UNIVERSITÄT
DES
SAARLANDES

Master's Thesis

Application of Suffix Trees as an Implementation Technique for Varied-Length N-gram Language Models

Casey Redd Kennington
2010-2011

Supervisor Saarland University: Martin Kay
Supervisor University of Nancy 2: Patrick Blackburn

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Declaration

I hereby confirm that the thesis presented here is my own work, with all assistance acknowledged.

Saarbrücken, 15 September 2011
Signature:

Casey Kennington

Abstract

Suffix trees have been used in various applications in natural language processing because of their ability to represent sequential data in a way that is efficient to build, store, and access. How suffix trees are built and their properties are discussed as well as how this relates to language modeling. Language modeling is explained generally, and in some areas with more detail to illustrate how the suffix tree is applied to language modeling. The suffix tree language model is then compared to other existing language models in perplexity scores and memory usage, and further compared to one language model in several machine translation experiments. It is shown that the suffix tree language model is competitive with the state of the art in machine translation experiments and that suffix trees are well-suited for language modeling.

Contents

1 Acknowledgements

Thanks and appreciation go to Professor Martin Kay who conjured up the idea of using suffix trees as language models, as well as his insights, experience, and valuable advise. Thanks also go to Dr. Jia Xu who was instrumental in some of the more technical aspects of this thesis, as well as being the catalyst for gaining access to corpora and computing clusters. Annemarie Friedrich was also essential as she implemented the first working version of a suffix tree language model upon which the implementation of this thesis was based. Her knowledge and advice have also been essential to the success of this thesis. Also much appreciation to Professor Dietrich Klakow, Grzegorz Chrupala, and Saeedeh Momtazi for their shared practical and technical knowledge about language modeling, as well as their time. Thanks also goes to Andreas Stolke for his useful insights into SRILM, and Victor Santos for his help in the project which was a forerunner to this thesis.

Finally, I must give a multitude of thanks to my wife, Katie, who supports me in everything.

2 Introduction

This thesis explores the use of a data structure called a *suffix tree* as the core component of a language model, specifically tailored for use in machine translation experiments. This approach allows for greater exploitation of training data by utilizing n-grams of varying size.

The main focus of this thesis is to greater expose the suffix tree data structure as an extremely useful approach to various natural language processing tasks. To this end, a fair amount of time is spent in Section 3 describing how suffix trees are built and what the represent. "Interesting" properties and patterns are pointed out along the way.

The fact that a suffix tree has been implemented into a usable language modeling program through this thesis is certainly important, but also of importance is that it is only one example of how suffix trees are useful in natural language processing. To explain this more fully, time is spent on the language modeling aspect of the suffix tree; differences with traditional n-gram models, similarities, and properties of the tree which are useful in language modeling, some information about what is stored in the suffix tree, as well as several applications of language models are all discussed in Section 4.

Section 7 shows several experiments which compare the suffix tree language model with the current state of the art using several metrics. These show that it is competitive in at least one domain: machine translation, and that the arbitrary length of n-gram representation is quite compact in size when compared to very large n-gram trained language models.

3 Suffix Trees

3.1 Overview

In 1973, Peter Weiner [Weiner, 1973] introduced a data structure which would soon be known as a *suffix tree*, a data structure which is often used for storing words or sequences of words so they can be looked up easily. In 1976, McCreight [McCreight, 1976] improved upon the idea with a simpler construction algorithm. This thesis, however, will focus on another algorithm introduced in 1995 by Esko Ukkonen [Ukkonen, 1995]. This algorithm is not only easier to understand and on-line in suffix tree construction, it offers several properties that make it useful for language modeling.

We use the term *suffix* not in the standard linguistic sense, but rather in a technical sense. Certainly the concept is of course related in principle. To explain a suffix tree, it is helpful to begin with a related, simpler structure known the *suffix trie*, a data structure which has interesting properties and many uses. A suffix trie represents every suffix of a sequence of data. That sequence can in principle be anything like words within a corpus of text or characters within a word. If the suffix tree were on the word level, the first branch would be the entire corpus, even across sentence boundaries. The second branch would be a copy of the first branch, minus the first word of that branch and the third branch would be a copy of the second branch minus the first word, and this process continues until the final branch which would represent the final word of the corpus. A simple example of a suffix trie in its most general form:

this is a short corpus of text
is a short corpus of text
a short corpus of text
short corpus of text
corpus of text
of text
text

The rest of this section will deal with how to create this suffix trie representation such that it loses no information, is built quickly, and uses a relatively small amount storage.

Figure 1 shows another suffix trie constructed with a small corpus that we are going to use as our example as we improve the suffix tree building. In this example, assume that the whole corpus consists of five words, 'c', 'a', 'c', 'a', 'o' (following the example used in Ukkonen [Ukkonen, 1995]) which, in this case, are just one-letter words. In principle, strings of any length can be used, and for a strings to be the same as another it must have the exact same spelling and case. The words are represented as nodes with arcs to connect the nodes. The left-most node is the root node of the tree and becomes the common starting place. Already this example has taken each suffix and stored it in a slightly better way than our first example, thus reducing much redundancy. It is still clear that each of the 5 suffixes of *cacao* (*acao*, *cao*, *ao*, and *o*) can be found by starting at the root node and following the sequence into the tree, as there can only be at most one branch for a given word branching from a given node.

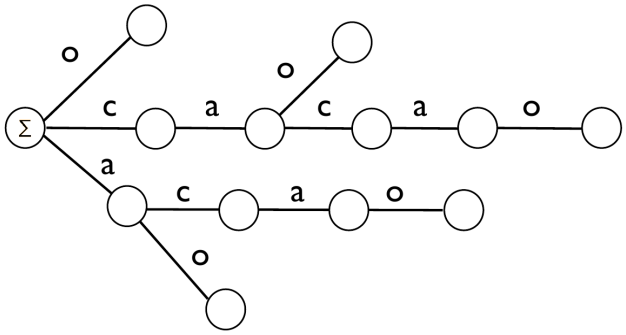

Figure 1: Expanded Suffix Tree for the corpus 'cacao'

This trie structure already has some useful properties, as pointed out by Martin Kay [Kay, 2004]. First, every different substring of the text is represented by a path from the root node to some node. Also notice the nonterminal nodes which have child branches; the node *a* directly off the root node, for example. We can conclude that the path from the root node to any node like this which we can reach in the tree was represented in our corpus more than once, because they branch. Thus, simply by looking at the tree we can tell which substrings occurred more than once simply by looking at the branching nodes. This kind of pattern has interest when one is seeking well-represented substrings. The number of branching nodes simply depends on the training corpus, but it may also be of use to note that the number of these branching nodes is always less than the number of suffixes. In our example, cacao has 5 suffixes including the entire corpus, but the number of branching nodes is two. Further, there are two other properties of this structure. First, when the initial suffix is placed in the tree, no branching nodes are created. Secondly, the entry into the trie of each succeeding suffix will create at most one new branching node.

However, despite these useful properties the practicality of building a trie is still exponential in memory and training time. This is where we shift our discussion from suffix tries to suffix *trees*, which is a way of building and storing suffix tries such that they are compact in size and quick to build and access without loss of information.

3.2 Memory-saving representation of Suffix Trees

Notice in Figure 1 that there are multiple nodes in the tree, each of which have only one child node all the way until the final leaf nodes. We can compact the tree by merging these nodes into one sub-branch and concatenate the words on the arcs into one string sequence which then is the label of that arc, as shown in Figure 2.

This brings the number of nodes down dramatically and all suffixes can still be reached, but this does not necessarily have a fundamental effect on size because of the length of most of the arc labels, which now carry most of the information. However, because all suffixes are sub-parts of the original corpus, the entire corpus can be made into a separate array of words, and the words in the tree now simply point to the words in the list. These pointers are simple integers which typically

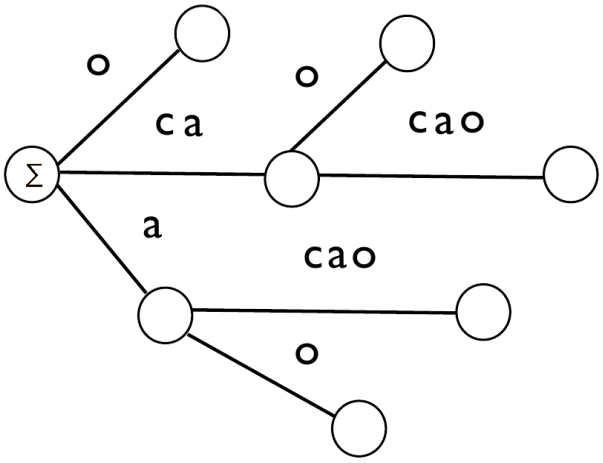

Figure 2: Compacted Suffix Tree for the corpus 'cacao'

take less space than words. Figure 3 illustrates this principle. Importantly, not every word needs to be represented in a node; only the beginning point and end point of what the node represents. For example, the string *ca* can be represented with integer pointers 1 and 2, and the string *acao* can be represented by 2 and 5. Hence, all nodes are constant in their storage of what they represent.

Again, using the same cacao corpus, now with integer representations of the indices of the vocabulary words:

1	2	3	4	5
1	2	1	2	3
c	a	c	a	o

Now, instead of 'ca', one can write $(1, 2)$ or instead of 'cao', one can simply write $(3, \infty)$ which means from field 3 to the end of the corpus.

Furthermore, as an implementation detail, each arc and the node which it leads into is collapsed to become single arc-node object, saving more storage space. Further, not only the corpus but the vocabulary set can be indexed. Thus all words are only represented once in their string form, which can be represented by a number which is further stored in the list of words which make up the corpus (one may point out here that this is simply a suffix array, a very similar data structure to a suffix tree, but there are some important differences which will be discussed later). In our cacao example, the vocabulary set is just c, a, and o. The vocabulary set indexing becomes interesting really only when dealing with words, not just letters, because it only really helps with large vocabularies.

This transformation brings the number of arcs and nodes from a maximum of $N(N+1)/2$, where N is the number of words in a corpus, and it is maximum because this is only reached if all branches are binary and divided by 2 because this is how many substrings there are, down to $2N - 1$, which means the number of nodes necessary to represent the tree is linear to the size of the corpus.

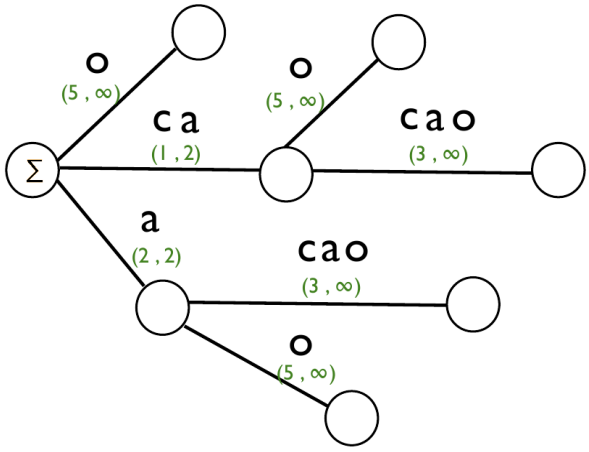

Figure 3: Compacted Suffix Tree for the corpus 'cacao' using Indices

3.3 Ukkonen's Algorithm

Now that the tree is small, the goal is now to build the tree more efficiently. To this end, [Ukkonen, 1995] presents an on-line algorithm for building suffix trees in linear time. The interested reader is referred also to [Gusfield, 1997] for a more detailed and technical explanation of the algorithm (with proofs). However, I do hope to explain the algorithm in sufficient detail to show that the very process of building the tree can expose some interesting patterns about sequential data, and therefore language modeling.

Further, because of the on-line nature of the algorithm, one can begin with the first word and building the tree such that it can stop at any time and the tree fully represents the training corpus to that particular point in the training. His method also builds the tree by the most compact and technical representation, as described previously. Importantly, for the suffix tree to be complete, a final *sentinel character* is added to the tree. The reason for this addition will be explained later.

An important detail for language modeling and distinction from suffix arrays is that Ukkonen's algorithm uses *suffix links* during the process of building the tree. Each node may have only one suffix link pointing to a node which is nearer to the root node and which has the same subtree, i.e. the same branches, and in general it usually has more branches. Figure 4 shows some of the suffix links. More explicitly, if one branch contains the string $\alpha\beta$, where α is the first word of that string, then the suffix link is a direct pointer to β. The tree in this example actually contains more suffix links which are not shown for readability reasons, such as the link from node 3 to node 5 and a link from node 5 to the root node. The circular arc at the root node represents the *unknown word*, or *UNK* which will be discussed later.

