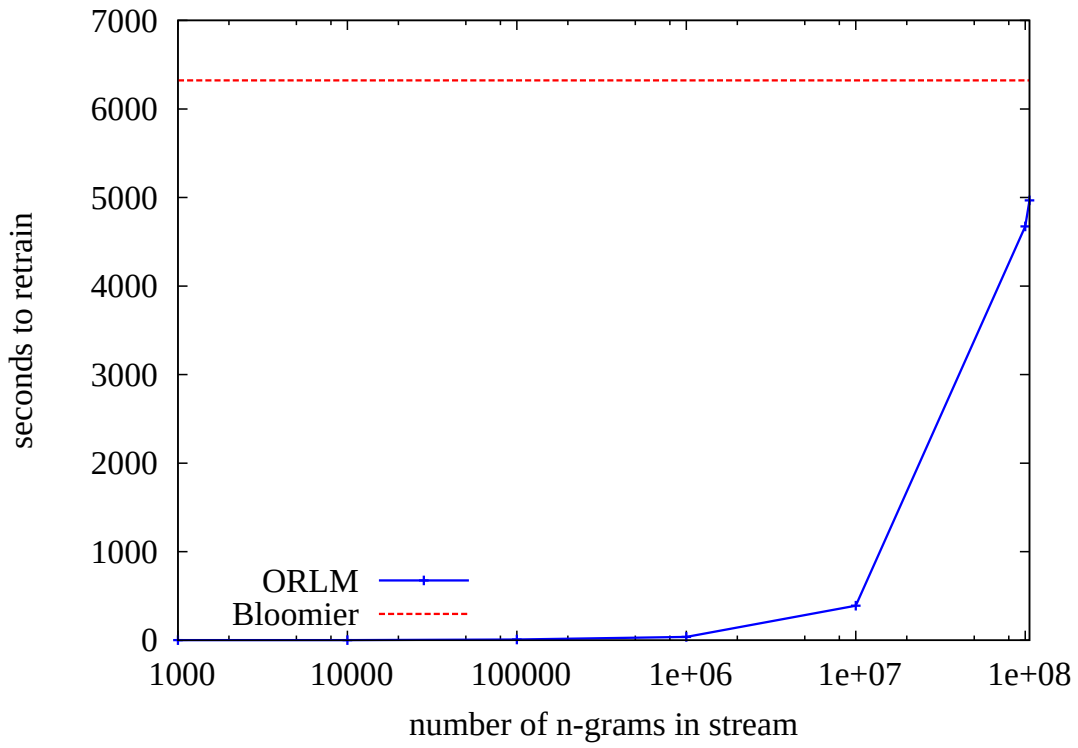


Figure 3.5: Time in seconds needed to add *n*-grams to the ORLM compared to (optimistic) batch retraining time as is necessary to add data into the batch Bloomier LM.

Retraining time differences between the batch and online LMs is shown in Figure 3.5.

3.6 Conclusion

In this chapter we introduced and analyzed our stream-based LM. We have shown the dynamic Bloomier filter can be used to encode an ORLM that effectively models an unbounded data stream over time in small space. Experiments show that the ORLM is efficient to retrain incrementally and alleviates the computational burden of frequent batch retraining while maintaining comparative translation performance with batch-based LMs. By adapting the LM we achieve significantly better streaming translation performance compared to a static LM. In Chapter 5 we describe a stream-based algorithm for the translation model and show how we can combine the ORLM and an online translation model for improved stream translation performance. In the next chapter we focus on how we can extend the ORLM to multiple incoming streams instead of restricting adaptation to a single in-domain input stream.

Chapter 4

Multiple Stream-based Translation

In the last chapter we introduced the ORLM, a stream-based LM, that is capable of adapting to an unbounded input data stream. Our experiments with the ORLM show that we can increase translation performance using an online LM that has access to recent data in the stream (potentially) relevant to the next test point. However, a drawback of the experiments reported in the previous chapter was the oversimplified scenario that all training and test data was drawn from the same distribution using a single, in-domain stream. In a real world scenario multiple incoming streams are readily available and test sets from dissimilar domains will be translated continuously. In this chapter we extend our work with the ORLM and consider the problem of *multiple* stream translation. We explore various strategies to model multiple unbounded streams within a single SMT system. The challenges in multiple-stream translation include dealing with domain differences, variable throughput rate of streams (the size of each stream per epoch), and the need to maintain constant space. Importantly, we impose the key requirement that our model match translation performance reached using the single stream approach on all test domains. The work presented in this chapter was previously published in Levenberg et al. (2011).

4.1 Overview

Recall that any source that provides a continuous sequence of natural language documents over time can be thought of as an *unbounded stream* which is time-stamped and access to it is given in strict chronological order. The ubiquity of technology and the Internet means there are many such text streams available already and their number is increasing quickly. For SMT, multiple text streams provide a potentially abundant

source of new training data that may be useful for combating model sparsity.

In Chapter 3 we described the streaming translation scenario where the LM is incrementally retrained by updating it arbitrarily often with previously unseen training data from a single stream. Then, when translating a recent test document drawn from an in-domain source stream, the LM contains potentially useful *n*-grams that aid in the translation. It is a gross oversimplification to assume that all test material for a SMT system will be from a single domain. In this chapter we report on using multiple incoming streams from variable domains to incrementally retrain our stream-based SMT system. Still of primary concern is building models whose space complexity is independent of the size of the incoming stream since allowing unbounded memory to handle unbounded streams is unsatisfactory. When dealing with more than one stream we must also consider how the properties of single streams interact in a multiple stream setting.

Every text stream is associated with a particular domain. For example, we may draw a stream from a newswire source, a daily web crawl of new blogs, or the output of a company or organisation. Obviously the distribution over the text contained in these streams will be very different from each other. As is well-known from the work on domain adaptation throughout the SMT literature, using a model from one domain to translate a test document from another domain would likely produce poor results.

Each stream source will also have a different rate of production, or *throughput*, which may vary greatly between sources. Blog data may be received in abundance but the newswire data may have a significantly lower throughput. This means that the text stream with higher throughput may dominate and overwhelm the more nuanced stream with less data in the LM during decoding. This is bad if we want to translate well for all domains in small space using a single model. These properties of multiple streams—high rate of throughput, unboundedness, and domain differences—and how they interact in a stream-based translation setting are what we tackle in the following sections.

4.2 Multiple Stream Retraining Approaches

In a stream-based translation setting we can expect to translate test points from various domains on any number of incoming streams. Our goal is a single LM that obtains equal performance in less space then when using a separate LM per stream. The underlying LMs could be exact, but here we use randomised versions based on the ORLM.

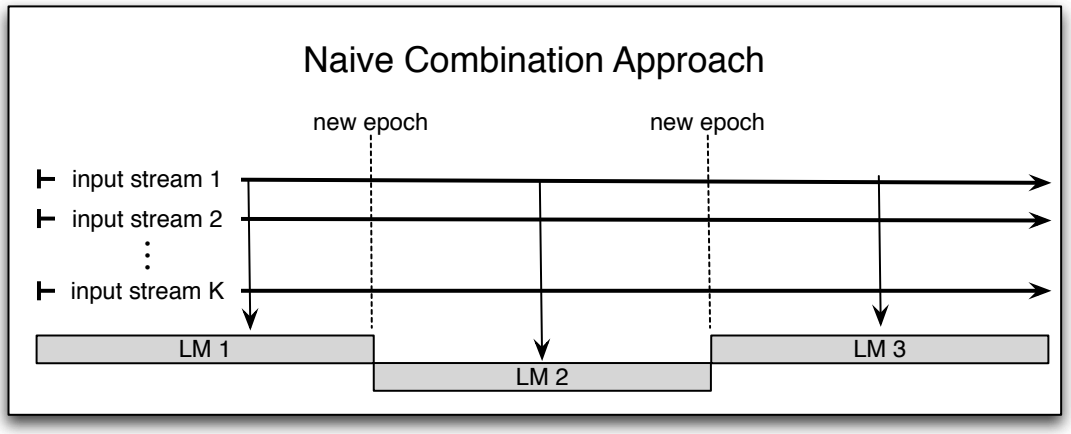


Figure 4.1: In the naive approach all K streams are simply combined into a single LM for each new epoch encountered.

Given an incoming number K of unbounded streams over a potentially infinite timeline T, with $t \subset T$ an *epoch* or windowed subset of the timeline, the full set of n-grams in all K streams over all T is denoted with S. By S_t we denote n-grams from all K streams and S_{kt}, $k \in [1,K]$, as the n-grams in the kth stream over epoch t. Since the streams are unbounded, we do not have access to all the n-grams in S at once. Instead we select n-grams from each stream $S_{kt} \subset S$. We define the collection of n-grams encoded in the LM at time t over all K streams as C_t. Initially, at time $t = 0$ the LM is composed of the n-grams in the stream so $C_0 = S_0$.

Since it is unsatisfactory to allow unbounded memory usage for the model and more bits are needed as we see more novel n-grams from the streams, we enforce a memory constraint and use an adaptation scheme to delete n-grams from the LM C_{t-1} before adding any new n-grams from the streams to get the current n-gram set C_t. Below we describe various approaches of updating the LM with data from the streams.

4.2.1 Naive Combinations

Approach The first obvious approach for an online LM using multiple input streams is to simply store all the streams in one LM. That is, n-grams from all the streams are only inserted into the LM once and their stream specific counts are combined into a single value in the composite LM.

