PBML

The Prague Bulletin of Mathematical Linguistics
NUMBER 98 OCTOBER 2012 5–24

Simple and Efficient Model Filtering in Statistical Machine Translation

Juan Pino, Aurelien Waite, William Byrne

Department of Engineering, University of Cambridge, Cambridge, CB2 1PZ, U.K.

Abstract

Data availability and distributed computing techniques have allowed statistical machine translation (SMT) researchers to build larger models. However, decoders need to be able to retrieve information efficiently from these models to be able to translate an input sentence or a set of input sentences. We introduce an easy to implement and general purpose solution to tackle this problem: we store SMT models as a set of key-value pairs in an HFile. We apply this strategy to two specific tasks: test set hierarchical phrase-based rule filtering and n-gram count filtering for language model lattice rescoring. We compare our approach to alternative strategies and show that its trade offs in terms of speed, memory and simplicity are competitive.

1. Introduction

Current machine translation research is characterised by ever increasing amounts of data available for research. For example, Figure 1 shows that for the WMT machine translation workshop (Callison-Burch et al., 2012) French-English constrained track translation task, the English side of parallel data has increased from 13.8M tokens in 2006 to 945.1M tokens in 2012 and that available English monolingual data has increased from 27.5M tokens to 6841.1M tokens. Along with growing amounts of data, the use of more powerful computers and distributed computing models such as MapReduce (Dean and Ghemawat, 2008; Lin and Dyer, 2010) has enabled machine translation researchers to build larger statistical machine translation (SMT) models. Examples include language modelling (Brants et al., 2007), translation rule extraction (Dyer et al., 2008; Weese et al., 2011), word alignment (Dyer et al., 2008; Lin and Dyer,

Corresponding author: jmp84@cam.ac.uk
Cite as: Juan Pino, Aurelien Waite, William Byrne. Simple and Efficient Model Filtering in Statistical Machine Translation. The Prague Bulletin of Mathematical Linguistics No. 98, 2012, pp. 5–24.
doi: 10.2478/v10108-012-0005-x.

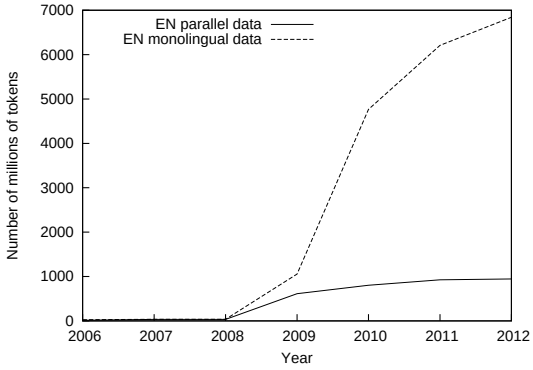

Figure 1. Number of English tokens (in millions) in parallel and monolingual data available for the WMT translation shared task constrained track for the years 2006 to 2012.

2010) as well as end-to-end toolkits for building entire phrase-based (Gao and Vogel, 2010) or hierarchical phrase-based models (Venugopal and Zollmann, 2009) using MapReduce.

Once SMT models are built, specifically the language model and the translation model, decoders or rescorers only need a fraction of the information contained in those models to be able to translate an input source sentence or a set of input source sentences. For example, in translation from French to English, given an input sentence "Salut toi", we don't need to know what translation probability the model assigns to other words than "Salut" and "toi" or what probability the English language model assigns to their possible English translation. With larger models, simply retrieving relevant translation or language model probabilities becomes a challenge. We use the HiFST system (Iglesias et al., 2009b; de Gispert et al., 2010), which involves a first-pass decoding followed by a 5-gram language model lattice rescoring step (Blackwood, 2010). Given a test set, the decoder only needs the rules whose source side matches part of one of the source sentences in the test set to be able to generate hypotheses. In the system described by Iglesias et al. (2009b), for each new test set, rules are re-extracted and filtered at extraction time. Similarly, for the task of 5-gram language model lattice rescoring (Blackwood, 2010), the rescorer only needs to retrieve counts for n-grams present in the lattice produced by the first-pass decoder to be able to assign a score to a hypothesis. As described by Blackwood (2010), obtaining relevant n-grams with their counts involves scanning a very large text file containing n-grams and counts and keeping the relevant records.

These two methods become progressively slower with larger amounts of data and we would like to improve on them for more rapid experimentation. We also would

like to use as lightweight a computing infrastructure as possible. For example, HBase has been applied to the use of distributed language models (Yu, 2008). However, we wish to address the question whether we can adapt this heavy infrastructure to our purposes with minimal effort. N-gram count filtering and rule filtering are two essential steps in our pipeline that can be a bottleneck. Our goal is to reduce their processing time from several hours to a few minutes.