The algorithm begins with the root node and simply adds the first word in the corpus as a child of the root node. The algorithm continues with the next word by seeing if a branch from the root node to that word already exists. If the branch doesn't exist then it creates a new branch for that node. If the branch does exist, it then

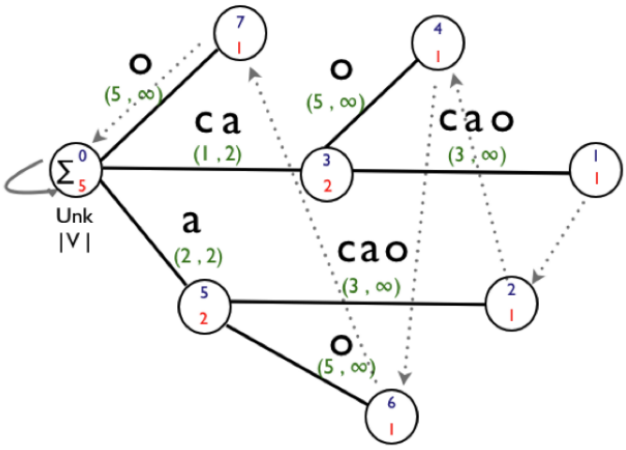

Figure 4: Compacted Suffix Tree including suffix links (dotted arcs) and suffix counts (red numbers of nodes). The darker numbers on top of each node are the node IDs.

follows the already-created branch and then processes the following word. Once a word is placed where it belongs, the fact that this particular branch for this word exists needs to be updated and represented in other parts of the tree in order for the tree to be a true and complete suffix tree. This is where the suffix links become essential. It was mentioned earlier that a suffix link is a pointer to another node in another branch of the tree which represents the same sequence minus the first word. If that suffix node doesn't already have the word as a child node, a new branch will need to be added to that suffix as well, and once it is added there it must continue to follow the suffix link of that node, adding a new branch to each necessary node in the path of suffixes. To summarize everything to this point, the tree adds a word to the longest suffix branch in the tree, starts at the end, and follows the suffix links back to the root node adding the new word as a child to each node which doesn't already have that word as a child node until it reaches the root node.

The use of these suffix links helps in the efficiency of the tree construction, but it is not enough to make building linear, which is the claim. To this end, the number of nodes which are handled each time a word is added must be optimized in such a way as to cut out visiting nodes which already have the word as a branch. To accomplish this, one must distinguish between two types of nodes: *leaf* and *internal*. No matter what the type, each node has information about where the words begin and end as represented by that node (known as the left and right indexes). In Figure 4, node 3 is represented by *ca*, which really means it knows that the sequence of words which it represents begins at index 1 and ends at index 2, representing two sequential words. Because that node has children it is an internal node and the first child which branches from that child node must have a left index which is one more than the node's right-most index. Notice in Figure 4 that the two nodes which have children, nodes 3 and 5, both follow this pattern. The other node type includes the nodes which have no children, the leaf (or *terminal*) nodes, which also have an interesting property. Notice that each leaf node in Figure 4 has infinity as the right index. This is really just a variable that points to the last word in the list of words in the training

corpus. It can also simply be seen as a variable which represents the number of words in the corpus. Thus, if another character is added to the cacao training corpus, the infinity variable will point to that without ever changing or needing to be updated. Leaf nodes therefore are dynamic in what they point to as the tree grows.

Having these two node types allow the tree to systematically grow while not needing to update every node. Here, we will look into greater detail on how this is accomplished. Before the tree began with the longest branch and added the node along the suffix path. However, because of the dynamic nature of leaf nodes, adding the new word to that node by changing the index is done automatically by virtue of the infinity pointer. This is the case for all leaf nodes which will need this update. We can therefore turn our focus to internal nodes where more involved updating must occur. But first, because leaf nodes need to special attention, making them a starting point for adding new words is meaningless. We therefore define an *active point* which points to the internal node which becomes the starting point for the newly added word, whereas before we just found the longest branch. Now, as a word is added to a tree it begins at the active point and follows the suffix link chain back to the root node, adding new branches as usual. However, there is another case which can allow the tree to stop short of an entire journey back to the root node and skip further unnecessary calculations. We know that if word α and string β are the branch of the tree, then β will also be in the tree. This means that anytime along the path of suffix links back to the root node, if the word already exists as a child of one of the suffix nodes, then it need proceed no further because that word will be a child node of all other nodes along the suffix links until it reaches (and including) the root node.

To summarize in adding a word to the tree with what we now know, the tree begins at the active point and adds the new child node. As it follows the suffix links back to the root node, it first checks to see if the child node already exists. If it doesn't it is added. If it does exist, then that word is completely added to the tree. The leaf nodes are updated automatically. By adding these two tricks, building the tree is further optimized. I wish to impress upon the reader that it is the suffix links which allow the algorithm to approach linearity in construction time. Without the suffix links, adding a new word to each part of the tree where it belongs would require starting at the root node and walking through the tree until all nodes which require the new word are found. Yet, the suffix links alone are not enough to make the jump to linear time. There are yet more optimizations which must be made in order for building to be truly linear. To accomplish this, we must look closer at the internal nodes.

Recall that some internal nodes represent more than one word. If one of these nodes lies along the suffix link path and needs to have a word added to them, then finding the place where that word must be placed would require stepping through each word which that node represents until the proper place is found for the new word to be added. Unless this traversal can somehow be optimized, then the worst case scenario for building a suffix tree would still approach quadratic. This is addressed by an *offset* variable which is an index on which word in a multi-word node needs to have the new word added. When a new word is added to the tree, it begins at the active point, as explained, which is some internal node. When the new word is added to this node, an offset is used to keep track of how many words into the sequence it was placed. Ukkonen called it a *reference pair*, but for my purposes, it is useful to

be seen as an offset. As the word is added to the nodes along the suffix link chain, if the number of words represented by each of those nodes is equal to or greater than that offset, then the word is added where the offset dictates, which is inside that sequence. If the node doesn't have enough words to accommodate a long offset, then a kind of traversal is necessary to find the correct placement of the word. The length of the sequence of words in the node is subtracted from the offset because the offset will have traversed at least that many words to get to the next node. Finding the next node requires finding the child node which represents the word represented by the first index of the offset. If this new node has more words than the offset, then the word to be added exists in that node. If it again is too short, then the process is repeated as many times as necessary.

To illustrate several of these cases, turn again to Figure 4. If node 1 is the starting point in adding a new character, x, it is trivial because it would only need to update the leaf nodes, and leaf nodes are already automatically updated by nature of the infinity variable. The only node where x would need to be added is to the root node. However, if an added character was already in the alphabet, c, for example, the leaf nodes would extend as before, but as the suffix links reach back to the root node, c already exists as a child of the root node in node 3, which contains the sequence ca. Because the c is already represented, it is technically done, but the fact that it has now been seen twice must somehow be represented. This is where the sentinel character becomes necessary. By using a special token, ca and be branched, creating a node out of c which has this special token as a child and a as another child. This still represents the sequence ca, but it also shows that c is a character which ends the corpus. Adding the sentinel character is done simply by adding it to the tree as the final token. Of course, this token must not already be in the vocabulary.

To illustrate another case, return to the original tree in Figure 4 and assume for a moment that a word was added to the tree which requires an offset of 3, and the suffix links have been followed back to the root node. The root node has an offset of 0 by definition, so subtracting that from the offset makes no difference. Hence, the first word pointed to by the index of the offset needs to be followed. Assume that it happens to be c, which means the length of that node is subtracted from the offset, leaving the offset to now be 1. Thus, the process needs to continued with the first character indexed by the offset. If the character happens to be o, then that child node exists and the rest of the offset can easily be satisfied. The same occurs if c is the next character, as it too is represented already. If the character happens to be a new one, then a new child node needs to be created which will then satisfy the offset.

Finally, the UNK character in Figure 4 is added. This is a placeholder for any word which doesn't exist in the corpus. For example, if I use a corpus to create a suffix tree, and I wish to see if a sentence or parts of that sentence are in the suffix tree, then I can begin with the first word of a sentence and see if that word is represented in a node directly off the root node. If so, it can continue onto the second word, seeing if that word follows the first word. The process continues until a word in the sentence cannot be found. It can then follow the suffix links back, looking for the word. If after following the suffix links back to the root node, thte word is not represented at all in the tree, then it is mapped to the UNK node, effecively skipped, and the rest of the sentence continues from the root node. This is very useful in language modeling, as will be explained.

Again after any word is added, it can stop at any point and the on-line nature of building the corpus from left to right allows it to have a full representation in the tree up to the point where it stopped. If another word was added it would continue as the active point and other pointers which are used to build the tree are ready to continue. Further, if another word is added, it is added via the suffix links to all parts of the tree which are necessary. This on-line building nature of the suffix tree gives it another useful property of being dynamic, where training can occur, the tree can be used for whatever purpose, and then further training can continue.

What we have now is a tree structure which represents all of the suffixes of a corpus. It can be built in linear time and requires much less storage than the original trie structure. It is on-line and dynamic, and the suffix links which were essential in building the tree can possibly be useful in other ways. It is a full representation of all substrings of the text without loss of data. With this data structure at hand, I will now introduce several applications of suffix trees in natural language processing.

3.4 Applications of Suffix Trees

Suffix trees can be used in fast string searching [Nelson, 1996]. If one needs to find all substrings of another string (or any sequence, DNA, for example), it would take N times M comparisons where N and M represent the lengths of the two strings. Using a suffix tree brings that number down to M comparisons because all substrings of N are represented in the suffix tree, and search is quickly done by starting at the root node. Seeking a substring is still proportional to size N because each substring of N needs to be compared, but it is still made much faster with suffix trees. Being able to match strings is also useful in data compression.

One of the first uses of suffix trees was plagiarism detection [Monostori et al., 2000]. With the increasing availability of professional documents and books on the Internet, it is easy to copy text and claim it as one's own. Suffix trees provide a fast way of finding long strings which occur more than once, thus exposing which parts of a text which may have been plagiarized. This is done simply by concatenating the text in question with the text that may have been plagiarized. After creating a suffix tree, the nodes which represent long strings show where the overlap is.

Parameterized suffix trees have been used to find code duplication in software systems [Baker, 1997]. This could be useful in code plagiarism detection, but in this case it was used to track duplication in large software systems, which is useful in software maintenance.

Suffix trees have been well used in information retrieval, more recently with Chinese documents [Huang and Powers, 2008]. Suffix trees are useful for languages such as Chinese which don't have word boundaries because a suffix tree can be made to set each individual character as a word, and each character sequence is fully represented, unlike other approaches which require error-prone word segmentation.

4 Language Modeling

We now move from suffix trees to discuss n-gram [Markov, 1913] language modeling, the applications of language modeling, and some existing implementations of language models. After which, I will discuss how a suffix tree can be made to house the same information necessary for effective language modeling.

4.1 N-gram Language Modeling

Language modeling in NLP is *statistical models* of word sequences [Jurafsky and Martin, 2008b]. The goal of a language model is to determine how well a sequence of words belongs to a certain language. A large class of language models are based on the idea that the probability of a word in a certain place in a text can be estimated based on context or based on the other words in the immediate vicinity. An *n-gram language model* further constrains this vicinity to only the previous words. It is assumed that the reader is familiar with the concept of n-grams in general, though we will see more detail on what n-grams mean in language modeling.

More formally, the probability of some word sequence (a sentence, for example, is a word sequence) is given statistically by using the *chain rule* where the probability of the first word by itself (unigram) multiplied by the probability of the second word following that first word (bigram), multiplied by the third word given the first two words (trigram), and so on until the end of the sentence. It is formalized in 1.

$$P(w_1^n) = P(w_1) * P(w_2|w_1) * P(w_3|w_1w_2) * .... * P(w_n|w_1...w_{n-1}) \tag{1}$$

How these probabilities are estimated is a matter of great interest in the area of language modeling. The most straightforward way is take a word history and count the different words which follow that word history. As language models are predictive models, one wants to model future possible word sequences given what was seen in training. This suggests a simple relative frequency as a probability estimate of a sequence of words, which is better known as the *maximum likelihood estimator* (see [Manning and Schütze, 1999]):

$$P_{MLE}(w_1...w_n) = \frac{C(w_1...w_n)}{C(w_1...w_{n-1})} \tag{2}$$

Though these histories can be set to be any length by most language modeling toolkits, the trouble is that the history must be pre-defined at training. This pre-defined n-gram history length is called the highest *order* of the language model and contains all smaller model orders. For example, a language model trained on the order of 3 will have all of the trigram, bigram, and unigram models.