Modelling the Stream In the naive case we retrain the LM C_t in full at epoch t using

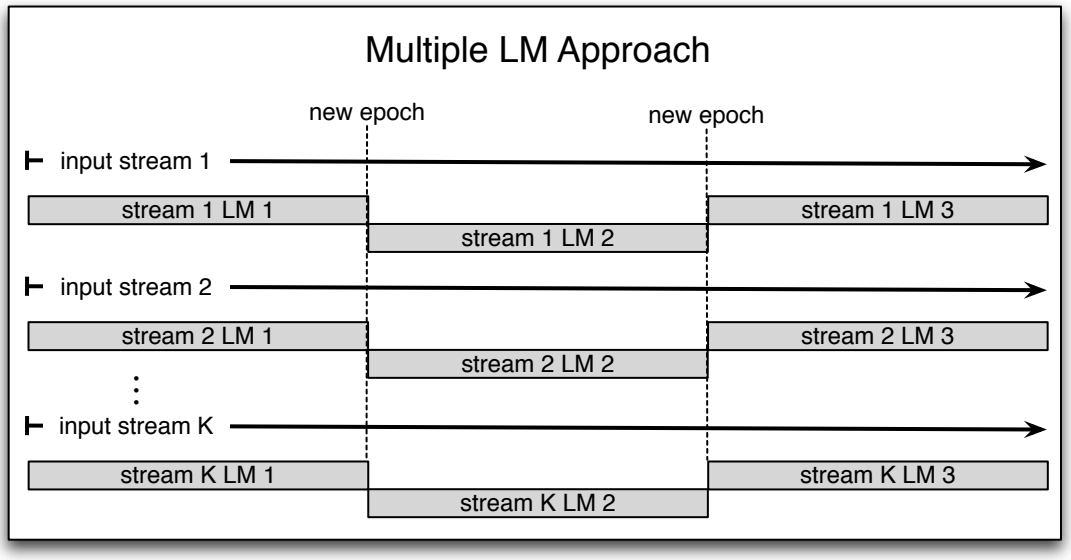


Figure 4.2: Each stream $1 \ldots K$ gets its own stream-based LM using the multiple LM approach.

all the new data from the streams. We have simply

$$C_t = \bigcup_{k=1}^{K} S_{kt} \tag{4.1}$$

where each of the K streams is combined into a single model and the n-grams counts are merged linearly. Here we carry no n-grams over from the LM C_{t-1} from the previous epoch. The space needed is the number of unique n-grams present in the combined streams for each epoch.

Resulting LM To query the resulting LM C_t during decoding with a test n-gram $w_i^n = (w_i, \ldots, w_n)$ we use Stupid Backoff (Section 2.5.0.2). Each stream provides a distribution over the n-grams contained in it and, for SMT, if a *separate* LM was constructed for each domain it would most likely cause the decoder to derive different 1-best hypotheses than using a LM built from all the stream data. Using the naive approach blurs the distribution distinctions between streams and negates any stream specific differences when the decoder produces a 1-best hypothesis. It has been shown that doing linear combinations of this type produces poor performance in theory (Mansour et al., 2008). In our experiments we demonstrate that naive combination does not perform well in practice. Given that the throughput of the streams differ greatly naive combination will degrade the translation performance for test domain from a stream with a much lower rate since the returned probabilities from the LM are dominated by the higher rate stream.

4.2.2 Weighted Interpolation

Approach An improved approach to using multiple streams is to build a separate LM for each stream and using a weighted combination of each during decoding. Each stream is stored in isolation and we interpolate the information contained within each during decoding using a weighting on each stream.

Modelling the Stream Here we model the streams by simply storing each stream at time t in its own LM so $C_{kt} = S_{kt}$ for each stream S_k. Then the LM after epoch t is

$$C_t = \{C_{1t}, \ldots, C_{Kt}\}.$$

We use more space here than all other approaches since we must store each n-gram/count occurring in each stream separately as well as the overhead incurred for each separate LM in memory.

Resulting LM During decoding, the probability of a test n-gram w_i^n is a weighted combination of all the individual stream LMs. We can write

$$P_t(w_i^n) := \sum_{k=1}^{K} f_k P_{C_{kt}}(w_i^n) \tag{4.2}$$

where we query each of the individual LMs C_{kt} to get a score from each LM using Stupid Backoff and combine them together using a weighting f_k specific to each LM. Here we impose the restriction on the weights that $\sum_{k=1}^{K} f_k = 1$. (We discuss specific weight selections in the next section.)

By maintaining multiple stream specific LMs we can maintain the particular distribution of the individual streams and keep the more nuanced translations from the lower throughput streams available during decoding without translations being dominated by a stream with higher throughput. From the learning perspective, interpolating multiple LMs can be seen as a type of *ensemble learning* where each of the individual LMs is considered a "weak" learner to the final combined LM used for the SMT system. However using multiple distinct LMs is wasteful of memory.

4.2.3 Combining Models via History

Approach We want to combine the streams into a single LM using less memory than but still achieving at least as good a translation for each test point as when storing each stream separately. Naively combining the streams removes stream specific translations but, from Chapter 3, using the history of n-grams selected by the decoder during the

previous test point in the single stream case obtained good results. This is applicable to the multi-stream case as well.

Modelling the Stream For multiple streams and epoch $t > 0$ we model the stream combination as

$$C_t = f_T(C_{t-1}) \cup \bigcup_{k=1}^{K} (S_{kt}). \tag{4.3}$$

where for each epoch a selected subset of the previous *n*-grams in the LM C_{t-1} is merged with all the newly arrived stream data to create the new LM set C_t. The parameter f_T denotes a function that filters over the previous set of *n*-grams in the model. It represents the specific adaptation scheme employed and stays constant throughout the timeline T. In this work we consider any *n*-grams queried by the decoder in the last test point as potentially useful to the next point. Since all of the *n*-grams S_t in the stream at time t are used the space required is of the same order of complexity as the naive approach.

Resulting LM Since all the *n*-grams from the streams are now encoded in a single LM C_t we can query it using Stupid Backoff during decoding. The goal of retraining using decoding history is to keep useful *n*-grams in the current model so a better model is obtained and performance for the next translation point is improved. Note that making use of the history for hypothesis combination is theoretically well-founded and is the same approach used here for history based combination. (Mansour et al., 2008)

4.2.4 Subsampling

Approach The problem of multiple streams with highly varying throughput rates can be seen as a type of class imbalance problem in the machine learning literature. Given a binary prediction problem with two classes, for instance, the imbalance problem occurs when the bulk of the examples in the training data are instances of one class and only a much smaller proportion of examples are available from the other class. A frequently used approach to balancing the distribution for the statistical model is to use *random under sampling* and select only a subset of the dominant class examples during training (Japkowicz and Stephen, 2002).

This subsampling approach is applicable to the multiple stream translation problem with imbalanced throughput rates between streams. Instead of storing the *n*-grams from each stream separately, we can apply a subsampling selection scheme directly to the incoming streams to balance each stream's contribution in the final LM. Note that subsampling is also related to weighting interpolation. Since all returned LM scores

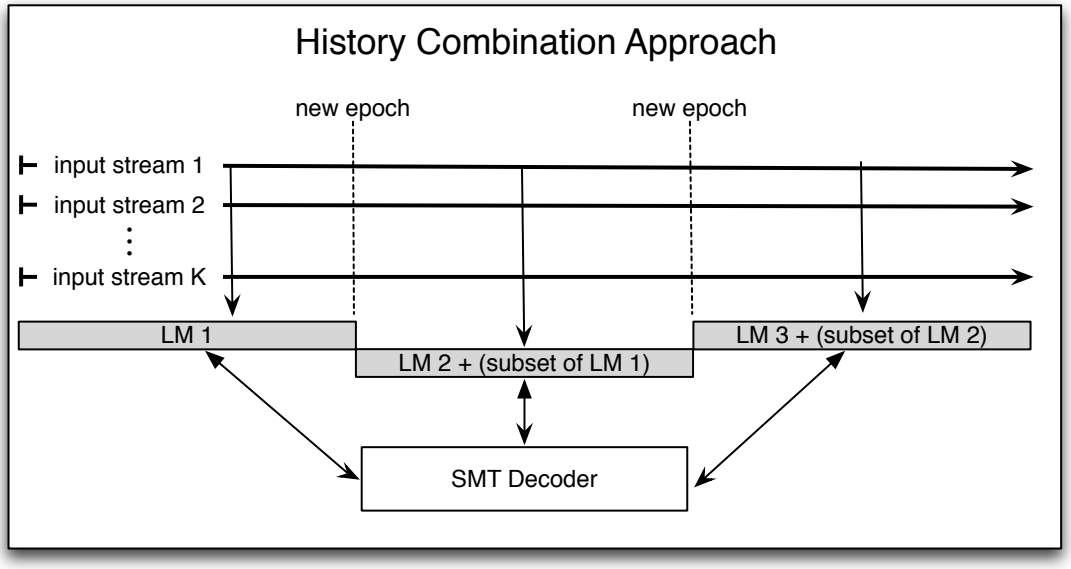


Figure 4.3: Using decoding history all the streams are combined into a unified LM.

are based on frequency counts of the *n*-grams and their prefixes, taking a weighting on a full probability of an *n*-gram is akin to having fewer counts of the *n*-grams in the LM to begin with.

Modelling the Stream To this end we use the weighted function parameter f_k from Equation 4.2 to serve as the sampling probability rate for accepting an *n*-gram from a given stream *k*. The sampling rate serves to limit the amount of stream data from a stream that ends up in the model. For $K > 1$ we have

$$C_t = f_T(C_{t-1}) \cup \bigcup_{k=1}^{K} f_k(S_{kt}) \tag{4.4}$$

where f_k is the probability a particular *n*-gram from stream S_k at epoch t will be included in C_t. The adaptation function f_T remains the same as in Equation 4.3. The space used in this approach is now dependent on the rate f_k used for each stream. Obviously lower acceptance rates require less space since fewer *n*-grams are (randomly) included in the final LM.

Resulting LM Again, since we obtain a single LM from all the streams, we use Stupid Backoff smoothing to get the probability of an *n*-gram during decoding.

The subsampling method is applicable to all of the approaches discussed in this section. However, since we are essentially limiting the amount of data that we store in the final LM we can expect to take a performance hit based on the rate of acceptance given by the parameters f_k. In the next section we show that by using subsampling

Stream	1-grams	3-grams	5-grams
EP	19K	520K	760K
GW (xie)	120K	3M	5M
RCV1	630K	21M	42M

Table 4.1: Sample statistics of unique *n*-gram counts from the streams from epoch 2 of our timeline. The *throughput* rate varies a lot between streams.

with the history combination approach we obtain good performance for all streams in small space.

4.3 Experiments

Given the approaches for multiple stream combination just described we show via experiments here how each affects translation performance and describe the system requirements in practice. We begin by showing that naively combining streams from different domains produces poor average results but also show that using information from an out-of-domain stream can help in the translation of a stream with low throughput. After establishing this we aim to get the same improved results with minimal space requirements.

4.3.1 Experimental Setup

The SMT setup we employ is standard and all resources used are publicly available. We translate from Spanish into English using phrase-based decoding again with Moses (Koehn and Hoang, 2007) as our decoder. Our parallel data came from Europarl.

We use three streams (all are timestamped): RCV1, Europarl (EP), and Gigaword (GW) (Graff et al., 2007). GW is taken from six distinct newswire sources but in our initial experiments we limit the incoming stream from Gigaword to one of the sources (xie). GW and RCV1 are both newswire domain streams with high rates of incoming data whereas EP is a more nuanced, smaller throughput domain of spoken transcripts taken from sessions of the European Parliament. As described previously, the RCV1 corpus only spans one calender year from October, 1996 through September, 1997 so we selected only data in this time frame from the other two streams so our timeline consists of the same full calender year for all streams.