This paper addresses the problem of retrieving relevant translation and language model probabilities by storing models in the HFile data structure.[1] To our knowledge, this is the first detailed proposed implementation of translation and language model storage and filtering using HFile data structures. We believe it offers a good compromise between speed, performance and ease of implementation. Although the HFile construction is done via MapReduce and a cluster of machines, the infrastructure for filtering is lightweight and requires the use of only one machine. We will apply this approach for two specific tasks, namely test set rule filtering prior to decoding and n-gram count filtering to build a stupid backoff model (Brants et al., 2007) for lattice rescoring (Blackwood, 2010). We will discuss alternative strategies as well as their strengths and weaknesses in terms of speed and memory usage. In Section 2, we will review approaches that have been used for model filtering. The HFile data structure that is used to store models will be presented in Section 3. Our method and alternative strategies will be compared empirically in Sections 4 and 5. We will finally conclude in Section 6.

2. Related Work

We now review techniques appearing in the literature that have been used to store SMT models and to retrieve the information needed in translation from these models. SMT models are usually discrete probabilistic models and can therefore be represented as a set of key-value pairs. To obtain relevant information from a model stored in a data structure, a set of keys called a *query set* is formed, then each key in this query set is looked up in the model. Strategies include storing the model as a simple data structure in memory, in a plain text file, in more complicated data structures in memory, storing fractions of the entire model, simply storing data as opposed to a precomputed model or storing models in a distributed fashion.

If small enough, it may be possible to fit the model into physical memory. In this case the model can be stored as a memory associative array, such as a hash table, for rapid query retrieval. In-memory storage has been used to store model parameters between iterations of expectation-maximisation for word alignment (Dyer et al., 2008; Lin and Dyer, 2010).

For larger models, the set of key-value pairs can be stored as a table in a single text file on local disk. Values for keys in the query set are retrieved by scanning through

[1] http://hbase.apache.org

the entire file. For each key in the file, its membership is tested in the query set. This is the approach adopted in the *Joshua 3.0* decoder (Weese et al., 2011), which uses regular expressions or n-grams to test membership. Venugopal and Zollmann (2009) use MapReduce to scan a file concurrently: a mapper is defined that tests if the vocabulary of a rule matches the vocabulary of a test set. The MapReduce framework then splits the grammar file into subsections for the mappers to scan over in parallel.

The model can also be stored using a trie associative array (Fredkin, 1960). A trie is a type of tree where each node represents a shared prefix of a set of keys represented by the child nodes. Each node only stores the prefix it represents. The keys are therefore compactly encoded in the structure of the trie itself. Querying the trie is a $\mathcal{O}(\log(n))$ operation, where n is the number of keys in the dataset. The trie may also be small enough to fit in physical memory to further reduce querying time. Tries have been used for storing phrase tables (Zens and Ney, 2007) and hierarchical phrase-based grammars (Ganitkevitch et al., 2012) as well as language models (Pauls and Klein, 2011; Heafield, 2011).

It is also possible to create a much smaller approximate version of the model. Randomised language models (Talbot and Osborne, 2007b,a; Talbot and Brants, 2008) store parameters or counts associated with n-grams in a structure similar to a Bloom filter (Bloom, 1970). This structure is small in comparison to the original language model, although the reduction in size comes at the cost of randomly corrupting model parameters or assigning model parameters to unseen n-grams. Guthrie and Hepple (2010) propose an extension which prevents the random corruption of model parameters but does not stop the random assignment of parameters to unseen n-grams. Levenberg and Osborne (2009) extend randomised language models to stream-based language models. Another way of building a smaller approximate version of a model is to retain items with high frequency counts from a stream of data (Manku and Motwani, 2002). This technique has been applied to language modelling (Goyal et al., 2009) and translation rule extraction (Przywara and Bojar, 2011).

Instead of pre-computing the dataset it is possible to compute the sufficient statistics at query time using a suffix array (Manber and Myers, 1990), so that the model can be estimated on the fly. A suffix array is a sequence of pointers to each suffix in a training corpus. The sequence is sorted with respect to the lexicographic order of the referenced suffixes. Suffix arrays have been used for computing statistics for language models (Zhang and Vogel, 2006), phrase-based systems (Callison-Burch et al., 2005; Zhang and Vogel, 2005), and hierarchical phrase-based systems (Lopez, 2007).

Finally, some approaches store language models in a distributed fashion. Brants et al. (2007) describe a distributed, fast, low-latency infrastructure for storing very large language models. Zhang et al. (2006) propose a distributed large language model backed by suffix arrays. HBase has also been used to build a distributed language infrastructure (Yu, 2008). The method we propose to use is closely related to the latter but we use a more lightweight infrastructure than HBase and we apply it to two different tasks, demonstrating the flexibility of the infrastructure.