Another trick to effective language modeling is sentence *padding*, also known as *start* and *stop* tokens. These are filler words which signify the beginning and ending

of a sentence. These tokens are used in training and evaluation as if they were actually part of the sentence. For example, the sentence *the boy kicked the ball* would have the added padding:

$< s >$ the boy kicked the ball $<\backslash s >$

These tokens give a more defined sentence boundary, which is useful because some words are more likely to begin a sentence while other words might be more likely to be at the end. Different language models use these in different ways, sometimes not at all.

4.2 Applications of Language Models

Language modeling has been used in automatic speech recognition (ASR) for some time. ASR takes audio input and, given an acoustic model (and possibly a grammar) determines a set of hypotheses as to what was uttered. Whether or not it was the actual utterance is not necessarily up to the language model, but the language model's purpose is to detect how well the utterance belongs to the language. In a similar way, *Automatic Speech Transcription* which is applied speech recognition without the use of a grammar, makes heavy use of language models (see also [Saykham et al., 2010]).

Statistical machine translation (SMT) has a similar problem. When an input sentence in some source language is given and run through the SMT system, it could give many hypotheses as to what the target language translation ought to be. It is assumed that the hypothesis which is assigned the highest probability by the language model will be the most fluent. That most fluent result will have the highest probability given by the language model. Machine translation takes this information and weights it along with other features to determine the most likely sentence given the input. To paraphrase Philip Koehn, he called a language model a function that takes a sentence and returns the probability that it was produced by a speaker of that language [Koehn, 2010]. Further, as translations often run into a problem of word order, language models offer a solution to prefer a correct word ordering to an incorrect word order.

Language models have also been used in information retrieval [Lv and Zhai, 2009]. Information retrieval is concerned with searching for documents or information within documents which is relevant to a query. For example, a search engine performs information retrieval when it seeks out which websites contain the information for which one is searching. A language model can be used in information retrieval such that each document is trained as its own language model, and the user query is then evaluated by that language model. Thus, documents are ranked based on the probability that that document's corresponding language model would generate the sequence of words in the user query.

A list of several language modeling toolkits can be found in Appendix A. For the purposes of this thesis, I will compare the suffix tree language model with SRI language model (SRILM) [Stolcke, 2002] written by Andreas Stolke, and the Suffix Array Language Model (SALM) [Zhang and Vogel, 2006] which was developed by Ying Zhang and Stephan Vogel.

5 Suffix Tree as a Language Model

With a knowledge of suffix trees and language modeling, we will now discuss how a suffix tree can be used as a language model.

5.1 Suffix Counts

Each node in a suffix tree is assigned a count which is called the *suffix count* which will be used in the probability estimation. This suffix count represents the number of times this word was crossed, or seen, in this node of the tree. Significantly, the sum of the counts of all the child nodes of that node is equal to the suffix count for that node. Figure 4 in Section 3.3 shows the 'cacao' suffix tree with suffix counts.

These counts can be found in a depth-first manner by beginning at each leaf node and incrementing each suffix count as it moves recursively to each parent node to the root node. This will leave the leaf nodes with a count of one, and every other node will have been crossed enough times to make the suffix counts become the sum of the suffix counts of the child nodes. This method of gathering the suffix counts is for intuition of how the nodes are related, but isn't actually implemented. During building, leaf nodes are given a count of 1 and counts are incremented when a node is split. Once the tree is built, a check is initiated from the root and recursively computed into the greater depth of the tree. The computation consists of checking to see if the count of the parent node is the sum of the child nodes' counts. If it is the same, then all children from that node all the way to the leaf nodes which branch from that node are assumed to be up-to-date. If not, then the parent node count is then updated to be the sum of the child nodes.

Just with these suffix counts, the suffix tree is already starting to look more like a language model. To compare what we currently have with traditional language models, notice that in the simple example corpus there are 5 unigrams: c, a, c, a, and o, where three words are unique: c, a, and o. The nodes directly branching off the root node are these three unique unigrams, and the counts correspond to the unigram counts (c is found twice, a twice, and o once). Now, if the root node is the total number of unigrams (that is, total number of tokens in the corpus), and a node directly off the root node is a count of that specific unigram, then the probability of the node off the root node is the count of that node divided by the count of the root node. This corresponds to the unigram probabilities in a normal n-gram model. Further, take any of those nodes directly off the root node and look at that node's child nodes. The counts on those nodes divided by their corresponding mother node is the bigram probability for those words where the word history begins with the node directly off the root node. This pattern is the same for any depth of the tree, the probability of a word given a history is the child node divided by the mother node, and the n-gram size is the depth of the tree where the computation is occurring. The node probabilities is illustrated in Figure 5.

The method described above is used for computing the probability of the first word in a node. However, some nodes represent sequences which are longer than just one word. For any word that is not the last word in a node, the probability is 1.0 because it is the only child node of its mother node. It is an interesting property of the suffix tree that one can easily find normal nodes which have direct children, and those nodes which have several words before branching to the children. For example,

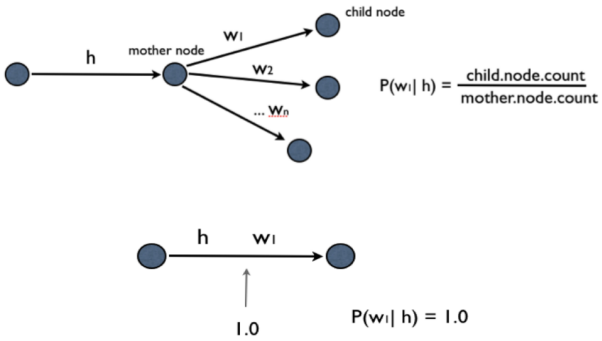

Figure 5: Computation of probabilities using the suffix tree nodes.

in Figure 2, *ca* is an arc which contains more than one word. As c has a as an only child, the maximum likelihood estimation of a following c would be 1.0 because that is what was seen in training.

Applied to this tree, the chain rule is straight forward. Finding a word sequence $w_1w_2...w_n$ is done by beginning with the probability of w_1 which is found by dividing the suffix count of node directly branched from the root node which represents w_1 by the suffix count of the root node, then continuing to follow that node into its children and finding w_2 in the sequence and multiplying that probability by the first probability. This process continues until the end of the sequence. Again turning to Figure 4, if one wished to find the probability of the sequence *ao*, start at the root node and find the node branching from the root node which represents *a*, node 5. This has a count of 2, the root has a count of 5, hence the probability will be 2 divided by 5. From node 5, it continues to find *o* as a child, which is node 6. Here the probability is 1 divided by 2 which would then be multiplied by the first calculation to give 0.2 for the sequence *ao*.

The reader will notice that it is very optimistic, and certainly naive, to think that every possible word sequence will be represented in the tree. Even if the previous example was to find the probability of the sequence *aa*, there would be a problem as there is no child *a* to node 5. Even if the training corpus which was used to build the tree was very large and contained billions upon billions of words, not every possible word sequence that has been experienced in the history of that language or every possible word sequence that will come later could possibly be represented, but it is a reasonable approximation.

We will now briefly discuss an important assumption that is made in language modeling known as the *Markov assumption* (see [Jurafsky and Martin, 2008b]). Though many things affect why a word is used in a specific word sequence, the Markov assumption states that only a limited number of previous words affect the probability of the next word. Formally, the Markov assumption can be stated in 3.

$$P(w_n|w_1 \dots w_{n-1}) \approx P(w_n|w_{n-M} \dots w_{n-1}) \tag{3}$$

It is certainly overly optimistic to assume that every word sequence will be represented, or even that every individual word will be represented. It is a fundamental part of language modeling to assume that future word sequences in fact won't exist in the training data and that must be accounted for, hence the Markov assumption is important. Furthermore, one must assume that other word sequences which are not in the training corpus are in fact probable sequences because language isn't static. We can at least start to address this problem by assuming that if a word sequence is not represented, then perhaps removing the first word from the history will allow us to continue computing the chain rule of that sequence. For example, if the probability of w_3 is conditioned on $w_1 w_2$, by further applying the Markov assumption and removing the first word from the history, w_3 is now only conditioned on w_2. We will now discuss suffix links as a means in accomplishing this.

5.2 Back Off using Suffix Links

When the suffix tree was being constructed, the Ukkonen Algorithm made use of suffix links, which linked a word sequence to the same word sequence branch where the first word is removed. Hence, a node links to the next suffix of itself via the suffix link. For example, if we have the sequence *is this is a nice sentence*, the last word, *sentence*, would link to the word *sentence* in the sequence *is a nice sentence* which exists elsewhere in the tree and this linking continues until the word *sentence* is the only word in the sequence and itself links back to the root node. This implicit part of the data structure becomes extremely useful during *back off*, which is crucial to an effective language model.

As another example, assume that there is a fully-trained tree with a large training corpus. Now we wish to evaluate some MT output and we must use our language model to assess the value of the sentences as members of the output language. To do this, the model takes a sentence which is to be evaluated and begins with the first word of that sentence. If that first word exists as a branch from the root node, then it gathers the probability of that word, again, by dividing the suffix count of that node by the suffix count of the root node. It then continues from that node to see if the second word in the sentence is a branch from that first node. If it is, it continues by finding the probability in the same manner as the first word and multiplying the two together as defined by the chain rule. But what if after several words into the sentence, the following word didn't exist in the training corpus? As mentioned before, it can follow the suffix link to the same sequence with a shorter word history, but what does that gain us? Would not the words which follow that node be the same words which follow the original node which doesn't have a branch to the next word in our sentence? The answer is yes, it will have all the same branches, but it will also have more branches. If the trigram *welcome to Switzerland* where *Switzerland* depends on *welcome to*, but that particular sequence isn't in the training corpus, it can back off to just *to*. Intuitively, one can imagine certain words which are likely to follow *welcome to* (just insert any place name), and one can certainly imagine many more examples of words which only follow *to*, and that list it will contain the same words as *welcome to* along with many others because *to* is a very common word. This is precisely why backing off is useful; it relaxes the restriction on which words can follow the current word. Normally, the model attempts to be as strict as possible

most of the time for the better estimators, but if those restrictions need to be relaxed to find the next best estimation, then it follows the suffix link to do so.

I now include another example which further shows how back off affects the chain rule. I don't wish to belabor this point, but back off is a very important part of language modeling. Take a short training corpus where the only sentence is: *the girls and the boys like the toys*. If we train a traditional trigram language model, we can break up the sentence into 6 trigrams, 7 bigrams, and 8 unigrams. If we now want to calculate the probability of the sentence *the boys like toys* we can do that with this language model with what we know so far. If we follow the chain rule, we can calculate the unigram probability of *the*, the probability of *girls* given *the*, and so on as shown in 4.

$$P(the) * P(boys|the) * P(like|the\ boys) * P(toys|boys\ like) \tag{4}$$

Remember that the corpus was trained as a trigram model, so it only contains unigrams, bigrams, and trigrams, therefore we are already making use of the Markov Assumption and approximating our word probabilities with two-word histories at most. Also notice that the last probability in the sentence does not exist in our training corpus. However, as was mentioned before, it can back-off to a shorter word history and see if that word sequence exists. So, it can remove *boys* giving us the bigram *like toys*, but that two-word sequence does not exist either. Nevertheless, we can back-off again in the same way by removing the first word in the sequence, leaving us with just *toys*, which is in the training corpus. The final sentence probability looks like 5.

$$P(the) * P(boys|the) * P(like|the\ boys) * P(toys) \tag{5}$$

5.3 The Sentinel Character

If the suffix tree requires a final character (word) in order to guarantee a complete tree, does this not affect the language model counts and on-line nature of the model? Indeed, adding a final character would create branches, and therefore increment counters of nodes, thus resulting in counts that aren't really accurate estimators. Yet without the sentinel character, the tree isn't considered truly complete. However, because of the way the tree is used as a language model, this sentinel character is not necessary for the tree to be usable. The reason the sentinel character guarantees a complete tree was illustrated in a previous example where adding a word resulted in a necessary branch of a node which represented multiple words. It had to create a node out of the first word and branch to the rest of the words in the sequence, making a child node out of them, and also branch to the sentinel character. This is not only inaccurate in the language model because now the new node would have

a count of 2, it is also unnecessary. Without the sentinel character, any character which is added to the tree, but is finally found to be in the middle of a node needs not branch because it is already represented and would receive the correct count without the sentinel character. This allows the language model to be complete and because it no longer needs such a final character, it is also able to be dynamic and on-line. Of course, the suffix counts would need to be updated if new words are added.