For this work we used the ORLM from the previous section. Recall the crux of the ORLM is an online perfect hash function that provides the ability to insert and delete from the data structure. Consequently the ORLM has the ability to adapt to an unbounded input stream whilst maintaining both a constant error rate and memory usage. The ORLMs were all 5-grams with the training data coming from the streams discussed above and Stupid Backoff smoothing used for *n*-gram scoring. Here again our results are reported using the BLEU metric.

For testing we held-out three random test points from both the RCV1 and EP stream's timeline for a total of six test points. This divided the streams into three epochs, and we updated our online LM using the data encountered in the epoch prior to each translation point. The *n*-grams and their counts from the streams are combined in the LM using one of the approaches from the previous section. [1]

Using the notation from Section 4.2 we have the RCV1, EP, and GW streams described above and $K = 3$ as the number of incoming streams from two distinct domains (newswire and spoken dialogue). Our timeline T is one year's worth of data split into three epochs, $t \in \{1,2,3\}$, with test points at the end of each epoch t. Since we have no test points from the GW stream it acts as a background stream for these experiments. [2]

4.3.2 Baselines and Naive Combinations

In this section we report on our translation experiments using a single stream and the naive linear combination approach with multiple incoming data streams from Section 4.2.1.

Using the RCV1 corpus as our input stream we tested single stream translation first. Here we are replicating the experiments from Section 3 so both training and test data comes from a single in-domain stream. Results are in Table 4.2 where each row represents a different LM type. *RCV1 (Static)* is the traditional baseline with no adaptation where we use the training data for the first epoch of the stream. *RCV1 (Online)* is the online LM adapted with data from the in-domain stream. For the reasons previously described we get improvements when using an online LM that incorporates recent data against a static baseline.

[1] A note on the significance of the results in this chapter. Our aim here is to *match* single-stream baselines while combining streams from multiple domains into a single succinct LM. In the context of significance testing the baselines are the null hypothesis and, since we are not trying to achieve gains against these, significance testing is out of context for our goals. Any improvement to translation (as measured in BLEU) is incidental to our approaches.

[2] A background stream is one that only serves as training data for all other test domains.

LM Type	Test 1	Test 2	Test 3
RCV1 (Static)	39.30	38.28	33.06
RCV1 (Online)	39.30	40.64	39.19
EP (Online)	30.22	30.31	26.66
RCV1+EP (Online)	39.00	40.15	39.46
RCV1+EP+GW (Online)	**41.29**	**41.73**	**40.41**

Table 4.2: Results for the RCV1 test points. RCV1 and GW LMs are in-domain LMs and EP is out-of-domain. Translation results are improved using more stream data since most *n*-grams are in-domain to the test points.

LM Type	Test 1	Test 2	Test 3
EP (Static)	**42.09**	44.15	36.42
EP (Online)	**42.09**	**45.94**	**37.22**
RCV1 (Online)	36.46	42.10	32.73
EP+RCV1 (Online)	40.82	44.07	35.01
EP+RCV1+GW (Online)	40.91	44.05	35.56

Table 4.3: EP results using in-domain and out-of-domain streams. The last two rows show the naive combination approach and get poor results compared to single stream approaches since the large amount of out-of-domain data hurts translation performance for the EP test points.

We then ran the same experiments using a stream generated from the EP corpus. EP consists of the proceedings of the European Parliament and is a significantly different domain than the RCV1 newswire stream. We updated the online LM using *n*-grams from the latest stream epoch before translating each in-domain EP test set. Results are in Table 4.3 and follow the same naming convention as Table 4.2 (except now in-domain is EP and out-of-domain is RCV1).

Using a single stream we also cross tested and translated each test point using the online LM adapted on the out-of-domain stream. As expected, translation performance decreases (sometimes drastically) in this case since the data of the out-of-domain stream are not suited to the domain of the current test point being translated.

We then tested the naive approach and combined both streams into a single LM by taking the union of the *n*-grams and adding their counts together. This is the *RCV1+EP*

(Online) row in Tables 4.2 and 4.3 and clearly, though it contains more data compared to each single stream LM, the naively combined LM does not help the RCV1 test points much and degrades the performance of the EP translation results. This translation hit occurs as the throughput of each stream is significantly different. The EP stream contains far less data per epoch than the RCV1 counterpart (see Table 4.1) hence using a naive combination means that the more abundant newswire data from the RCV1 stream overrides the probabilities of the more domain specific EP *n*-grams during decoding.

When we added a third newswire stream from a portion of GW, shown in the last row of Tables 4.2 and 4.3, improvements are obtained for the RCV1 test points due to the addition of in-domain data but the EP test performance still suffers.

This highlights why naive combination is not satisfactory. While using more in-domain data aids in the translation of the newswire tests, for the EP test sets, when we naively combine the *n*-grams from all streams, the hypotheses the decoder selects are weighted heavily in favour of the out-of-domain data. This is because the out-of-domain streams throughput is significantly larger and swamps the model.

4.3.3 Interpolating Weighted Streams

Straightforward linear combinations of the streams into a single LM results in degradation of translations for test points whose in-domain training data is drawn from a stream with lower throughput than the other data streams. We could maintain a separate MT system for each streaming domain but intuitively some combination of the streams may benefit average performance since using all the data available should benefit those test points from streams with low throughput. To test this we used an alternative approach described in Section 4.2.2 and used a weighted combination of the single stream LMs during decoding.

We tested this approach using our three streams: RCV1, EP and GW (xie). We used a separate ORLM for each stream and then, during testing, the result returned for an *n*-gram queried by the decoder was a value obtained from some weighted interpolation of each individual LM's score for that *n*-gram. To get the interpolation weights for each streaming LM we minimised the perplexity of all the models on held-out development data from the streams. [3] Then we used the corresponding stream specific weights to decode the test points from that domain.

Results are shown in Tables 4.4 and 4.5 using the weighting scheme described

[3]The lossy nature of the encoding of the ORLM means that the perplexity measures were approximations. Nonetheless the weighting from this approach had the best performance.

Weighting	Test 1	Test 2	Test 3
$1.0_R + 0.0_E + 0.0_G$	*39.30*	*40.64*	*39.19*
$.33_R + .33_E + .33_G$	38.97	39.78	35.66
$.50_R + .25_E + .25_G$	39.59	40.40	37.22
$.25_R + .50_E + .25_G$	36.57	38.03	34.23
$.70_R + 0.0_E + .30_G$	**40.54**	**41.46**	**39.23**

Table 4.4: Weighted LM interpolation results for the RCV1 test points where $E =$ Europarl, $R =$ RCV1, and $G =$ Gigaword (xie). The first row is the single-stream baseline and we compare the results using perplexity based weight optimization in the final row with various other random weightings.

Weighting	Test 1	Test 2	Test 3
$1.0_E + .0.0_R + 0.0_G$	*42.09*	*45.94*	*37.22*
$.33_E + .33_R + .33_G$	40.75	45.65	35.77
$.50_E + .25_R + .25_G$	41.46	46.37	36.94
$.25_E + .50_R + .25_G$	40.57	44.90	35.77
$.70_E + .20_R + .10_G$	**42.47**	**46.83**	**38.08**

Table 4.5: EP results in BLEU for the interpolated LMs. We easily match single-stream results (first row) using some information from the OOD stream and an optimal weighting between the LMs (final row). We show other random weightings for comparison.

above plus a selection of random parameter settings for comparison. Using the notation from Section 4.2.2, a caption of "$.5_R + .25_E + .25_G$", for example, denotes a weighting of $f_{RCV1} = 0.5$ for the scores returned from the RCV1 stream LM while f_{EP} and $f_{GW} = 0.25$ for the EP and GW stream LMs.

The weighted interpolation results suggest that while naive combination of the streams may be misguided, average translation performance can be improved upon when using more than a single in-domain stream. Comparing the best results in Tables 4.2 and 4.3 to the single stream baselines in Tables 4.4 and 4.5 we achieve comparable, if not improved, translation performance for *both* domains. This is especially true for test domains such as EP which have low training data throughput from the stream. Here adding some information from the out-of-domain stream that contains a lot more data aids significantly in the translation of in-domain test points.

LM Type	Test 1	Test 2	Test 3
Single-stream	*39.30*	*40.64*	*39.19*
Multi-f_k	41.19	41.73	39.23
Multi-f_T	**41.29**	42.23	**40.51**
Multi-$f_k + f_T$	41.19	**42.52**	40.12

Table 4.6: RCV1 test results using history and subsampling approaches. The top row is the single-stream baseline.

LM Type	Test 1	Test 2	Test 3
Single-stream	*42.09*	*45.94*	*37.22*
Multi-f_k	**40.91**	43.50	36.11
Multi-f_T	**40.91**	47.84	**39.29**
Multi-$f_k + f_T$	**40.91**	**48.05**	39.23

Table 4.7: Europarl test results with history and subsampling approaches.

However, the optimal weighting scheme differs between each test domain. For instance, the weighting that gives the best results for the EP tests results in much poorer translation performance for the RCV1 test points requiring us to track which stream we are decoding and then select the appropriate weighting. This adds unnecessary complexity to the SMT system. And, since we store each stream separately, memory usage is not optimal using this scheme.

4.3.4 History and Subsampling

For space efficiency we want to represent multiple streams non-redundantly instead of storing each stream/domain in its own LM. Here we report on experiments using both the history combination and subsampling approaches from Sections 4.2.3 and 4.2.4.

Results are in Tables 4.6 and 4.7 for the RCV1 and EP test sets respectively with the column headers denoting the test point. The row *Multi-*f_k shows results using only the random subsampling parameter f_k and the rows *Multi-*f_T show results with just the time-based adaptation parameter without subsampling. The final row *Multi-*$f_k + f_T$ uses both the f parameters with random subsampling as well as taking decoding history into account.

	RCV1 Test Points			EP Test Points		
RCV1 f_k	Test 1	Test 2	Test 3	Test 1	Test 2	Test 3
10%	40.51	41.64	39.51	**41.05**	**44.19**	35.82
30%	41.19	**41.73**	39.23	40.91	43.50	**36.11**
50%	**41.43**	41.59	39.52	40.86	43.73	35.66
100%	41.29	**41.73**	**40.41**	40.91	44.05	35.56

Table 4.8: Variation in translation quality caused by changing the subsampling rate over the streams with high data rates.