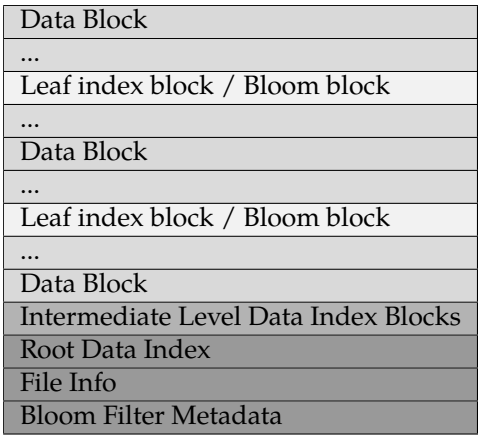

Figure 2. HFile internal structure [2]

3. HFile Description

We now describe the data structure we use to store models and we review relevant features to the design of our system. To store a model represented as key-value pairs, we use the HFile file format,[3] which is a reimplementation of the SSTable file format (Chang et al., 2008). The HFile is used at a lower level in the HBase infrastructure. In this work, we reuse the HFile format directly without having to install an HBase system. The HFile format is a lookup table with key and value columns. The entries are free to be an arbitrary string of bytes of any length. The table is sorted lexicographically by the key byte string for efficient record retrieval by key.

3.1. Internal structure

As can be seen in Figure 2, the data contained in an HFile is internally organised into blocks called data blocks. The block size is configurable, with a default size of 64KB. Note that HFile blocks are not to be confused with Hadoop Distributed File System (HDFS) blocks whose default size is 64MB. If an HFile is stored on HDFS, several HFile blocks will be contained in an HDFS block. A block index is constructed which maps the first key of an HFile block to the location of the block in the file. For large HFiles the block index can be very large. Therefore the block index is itself organised into blocks, which are called leaf index blocks. These leaf index blocks

[2] after http://hbase.apache.org/book/book.html (simplified)

[3] http://hbase.apache.org

are interspersed with the data blocks in the HFile. In turn, the leaf index blocks are indexed by intermediate level data index blocks. The intermediate blocks are then indexed by a root data index. The root data index and optionally the Bloom filter metadata, described next, are stored at the end of the HFile. In order to distinguish block types (data block, index block, etc.), the first 8 bytes of a block will indicate the type of block being read. The HFile format allows for the blocks to be compressed. The choice of compression codec is selected when the file is created. We choose the GZip compression codec for all our experiments. Block compression is also used in other related software (Pauls and Klein, 2011). For more details, the interested reader can refer to the HBase documentation.[4]

3.2. Record retrieval

When the HFile is opened for reading, the root data index is loaded into memory. To retrieve a value from the HFile given a key, the appropriate intermediate index block is located by a binary search through the root data index. Binary searches are conducted on the intermediate and leaf index blocks to identify the data block that contains the key. The data block is then loaded off the disk into memory and the key-value record is retrieved by scanning the data block sequentially.

3.3. Bloom filter optimization

It is possible to query for a key that is not contained in the HFile. This very frequently happens in translation because of language data sparsity. Querying the existence of a key is expensive as three blocks have to be loaded from disk and binary searched. For fast existence check queries, the HFile format allows the inclusion of an optional Bloom filter (Bloom, 1970). A Bloom filter provides a probabilistic, memory efficient representation of the key set with an $O(1)$ membership test operation. The Bloom filter may provide a false positive, but never a false negative for existence of a key in the HFile. For a large HFile, the Bloom filter may also be very large. Therefore the Bloom filter is also organised into blocks called Bloom blocks. Each block contains a smaller Bloom filter that covers a range of keys in the HFile. Similar to the root data index, a Bloom filter metadata or Bloom index is constructed. To check for the existence of a key, a binary search is conducted on the Bloom index, the relevant Bloom block is loaded, and the membership test performed. Contrary to work on Bloom filter language model (Talbot and Osborne, 2007a,b), this filter only tests the existence of a key and does not return any statistics from the value. If a membership test is positive, the HFile data structure still requires to do a usual search. During the execution of a query, two keys may reference the same index or Bloom blocks. To prevent these blocks from being repeatedly loaded from disk, they are cached after reading.

[4] http://hbase.apache.org/book/book.html

3.4. Local disk optimization

The HFile format is designed to be used with HDFS, a distributed file system based on the Google File System (Ghemawat et al., 2003). Large files are split into HDFS blocks that are stored on many nodes in a cluster. However, the HFile format can also be used completely independently of HDFS. If its size is smaller than disk space, the entire HFile can be stored on the local disk of one machine and accessed through the machine's local file system. We find in Sections 4 and 5 that using local disk is faster than using HDFS.

3.5. Query sorting optimization

Prior to HFile lookup, we sort keys in the query set lexicographically. If two keys in the set of queries are contained in the same block, then the block is only loaded once. In addition, the computer hardware and operating system allow further automatic improvements to the query execution. Examples of these automatic improvements include reduced disk seek time, the operating system caching data from disk,[5] or CPU caching data from main memory (Patterson and Hennessy, 2009).

4. Hierarchical Rule Filtering for Translation

In this section, we describe how the HFile data structure can be used to store a hierarchical phrase-based translation model (Chiang, 2007) and to retrieve rules from a given test set. We describe our system called *ruleXtract*, and compare it to other methods through time and memory measurements.