6 Smoothing

6.1 Introduction

Any linguist will argue that there is much more involvement in language processing than simply using the previous words to predict the next word. That linguist would certainly be correct. However, this kind of language modeling is easy to process, requires minimal pre-processing or annotation, and works well when determining if a sequence of words belongs to a language. Even so, back off doesn't solve all of the problems. What if, for example, a word which exists in a sentence which we are trying to evaluate didn't exist at all in the training corpus? This happens a lot with proper names, also known as *named entities*. Even if it backs off until the root node, the word is not represented anywhere in the tree. In this case the conditional probability of that word would be zero, thus making the entire sentence probability zero by nature of the chain rule multiplication. There may be a case for making the sentence zero, but it is not uncommon to have a legitimate word show in evaluation which was not in training, something which must be taken into account. Enter now a very important aspect of language modeling called *smoothing*.

Smoothing is a way of assigning a small amount of probability mass to be used in times where a word doesn't exist in a context, known as the *unseen event*, also known as *UNK*, unseen because it was not in the training corpus. I must insert here that back off as I have explained it isn't separate from smoothing, but is an integral part of smoothing as it addresses part of the problem on its own. I made mention of back off first because it was an implicit part of the suffix tree data structure, and it lead naturally into general smoothing.

This is not, however, a complete handbook or history on smoothing. There are various ways to approach smoothing, each method with advantages and disadvantages. All have the same thing in common: to avoid the zero probability by assigning a small amount of probability mass to the unseen event while holding true to being a fully *normalized* model (everything in the probability space sums to 1). Having a normalized model is also important for certain types of evaluation which will be discussed later. The reader is referred to [Koehn, 2010] and [Jurafsky and Martin, 2008a] for detailed explanations on various smoothing techniques. I will only discuss two techniques: simple back-off and Kneser-Ney, the first because of it's simplicity and ease of implementing in the suffix tree, and the second because of its effectiveness in most areas where language models are used in natural language processing.

A very simple form of smoothing which can easily be implemented in the suffix tree makes heavy use of back off. In fact, if a word doesn't exist in a context, it always backs off and looks at each shorter word history for the word in question, as discussed previously. If it must back off all the way to the root node and the word still isn't found in any branch, it then assigns that word the probability which has been set aside as *UNK*. This UNK probability is made by removing a very small amount of probability mass from each suffix count such that the amount of resulting mass becomes Equation 6. This is what was represented in the UNK arc in Figure 4.

$$p(UNK) = \frac{1}{SIZE - OF - VOCABULARY} \tag{6}$$

This is easily done as the SIZE-OF-VOCABULARY is the number of branches from the root node, or the number of unigrams. One can simply loop through each of these nodes and subtract $1/SIZE - OF - VOCABULARY$ from its suffix count, also known as *discounting*. This is a small enough amount to barely affect the suffix counts, but enough to create a small amount of mass for the unseen events and thus allowing the chain rule to not evaluate to zero. In fact, the amount removed from each node off the root node is typically so small that it is arguably permitted to not bother removing it in the first place and still resulting in the desired normalization. This is certainly a very greedy approach to finding the sentence probability; always follow the path as deep as possible and only back off when necessary.

This fully solves the problem of a zero-probability. It seems intuitive to assign a small amount like this only if the model backed off entirely to the root node. It further seems intuitive that a word which doesn't exist in training should be assigned a very small probability mass. However, this simple approach isn't the most ideal way of smoothing because it doesn't produce as good results as other known methods. We now discuss a smoothing technique which is very widely used today.

6.2 Kneser-Ney

Kneser-Ney smoothing[Kneser and Ney, 1995] has been around for some time and at the time of this paper, is currently the best smoothing used in MT and ASR evaluation. I will spend some time describing Kneser-Ney smoothing because it plays an important part in making the suffix tree language model perform well. My explanation of Kneser-Ney smoothing follows Philip Koehn [Koehn, 2010], which also contains a detailed explanation of other smoothing approaches.

There is more to a language model than the simple fact that w_2 follows w_1 by some probability which depends on all words which follow w_1. We can look at language phenomenon more closely and incorporate more things which will impact our decision in how much probability we assign to a sequence of words.

Koehn gives an excellent example in his book. The word *york* is as frequent in the Europarl [Koehn, 2005] corpus as *foods*, *indicates*, and *providers*. That means in a unigram model these four words would have the same probability. However, the word *york* almost always occurs after *new*, so *york* is rare unless seen together with *new* and it should be treated that way. Thus, in the unigram model less probability would be assigned to the word *york*.

Kneser-Ney smoothing therefore takes a second pattern into account. It doesn't just look at the raw probability counts, it also looks at what is called the *diversity of histories*, or *count of counts*. For example, a bigram model which has a one-word history, will also store information on how many different words came before w_2. We expect much fewer histories for *york* than for the equally represented *food*. This diversity of histories is represented by $N_{1+}(\bullet w)$, and the normalizing factor is the sum of all the histories. This normalizing factor is simply a count of all the instances of all histories of a word for a specific history length. Formally, in 7.

$$pKN(w) = \frac{N_{1+}(\bullet w)}{\sum_{w_i} N_{1+}(w_i w)} \tag{7}$$

For lower order models like a unigram or bigram where the diversity of histories makes a difference, we can use this as our probability. In higher-order models we can use our normal, raw counts as the probabilities.

Add to this something called *interpolation* which is a way of combining lower order and higher order n-gram language models. This interpolation is done by a simple linear combination of all the models. For example, if one has a trigram language model, the combination of the unigram, bigram, and trigram can be presented in 8.

$$pI(w_3|w_1, w_2) = \lambda_1 p_1(w_3) * \lambda_2(w_3|w_2) * \lambda_3(w_3|w_1, w_2) \tag{8}$$

where each lambda is greater than or equal to zero, and less than or equal to 1, and the three lambdas all sum to 1. These values can be found empirically. The lambda value for the trigram model, for example, could have a higher weight than the other two combined to give it the most say in what the probability ought to be.

Modified Kneser-Ney smoothing [Chen and Goodman, 1998] has yet another improvement. In this already powerful smoothing technique, *absolute discounting* is added to the already established interpolation, back off, diversity of histories, and raw counts. I introduced a similar concept in the previous section which described my method of simple smoothing. In that smoothing technique, I remove a small amount of mass from each of the node counts off the root node (the unigram counts) which summed to the amount of probability mass which I assigned to the unseen event. The principle here is similar, but finding that amount of mass to discount and how to apply it is much more sophisticated. Chen and Goodman use three different discount values. The values are found by counting up the total number of n-grams (unigrams, bigrams, etc) which have count x. For example, if the bigram *the dog* only showed up once in a training corpus, then it has only one count and would be part of N_1. After counting up all n-grams with counts 1, 2, 3, and 4, we have what we need to find the discounts. These can be computed easily as seen in the following equations:

$$Y = \frac{N_1}{N_1 + 2N_2} \tag{9}$$

$$D_1 = 1 - 2Y\frac{N_2}{N_1} \tag{10}$$

$$D_2 = 2 - 3Y\frac{N_3}{N_2} \tag{11}$$

$$D_3 = 3 - 4Y\frac{N_4}{N_3} \tag{12}$$

Combining all of these tools together; absolute discounting, interpolation, back off, diversity of histories, and raw counts, we have a powerful language model with sophisticated smoothing. The final equation which combines all of these factors is akin to Equation 13. As a sentence is evaluated, and a certain word in that evaluation is reached based on the history up to that word, there are two possibilities. The first possibility is when the raw counts are greater than zero, meaning the n-gram was seen in the training data (or, if it is a lower-order model it uses the diversity of histories instead of the raw counts). The other case is when there is no raw count for that word given that history (it was not seen in the training data in that context). In that case, the left part represents the discounting equation and the right side is the probability in the back off model. Hence the discounting equation also acts as the interpolation. If one had to back off multiple times, each back off instance would result in evaluating the bottom equation, thus always incorporating some amount of mass that would be given at that order.

$$p(w) = \begin{cases} \alpha_n(w_i|w_{i-n+1},...,w_{i-1}) & if\ count_n(w_{i-n+1},...w_i) > 0 \\ d_n(w_{i-n+1},...,w_{i-1})p_{n-1}(w_i|w_{i-n+2},...,w_{i-1}) & otherwise \end{cases} \tag{13}$$

In the suffix tree language model, much of what is needed exists ready for use. We can define the depth of the tree as the order of the current model we are working with, that is, the depth of the tree corresponds to the history or model size. It also has raw counts and back off implicit in the data structure. Interpolation can also be found by recursively moving towards the root node by following the suffix links to the lower-order models, combining them along the way. The diversity of histories is also found in the tree, but it is more difficult to compute. To illustrate in the cacao example, a shows up twice, and its one-word history is c both times, thus the diversity of histories would be one because it only has one history. However, there really isn't anything in the tree that allows this to easily be computed. Finding the diversity of histories would therefore require stepping through each branch and keeping track of how many times branches with the same histories occur. A simpler way to do this will now be explained.

As the entire corpus is also indexed in an array, it is not difficult to step through that array and keep track of the histories of all the words. As the array is stepped through and a word is encountered, the algorithm looks to the history of that word. If the history has been seen, then it only increments a counter which keeps track of the total number of histories have been seen for that particular word. If it is a new history, then that history is added to a data structure which holds all of the histories for that word. This can potentially take an enormous amount of space and stepping through the corpus once is done in linear time. The following pseudo code shows a way of accomplishing this which will work for any length of history, though Kneser-Ney smoothing is defined to find the diversities of histories up to a trigram:

```
kneser-ney-depth = 3
let i = 0 where i is an index to the training corpus
while i is less than the size of the training corpus
        let j = i
        let hist = i
        let hist-length = j-i+1
        while j is less than i + kneser-ney-depth
                if history of j does not contain hist
                  add hist to history of j[hist-length]
                increment counter for j[hist-length]
                hist = hist + j
                hist-length = hist-length + 1
```

Once complete, the discounting values are easily computed based on the total diversities of histories, a computation which is completed once.

The suffix tree language model implementation for this thesis is similar to the modified Kneser-Ney smoothing as explained, but it also differs in some respects. For example, it worked better (at least in English) to always interpolate the diversity of histories with the raw counts no matter what depth an evaluation sentence was in the tree. To find the Chen and Goodman discount values, it made no difference to use a depth of 2 or 6. That is, the discounting values didn't make any difference in evaluation whether it was using n-grams up to a 6-gram, or just unigrams and bigrams. The weight given to the diversity of history is 0.05, and 0.95 to the raw counts. Interpolation as defined in Kneser-Ney smoothing always made the model perform worse, so interpolation is not used. It was also difficult to determine what weights to give to each depth as it cannot be known before hand what depth the sentence would reach and always be normalized. I attempted a pareto distribution which could be made to change depending on the current depth, but that too did not perform as well as just having no interpolation at all.

Another thing in the suffix tree language model which needs to be addressed are the nodes which return a probability of 1.0. Recall that during training, nodes contain arc and node information and some nodes represent a long sequence of words. In these cases, the training showed that multiple words represented by a single node are only seen once in training. If one of these sequences is encountered during evaluation, the estimation would find that the parent and the child would each have a count of 1, thus resulting in a 1.0 probability. Intuitively, Kneser-Ney smoothing which requires a small amount of discounted mass to be removed from all counts should also require a small amount to be removed from these instances where 1.0 is returned. However, the simple fact of the matter is removing any mass from from these nodes results in worse results no matter what evaluation metric is being used. To get an idea of how much this affects the model, I trained the suffix tree language model using 100k lines of a the English Europarl corpus [Koehn, 2005] and ran some statistics using a corpus from another domain, JRC-Acquis [Steinberger et al., 2006].

Table 1 has a lot of useful information about the suffix tree and how it handles evaluation. As expected, the number of leaf nodes approaches the number of words and makes up a large percentage of the nodes. Internal nodes show how much overall branching occurs, and in this case it's about once out of every 5 nodes. In evaluation,

Table 1: Suffix Tree Corpus Statistics

Training (100k lines of Europarl)	
Number of words	3,005,366
Size of vocabulary	34,363
Total number of nodes	3,533,984
Number of internal nodes	727,774
Number of leaf nodes	2,806,210
Percentage of leaf nodes	79.4
Percentage of internal nodes	20.6
Evaluation (JRC-Acquis)	
Number of sentences	4,108
Number of words	128,879
Average sentence length	31.37
Maximum depth reached	15
Average depth reached	3.38
Number of back off instances	100,688
Percentage of normal probabilities	91.4
Percentage of UNK probabilities	3.4
Percentage of 1.0 probabilities	5.2

there are over 128k words which means that there are over 128k individual probability calculations, which are combined to become the more than 4k sentences. Of those more than 128k probability calculations, 91.4% are found normally, that is, they were in training (though back off could have occurred somewhere along the way before the probability was found). Over 3% belongs to those words which are not in training at all, but show up in evaluation. The interesting statistic here are the 1.0 probabilities that are returned, over 5%. That's more than the number of unknown words, and how the unknown words are dealt with makes a huge difference to the performance of the model. The number of 1.0 probabilities, therefore, isn't just some negligible amount.