Multi-f_k uses the random subsampling parameter f_k to filter out higher order *n*-grams from the streams. All *n*-grams that are sampled from the streams are then combined into the joint LM. The counts of *n*-grams sampled from more than one stream are added together in the composite LM. The parameter f_k is set dependent on a stream's throughput rate, we only subsample from the streams with high throughput, and the rate was chosen based on the weighted interpolation results described previously. In Tables 4.6 and 4.7 the subsampling rate $f_k = 0.3$ for the combined newswire streams RCV1 and GW and we kept all of the EP data.

We also tested various other values for the f_k sampling rates and found translation results only minorly impacted as shown in Table 4.8. Note that the subsampling is random sampling so two adaptation runs with equal subsampling rates may produce different final translations. Nonetheless, in our experiments we saw expected performance, observing slight variation in performance for all test points that correlated to the percentage of in-domain data residing in the model. So we take a slight hit on performance for the RCV1 test points when the subsampling rate is lower since more stream data is better since it is all in-domain to the test. For the EP test domain we see largely the opposite effect. We also see that the newswire streams still act only as noisy data and negatively effect the translation for the EP test sets. Consequently results still do not match the performance of translations based solely on an EP-stream specific ORLM.

The next row in Tables 4.6 and 4.7, *Multi-*f_T, uses recency criteria to keep potentially useful *n*-grams but uses no subsampling and accepts all *n*-grams from all of the streams into the LM. Here we get better results than naive combination and plain subsampling at the expense of using more memory for the same error rate for the ORLM.

The final row in the tables, *Multi-*$f_k + f_T$ uses both the subsampling function f_k

and f_T so maintains a history of the *n*-grams queried by the decoder for the prior test points. This approach achieves significantly better results than naive adaptation and compares to using all the data in the stream. Combining translation history as well as doing random subsampling over the stream means we match the performance of but use far less memory than when using multiple online LMs whilst maintaining the same error rate. This is simply because, when we subsample, our LM contains less data and therefore can achieve the same false positive rate using less bits overall. By simply storing fewer *n*-grams we are able to achieve space savings while still improving the translation results by using recent data from the stream. For the sampling the exact amount of memory is of course dependent on the sampling rate used. For the results in Tables 4.6 and 4.7 we used significantly less memory (300MB) but still achieved comparable performance to approaches that used more memory by storing the full streams separately (600MB).

4.4 Scaling Up

The experiments described in the proceeding section used combinations of relatively small (compared to current industry standards) input streams. The question remains if using such approaches aids in the performance of translation if used in conjunction with large static LMs trained on large corpora. In this section we describe scaling up the previous stream-based translation experiments using a large background LM trained on a billion *n*-grams. [4]

We used the same setup described in Section 4.3.1. However, instead of using only a subset of the GW corpus as one of our incoming streams, we trained a static LM using the *full* GW3 corpus of over three billion tokens and used it as a background LM. As the *n*-gram statistics for this background LM show in Table 4.9, it contains far more data then each of the stream specific LMs (Table 4.1). We tested whether using streams atop this large background LM had a positive effect on translation for a given domain.

Baseline results for all test points using only the GW background LM are shown in the top row in Tables 4.10 and 4.11. We then interpolated the ORLMs with this LM. For each stream test point we interpolated with the big GW LM an online LM built

[4] Special thanks to David Matthews who assisted with some of the computationally expensive experiments reported in this section by running them under his login and reporting the results back to the author.

Order	Count
1-grams	3.7M
2-grams	46.6M
3-grams	195.5M
4-grams	366.8M
5-grams	454.2M
Total	1067M

Table 4.9: Singleton-pruned statistics of unique *n*-gram counts (in millions) for the Gigaword background LM.

LM Type	Test 1	Test 2	Test 3
RCV1	*39.30*	*40.64*	*39.19*
GW	41.69	42.40	35.48
GW+RCV1	42.44	43.83	**40.55**
GW+RCV1+EP	**42.80**	**43.94**	38.82

Table 4.10: Test results for the RCV1 stream using the large background LM (GW) along with the stream-based incremental LMs (RCV1 and EP). Using stream data benefits translation.

with the most recent epoch's data. Here we used separate models per stream so the RCV1 test points used the GW LM along with a RCV1 specific ORLM. We used the same mechanism to obtain the interpolation weights as described in Section 4.3.3 and minimised the perplexity of the static LM along with the stream specific ORLM. Interestingly, the tuned weights returned gave approximately a 50-50 weighting between LMs and we found that simply using a 50-50 weighting for all test points resulted had no negative effect on BLEU. In the third row of the Tables 4.10 and 4.11 we show the results of interpolating the big background LM with ORLMs built using the approach described in Section 4.2.4. In this case all streams were combined into a single LM using a subsampling rate for higher order *n*-grams. As before our sampling rate for the newswire streams was 30% chosen by the perplexity reduction weights.

The results show that even with a large amount of static data adding small amounts of stream specific data relevant to a given test point has an impact on translation quality. Compared to only using the large background model we obtain significantly better

LM Type	Test 1	Test 2	Test 3
EP	*42.09*	*45.94*	*37.22*
GW	40.78	44.26	34.36
GW+EP	**43.94**	**47.82**	38.71
GW+EP+RCV1	43.07	47.72	**39.15**

Table 4.11: EP test results using a static, large background GW LM plus adding domain specific information using ORLMs trained on the EP and RCV1 streams.

results when using a streaming ORLM to compliment it for all test domains. However the large amount of data available to the decoder in the background LM positively impacts translation performance compared to single-stream approaches (Tables 4.2 and 4.3). Further, when we combine the streams into a single LM using the subsampling approach we get, on average, comparable scores for all test points. Thus we see that the patterns for multiple stream adaptation seen in previous sections hold in spite of big amounts of static data.

4.5 Conclusion

In this chapter we have reported various approaches for multiple stream based translation for SMT. We have shown that using data from multiple streams benefits SMT performance. Our experiments demonstrate that naive linear combinations of the streams can hurt performance for some test domains if the input streams have significantly different incoming rates of data but some combination of the individual streams aids in translation quality for all test points. However, the interpolated multiple LM approach is unnecessarily complex and uses the most memory since this requires all overlapping *n*-grams in the streams to be stored separately.

Our best approach, using history based combination along with subsampling, combines all incoming streams into a single, succinct LM and obtains translation performance equal to single stream, domain specific LMs on all test domains. Crucially we do this in bounded space, require less memory than storing each stream separately, and do not incur translation degradations on any single domain. As well, these results can be additive. Even when using large amounts of additional background data, adding stream specific data continues to improve translation.

Chapter 5

A Stream-based Translation Model

Since the TM is the starting point by which possible translations are selected by the decoder, arguable the data available in the TM affects the performance of an SMT system more than any other model in the pipeline. In this chapter we extend our stream-based SMT system from the LM to include the TM. This allows it to be incrementally updated with new, useful parallel data efficiently without incurring the substantial computational cost associated with batch training. The online TM presented here is the first reported in the literature and represents another contribution of this thesis. We extend the traditional EM algorithm approach for word alignments to operate online. We also make use of dynamic suffix arrays to incorporate the novel parallel sentences that have been received from the stream. Finally we demonstrate that the ORLM and incremental TM compliment each other in a stream-based SMT system that significantly outperforms a traditional batch-based approach. The stream-based TM was previously reported in Levenberg et al. (2010).

5.1 Overview

There is more parallel training data available today than there has ever been and it keeps increasing. For example, the European Parliament [1] releases new parallel data in 22 languages on a regular basis. Project Syndicate[2] translates editorials into seven languages (including Arabic, Chinese and Russian) every day. Existing translation systems often get 'crowd-sourced' improvements such as the option to contribute a better translation to GoogleTranslate [3]. In these examples and many other instances,

[1] http://www.europarl.europa.eu
[2] http://www.project-syndicate.org
[3] http://www.translate.google.com

the data can be viewed as an incoming unbounded stream of parallel sentences since the bitext corpus grows continually with time.

Recall in Chapter 1 that we showed the effect of recency on OOV rates and SMT translation performance by using more recent data to train TMs that outperformed batch based models. However, TMs for SMT systems are typically batch trained, often taking many CPU-days of computation when using large volumes of training material. Incorporating any new sentence pairs into these models forces us to batch retrain entirely from scratch. Clearly, this makes the standard approach infeasible for streaming translation where we desire to rapidly add new parallel sentences into the TM.

Here we introduce an adaptive training regime that uses an online variant of the EM algorithm that is capable of incrementally aligning new sentences without incurring the burdens of full retraining. After a new sentence has been word aligned it needs to be added to the current corpus from which translation probabilities are extracted and measured. Instead of reestimating translation probabilities offline for the full corpus we employ dynamic suffix arrays to allow incremental updates to the parallel data available to the decoder and compute needed statistics on the fly. The dynamic suffix arrays allow deletions of old sentences so the incremental TM presented here has the ability to operate within bounded memory.

5.2 Online EM

In this section we describe the online EM algorithm employed, *stepwise online EM* (SOEM), and how it is used for stream-based incremental word alignments in our streaming SMT system.

5.2.1 Stepwise Online EM

When we move from batch training to processing an incoming data stream, the batch EM algorithm's (Section 2.3) requirement that all data be available for each iteration becomes impractical since we do not have access to all n examples at once. Instead we receive examples from the input stream incrementally. For this we make use of online EM algorithms that update the probability model $\hat{\theta}$ incrementally without needing to store and iterate through all the unlabeled training data repeatedly.

Various online EM algorithms have been investigated (see Liang and Klein (2009)

for an overview) but our focus is on the *stepwise online* EM (sOEM) algorithm (Cappe and Moulines, 2009). Instead of iterating over the full set of training examples, sOEM stochastically approximates the batch E-step and incorporates the information from the newly available streaming observations in smaller steps. Each step is called a *mini-batch* and is comprised of one or more new examples encountered in the stream. For each mini-batch of an arbitrary size m of new examples, $1 \leq m \leq n$, we use the current model's distribution to inform the new observations and then update the model with the newly obtained evidence from the incoming data.

Unlike in batch EM, in sOEM the expected counts are retained between EM iterations and not cleared. That is, for each new example we interpolate its expected count with the existing set of sufficient statistics. For each step we use a *stepsize* parameter γ which mixes the information from the current example with information gathered from all previous examples.