4.1. Task Description

Given a test set and a hierarchical phrase-based translation model, we would like to retrieve all the relevant rules from the model. A phrase-based rule is relevant if its source is a substring of a sentence in the test set. A hierarchical rule is relevant if, after instantiation of its nonterminals, it is a substring of a sentence in the test set. For example, with a test set containing one sentence "Salut toi", the phrase-based rules with sources "Salut", "toi", "Salut toi" are relevant and the hierarchical rules with sources "Salut X" and "X toi" are relevant.

4.2. HFile for Hierarchical Phrase-Based Grammars

The input to our system *ruleXtract* is a word aligned parallel corpus. First, hierarchical phrase-based rules are extracted using a MapReduce job with no reducer.

[5] The Linux Documentation Project, The File System, http://tldp.org

Then, features that require a pass over the whole training material, such as the source-to-target probability, are computed in parallel using MapReduce jobs. We call these features MapReduce features. We follow Method 3 described by Dyer et al. (2008) to compute translation probabilities. Finally, the outputs of the feature jobs are merged in sorted order and the merged output is converted to an HFile. This step is preferably run on a cluster of machines.

Given a test set and an HFile storing a hierarchical phrase-based grammar, we first generate queries from the test set, then retrieve relevant rules along with their MapReduce features from the HFile. To generate queries, we have a set of allowed *source patterns* and instantiate these patterns against the test set. A source pattern is simply a regular expression. For example, the pattern $\Sigma^{+}X$ represents a rule source side containing a sequence of terminals followed by the nonterminal X. If the input sentence is "Salut à toi", the pattern will be instantiated as "Salut X" and "Salut à X". We impose the following constraints on source pattern instantiation where the first three relate to constraints in extraction and the last one relates to a decoding constraint:

- *max_source_phrase*: maximum number of terminals for phrase-based rules,
- *max_source_elements*: maximum number of terminals and nonterminals,
- *max_terminal_length*: maximum number of consecutive terminals for hierarchical rules,
- *max_nonterminal_span*: maximum nonterminal span in a hierarchical rule.

The source pattern instances are then sorted for more efficient HFile lookup (see Section 3). Each query is then looked up in the HFile and if present, an HFile record is retrieved. We typically run this retrieval step on one machine only.

We now compare our approach to similar approaches whose aim is to obtain rules for a test set.

4.3. Suffix Array for Hierarchical Phrase-Based Grammars

We use the *cdec* software (Dyer et al., 2010) for hierarchical phrase-based rule extraction. The implementation is based on earlier work (Lopez, 2007) which extends suffix array based rule retrieval from phrase-based systems to hierarchical phrase-based systems.

Given a test set, a set of source pattern instances is generated similarly to what is done for *ruleXtract*. Then these source pattern instances are looked up in a suffix array compiled from the source side of a parallel corpus. Rules are then extracted using the word alignment and source-to-target probabilities are then computed on the fly.

4.4. Text File Representation of Hierarchical Phrase-Based Grammars

We now describe an implementation for storing and retrieving from a translation model by the *Joshua* decoder (Weese et al., 2011). The first implementation variant,

which we call *Joshua*, stores the translation model in a text file. Given a test set, each word in the test set vocabulary is mapped to the list of sentences in which it appears. Then, each rule in the translation model is compiled to a regular expression, and each sentence that contains at least a vocabulary word of the rule is matched against this regular expression. If at least one match is successful, the rule is retained. A faster version is provided that matches consecutive terminals in the source side of a rule to the set of n-grams extracted from the test set. We call this version *Joshua Fast*. A parallel version also exists that chunks the grammar file and distributes each chunk processing as a separate process on a cluster running Sun Grid Engine (Gentzsch, 2001). We call this version *Joshua Parallel*. The parallel version using the faster matching algorithm is called *Joshua Fast Parallel*.

4.5. Experimental Setup

We use the following setup:

- Data: we use a small parallel corpus of 750,950 word-aligned sentence pairs and a larger corpus of 9,221,421 word-aligned sentence pairs from the NIST'12 Chinese-English evaluation, to show how systems scale up with more data.
- Grammar extraction: from the parallel corpora, we extract hierarchical grammars with the source-to-target probability feature only, because we do not want feature computation to introduce noise in timing results when comparing different strategies and software implementations. In addition, the suffix array implementation of rule extraction does not generate target-to-source probabilities. Note that in practice, given a vector of parameters, we could simply replace multiple features in the translation model by a single value representing the dot product of the features with the parameter vector. The extraction constraints are
  - max_source_phrase = 9,
  - max_source_elements = 5,
  - max_terminal_length = 5 (redundant with max_source_elements),
  - max_nonterminal_span = 10.