It is further interesting to note that the maximum depth reached during evaluation is 15. This is the final depth of a sentence, which means at some point back off may have occurred and the depth may have actually been bigger, but 15 was the depth when the sentence was done with evaluation. This gives the suffix tree language model more credence in that there are times when long sentences, or long quotes, which are recognized as in the language. On the other hand, the average depth upon completion of sentence evaluation is under 4. This again could mean that a sentence went deep into the tree, but by the the time the final word in the sentence was evaluated, it was at a shallower depth. Back off also proves to be a very important part of language modeling, as the number of times individual back off occurred approaches the number of words which were evaluated. Hence, on average, each word needs to utilize back off about one time.

I wish to add here that in order to compare and contrast with other models, it was necessary to add the ability to limit the depth of the suffix tree language model. This is done simply by checking the depth of the tree at any time and if it exceeds some depth limit, then the suffix link is followed.

Table 2:

count	string
2	Substantially Less Interference by Members
3	overstepping the
4	callous disregard
4	five-legged sheep
4	nautical miles
5	non-economically active
6	oriental carpets
6	revitalization of the
7	sportsman or woman
24	Camp David

Table 3:

span	count	span	count
2	1237	15	9
3	359	16	3
4	142	17	3
5	69	18	3
6	44	19	1
7	27	20	3
8	16	21	5
9	14	22	1
10	8	24	1
11	9	25	2
12	9	26	3
13	6	27	4
14	3	28	1

Table 2 contains several examples of strings that occur more than once, but only in the represented sequence. In these cases, if any word beyond the first word of the sequence were queried for a probability, the result would be 1.0. These were similarly taken from the first 100k lines of the Europarl English corpus. After the suffix tree was built, the nodes directly off the root node were queried and if they represented a string of more than one words, they were displayed along with how many times they occurred in the corpus. Many of these multiple-word strings are named entities, as would be expected.

To get a feel of just how many of these multiple-word nodes exist, see Table 3. The *span* column shows the number of words represented by the node, and the *count* column represents the count of how many times nodes with that span occurred. Only nodes which had a span greater than 1 were included. Again, these are only nodes directly connected to by the root node. After 28, the counts hold steady at 1 for those spans which are represented at all. As expected, this follows a Zipfian distribution. The average span of these nodes is 4.24.

With this data in hand, it was my intuition that I wasn't making the best use of the suffix tree as a language model because the data structure has more useful

properties than the traditional n-gram model uses. A sentence in the suffix tree language model is processed by starting with the first word and finding its probability from the root node as before explained. The second word depends on the node of the first word, the third continues as a branch from the second word, and so on unless back off is necessary. A sentence potentially can be represented in the tree without back off in its entirety. For example, if a sentence has 12 words and it is represented in the tree without needing back off to evaluate it, it is as if there existed a 12-gram model. It doesn't really matter what length each sentence is as it always begins from the root node and can potentially go to the depth which is the length of the sentence. Therefore, it always seemed intuitive to me to give a boost to a probability that had a higher depth because a higher depth meant the model better represented, and therefore could estimate, the sentence being evaluated. I thus multiply each probability by the depth which the model is currently at for evaluation. This was the last thing that caused the model to push past the SRILM model in MT evaluation as shown in section 7. In fact, multiplying by twice the depth showed better performance than interpolation with lower-depth models. However, this takes away the normalization of the model.

As a simple example of this depth boost, take a sentence with 8 words, $w_1 w_2 ... w_8$. Assume that w_7 does not follow w_6 directly, but needs to back off once in order to continue. The probability of the sentence would be found by starting with w_1 at depth 1, w_2 at depth 2, etc, as shown in 14.

$$p(w) = 1p(w_1) * 2p(w_2) * 3p(w_3) * 4p(w_4) * 5p(w_5) * 6p(w_6) * 6p(w_7) * 7p(w_8) \quad (14)$$

It may seem like too much to give each individual probability such a large boost, but it works quite well. Those sentences which are long and well represented get an almost exponential boost, whereas sentences which require back off are otherwise penalized for backing off. Even "good" shorter sentences get a small boost, which is important because language models can often be used to distinguish short phrases.

A simple example to see how much the boost helps, below are 5 sentences with the punctuation removed and numbers added for clarity. The first sentence is the original sentence and the rest are scrambled versions of the original sentence. They are scrambled enough to show a distinction between "correct" and "incorrect" sentences:

1. as my prepared statement is in tibetan i want to read in tibetan to show respect for my own unique separate language
2. i want to read in tibetan to show respect for my own unique separate language as my prepared statement is in tibetan
3. is in tibetan i want to read in tibetan to show respect for my own unique separate language as my prepared statement
4. statement is in tibetan i want to read in tibetan to show respect for my own unique separate language as my prepared
5. show respect for my own unique separate language as my prepared statement is in tibetan i want to read in tibetan to

A language model will assign a probability to these sentences. The first sentence should receive the highest probability. Following are the assigned *log* probabilities:
1. -53.5931
2. -54.2598
3. -55.7571
4. -55.536
5. -58.4584

The following are the corresponding log probabilities with the added boost:
1. -44.1522
2. -44.886
3. -46.5812
4. -46.3991
5. -49.4989

As expected, the boost sentences have larger (non-normalized) probabilities because of the extra multiplication of whole numbers. The question now is, why does this matter if in both cases it assigned the highest probability to the first sentence? The answer is in the average differences. If we subtract the first score by each of the other scores and take the average of those four differences, we can see what the spread is for the probabilities. Without boost, this spread is 2.4097. With the boost, the spread grows to 2.6741. This may not seem like much, but this small difference may be the difference between discriminating two very close sentences in MT. This allows the language model to be able to better distinguish good sentences from bad ones, its very function and purpose.

Now, a shorter sentence presented in the same manner as the previous long sentence:

1. i thank you for this kind invitation
2. this kind invitation i thank you for
3. kind invitation i thank you for this
4. for this kind invitation i thank you
5. you for this kind invitation i thank

Without the added boost, the scores are as follows:
-11.5799
-14.7026
-15.3584
-15.242
-16.8328

Which has an average spread of 3.95. With the boost:
-7.29299
-10.6892
-11.4022
-11.3251

-13.3401

the average spread is 4.39, again making the discrimination more distinct.

Further, in a small scale MT experiment where 10k sentences were used for training the decoder and the language model, and 1k for evaluation, the BLEU score (we will discuss MT evaluation and BLEU in detail in Section 7.2) is 12.37, whereas with the boost the same set receives 13.32, which is a significant gain for such a small amount of training and evaluation data.

7 Experiments

7.1 Perplexity

The standard way of evaluating a language model is *perplexity*. Perplexity is a fast way of showing how well a language model functions. Though it may not correlate completely with MT or ASR evaluations, it is still a high indicator of improvements. As the fundamental purpose of a language model is to give a high probability to good word sequences and a low probability to bad sequences, it seems intuitive to have an easy way to combine a large text into a single number which represents just how well the language model performs given the text. This is precisely what perplexity is. Various things affect perplexity which would affect evaluation in any area, such as n-gram order, amount of training data, and smoothing techniques. A more detailed explanation of perplexity is given in Appendix B.

I did not use the boost as described earlier in the perplexity scoring. My version of Kneser-Ney smoothing is implemented as close to the actual Kneser-Ney smoothing as I could understand it. This reflects a near-normalized model (the 1.0 probabilities which are sometimes returned did not have any discounting).

I modeled the perplexity measure of the suffix tree language model to follow the SRILM equation in 15. Because the suffix tree language model always uses UNK, the OOVs variable will always be evaluated to zero. I therefore trained SRILM models to include UNK so it too would evaluate the OOVs to be zero. I also include sentence padding in the calculation, so Equation 15 is the best comparison, though other settings for SRILM might result in better perplexities. The purpose is to establish my model as comparable to SRILM. The fact that STLM performs better overall with these comparable settings doesn't necessarily mean that it is a better model. This is due in part to the 1.0 probabilities, which result in higher overall sentence probabilities which in turn result in lower perplexity scores. It is interesting to note that the STLM language model only levels off; it never worsens as is the case with the other two models, even up to the unlimited order model, as denoted by *NA* This is because the entire model is created and then the order is limited, so the estimates aren't affected by a change in order. This is not necessarily the case for the other models. Certainly for SRILM, a change of order affects interpolation which affects most probability estimates.

$$PP = 10^{\left(\frac{-logprob}{words-OOVs+sentences}\right)} \tag{15}$$

In order to find the perplexity for SALM, I had to change the logarithm from base 2 to base 10, then combine the sentence log probabilities as shown in 15. I only show enough results for SALM to establish that the perplexities are sufficiently high (because of lack of Kneser-Ney smoothing) and that it isn't really necessary to use it in other evaluations. I realize that it is not typical do compare perplexities like this, but does who how the models compare using a simple metric.

Table 4: Perplexities

Order	Perplexity
SRILM	
1	924.78
2	197.725
3	163.386
4	156.165
5	154.226
6	153.439
7	152.331
8	152.492
STLM	
1	505.602
2	174.071
3	159.926
4	153.833
5	151.595
6	151.437
7	151.191
8	151.19
9	151.19
NA	151.19
SALM	
1	2979.6309
2	867.4389
3	831.4024
4	993.9009

7.2 Statistical Machine Translation

As mentioned earlier, language models are used in SMT to determine the acceptability of a candidate translation. Typically, MT systems provide hundreds of hypotheses which could be translations of a single sentence, and each of those hypotheses are in turn checked against the language model to see just how probable they are as members of the target language. In the end, a single hypothesis has the highest probability according to all the factors in the MT system and is offered as the translation for the sentence.

In order to learn the patterns of translation, like language models, a SMT system needs corpora for training, tuning, and evaluation. However, the corpora necessary for a SMT system is somewhat different. SMT requires a *parallel corpus*, that is, a corpus of text in two languages. Typically a corpus has one sentence per line and each sentence corresponds to a translation in a different language in the other language's corpus. These sentence-aligned corpora are fed into a SMT system and, given some machine learning techniques, is able to learn to some degree how the source language translates into the target language. SMT systems typically are one-directional, which means that in order to translate from a source language to a target language, it needs to be fully trained in each language pair; it cannot translate backwards from the target language to the source language unless it is trained to do so. There only needs to be a language model in the target language as that is what the desired output is. Often the language model is trained on the same target-language side of the parallel corpus as is used in the SMT system. Evaluation of the machine translation output was done by the *Bilingual Evaluation Understudy* (BLEU) [Papineni and et al, 2002], the *National Institute of Standard and Technology*[1] (NIST) metric, METEOR [Lavie and Denkowski, 2010], *translation edit rate* (TER) [Snover et al., 2006], as well as precision and recall (which are used by METEOR). A useful script that produces all of these outputs given the machine translation hypotheses along with the reference target translation was provided by Kenneth Heafield[2].

To manage the machine translation experiments, I coupled the suffix tree language model with the Moses Machine Translation System [Koehn et al., 2007], which is further explained in Appendix F.3. The corpora I used were Europarl [Koehn, 2005], an eval set of JRC-Acquis [Steinberger et al., 2006], and for some other evaluations I used the newstest2009 corpus. I will explain each corpus used as well as the language pairs in each experiment. Before moving into that, I first used BLEU to establish which settings to use for both language models before the final set of experiments. This will now be discussed.

7.2.1 SRILM Best Settings

The established standard for SRILM in machine translation is to use Kneser-Ney smoothing with interpolation. I verified this by a short email correspondence with the author of SRILM, Andreas Stolke. To be even more certain, I ran several experiments with different settings in SRILM. The BLEU results of these experiments told me the best settings and those were the settings used to train all future SRILM language

[1] http://www.itl.nist.gov/iad/mig/tests/mt/2009/

[2] http://kheafield.com/code/mt/

Table 5: SRILM Settings Comparison

Order	BLEU	Settings
7	10.40	kndiscount, interpolate, unk
5	10.37	kndiscount, interpolate, unk
4	10.45	kndiscount, interpolate, unk
4	10.51	kndiscount, interpolate
4	10.41	kndiscount

Table 6: STLM Settings Comparison

BLEU	Settings
9.26	kndiscount, interpolate, unk, no depth limit
11.55	kndiscount, interpolate, unk, boost, no depth limit
11.96	kndiscount, interpolate, unk, boost, depth limit 4

models for other evaluations. The training corpus was the first 200k lines of Europarl (German-English), and the newstest2009 corpus for evaluation. The comparison is in Table 5.