As we are updating the model with the sufficient statistics of only m sentences, if $m \ll n$ the statistics collected for just the current mini-batch will be a bad approximation of the true distribution over all n. To account for this we interpolate, based on stepsize parameter γ_t, the set of sufficient statistics from the current mini-batch with the counts from all previous observations. If $\bar{s}_{tm}$ represents the most recent sufficient statistics for the last mini-batch of m examples, then we interpolate the new sufficient statistics $\bar{s}_{tm}$ as $\mathbf{S} \leftarrow (1-\gamma_t)\mathbf{S} + \gamma_t \bar{s}_{tm}$. Over time the sOEM model probabilities begin to stabilize and are guaranteed to converge to a local maximum (Cappe and Moulines, 2009).

Note that the stepsize γ has a dependence on the current mini-batch. As we observe more incoming data the model's current probability distribution is likely closer to the true distribution so the new observations receive less weight. From Liang and Klein (2009), if we set the stepsize as $\gamma_t = (t+2)^{-\alpha}$, with $0.5 < \alpha \leq 1$, we can guarantee convergence in the limit as $n \to \infty$. If we set α low, γ weighs the newly observed statistics heavily whereas if γ is low new observations are down-weighted.

5.2.2 Stepwise EM for Word Alignments

Application of sOEM to HMM (Equation 2.3) and IBM Model 1 (Equation 2.4) based word aligning is straightforward. The process of collecting the counts over the expected conditional probabilities inside each iteration loop remains the same as in the batch case. However, instead of clearing the sufficient statistics between the iterations

Algorithm 4: sOEM Algorithm for Incremental Word Alignments

Input: mini-batches of sentence pairs $\{m : m \subset \{(f(source), e(target))\}\}$

Input: stepsize weight α

Output: MLE $\hat{\theta}_T$ over alignments **a**

$\hat{\theta}_0 \leftarrow$MLE initialization;

$S \leftarrow 0$; $k = 0$;

foreach *mini-batch* $\{m\}$ **do**

  for *iteration* $t = 0, \ldots, T$ **do**

    foreach $(f, e) \in \{m\}$ **do** `// E-step`

      $\bar{s} \leftarrow \sum_{a' \in \mathbf{a}} \Pr(f, a' | e; \hat{\theta}_t)$;

    end

    $\gamma = (k+2)^{-\alpha}; k = k+1$; `// stepsize`

    $S \leftarrow \gamma \bar{s} + (1-\gamma) S$; `// interpolate`

    $\hat{\theta}_{t+1} \leftarrow \bar{\theta}_t(S)$; `// M-step`

  end

end

we retain them and interpolate the last set of statistics with the batch of counts gathered in the next iteration.

Algorithm 4 shows high level pseudocode of our sOEM framework for HMM-based word alignments. We have an unbounded input stream of source and target sentences pairs $\{(\mathbf{f}, \mathbf{e})\}$ which we observe as a stream of mini-batches $\{\mathbf{m}\}$ comprised of chronologically ordered strict subsets of the full stream. To word align the sentences for each mini-batch **m**, we use the probability assigned by the current model parameters and then interpolate the newest sufficient statistics $\bar{s}$ with our full count vector **S** using an interpolation parameter γ. The interpolation parameter γ has a dependency on how far along the input stream we are processing. We can choose our interpolation parameter so that new statistics gathered are weighted strongly or weakly compared to previous observations. An in-depth analysis of the interpolation parameter as applied to word alignments is given in Liang and Klein (2009).

5.3 Dynamic Suffix Arrays

We can implement a sOEM based alignment algorithm to incrementally align an incoming stream of parallel sentences in mini-batches. However, we still need to consider how to insert newly aligned sentences, along with the associated new or adjusted translation probabilities, into the grammar or phrase table so they become quickly available to the decoder for translation. Continuous batch recomputing of phrase or grammar rule probabilities is slow and resource heavy and so is ill-suited for stream-based SMT systems. We may also need to bound the space used for the TM when processing (potentially) unbounded streams of parallel data. Dynamic suffix arrays allow us to update the data in a TM online and provide fast access to it to compute statistics needed by the decoder.

Recall from Section 2.4 that we can use a suffix array to compactly represent a phrase table implementation for either phrase-based or hierarchical phrase-based TMs. The suffix array phrase tables store the set of source and target sentences and compute the translation probabilities needed by the decoder on demand. Standard suffix arrays are static, store a fixed corpus and do not support online adaptation that allow insertions or deletions. In our stream-based approach we make use of a dynamic variant of the suffix array (Salson et al., 2009). This allows us to incorporate new sentence pairs into the phrase table by inserting them into the suffix array and, if we need to maintain constant space usage, we can delete an equivalent number.

Recall that the suffix array phrase table keeps the suffixes of a sorted list of the source and target corpus. To make a suffix array dynamic, we need to be able to insert the new incoming sentence pairs while keeping the suffix array lexicographically ordered. If we have a constrained memory we may also need to be able to delete sentences from but still maintain the ordering of the suffix array so we are able to reconstruct the original sentences to retrieve the necessary phrases and lexical rules. We describe how to accomplish this below.

5.3.1 Burrows-Wheeler Transform

To accomplish this ordered dynamism we introduce the *Burrows-Wheeler transform* (BWT) (Schindler, 1997), a well studied transformation that reorders the characters or words of a text to allow for better compression. Given a text corpus $\mathbf{T} = T[0 \dots n]$, we add a special stop symbol at $T[n+1]$ to the text and then build a conceptual matrix of lexically ordered cyclic shifts of the text. Each row begins with the word in the text at

some index T_i and ends with the word prior at index T_{i-1} so each row is of the form $T[i \ldots n+1]T[0 \ldots i-1]$. The output of the BWT is the last column L of the matrix consisting of a complete reordering of the original text but since the BWT matrix is sorted, the first column F contains the words of the text that the suffix array represents as integers in its corresponding rows. Since we keep only the transformed last column L we need a formula to reconstruct the original text after the BWT. We can use the fact that there is a one-to-one correspondence between the rank of a word in L and that word's ranking in F. That is, the first occurrence of a word type in L corresponds to the first occurrence of the equivalent word type in F, the second occurrence in L with the second in F and so forth. We can use this relationship and create a mapping function, LF, that provides us with the next row to visit. Since we added a special stop symbol to the original text, we can start with the stop symbol and use the LF function to reconstruct our original text (backwards) by visiting the cycle built during the BWT.

5.3.2 Dynamic BWT and Suffix Arrays

To allow the BWT and the entailed suffix array to adapt to new text online, when inserting a word into (or deleting from) our corpus T we find the correct insertion (or deletion) point for the BWT via an *inverse suffix array* (ISA), a mapping of indexes T_i in the original text to their location in the suffix array. Then we increment (or decrement) all indexes in the suffix array and ISA greater than the insertion point to handle the changes introduced by the text modifications. Finally, we compare the "expected" LF value against the "actual" LF value and reorder until they match. We can compute the expected LF value by using sub-functions that *rank* the number of occurrences of a word in the range $T[0 \ldots i]$ and *count* the index of the first row where the word is found in the array F. Here we are only interested in a dynamic suffix array but to keep the lexical ordering intact we must borrow the dynamic BWT. Thus, besides the suffix array itself, we also need to store the BWT's L array to make use of the LF function and corresponding *rank* and *count* sub-functions [4]. When we append new sentences into the suffix array, we add each word backwards based on the location given by the ISA beginning with the final index in the corpus. Deletions are done similarly starting with the first index (if we are deleting the oldest sentences). Then we use an algorithm that reorders the indexes in the dynamic suffix array until the

[4] Storing L means we no longer need to store the original corpus in memory since we can reconstruct this from L and the LF function.

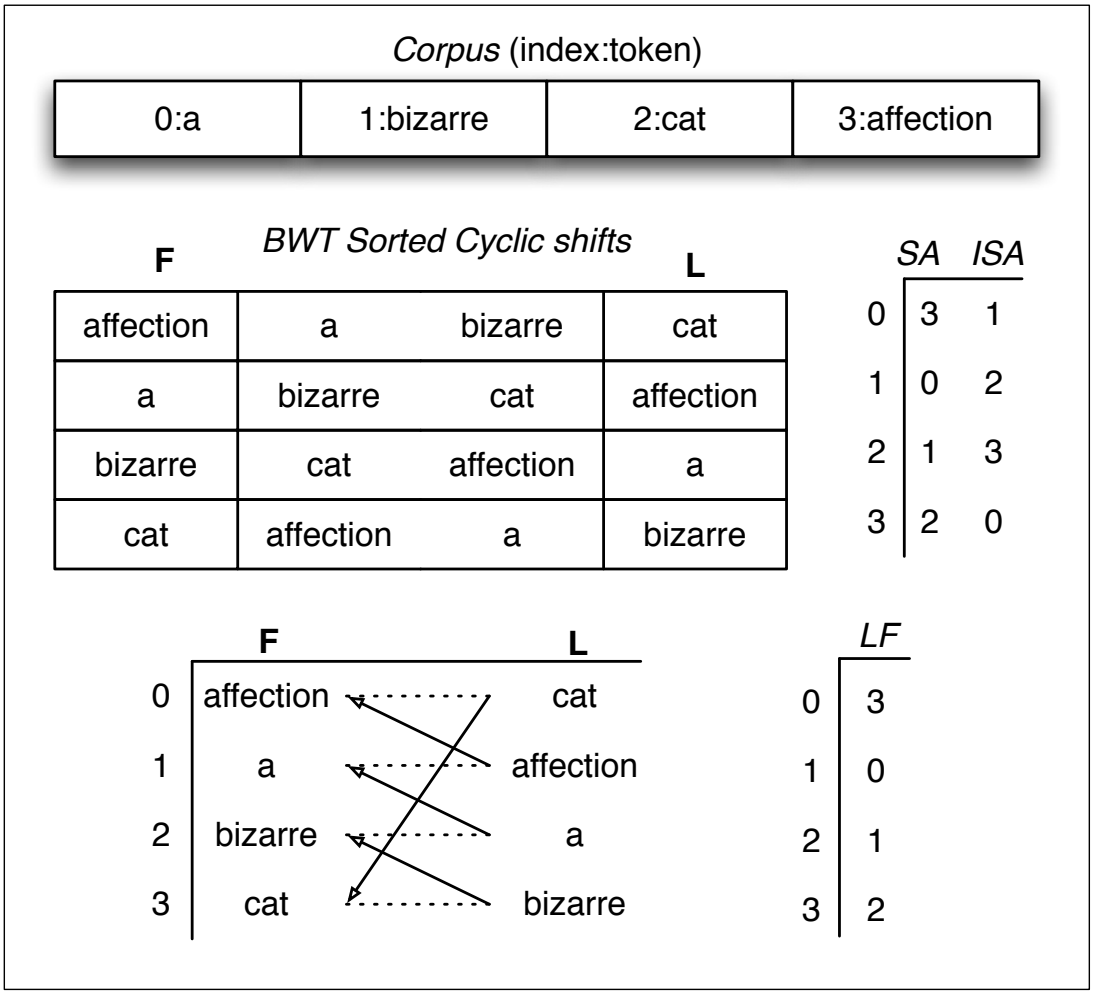


Figure 5.1: Example BWT and suffix array. The LF function (bottom right) is used to reorder the text when inserting new sentences into the dynamic suffix array.

expected and actual LF function values match. While we used the dynamic suffix array in the experiments reported in this section, the technical details of the reordering for the dynamic suffix array algorithm is extremely involved. Its description here would add no clarity to stream-based SMT which is the scope of this thesis. We refer the interested reader to the full algorithm description in Salson et al. (2009).