  The small grammars contains approximately 60M rules while the larger grammar contains approximately 726M rules. The grammar we obtain is converted to the *Joshua* format.
- Grammar filtering: for *ruleXtract*, we use these constraints for source pattern instantiation:
  - max_source_phrase = 9,
  - max_source_elements = 5,
  - max_terminal_length = 5 (redundant with max_source_elements),
  - max_nonterminal_span = ∞,

  so that the number of rules obtained after filtering is identical between *ruleXtract* and *Joshua*. For the *Joshua Parallel* configurations, we use 110 jobs for the larger

grammar on a cluster of 9 machines. For this latter configuration, we report the maximum time spent on a job (not the sum) and the maximum memory usage by a job.
- Measurements: we report time measurements for query processing and query retrieval and the total time used to obtain a set specific rule file for a test set of 1755 Chinese sentences and 51008 tokens. We also report peak memory usage. For *ruleXtract*, query processing involves generating source pattern instances and sort them according to the HFile sorting order. If we use a Bloom filter, it also involves pre-filtering the queries with the Bloom filter. Query retrieval involves HFile lookup. For the *Joshua* configurations, query processing involves indexing the test set and generating test set ngrams and query retrieval involves regular expression matching.
- Hardware configuration: the machine used for the query has 94GB of memory and an Intel Xeon X5650 CPU. The distributed file system is hosted on the querying machine and other machines with the same specification, which are used to generate the HFile.

The setup was designed for accurate comparisons between strategies, however these strategies are not necessarily used with this setup in an end-to-end translation system. For example the grammar extracted by *Joshua* is smaller than the grammar extracted by *ruleXtract* because of target side constraints but *ruleXtract* uses filter criteria (Iglesias et al., 2009a) to reduce the test set specific grammar.

4.6. Results and Discussion

Results are summarized in Table 1, from which we can draw the following observations:
- Speed: column *Total Time* shows that *ruleXtract* is competitive with alternative strategies in terms of speed.
- Memory: column *Peak Memory* shows that both *ruleXtract* and *Joshua* memory usage is important. In the case of *ruleXtract*, this is because we keep all source pattern instances in memory. In the case of *Joshua*, this is due to a caching optimization.
- HFile optimization: comparing *HDFS* and *Local* rows, we can see that using the local filesystem as opposed to HDFS gives a small decrease in query retrieval time, more important for the larger grammar. This is due to the fact that HDFS blocks are located on different data nodes. Since the HFile size is smaller than the disk space, it is preferable to work locally, although it requires copying the HFile from HDFS to the hard disk. Comparing rows with and without *Bloom*, we can see that the use of a Bloom filter gives an important decrease in query retrieval time. This is due to the fact that the number of source pattern instances queries is 31,552,746 and after Bloom filtering, the number of queries is 1,146,554 for the small grammar and 2,309,680 for the larger grammar, reducing the num-

Small Grammar					
System	**Query Processing**	**Query Retrieval**	**Total Time**	**Peak Memory**	**# Rules**
ruleXtract HDFS	9m1s	7m36s	16m40s	40.8G	6435124
ruleXtract Bloom, HDFS	8m57s	2m16s	11m15s	39.9G	6435124
ruleXtract Local	8m54s	7m33s	16m30s	40.4G	6435124
ruleXtract Bloom, Local	8m50s	2m19s	11m11s	38.8G	6435124
Joshua	0.9s	29m51s	29m54s	42.2G	6435124
Joshua Fast	0.9s	7m25s	7m28s	40.1G	7493178
Large Grammar					
System	**Query Processing**	**Query Retrieval**	**Total Time**	**Peak Memory**	**# Rules**
ruleXtract HDFS	8m56s	22m18s	31m17s	42.2G	47978228
ruleXtract Bloom, HDFS	9m12	15m33s	24m49s	40.7G	47978228
ruleXtract Local	8m55s	21m3s	30m1s	41.6G	47978228
ruleXtract Bloom, Local	9m0s	14m43s	23m46s	40.6G	47978228
Joshua	0.9s	out of memory	out of memory	out of memory	out of memory
Joshua Fast	0.9s	out of memory	out of memory	out of memory	out of memory
Joshua No Cache	0.9s	537m10s	537m11s	10.1G	47978228
Joshua Fast No Cache	0.9s	78m53s	78m54s	10.1G	83339443
Joshua Parallel	total time (not sum): 43m36s			4G	47978228
Joshua Fast Parallel	total time (not sum): 44m29s			4G	83339443

Table 1. Time and memory measurements for rule filtering with different strategies for a small and a large grammar.

ber of time consuming HFile lookups respectively by 96% and 93%. Note that Bloom filters increase query processing time only in the case of a large grammar and more so when using HDFS.
- Parallelization: in order to run *Joshua* on the larger grammar and avoid memory problems, we needed to use parallelization, which provided competitive speeds and a low memory footprint (maximum 4G per job). We are currently looking into making a parallel version of ruleXtract by parallelizing the query.

For information, *cdec*'s total processing time is 57m40s for the small grammar, which is significantly slower than the other methods. However, we do not include a direct comparison to *cdec* in Table 1 because the suffix array method involves much on-the-fly computation that has been precomputed in the case of *Joshua* and *ruleXtract*. Despite this apparent slowness, the use of suffix array methods for rule extraction favors rapid experimentation because no precomputation is required. But we note that the HFile generation from the larger parallel corpus took 5 hours and from this HFile it is possible to run multiple experiments by varying test sets and/or filtering parameters.