It worked well to include UNK in the perplexity evaluations to better match with how STLM functions, but it worked best in BLEU scoring to simply use the order 4 model with Kneser-Ney discount with interpolation, which I will use for the rest of the experiments.

7.2.2 STLM Best Settings

Using the same corpora for training and evaluation (200k lines of Europarl for training, newstest2009 for evaluation, German-English), I tried various settings of the suffix tree language model (now denoted as STLM), as shown in Table 6.

After all which has been done, the best settings include a depth limit of 4. One would expect that an arbitrary n-gram size would always score the highest. However, it is not that strange because the best settings for SRILM are also at order 4. The suffix tree language model has been implemented to mimic traditional n-gram language models as closely as possible in how it estimates probabilities. The smoothing is very similar. It should be no real surprise to find that it in fact does function in a very similar way to a traditional n-gram model. However, even with these scores it is already showing promise because with or without the depth limit, it is scoring higher than SRILM with this small set because of some differences in estimation, such as the added boost.

7.2.3 Experiments: Small-Scale

With the best settings for each language model established, we now move to larger experiments using these best settings for each model. In the following experiments, various sizes of training sets were used. For each training set size, the same corpus is used to train the SMT system and the language model. Training sets of 100k, 200k, and 300k sentences were used from Europarl for training.

Table 7 shows the results for the newstest2009 evaluation. German is the source

Table 7: DE-EN comparison newstest2009

LM	Size	BLEU	NIST	TER	METEOR	Precision	Recall
STLM	100k	13.88	4.96	66.41	43.84	59.56	52.85
SRILM	100k	13.04	4.56	65.53	42.75	61.83	49.60
STLM	200k	15.12	5.17	65.12	46.07	61.39	54.71
SRILM	200k	13.52	4.69	64.47	44.45	63.49	50.86
STLM	300k	15.76	5.30	64.28	47.14	62.69	55.57
SRILM	300k	14.01	4.77	63.77	45.34	65.02	51.68

Table 8: DE-EN comparison acquis

LM	Size	BLEU	NIST	TER	METEOR	Precision	Recall
STLM	100k	23.03	6.24	59.07	50.98	69.06	58.95
SRILM	100k	21.83	5.48	59.87	49.49	72.14	54.77
STLM	200k	24.23	6.50	57.81	52.87	70.58	60.75
SRILM	200k	22.94	5.66	58.71	50.83	73.59	55.76
STLM	300k	24.96	6.64	57.37	53.75	71.15	61.53
SRILM	300k	23.57	5.75	57.93	51.86	74.83	56.40

language and English is the target language (DE-EN). Table 8 shows scores for the same training data; Europarl at various sizes of training, but using the acquis evaluation set.

Two more tables where English is the source language and German the target are represented by 9 and 10. This is where the story changes. When English was the target language, STLM performed better with all metrics except for TER, where a lower number means a better score. In fact, SRILM always performs better with the TER metric in all cases. For the newstest2009 evaluation from English to German, STLM is still ahead, but SRILM is ahead for the acquis evaluation set. This shows that the STLM doesn't perform as well when German is the target language. This is partially due to the fact that most of the development and optimizing of the smoothing parameters of STLM were done with a small training and evaluation set with German as the source and English as the target language. The STLM smoothing is quite greedy and a language with strict word order like English would benefit from that kind of language modeling approach. It still worked reasonably well when German was the target language, but there is much room for improvement. The fact that TER is always better for SRILM than STLM could provide a way of analyzing STLM to find ways to improve. Notice also that the precision and recall are overall more similar for the STLM. SRILM tends to favor precision.

A statistical significance test [Zhang and Vogel, 2004] was also performed to verify the significance in BLEU score differences. It can be verified that all evaluations are significant if the ones with the smallest difference are significant for the smallest evaluation set. This smallest difference happened in the English-German language pair. The set was the Europarl 100k for training, newstest2009 for evaluation. The difference here was statistically significant with a confidence of 100%. Because this set is smaller than aquis (it contains fewer sentences to evaluate) and the difference was smaller than all others, all other BLEU scores are statistically significant whether

Table 9: EN-DE comparison newstest2009

LM	Size	BLEU	NIST	TER	METEOR	Precision	Recall
STLM	100k	8.72	4.09	74.53	13.15	46.21	41.18
SRILM	100k	8.62	3.85	72.02	12.94	48.62	39.34
STLM	200k	9.76	4.27	72.61	13.95	47.82	42.27
SRILM	200k	9.14	3.93	71.33	13.28	49.82	39.78
STLM	300k	10.12	4.32	72.21	14.34	48.95	42.93
SRILM	300k	9.19	3.93	70.86	13.87	51.41	40.44

Table 10: EN-DE comparison acquis

LM	Size	BLEU	NIST	TER	METEOR	Precision	Recall
STLM	100k	18.92	5.55	66.65	21.03	56.06	52.22
SRILM	100k	19.45	5.48	64.30	21.20	58.58	50.01
STLM	200k	19.67	5.71	65.43	21.77	57.30	53.17
SRILM	200k	20.52	5.64	62.80	22.48	60.51	50.95
STLM	300k	20.40	5.85	64.58	22.77	58.29	53.82
SRILM	300k	21.27	5.72	62.09	23.08	61.43	51.21

SRILM or STLM scores higher.

7.2.4 Experiments: Small-Scale Combined

Another final experiment for these smaller sets was performed which combined the machine translation hypotheses of both the STLM and SRILM. The set of English-German Europarl 100k training, acquis evaluation was used (an instance where SRILM performed better). The Carnegie Mellon University *Multi-Engine Machine Translation* (MEMT) scheme was used to combine the hypotheses from both systems and tune a final system which yielded the combine results as shown in Table 11. The results with a combined system show a significant increase in all metrics except for TER, which is worse when compared to the SRILM score. Note that the combined score is significantly better than even the 300k scores in Table 10. Hence, even if SRILM is a better language model than STLM, together they perform very well in certain scoring metrics.

7.2.5 Experiments: Large-Scale

Large scale experiments were also performed using the entire Europarl corpus as training and Acquis for evaluation. German (DE), French (FR), and Spanish (ES) are all trained with English as the target language, then the same sets are used where

Table 11: EN-DE comparison MEMT

LM	Size	BLEU	NIST	TER	METEOR	Precision	Recall
STLM	100k	18.92	5.55	66.65	21.03	56.06	52.22
SRILM	100k	19.45	5.48	64.30	21.20	58.58	50.01
Both	100k	24.32	6.26	65.47	25.08	58.59	57.57

Table 12: English **target** comparison, full Europarl

LM	LANG	BLEU	NIST	TER	METEOR	Precision	Recall
STLM	DE	26.93	7.05	55.49	56.27	72.86	64.03
SRILM	DE	21.92	5.25	58.26	51.34	76.97	55.25
STLM	FR	37.29	8.45	45.73	66.65	75.59	75.88
SRILM	FR	36.69	8.59	43.85	66.62	80.87	71.57
STLM	ES	33.73	7.91	49.38	64.08	72.84	74.43
SRILM	ES	33.93	8.19	46.66	64.24	78.36	70.03

Table 13: English **source** comparison, full Europarl

LM	LANG	BLEU	NIST	TER	METEOR	Precision	Recall
STLM	DE	23.17	6.15	63.82	24.76	58.96	56.26
SRILM	DE	23.96	6.15	59.73	25.48	64.29	53.21
STLM	FR	33.61	8.05	49.37	20.45	70.23	67.14
SRILM	FR	35.11	8.24	47.26	22.42	74.33	64.17
STLM	ES	31.15	7.62	51.47	30.01	69.27	65.87
SRILM	ES	32.35	7.79	49.24	30.32	73.68	63.05

English is the source language. Europarl was used for training in the SMT system as well as the language model, using the same settings as before. Table 12 contains the scores for the two language models where English is the target language. Table 13 contains the scores when English is the source language.

Note that TER scores in Table 13 where English is the source language only uses parts of the TER metric (specifically, HTER can only accommodate English as the target language). These large-scale experiments show that overall SRILM has better performance in machine translation experiments. It isn't surprising that DE-EN has the widest gap between scores with STLM as the clear winner for all metrics as that was the development language pair and given the greedy nature of STLM. The only other time STLM came out on top was FR-EN, but only for certain scoring metrics. In all other cases, SRILM scores better (though EN-DE does have a tie with NIST). However, these scores are competitive, especially when English is the target language. It shows that STLM is indeed a powerful approach to language modeling in certain domains.

7.3 Memory Usage

A final comparison between the suffix tree language model and SRILM will be in memory usage. This isn't necessarily an experiment, but memory usage in language models is a hot topic, as language models with large amounts of training data are being pushed. I used a small training set, only 10k sentences (1.5 megabytes of text) of Europarl English for training, and 1k to run an evaluation. The evaluation wasn't for any purpose other than to see how much memory each model uses during training and evaluation.

I utilized the valgrind program which is a toolkit for profiling C++ compiled programs. The chart was generated automatically by means of the *massif* profiler. We are mostly interested in the highest, darkest line, which is the total heap usage.

Figure 6 shows the memory usage for SRILM during training. The same settings as in the BLEU experiment were used; order 4, Kneser-Ney smoothing with interpolation. The memory usage grows to 1.6 megabytes, which is only 1.066 times more memory than the size of the training corpus. Runtime evaluation (perplexity scoring) uses even less space; only about 0.35 megabytes.

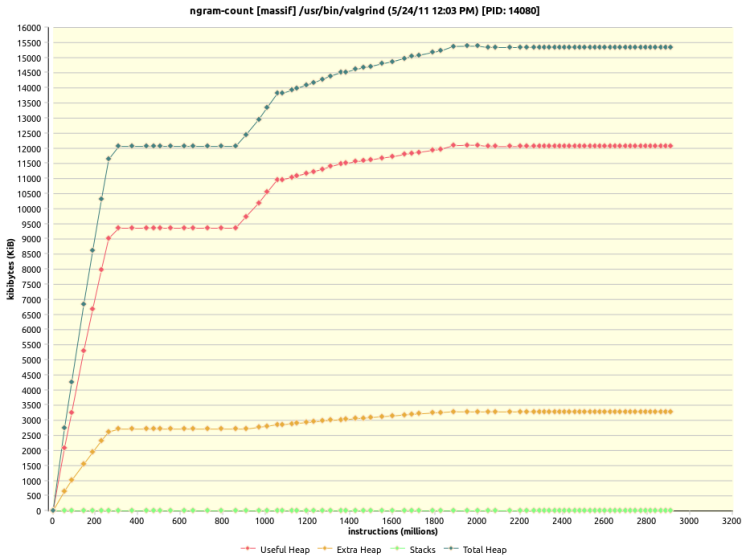

Figure 6: SRILM memory usage over time (training)

Figure 7 shows the memory usage for the suffix tree language model. Training grows linearly with the size of the corpus. At about 25, the training is complete and the rest of the growth is due to the Kneser-Ney diversities of histories (count of counts) gathering until it peaks, utilizing 48 megabytes of memory. Once those are found, the counts are kept but the data used to keep track of histories is discarded. The second peak is more necessary processing for Kneser-Ney smoothing. At 38, that is too complete and the book keeping data discarded. The rest of the evaluation holds steady at 31 megabytes. Thus, the peak in training is 48 megabytes, but when training is over, runtime evaluation is 31 megabytes. To better compare, at peak training, the suffix tree language model requires 32 times more megabytes in memory than the size of the corpus. During runtime evaluation, that number drops to 21 times more memory.

Figure 8 shows the maximum memory usage for SRILM during training for various language model orders. The significant thing to note here is that the maximum usage for SRILM training grows linearly as the order increases. The point where SRILM matches the suffix tree language model for this corpus is at an order of 11. Everything beyond 11 takes more memory during training. If one is concerned over memory usage, then and order up to 11 for SRILM will use less memory, but for any order above 11, the suffix tree language model uses less memory. With this in view, the price seems quite small for having the entire corpus with all n-grams represented.