5.4 Experiments

We have described the sOEM algorithm for incremental word alignments and an efficient data structure that can incorporate new sentence pairs quickly. The question still remains whether using such an online TM is worthwhile for translation performance. In this section we describe the experiments conducted comparing various batch trained

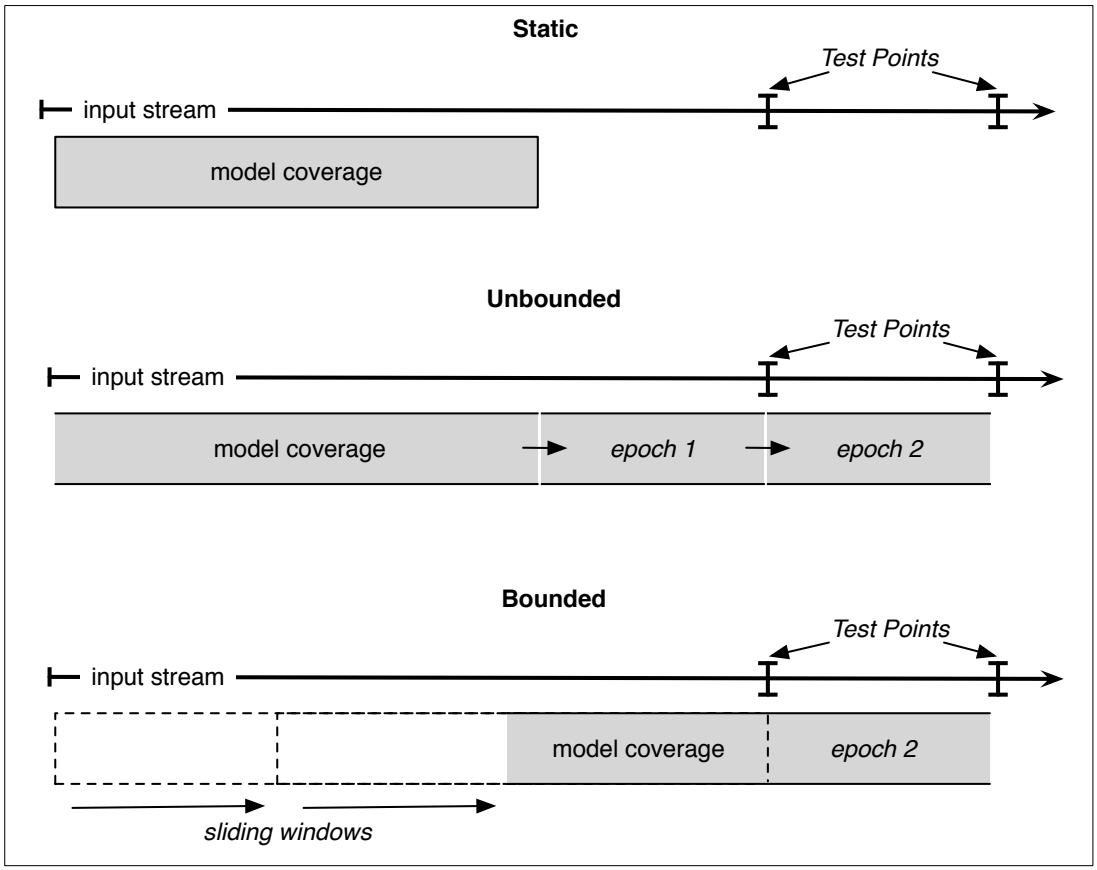


Figure 5.2: Streaming coverage conditions. In traditional batch based modeling the coverage of a trained model never changes. Unbounded coverage operates without any memory constraints so the model is able to continually add data from the input stream. Bounded coverage uses just a fixed window and deletions are required to maintain a strict memory bound.

TMs versus online, incrementally retrained TMs in a full SMT setting with different conditions set on model coverage.

5.4.1 Experimental Setup

We used publicly available resources for all our tests. Our parallel input stream was generated from the German-English and French-English language pairs of Europarl with English as target for both pairs. For testing we divided each input stream into 10 evenly spaced epochs and held out a total of 22k test sentences in total from each source stream. Stream statistics for three example epochs in the German-English language pair are shown in Table 5.1. For each stream, we held out 4.5k sentence pairs from each language pair as development data to optimize the feature function weights using

Ep	From–To	Sent Pairs	Words (source/target)
00	04/1996–12/2000	580k	15.0M/16.0M
03	02/2002–09/2002	70k	1.9M/2.0M
06	10/2003–03/2004	60k	1.6M/1.7M
10	03/2006–09/2006	73k	1.9M/2.0M

Table 5.1: Date ranges, total sentence pairs, and source and target word counts encountered in the German input stream for example epochs. Epoch00 is baseline data that is also used as a seed corpus for the online models.

minimum error rate training (Och, 2003).

We used a 5-gram, Kneser-Ney smoothed LM trained on the initial segment of the target side parallel data used in the baseline as described further in the next subsection. As our initial experiments aim to isolate the effect of changes to the TM on overall translation system performance, our in-domain LM remains static for every decoding run.

We used the open-source toolkit GIZA++ (Och and Ney, 2003) for all word alignments. We modified IBM Model 1 and the HMM model in GIZA++ to use the sOEM algorithm for the online experiments. Batch baselines were aligned using the standard version of GIZA++. We ran the batch and incremental versions of Model 1 and HMM for the same number of iterations each in both directions. We used Joshua (Li et al., 2009), a syntax-based decoder with a suffix array implementation, and rule induction via the standard Hiero grammar extraction heuristics (Chiang, 2007) for the grammar-based TMs. For the standard phrase-based models we used the Moses decoder. For both decoders we implemented a dynamic variant of the suffix arrays. Both the German-English and French-English language pairs were translated using the modified Joshua (with a Heiro grammar) and Moses (using phrase-based translation) decoders. Equivalent results were seen regardless of the TM formalism used. In the results below we only report on the German-English results using Heiro and the French-English using phrases.

We considered how to process a stream along two main axes: by bounding time (batch versus incremental retraining) and by bounding space (either using all the stream seen so far, or only using a fixed sized sample of it).

			Baseline/Bounded		Unbounded	
Epoch	Test Date	Test Sent.	Train Sent.	Rules	Train Sent.	Rules
03	23/09/2002	1.0k	580k	4.0M/4.2M	800k	5.0M
06	29/03/2004	1.5k	580k	5.0M/5.5M	1.0M	7.0M
10	26/09/2006	3.5k	580k	8.5M/10.0M	1.3M	14.0M

Table 5.2: Translation model statistics for example epochs and the next test dates grouped by experimental condition. 'Test and Train Sent.' is the number of sentence pairs in test and training data respectively. 'Rules' is the count of unique Hiero grammar rules extracted for the corresponding test set.

5.4.2 Time and Space Bounds

For both batch and sOEM we ran a number of experiments listed below corresponding to the different training scenarios diagrammed in Figure 5.2.

1. **Static**: We used the first half of each input stream, approximately 600k sentences and 15/16 million source/target words, as parallel training data. We then translated each of the 10 test sets using the static model. This is the traditional approach and the coverage of the model never changes.

2. **Unbounded Space**: Batch or incremental retraining with no memory constraint. For each epoch in the stream, we retrained the TM using all the data from the beginning of the input stream until just before the present with respect to a given test point. As more time passes our training data set grows so each batch run of GIZA++ takes more time. Overall this is the most computationally expensive approach.

3. **Bounded Space**: Batch and incremental retraining with an enforced memory constraint. Here we batch or incrementally retrain using a sliding window approach where the training set size (the number of sentence pairs) remains constant. In particular, we ensured that we used the same number of sentences as the baseline. As the window slides, old sentence pairs from earlier in the timeline are deleted from the dynamic suffix array to make room for newly observed data incoming from the stream. Each batch run of GIZA++ takes approximately the same time.

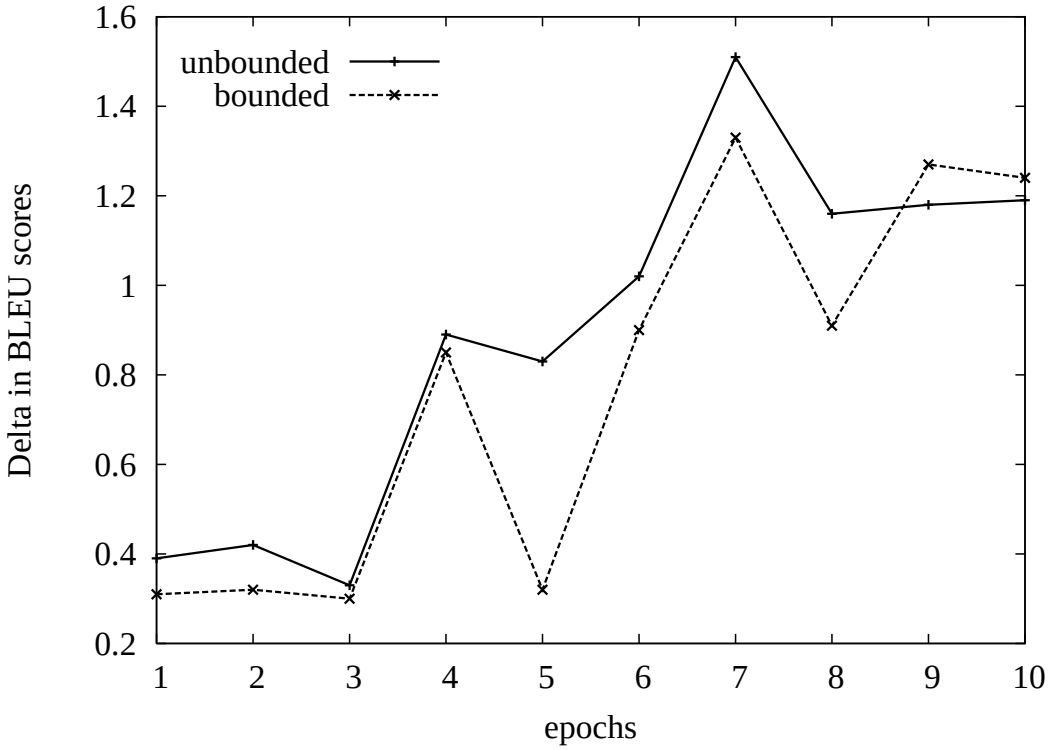


Figure 5.3: Static vs. online TM phrase-based performance. Gains in translation performance measured by BLEU are achieved when recent German-English sentence pairs are automatically incorporated into the TM. Shown are relative BLEU improvements for the online models against the static baseline.