The *ruleXtract* system works in batch mode and dividing the number of words in the test set by the total time in the best configuration (*ruleXtract, Bloom, Local*) for the large grammar yields a speed of 35.8 words per second which is a real time system speed for batch processing tasks in which latency has little effect. However, running the system in that configuration gives a speed of 2.5 words per second for the longest sentence in the test set (135 words) and 1.3 words per second for a sentence of length 20. Future work will be dedicated to reduce latency and obtain an actual real time system.

5. N-Gram Count Filtering for Language Model Lattice Rescoring

In this section, we describe an HFile based infrastructure that supports a stupid backoff (Brants et al., 2007) n-gram language model. We conduct timed queries with comparison to a suffix array baseline.

5.1. Task Description

Blackwood (2010) motivates the use of 5-gram language model rescoring as a way of avoiding memory limitations in language model estimation and decoding. Depending on translation grammar and language model complexity, pruning thresholds in search can be set so that search errors are inconsequential. 5-gram rescoring requires first to obtain n-gram counts for $n \leq 5$ for a large monolingual corpus. Given a test set, n-grams present in the output lattices generated by a first-pass decoder are then extracted. The stupid backoff n-gram model (Brants et al., 2007) is described

with the pseudo-probability $S(\cdot)$. It has the form:

$$S(W_i|W_{i-n+1}^{i-1}) = \begin{cases} \frac{f(W_{i-n+1}^{i})}{f(W_{i-n+1}^{i-1})} & \text{if } f(W_{i-n+1}^{i}) > 0 \\ \alpha S(W_i|W_{i-n+2}^{i}) & \text{otherwise} \end{cases} \quad (1)$$

where W_i^j is a sequence of words contained in a machine translation hypothesis, $f(W_i^j)$ is the count of the occurrences of the word sequence W_i^j in a large monolingual corpus, and α is a constant that is set heuristically. To compute the pseudo-probability of an n-gram $S(W_i|W_{i-n+1}^{i-1})$ the only statistics required are the counts for the constituent word sequences extracted from the monolingual corpus. Brants et al. (2007) show that with large amounts of data, stupid backoff smoothing performs similarly to Kneser-Ney smoothing (Kneser and Ney, 1995).

5.2. HFile for n-gram count filtering

The HFile stores n-grams W_i^j as keys, and their counts $f(W_i^j)$ as values. Each word of the key is mapped to an integer so that the n-gram becomes a string of integers. Each integer is then converted into a binary representation with a three byte width, which is adequate for the vocabulary used by our collections. The count is stored using a four byte integer representation.

5.3. Suffix array

The suffix array baseline is a modified version of the Suffix Array Language Model toolkit (SALM) toolkit (Zhang and Vogel, 2006). The original SALM toolkit used a 32-bit integer representation for each element in the suffix array. This representation has been widened to 64-bits to allow a larger corpus to be indexed. SALM loads the suffix array and monolingual corpus into memory for fast computation of the counts.

5.4. Experimental setup

We use the following setup:

- Data: We use a concatenation of the Gigaword Fifth Edition (Parker et al., 2011) with the English side of the NIST'12 parallel data for the constrained track. The SALM toolkit imposes a 256 word limit on sentence length in the corpus, therefore we truncated all sentences to 256 words. The corpus contains 5.4 billion words. From the monolingual corpus we extract 2.5 billion word sequences and counts. These are stored in an HFile with 8 KB data block size.
- Translation lattices: we replicate an experiment where a set of 2816 translation lattices are rescored using a 5-gram stupid backoff language model (Blackwood, 2010). The n-gram keys required to build the set-specific language model are extracted from the lattices using modified counting transducers (Mohri, 2003).

The queries take the form of 8.4 million keys, of which 7.3 million of the keys are unique.
- HFile optimization: we execute four HFile based queries based on whether the HFile contains a Bloom filter index, and whether the HFile is stored on local disk or a distributed file system.
- Time measurement phases: we split the query execution into distinct phases. For SALM we record the time taken to load the suffix array and monolingual corpus into memory, which we label index load time. We then enumerate through the unsorted keys in the query and compute the count associated with the key. Note that for any duplicate key in the query a duplicate count is computed. We call this phase query retrieval. For the HFile based infrastructure, the query has to be sorted. A Bloom filter may also be applied after the sort. We call this phase query processing. We then look up the HFile to locate the query keys. The look up phase is also labelled query retrieval.
- Hardware configuration: identical to the one in Section 4.

5.5. Results and Discussion

The results are shown in Table 2 from which we can draw the following observations:
- Speed: column 4 shows that the HFile infrastructure provides a competitive query speed with respect to SALM.
- Memory: column 5 shows that the memory overhead of the HFile infrastructure is much lower than SALM. We could reduce the suffix array memory usage by doing an on-disk binary search but this would increase the query processing time.
- HFile optimizations: an interesting result is the effect that the Bloom filter has on the query processing time for the distributed query. The time spent loading the blocks that comprise the Bloom filter offsets the time saved retrieving the counts. However, when using local disk the Bloom filter has only a small impact on the query processing time.