However, one of the drawbacks of the suffix tree language model is the memory usage during evaluation. It uses 89 times more memory than SRILM during evalu-

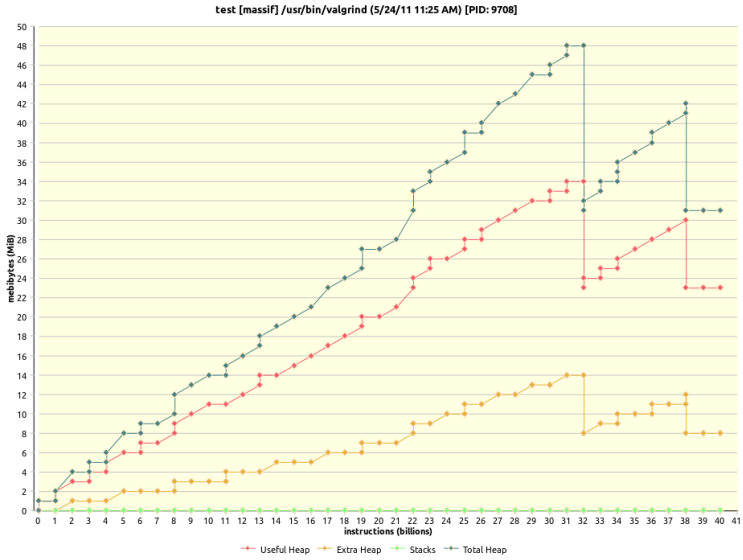

Figure 7: STLM memory usage over time

ation for a 4-gram, and even a 15-gram SRILM language model requires less than one megabyte during perplexity evaluation. In practical terms, when using the suffix tree language model, be prepared to have 32 times more memory than the size of the training corpus. For example, the English Europarl corpus is 251 megabytes, which would require over 8 gigabytes of memory for training.

A third model in the comparison, the suffix array language model, is very efficient in memory usage, but has some differences with training and evaluation when compared to the other two models. Training consists of indexing a corpus, which required less than 0.5 megabytes to do using the same training corpus as the other two models. Evaluation of SALM is where most of the work is done. It is actually during evaluation of the language model where one specifies the order of the model, so the n-gram information is calculated on the fly, which essentially means training is partially computed during evaluation, but that doesn't seem to affect the speed or the memory usage. Evaluation only required about one third of a megabyte, even for large orders.

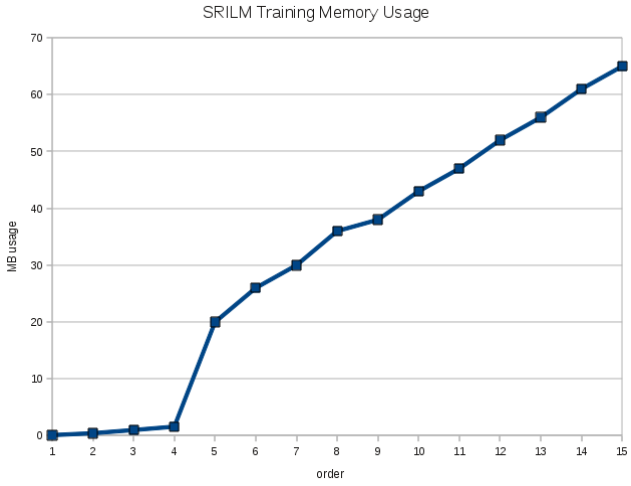

Figure 8: SRILM training memory usage with various orders

8 Conclusions and Future Work

8.1 General

In natural language processing, suffix trees carry great potential. Any application which needs n-gram information, substrings, text overlap, sentence alignment, or language models can benefit from suffix trees.

In language modeling specifically, the suffix tree can store an arbitrary length of word histories, up to the entire length of the corpus. Language model order need not be specified at training like other language models. The suffix links which are implicit in the data structure make parts of smoothing, more specifically back off, an efficient exercise.

The use of suffix trees is also a promising approach to language modeling when used in machine translation experiments. It outperforms a well-established language model, SRILM, by a significant amount in some experiments. In experiments where SRILM performs better, STLM is still quite competitive.

8.2 Incrementality

The nature of the suffix tree data structure which uses the Ukkonen Algorithm does more than just give us implicit back off and any-order ngrams. It also offers a dynamic language model which might be useful in more domains than just MT and ASR. Recall that building the tree begins with the first word of the corpus and the tree is built word by word. It can stop at any point and the tree represents a full suffix tree up to that point in the training corpus. If one wished to add more to the suffix tree, one could easily just add more words. The tree could also be used for evaluation at any point of training, though the more training data, typically, the better. This kind of easy adding of new information could be useful for dynamic systems, such as dialogue systems or information retrieval. Imagine two robots speaking to each other, and each robot is able to take in data spoken by the other robot and incorporate that

data in the language model at real time without a lot of extra computation, and then be able to turn around and use that newly acquired data to make decisions about future utterances. Removing words or sequences from the model in any context is another, more difficult matter.

The suffix tree language model also processes the evaluation sentences slightly differently than the traditional language models do. It trains incrementally as explained, but evaluation is also incremental, that is, word by word. As it processes a sentence probability, it begins with the first word then it looks only at the second word as it follows the first word, etc., whereas traditional language models will need the previous words in a window, up to the size of the order of the language model. Psycholinguistic research has found that humans process sentences incrementally and integrate information word by word [Altmann and Mirković, 2009]. The suffix tree language model could perhaps be used in psycholinguistic research because it appears to be a more cognitively plausible approach, though more research would be needed to establish its cognitive plausibility.

8.3 Future Work

More needs to be done to improve memory usage. To this end, perhaps finding a middle ground between suffix arrays and suffix trees would allow the memory usage and speed of suffix arrays while keeping some of the implicit parts of the suffix tree which are useful to language modeling. Further, I would like to find a way to fully normalize the model while still having the benefit of the depth boost. Attempting other smoothing techniques which better exploit the properties of the data structure might be beneficial and allow for broader use. The model certainly needs to be to evaluated in other domains like ASR and information retrieval. It would also be useful to see if the model performs well with sparse data.

References

[Altmann and Mirković, 2009] Altmann, G. T. and Mirković, J. (2009). Incrementality and prediction in human sentence processing. *Cognitive Science*.

[Baker, 1997] Baker, B. S. (1997). Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.*, 26:1343–1362.

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

[Chen and Goodman, 1998] Chen, S. and Goodman, J. (1998). An empirical study of smoothing techniques for language modeling. Technical Report TR-10-98, Computer Science Group Harvard University.

[Clarkson and Rosenfeld, 1997] Clarkson, P. and Rosenfeld, R. (1997). Statistical language modeling using the cmu-cambridge toolkit. Eurospeech.

[Gusfield, 1997] Gusfield, D. (1997). *Linear-Time Construction of Suffix Trees, Algorithms on strings, trees, and sequences : computer science and computational biology.* Cambridge Univ. Press.

[Huang and Powers, 2008] Huang, J. H. and Powers, D. (2008). Suffix tree based approach for chinese information retrieval. In *Eighth International Conference on Intelligent Systems Design and Applications*.

[Jurafsky and Martin, 2008a] Jurafsky, D. and Martin, J. H. (2008a). N-grams. In *Speech and Language Processing (2nd Edition) (Prentice Hall Series in Artificial Intelligence)*. Prentice Hall.

[Jurafsky and Martin, 2008b] Jurafsky, D. and Martin, J. H. (2008b). *Speech and Language Processing (2nd Edition) (Prentice Hall Series in Artificial Intelligence)*. Prentice Hall, 2 edition.

[Kay, 2004] Kay, M. (2004). Substring alignment using suffix trees. Computational Linguistics and Intelligent Text Processing, 5th International Conference, CICLing, Springer.

[Kneser and Ney, 1995] Kneser, R. and Ney, H. (1995). Improved backing-off for m-gram language modeling. In *In Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume I, page 181â[U+0080][U+0093]184, Detroit, Michigan.

[Koehn, 2005] Koehn, P. (2005). Europarl: A parallel corpus for statistical machine translation. In *Conference Proceedings: the tenth Machine Translation Summit*, pages 79–86, Phuket, Thailand. AAMT, AAMT.

[Koehn, 2010] Koehn, P. (2010). Langauge models. In *Statistical Machine Translation*, chapter 7, pages 181–216. Cambridge University Press.

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

[Lavie and Denkowski, 2010] Lavie, A. and Denkowski, M. (2010). The meteor metric for automatic evaluation of machine translation. In *Machine Translation*.

[Lv and Zhai, 2009] Lv, Y. and Zhai, C. (2009). Positional language models for information retrieval. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '09, pages 299–306, New York, NY, USA. ACM.

[Manning and Schütze, 1999] Manning, C. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT Press.

[Markov, 1913] Markov, A. A. (1913). An example of statistical investigation in the text of "eugene onyegin" illustrating coupling of "tests" in chains. In *Proceedings of Academic Scienctific St. Petersburg, VI*, pages 153–162.

[McCreight, 1976] McCreight, E. M. (1976). A space-economical suffix tree construction algorithm. *J. ACM*, 23:262–272.

[Monostori et al., 2000] Monostori, K., Zaslavsky, A., and Schmidt, H. (2000). Identifying overlapping documents in semi-structured text collections. Australasian Computer Science Conference.

[Nelson, 1996] Nelson, M. (1996). Fast string searching with suffix trees. *Dr. Dobb's Journal*.

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

[Papineni and et al, 2002] Papineni, K. and et al, S. R. (2002). Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics*.

[Saykham et al., 2010] Saykham, K., Chotimongkol, A., and Wutiwiwatchai, C. (2010). Online temporal language model adaptation for a thai broadcast news transcription system. In Chair), N. C. C., Choukri, K., Maegaard, B., Mariani, J., Odijk, J., Piperidis, S., Rosner, M., and Tapias, D., editors, *Proceedings of the Seventh conference on International Language Resources and Evaluation (LREC'10)*, Valletta, Malta. European Language Resources Association (ELRA).

[Snover et al., 2006] Snover, M., Dorr, B., Schwartz, R., Micciulla, L., and Makhoul, J. (2006). A study of translation edit rate with targeted human annotation. In *Proceedings of Association for Machine Translation in the Americas*.

[Stehouwer and van Zaanen, 2010] Stehouwer, H. and van Zaanen, M. (2010). Enhanced suffix arrays as language models: Virtual k-testable languages. *Grammatical Inference: Theoretical Results and Applications*, 6339:305–308.

[Steinberger et al., 2006] Steinberger, R., Pouliquen, B., Widiger, A., Ignat, C., Erjavec, T., Tufiş, D., and Varga, D. (2006). The jrc-acquis: A multilingual aligned parallel corpus with 20+ languages. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC'2006)*. LREC.

[Stolcke, 2002] Stolcke, A. (2002). Srilm - an extensible language modeling toolkit. International Conference on Spoken Language Processing.

[Talbot and Osborne, 2007] Talbot, D. and Osborne, M. (2007). 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. Association for Computational Linguistics.

[Ukkonen, 1995] Ukkonen, E. (1995). On-line construction of suffix trees. *ALGORITHMICA*.

[Weiner, 1973] Weiner, P. (1973). Linear pattern matching algorithm. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11.

[Zhang and Vogel, 2004] Zhang, Y. and Vogel, S. (2004). Measuring confidence intervals for the machine translation evaluation metrics. In *Proceedings of The 10th International Conference on Theoretical and Methodological Issues in Machine Translation*. International Conference on Theoretical and Methodological Issues in Machine Translation.

[Zhang and Vogel, 2006] Zhang, Y. and Vogel, S. (2006). Suffix array and its applications in empirical natural language processing. Technical report, CMU, Pittsburgh PA.

A Appendix: Language Modeling Toolkits

A.0.1 SRI Language Model

The [3]SRI Language Model (SRILM) [Stolcke, 2002] is a toolkit for building and applying statistical language modeling written by Andreas Stolke. It is a powerful toolkit that has been in development since 1995. It is a set of C++ class libraries, executable, programs, and helper scripts. It supports most of the known smoothing methods (see Section 6) and is easily coupled with the Moses Statistical Machine Translation toolkit [Koehn et al., 2007]. SRILM also has the ability to train on very large corpora. I will compare my model extensively with SRILM because it works well with Moses and it is possibly the most used language modeling toolkit in use. SRILM supports the ARPA file format. Language models can also be binarized for faster access.

A.0.2 RandLM

[4]RandLM is a result of efforts to push even more training data into a language model by using space-efficient n-gram-based language models. It uses randomized representations known as bloom filters (see [Talbot and Osborne, 2007]). RandLM was made primarily for machine translation and can be easily coupled with Moses, but the developers hope that the code will be useful in other areas. I will not compare my model to RandLM because my focus isn't in large models.

A.0.3 IRST Language Modeling Toolkit

The [5]IRST Language Modeling Toolkit (IRSTLM) is a language model by the Fondazione Bruno Kessler Human Language Technology group, according to their website, features algorithms and data structures to estimate, store, and access very large langauge models. It is also easily coupled with moses and is compatible with SRILM models. I will not compare to the IRSTLM for the same reason as RandLM.