The time for aligning in the sOEM model is unaffected by the bounded/unbounded conditions since we always only align the mini-batch of sentences encountered in the last epoch. In contrast, for batch EM we must realign all the sentences in our training set from scratch to incorporate the new training data.

Similarly space usage for the batch training grows with the training set size. For sOEM, in theory memory used is with respect to vocabulary size (which grows slowly with the stream size) since we retain count history for the entire stream. To make space usage truly constant, we filter for just the needed word pairs in the current epoch being aligned. This effectively means that online EM is more memory efficient than the batch version. As our experiments will show, the sufficient statistics kept between epochs by sOEM benefits performance compared to the batch models which can only use information present within the batch itself.

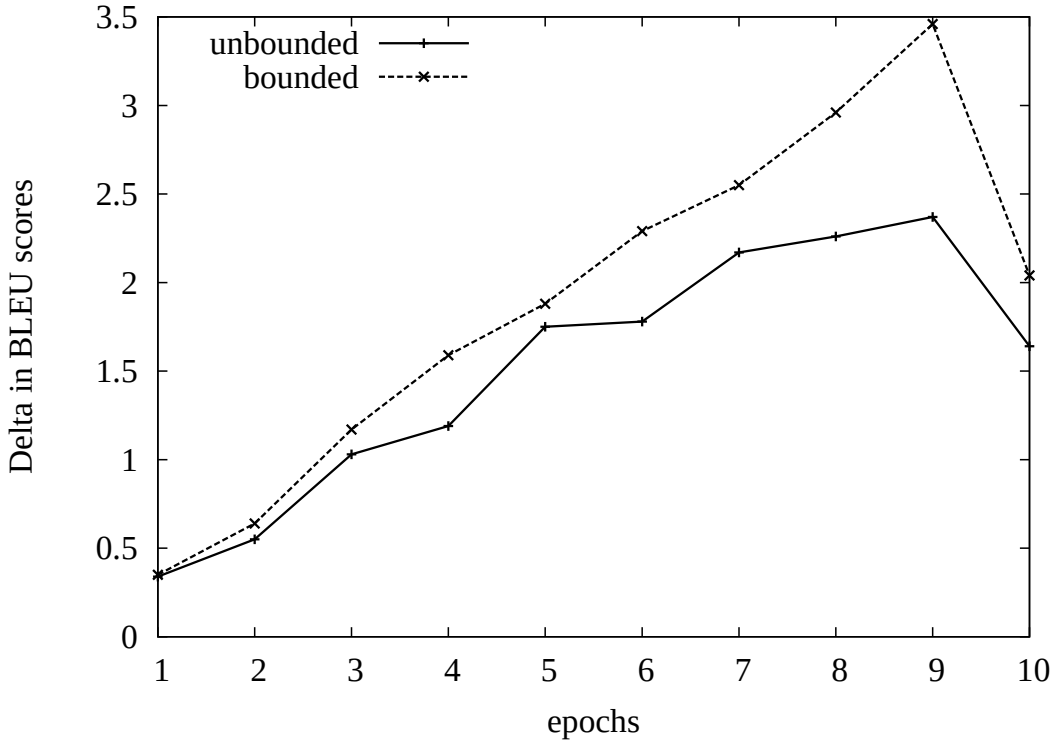


Figure 5.4: Static vs. online grammar-based TM performance. As in Figure 5.3, shown are the deltas in BLEU score achieved when adding recent sentence pairs to the stream using the stream-based TM described in this chapter.

5.4.3 Incremental Retraining Procedure

Our incremental adaptation procedure was as follows: after the latest mini-batch of sentences had been aligned using GIZA++ and the sOEM algorithm we added all newly aligned sentence pairs to the dynamic suffix arrays. For the experiments where our memory was bounded, we also deleted an equal number of sentences from the suffix arrays before extracting the phrase probabilities or the Hiero grammar for the next test point. For the unbounded coverage experiments we deleted nothing prior to decoding. Table 5.2 presents statistics for the number of training sentence pairs and phrase/grammar rules extracted for each coverage condition for various test points in the German-English pair.

5.4.4 Results

Figures 5.3 and 5.4 shows the results of the static baseline against both the unbounded and bounded online EM models. We can clearly see that the online models outperform the static baseline. For the grammar-based translation results we found that, on

	Static Baseline	Retrained (Unbounded)		Retrained (Bounded)	
Epoch	Test Date	Batch	Batch	Online	Batch	Online
3	23/09/2002	26.10	**26.60**	26.43	26.19	*26.40*
6	29/03/2004	27.40	28.33	**28.42**	28.06	*28.38*
10	26/09/2006	28.56	29.74	**29.75**	29.73	*29.80*

Table 5.3: Sample BLEU results for all German-English baseline and online EM model conditions. The static baseline is a traditional model that is never retrained. The batch unbounded and batch bounded models incorporate new data from the stream but retraining is slow and computationally expensive (best results are in bold text). In contrast both unbounded and bounded online models incrementally retrain only the mini-batch of new sentences collected from the incoming stream so quickly adopt the new data (best results are italicized).

average, the unconstrained model that contains more sentence pairs for rule extraction slightly outperforms the bounded condition which uses less data per epoch as in Figure 5.3. As evident in Figure 5.4, this was not the case for the phrase-based system results where the opposite occurs and the bounded model slightly outperforms the unbounded model. This is due to the fact that the bounded model produces more nuanced phrase and lexical probabilities for the given test point compared to the unbounded model which returns probabilities based on the whole corpus whereas the default Hiero extraction heuristics employ rigid pruning so more local context rules are ignored. However, in all conditions we see there is a clear gain by incorporating recent parallel sentences made available by the stream.

Table 5.3 gives explicit BLEU scores of the online German-English models compared to batch retrained models. That is, for completeness we compared the results of batch retrained word alignments using the same training data sets from the stream epochs that were used for the sOEM aligned TMs. For presentation clarity we show only a sample of three of the full set of ten test points shown in Figure 5.3. For the same coverage constraints not only do we achieve comparable performance to batch retrained models using the sOEM method of incremental adaptation, we are able to align and adopt new data from the input stream orders of magnitude quicker since we only align the mini-batch of sentences collected from the last epoch. Interestingly, in the bounded condition we also see that sOEM models slightly outperform the batch based models due to the online algorithm employing a longer history of count-based

Epoch	Test Date	Static	Unbounded	Bounded
3	24/09/2002	29.99	31.55	31.58
6	29/03/2004	35.74	38.11	38.30
10	12/10/2006	34.78	37.50	37.64

Table 5.4: Unbounded LM coverage improvements for the French-English phrase-based translation experiments. Significant gains in translation quality are achieved by using recent data in both the TM and the LM.

evidence to draw on when aligning new sentence pairs.

Figure 5.5 shows two example test sentences that benefited from the online TM adaptation. Translations from the online model produce more and longer matching phrases for both sentences (e.g., “creation of such a”, “well known”) leading to more fluent output as well as the improvements achieved in BLEU scores. As well we are able to incorporate new vocabulary into the phrase table (“occupying forces”) that is missing from the static baseline model that affects translation quality.

We experimented with a variety of interpolation parameters (see Algorithm 4) but found no significant difference between them with the biggest improvement gained over all test points for all parameter settings was less than 0.1% BLEU.

5.5 Stream-based SMT

A natural and interesting extension to the experiments above is to use the target side of the incoming stream to extend the LM coverage alongside the TM. It is well known that more LM coverage (via larger training data sets) is beneficial to SMT performance (Brants et al., 2007) so we investigated whether recency gains for the TM were additive with recency gains afforded by a LM.

Here we combined the ORLM from Chapter 3 with the stream-based TM to create a fully online, stream-based SMT system. We added all the target side data from the beginning of the German and French streams to the current epoch into the LM training set before each test point for the given pair. We then used the new LM with greater coverage for the next decoding run. We tested with the same coverage conditions imposed on the TM – bounded and unbounded amounts of streaming data.

Example results are reported in Tables 5.5 and 5.4. We can see that increasing LM coverage is indeed complimentary to adapting the TM with recent data. Compar-

Source: Die Kommission ist bereit, an der Schaffung eines solchen Rechtsrahmens unter Zugrundelegung von vier wesentlichen Prinzipien mitzuwirken.

Reference: The commission is willing to cooperate in the creation of such a legal framework on the basis of four essential principles.

Static: The commission is prepared, in the creation of a legal framework, taking account of four fundamental principles them.

Online: The commission is prepared to participate in the creation of such a legal framework, based on four fundamental principles.

Source: Unser Standpunkt ist klar und allseits bekannt: Wir sind gegen den Krieg und die Besetzung des Irak durch die USA und das Vereinigte Königreich, und wir verlangen den unverzüglichen Abzug der Besatzungsmächte aus diesem Land.

Reference: Our position is clear and well known: we are against the war and the US-British occupation in Iraq and we demand the immediate withdrawal of the occupying forces from that country.

Static: Our position is clear and we all know: we are against the war and the occupation of Iraq by the United States and the United Kingdom, and we are calling for the immediate withdrawal of the besatzungsmächte from this country.

Online: Our position is clear and well known: we are against the war and the occupation of Iraq by the United States and the United Kingdom, and we demand the immediate withdrawal of the occupying forces from this country .

Figure 5.5: Example sentences and improvements to their translation fluency by the adaptation of the TM with recent sentences. In both examples we get longer matching phrases in the online translation compared to the static one.