In addition, although disk usage is not an issue, it is worth mentioning that the English monolingual data together with the suffix array represent 90G of uncompressed data and the HFile size is 11G without Bloom filter and 14G with Bloom filter. We store the English monolingual data in a decompressed file for more efficient loading into a suffix array. On the other hand, only HFile blocks potentially containing a key are uncompressed during an HFile query.

We did not report comparisons to the KenLM toolkit (Heafield, 2011), which is designed for retrieving n-gram probabilities from an ARPA file as opposed to raw n-gram counts. It might be possible to build an ARPA file containing n-gram counts; we leave this study to future work and hope to obtain improvements in speed.

	Index Load	Query Processing	Query Retrieval	Total Time	Peak Memory
Suffix Array	8m39s	-	3m20s	11m59s	90.7G
HFile, HDFS	-	18s	3m54s	4m12s	3.1G
HFile, Bloom, HDFS	-	1m11s	2m52s	4m3s	5.8G
HFile, Local	-	18s	3m5s	3m23s	3.1G
HFile, Bloom, Local	-	25s	1m56s	2m21s	5.8G

Table 2. Timing results for n-gram count queries.

6. Conclusion

We have presented a strategy to filter SMT models to a given test set. This strategy is easy to implement, flexible as it was applied to two different tasks and it does not require extensive computing resources as it is run on one machine. We have demonstrated that its performance in terms of speed and memory usage is competitive with other current alternative approaches.

In the future, we would like to provide two extensions to our HFile infrastructure. First, in order to increase the speed in batch mode, we would like to implement a MapReduce version that would split the queries (as opposed to the HFile). Second, in order to provide a real time system, we would like to reduce latency by optimizing the source pattern instance creation phase.

Acknowledgements

This research was funded by two EPSRC DTA studentships for the first two authors, the European Union Seventh Framework Programme (FP7-ICT-2009-4) under grant agreement number 247762 and the DARPA BOLT program. We thank Gonzalo Iglesias for providing advice on an earlier draft, Juri Ganitkevitch and Matt Post for their help with the Joshua software and Kenneth Heafield and Jonathan Clark for answering questions about the KenLM software.

Bibliography

Blackwood, Graeme. *Lattice Rescoring Methods for Statistical Machine Translation*. PhD thesis, University of Cambridge, 2010.

Bloom, Burton H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, July 1970. ISSN 0001-0782. doi: 10.1145/362686.362692.

Brants, Thorsten, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. 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, Prague, Czech Republic, June 2007. Association for Computational Linguistics.

Callison-Burch, Chris, Colin Bannard, and Josh Schroeder. 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, June 2005. Association for Computational Linguistics. doi: 10.3115/1219840.1219872.

Callison-Burch, Chris, Philipp Koehn, Christof Monz, Matt Post, Radu Soricut, and Lucia Specia. Findings of the 2012 workshop on statistical machine translation. In *Proceedings of the Seventh Workshop on Statistical Machine Translation*, pages 10–51, Montréal, Canada, June 2012. Association for Computational Linguistics.

Chang, Fay, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4:1–4:26, June 2008. doi: 10.1145/1365815.1365816.

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

de Gispert, Adrià, Gonzalo Iglesias, Graeme Blackwood, Eduardo R. Banga, and William Byrne. Hierarchical phrase-based translation with weighted finite-state transducers and shallow-n grammars. *Computational Linguistics*, 36(3):505—533, 2010. ISSN 0891-2017.

Dean, Jeffrey and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492.

Dyer, Chris, Aaron Cordova, Alex Mont, and Jimmy Lin. Fast, easy, and cheap: Construction of statistical machine translation models with MapReduce. In *Proceedings of the Third Workshop on Statistical Machine Translation*, pages 199–207, Columbus, Ohio, June 2008. Association for Computational Linguistics.

Dyer, Chris, Adam Lopez, Juri Ganitkevitch, Jonathan Weese, Ferhan Ture, Phil Blunsom, Hendra Setiawan, Vladimir Eidelman, and Philip Resnik. cdec: A decoder, alignment, and learning framework for finite-state and context-free translation models. In *Proceedings of the ACL 2010 System Demonstrations*, pages 7–12, Uppsala, Sweden, July 2010. Association for Computational Linguistics.

Fredkin, Edward. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960. ISSN 0001-0782. doi: 10.1145/367390.367400.

Ganitkevitch, Juri, Yuan Cao, Jonathan Weese, Matt Post, and Chris Callison-Burch. Joshua 4.0: Packing, PRO, and paraphrases. In *Proceedings of the Seventh Workshop on Statistical Machine Translation*, pages 283–291, Montréal, Canada, June 2012. Association for Computational Linguistics.