Epoch	Test Date	Static	Unbounded	Bounded
3	23/09/2002	26.46	27.11	26.96
6	29/03/2004	28.11	29.53	29.20
10	26/09/2006	29.53	30.94	30.88

Table 5.5: Unbounded LM coverage improvements for the German-English translation experiments. Shown are the BLEU scores for each experimental conditional when we allow the LM coverage to increase.

ing Tables 5.3 and 5.5, for the bounded condition, adapting only the TM achieved an absolute improvement of +1.24 BLEU over the static baseline for the final test point. We get another absolute gain of +1.08 BLEU by allowing the LM coverage to adapt as well. Using an online, adaptive model gives a total gain of +2.32 BLEU over a static baseline that does not adapt. We gain similar translation improvements using the phrase-based model with the French-English pair.

5.6 Conclusion

We described a stream-based TM that allows for incremental updates. Our results show that, like the stream-based LM, using recent data from an incoming stream reduces sparsity in the models and leads to improved translation performance when translating test points in the stream. The sOEM algorithm and dynamic suffix arrays presented allow online updates to be done efficiently so that even a single sentence pair can be added quickly.

Chapter 6

Conclusion

This thesis presented original work on stream-based algorithms for building a complete online SMT system that has the ability to efficiently adapt to high rates of novel, incoming training data without the need for expensive batch retraining. This is the first complete system of its kind reported in the literature. We have presented a stream-based LM via the ORLM —a dynamic randomised LM that allows for incremental online adapation to unbounded unilingual text streams—and the stream-based TM which has the ability to incorporate bilingual sentence pairs incrementally into the SMT system very quickly. The translation experiments reported show that making use of recent incoming data from the stream is indeed beneficial to stream-based translation performance. We have looked at some heuristics for online adaptation from the streams and showed how to combine multiple, domain diverse streams in a single system without taking a hit on, in fact improving, translation performance for all streaming test domains.

The contribution of this thesis is two-fold. First we have introduced a novel way of viewing translation as a streaming problem where the training data in the SMT models should be related to the next test point being translated. The algorithms described, however, could also be applied to traditional SMT systems as a way of quickly updating the models with new training data from any source and thereby saving computational resources, time, and energy in contrast with full batch retraining.

The algorithms described here are efficient enough to allow even single-sentence updates to the models. Besides the potential impact this has for large-scale SMT systems in industry, recently published literature has shown the usefulness of such small updates within the context of interactive, computer-aided translation systems. In this setting an automatically translated document is used as the basis for a human translator

who then post-edits the sentences to correct the SMT system's errors so the translation reads fluently in the target language. Using similar techniques as presented here, Ortiz-Martínez et al. (2010) showed the potential usefulness of online learning for an interactive SMT system. Hardt and Elming (2010) further showed that updating a phrase table using within-document sentences increases the BLEU score of that document significantly compared with a baseline model that does not incorporate within-document translations. Our approach makes it feasible for the human translator to update the TM with a novel translation whilst editing and for that improved translation to be propagated throughout the remainder of the document quickly. Further work is currently being conducted to apply our stream-based TM within a computer-aided translation system.

Consider also the following related scenario for a company that either provides professional translation as their main service or one that uses translators in-house and would like to use a SMT system to improve turnover. The documents being translated are domain specific, e.g., patents or legal documents, and little in-domain training data is available initially. Beginning with a system built using out-of-domain data, the techniques in this thesis can be directly applied to efficiently update the in-house SMT system continually with focused training data taken from the human translated documents. Over time automatic translation would improve as more domain specific data is input into the SMT system. The work in this thesis has already been sought after and applied by various companies to accomplish just this.

The demand for SMT is growing rapidly. So too is the amount of textual data being churned out daily. In many cases, attempting to improve translation quality in SMT will only become increasingly unweidly if the focus remains solely on acquiring huge amounts of context inspecific data and computers to distribute computation across. This thesis provides a foundation for investigating SMT systems that can tailor themselves online to provide accurate, quality translations. The streaming SMT framework provided here begs further work in more advanced learning algorithms for selecting stream data, learning from prior translations, and building better models online.

Bibliography

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., Cocke, J., Pietra, S. A. D., Pietra, V. J. D., Jelinek, F., Lafferty, J. D., Mercer, R. L., and Roossin, P. S. (1990). A statistical approach to machine translation. *Comput. Linguist.*, 16:79–85.

Brown, P. F., Pietra, V. J. D., Pietra, S. A. D., and Mercer, R. L. (1993). The mathematics of statistical machine translation: parameter estimation. *Computational Linguistics*, 19(2):263–311.

Callison-Burch, C., Bannard, C., and Schroeder, J. (2005). Scaling phrase-based statistical machine translation to larger corpora and longer phrases. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 255–262, Ann Arbor, Michigan. Association for Computational Linguistics.

Cappe, O. and Moulines, E. (2009). Online EM algorithm for latent data models. *Journal Of The Royal Statistical Society Series B*, 71:593.

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

Chiang, D. (2007). Hierarchical phrase-based translation. *Computational Linguistics*, 33(2):201–228.

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

Cormode, G. (2008). Data stream algorithms: Tutorial at Bristol summer school on probabilistic techniques in computer science. Available at `http://www.dimacs.rutgers.edu/ graham/pubs/html/TalkBristol08.html`.

Cormode, G. and Muthukrishnan, S. (2005). An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75.

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.

Curran, J. R. and Osborne, M. (2002). A very very large corpus doesn't always yield reliable estimates. In *COLING-02: proceedings of the 6th conference on Natural language learning*, pages 1–6, Morristown, NJ, USA. Association for Computational Linguistics.

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39:1–38.

Graff, D. (2003). English Gigaword. Linguistic Data Consortium (LDC-2003T05).

Graff, D., Kong, J., Chen, K., and Maeda, K. (2007). English Gigaword Third Edition. Linguistic Data Consortium (LDC-2007T07).

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.

Hardt, D. and Elming, J. (2010). Incremental re-training for post-editing SMT. In *Proceedings of AMTA*.

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

Japkowicz, N. and Stephen, S. (2002). The class imbalance problem: A systematic study. *Intell. Data Anal.*, 6:429–449.

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. (2010). *Statistical Machine Translation*. Cambridge University Press.

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.

Levenberg, A., Callison-Burch, C., and Osborne, M. (2010). Stream-based translation models for statistical machine translation. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 394–402, Los Angeles, California. Association for Computational Linguistics.

Levenberg, A. and Osborne, M. (2009). Stream-based randomised language models for SMT. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Levenberg, A., Osborne, M., and Matthews, D. (2011). Multiple-stream language models for statistical machine translation. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, Edinburgh, UK. Association for Computational Linguistics.

Li, Z., Callison-Burch, C., Dyer, C., Ganitkevitch, J., Khudanpur, S., Schwartz, L., Thornton, W. N. G., Weese, J., and Zaidan, O. F. (2009). Joshua: an open source toolkit for parsing-based machine translation. In *WMT09: Proceedings of the Fourth Workshop on Statistical Machine Translation*, pages 135–139, Morristown, NJ, USA. Association for Computational Linguistics.

Liang, P. and Klein, D. (2009). Online EM for unsupervised models. In *North American Association for Computational Linguistics (NAACL)*.

Lopez, A. (2008a). Statistical machine translation. *ACM Comput. Surv.*, 40:8:1–8:49.

Lopez, A. (2008b). Tera-scale translation models via pattern matching. In *Proceedings of the 22nd International Conference on Computational Linguistics (Coling 2008)*, pages 505–512, Manchester, UK. Coling 2008 Organizing Committee.

Manber, U. and Myers, G. (1990). Suffix arrays: A new method for on-line string searches. In *The First Annual ACM-SIAM Symposium on Dicrete Algorithms*, pages 319–327.

Mansour, Y., Mohri, M., and Rostamizadeh, A. (2008). Domain adaptation with multiple sources. In *NIPS*, pages 1041–1048.

Mortensen, C. W., Pagh, R., and Pătraçcu, M. (2005). On dynamic range reporting in one dimension. In *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 104–111, New York, NY, USA. ACM.

Motwani, R. and Raghavan, P. (1995). *Randomized algorithms*. Cambridge University Press, New York, NY, USA.

Muthukrishnan, S. (2003). Data streams: algorithms and applications. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 413–413, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Och, F. (2005). The Google statistical machine translation system for the 2005 Nist MT evaluation. oral presentation at the 2005 Nist MT evaluation workshop.

Och, F. J. (2003). Minimum error rate training in statistical machine translation. In *ACL '03: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics*, pages 160–167, Morristown, NJ, USA. Association for Computational Linguistics.

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.

Och, F. J. and Ney, H. (2003). A systematic comparison of various statistical alignment models. *Computational Linguistics*, 29(1):19–51.

Ortiz-Martínez, D., García-Varea, I., and Casacuberta, F. (2010). Online learning for interactive statistical machine translation. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 546–554, Los Angeles, California. 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. (2001). Cuckoo hashing. *Lecture Notes in Computer Science*, 2161:121–129.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2001). Bleu: a method for automatic evaluation of machine translation. In *ACL '02: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 311–318, Morristown, NJ, USA. Association for Computational Linguistics.

Rose, T., Stevenson, M., and Whitehead, M. (2002). The reuters corpus volume 1 - from yesterdays news to tomorrows language resources. In *In Proceedings of the Third International Conference on Language Resources and Evaluation*, pages 29–31.

Rosenfeld, R. (1995). Optimizing lexical and n-gram coverage via judicious use of linguistic data. In *In Proc. European Conf. on Speech Technology*, pages 1763–1766.

Salson, M., Lecroq, T., Léonard, M., and Mouchard, L. (2009). Dynamic extended suffix arrays. *Journal of Discrete Algorithms*.

Schindler, M. (1997). A fast block-sorting algorithm for lossless data compression. In *DCC '97: Proceedings of the Conference on Data Compression*, page 469, Washington, DC, USA. IEEE Computer Society.

Stolcke, A. (1998). Entropy-based pruning of backoff language models. In *In Proc. DARPA Broadcast News Transcription and Understanding Workshop*, pages 270–274.

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

Talbot, D. and Brants, T. (2008). Randomized language models via perfect hash functions. In *Proceedings of ACL-08: HLT*, pages 505–513, Columbus, Ohio. Association for Computational Linguistics.

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.

Vogel, S., Ney, H., and Tillmann, C. (1996). HMM-based word alignment in statistical translation. In *Proceedings of the 16th conference on Computational linguistics*, pages 836–841, Morristown, NJ, USA. Association for Computational Linguistics.

Whittaker, E. W. D. (2001). Temporal adaptation of language models. In *In Adaptation Methods for Speech Recognition, ISCA Tutorial and Research Workshop (ITRW)*, pages 203–206.