Gao, Qin and Stephan Vogel. Training phrase-based machine translation models on the cloud: Open source machine translation toolkit Chaski. *The Prague Bulletin of Mathematical Linguistics*, 93:37–46, January 2010. doi: 10.2478/v10108-010-0004-8.

Gentzsch, Wolfgang. Sun grid engine: Towards creating a compute power grid. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, CCGRID '01, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1010-8.

Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945450.

Goyal, Amit, Hal Daume III, and Suresh Venkatasubramanian. Streaming for large scale NLP: Language modeling. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 512–520, Boulder, Colorado, June 2009. Association for Computational Linguistics.

Guthrie, David and Mark Hepple. Storing the web in memory: Space efficient language models with constant time retrieval. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 262–272, Cambridge, MA, October 2010. Association for Computational Linguistics.

Heafield, Kenneth. KenLM: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, pages 187–197, Edinburgh, Scotland, July 2011. Association for Computational Linguistics.

Iglesias, Gonzalo, Adrià de Gispert, Eduardo R. Banga, and William Byrne. Rule filtering by pattern for efficient hierarchical translation. In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*, pages 380–388, Athens, Greece, March 2009a. Association for Computational Linguistics.

Iglesias, Gonzalo, Adrià de Gispert, Eduardo R. Banga, and William Byrne. Hierarchical phrase-based translation with weighted finite state transducers. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 433–441, Boulder, Colorado, June 2009b. Association for Computational Linguistics.

Kneser, Reinhard and Hermann Ney. Improved backing-off for m-gram language modeling. In *Proceedings of ICASSP*, volume 1, pages 181–184, 1995.

Levenberg, Abby and Miles Osborne. Stream-based randomised language models for SMT. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 756–764, Singapore, August 2009. Association for Computational Linguistics.

Lin, Jimmy and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

Lopez, Adam. Hierarchical phrase-based translation with suffix arrays. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 976–985, Prague, Czech Republic, June 2007. Association for Computational Linguistics.

Manber, Udi and Gene Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first ACM-SIAM symposium on Discrete algorithms*, pages 319–327. Society for Industrial and Applied Mathematics, 1990.

Manku, Gurmeet Singh and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 346–357. VLDB Endowment, 2002.

Mohri, Mehryar. Edit-distance of weighted automata: General definitions and algorithms. *International Journal of Foundations of Computer Science*, 14(6):957–982, 2003.

Parker, Robert, David Graff, Junbo Kong, Ke Chen, and Kazuaki Maeda. English gigaword fifth edition. In *Linguistic Data Consortium, Phildelphia*, 2011.

Patterson, David A. and John L. Hennessy. *Computer Organization and Design: The Hardware/software Interface*. Morgan Kaufmann, 2009.

Pauls, Adam and Dan Klein. Faster and smaller n-gram language models. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 258–267, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

Przywara, Česlav and Ondřej Bojar. eppex: Epochal phrase table extraction for statistical machine translation. *The Prague Bulletin of Mathematical Linguistics*, 96:89–98, 2011.

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

Talbot, David and Miles Osborne. 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, June 2007a. Association for Computational Linguistics.

Talbot, David and Miles Osborne. 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, Prague, Czech Republic, June 2007b. Association for Computational Linguistics.

Venugopal, Ashish and Andreas Zollmann. Grammar based statistical MT on Hadoop: An end-to-end toolkit for large scale PSCFG based MT. *The Prague Bulletin of Mathematical Linguistics*, 91:67–77, January 2009. doi: 10.2478/v10108-009-0017-3.

Weese, Jonathan, Juri Ganitkevitch, Chris Callison-Burch, Matt Post, and Adam Lopez. Joshua 3.0: Syntax-based machine translation with the Thrax grammar extractor. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, pages 478–484, Edinburgh, Scotland, July 2011. Association for Computational Linguistics.

Yu, Xiaoyang. Estimating language models using Hadoop and HBase. Master's thesis, University of Edinburgh, Edinburgh, 2008.

Zens, Richard and Hermann Ney. Efficient phrase-table representation for machine translation with applications to online MT and speech translation. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pages 492–499, Rochester, New York, April 2007. Association for Computational Linguistics.

Zhang, Ying and Stephan Vogel. An efficient phrase-to-phrase alignment model for arbitrarily long phrase and large corpora. In *Proceedings of the 10th Conference of the European Association for Machine Translation (EAMT-05*, pages 294–301, 2005.

Zhang, Ying and Stephan Vogel. Suffix array and its applications in empirical natural language processing. Technical Report CMU-LTI-06-010, Language Technologies Institute, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, December 2006.

Zhang, Ying, Almut Silja Hildebrand, and Stephan Vogel. Distributed language modeling for n-best list re-ranking. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pages 216–223, Sydney, Australia, July 2006. Association for Computational Linguistics.

Address for correspondence:
Juan Pino
jmp84@cam.ac.uk
Department of Engineering
University of Cambridge
Cambridge
CB2 1PZ, U.K.