A.0.4 CMU-Cambridge Statistical Language Modeling Toolkit

[6]The CMU-Cambridge SLM toolkit comes in two versions. Version 1 is limited to trigrams and simple forms of smoothing. Version 2 [Clarkson and Rosenfeld, 1997] is more like SRILM in that it supports n-grams of arbitrary size, has the ability to store large amounts of data, is faster at runtime, as well as more advanced smoothing methods. The toolkit was originally made for the [7]CMU Sphinx speech recognizer, though it has always been freely available for use in other areas. This toolkit supports the ARPA file format.

[3]http://www-speech.sri.com/projects/srilm/
[4]http://randlm.sourceforge.net
[5]http://hlt.fbk.eu/en/irstlm
[6]http://www.speech.cs.cmu.edu/SLM/toolkit.html
[7]http://cmusphinx.sourceforge.net/

A.0.5 KenLM

[8]KenLM was written by Kenneth Heafield. It boasts better speed and less memory usage than SRILM or IRSTLM. It takes already-trained ARPA files, for example, by SRILM, and can store the information faster and with less memory than is utilized when SRILM reads in its own ARPA formatted file language model file. KenLM is freely available and is compiled and ready for use (unless otherwise directed) in Moses. As my thesis work isn't as much focused on speed and memory, I won't compare my model to this one.

A.0.6 Suffix Array Language Model

The [9]Suffix Array Language Model (SALM) [Zhang and Vogel, 2006] was developed by Ying Zhang and Stephan Vogel of CMU. The suffix array approach to language modeling is very similar to the suffix tree approach. In a theoretical sense, they are one and the same as both can store the same information. SALM runs extremely fast and doesn't require as much storage space as the suffix tree approach, but lacks in other areas, and certainly there are some differences when applied to language modeling. The current version of SALM does have the capability to be a language model, but it doesn't have the most effective language modeling techniques for estimation implemented. It also requires a limit on the order of the model (according to my own experiments) defined during runtime, though defining different orders on the fly doesn't seem to affect its efficiency. The toolkit can also be used for things besides language modeling such as fast text searching, n-gram frequency counting, duplicate sentence searching, among others. Other work has been done using suffix array language models in classification tasks where the order is not limited [Stehouwer and van Zaanen, 2010]. It doesn't appear to have been developed since 2007.

[8]http://kheafield.com/code/kenlm/

[9]http://projectile.sv.cmu.edu/research/public/tools/salm/salm.htm

B Appendix: Perplexity

Perplexity is based on cross-entropy (again, see Chapter 7 of Philip Kohen's book, on which this explanation is based), which is formalized in 16.

$$H(p_{LM}) = -\frac{1}{n}\sum_{i=1}^{n} logp_{LM}(w_i|w_1...w_{i-1}) \tag{16}$$

Perplexity is therefore a transformation of cross-entropy:

$$PP = 2^{H(p_{LM})} \tag{17}$$

The cross entropy is an average of the negative logarithm of the word probabilities. More simply, entropy, on which perplexity is based, measures uncertainty in a probability distribution, formalized in 18.

$$H(p) = -\sum_{x} p(x)log_2 p(x) \tag{18}$$

Because of the log base 2 nature, entropy is a common in computer science because it relates to how many bits are necessary to encode something. For very certain outcomes, the entropy will be 0. If there are more possibilities, hence more confusion, entropy will be higher. Hence, the lower the entropy for some real text, the better. Perplexity can be found for a small sequence of words like a sentence, or it can be calculated for an entire corpus. This is where our evaluation corpora are used.

SRILM has [10]two different perplexity measures. The first is the geometric average of $\frac{1}{probability}$ of each word, which is perplexity, shown in 19.

$$PP = 10^{(\frac{-logprob}{words-OOVs+sentences})} \tag{19}$$

But this takes the $< /s >$ tokens into account. To be complete, SRILM also offers a second perplexity measure which excludes those tokens, shown in 20

$$PP = 10^{(\frac{-logprob}{words-OOVs})} \tag{20}$$

[10]http://www.speech.sri.com/projects/srilm/manpages/srilm-faq.7.html

The *words* variable represents the number of words in the evaluation corpus. The *OOVs*, or *out-of-vocabulary* words variable represents the number of times a word was found which wasn't in the corpus. How this is handled depends on how the language model was trained. SRILM can be trained on Kneser-Ney discounting with interpolation, which I explained to be the best current method. It also has the added functionality of including or excluding an *UNK* variable which acts as a placeholder for all unseen words. When SRILM is trained without UNK, then it counts the OOVs. When UNK is used, UNK becomes a placeholder, so the model thinks that there are no OOVs, so it always evaluates to zero. Also, notice that SRILM uses 10 as the base, which means log probabilities must be in base 10. The base doesn't really affect the usefulness of perplexity, as it is an internal measure of the model and will show improvements as the perplexity decreases. I show increasing orders for all language models until the perplexity levels off or worsens.

C Appendix: Smoothing Comparison

It has been shown that large amounts of training corpora lead to a lowered dependence on sophisticated smoothing techniques. [Brants et al., 2007] introduced a smoothing technique which they called *Stupid Backoff* which is a very simple approach to smoothing. In their approach, there is no discounting and back off is relied upon when words don't appear in a context. They showed that, given large amounts of data (2 trillion tokens), this simple smoothing technique approached the performance of Kneser-Ney smoothing in MT experiments.

To illustrate this with the Suffix Tree language model, I implemented a simple smoothing technique that similarly relies heavily on back off, but does incorporate some discounting and it also applies boost if it reaches a depth of the tree greater than 2. It was similarly limited to a 4-gram like the original experiments of this thesis. The results of translating from German (DE), French (FR), Spanish (ES) into English are shown in table 14. The results where English was the source language are shown in table 15. The results from the original experiments of this theses are shown again for convenience, where STLM SS represents the simple smoothing, and STLM KN represents the original Kneser-Ney-like smoothing. I predicted before the experiments were complete that all of the results would be worse than the STLM and SRILM results, and this was mostly the case. It does still perform better in some instances. The important thing to note is that the performance isn't as terrible as one might expect with such a simple smoothing approach. Because it does not need to compute the count of counts, it takes less memory and processing time. Everything necessary to compute the smoothing exists in the tree.

Table 14: English **target** comparison, full Europarl, simple smoothing

LM	LANG	BLEU	NIST	TER	METEOR	Precision	Recall
STLM SS	DE	26.79	7.03	55.83	56.15	73.16	63.87
STLM KN	DE	26.93	7.05	55.49	56.27	72.86	64.03
SRILM	DE	21.92	5.25	58.26	51.34	76.97	55.25
STLM SS	FR	36.14	8.18	47.48	66.14	73.35	76.40
STLM KN	FR	37.29	8.45	45.73	66.65	75.59	75.88
SRILM	FR	36.69	8.59	43.85	66.62	80.87	71.57
STLM SS	ES	33.05	7.79	50.47	63.52	71.83	74.19
STLM KN	ES	33.73	7.91	49.38	64.08	72.84	74.43
SRILM	ES	33.93	8.19	46.66	64.24	78.36	70.03

D Appendix: Moses SMT

I made extensive use of the [11]Moses Statistical Machine Translation System, particularly the very useful [12]*Experiment Management System* (EMS). The EMS is a set of perl scripts which, given a configuration file, will pre-process, tokenize, train the translation system, train the language model, and run evaluations. It is easily configured and if an experiment were to fail at some point, it gives information on the error and can pick up where it left off after the error is fixed. I also made extensive use of the GIZA++ statistical translation models toolkit [Och and Ney, 2003], which is used by Moses and EMS to train the translation models.

Table 15: English **source** comparison, full Europarl, simple smoothing

LM	LANG	BLEU	NIST	TER	METEOR	Precision	Recall
STLM SS	DE	23.13	6.18	63.22	24.49	59.95	55.77
STLM KN	DE	23.17	6.15	63.82	24.76	58.96	56.26
SRILM	DE	23.96	6.15	59.73	25.48	64.29	53.21
STLM SS	FR	32.77	7.86	51.09	19.87	69.32	67.39
STLM KN	FR	33.61	8.05	49.37	20.45	70.23	67.14
SRILM	FR	35.11	8.24	47.26	22.42	74.33	64.17
STLM SS	ES 31.27	7.56	52.16	29.96	68.67	66.80	
STLM KN	ES	31.15	7.62	51.47	30.01	69.27	65.87
SRILM	ES	32.35	7.79	49.24	30.32	73.68	63.05

[11]http://www.statmt.org/moses/index.php?n=Main.HomePage
[12]http://www.statmt.org/moses/?n=FactoredTraining.EMS

E Appendix: Usage

E.1 Features

The suffix tree language model is a fully-implemented language model which uses a suffix tree data structure to store an entire corpus. It has an arbitrary length on accessible n-grams and is on-line and dynamic. It must be given a training corpus, then it can run a test on an evaluation corpus outputting a log probability for each line, or perplexity for the entire corpus. It can also run in server mode and be accessible by other XMLRPC clients. Useful statistics and tree information can be displayed. It has a unit test suite which tests most of the individual classes and methods. The number of padding, the maximum depth of the tree (if desired) which is used, the Kneser-Ney smoothing diversity of histories depth, the text compression, as well as a debug mode can all be set in the Constants header file. It can be coupled with the Moses SMT system.

E.2 Command Line Arguments

When invoking the suffix tree language model, these are the possible command line parameters:
-text path to the tokenized training corpus - required
-server a binary flag which invokes the server mode - a message and the utilized port number will be displayed when ready for use
-port sets the port which is used by the server (default 2001) - requires the *-port* flag
-test path to an evaluation corpus which is assumed to have one string on each line to evaluate - prints one log probability per line
-stats outputs some interesting information about the tree and an evaluation corpus - requires the *-test* flag
-perp when used in conjunction with *-text*, this binary flag outputs a perplexity score for the entire evaluation corpus
-unit runs the unit tests

E.3 Examples

Training and evaluation:

```
./stlm -text /path/to/training.corpus -test /path/to/eval.corpus
```

Training and server mode using port 3003:

```
./stlm -text /path/to/training.corpus -server -port 3003
```

Test using perplexity:

```
./stlm -text /path/to/training.corpus -test /path/to/eval.corpus -perp
```

Unit test suite:

```
./stlm -text /path/to/training.corpus -unit
```

The server uses XMLRPC. A sample python script that could connect to the server would look like this:

```
import xmlrpclib
stlm = xmlrpclib.ServerProxy("http://localhost:2001/")
prob = stlm.logProb(['this','is','a','sentence'])
print prob
```

Note that logProb is the only registered function with the server. It takes in a sentence and returns the log probability.

F Appendix: Code Design

F.1 Environment

The main part of the suffix tree language model is implemented in the C++ programming language. I attempted to mimic the code style of the moses documentation.[13]. Several of the parameters including Kneser-Ney smoothing diversity of histories depth, a limiter on the tree depth, sentence padding, among other variables, can be set in the Constants header file. I used the Eclipse[14] programming environment.

There is a main driver class called *STLM*. This class will reference and use the *SuffixTree* class, which takes a corpus file as input to build a tree. Given this file, it will assume that the text is tokenized and proceed to process the text by segmenting words based on white space and training will commence.

A *SuffixTree* is made up of *Segment* classes. These *Segment* classes can be *InternalSegment*, *RootSegment*, *BaseSegment*, and *LeafSegment*. All segments are similar in function, hence the inheritance. Each *Segment* will contain a link to its left (or parent) node, and a suffix link which links to the node with a one-word shorter history. There will be only one *RootSegment* which contains root node information. *InternalSegment* has some special functions. It is neither in a root or leaf position in the tree, and because of the way the tree is created, it needs to be able hold more information than just one word for a node. An *InternalSegment* will have a link to its parent node, a suffix link, and a link to all of its children. The children only appear on the last word of the segment. That is, if there is an arbitrary number of words that follow each other (each word only has one child), then there is no need to branch at that point, so the segment holds all information pertaining to that string of words until a word with multiple children is found. In many cases *InternalSegments* will only have one word and many children.

Many programming languages offer the ability to monitor and take care of memory that is no longer being used by a program. C++ doesn't have that option, so everything must be deallocated in a destructor method in order to free the memory for further use. If there is memory which becomes unusable during runtime because it was not deallocated properly, then it is considered a *memory leak*. A program called *valgrind*[15] can be used to find memory leaks, which I will employ. Valgrind also has a tool which can graph the memory usage over time, which I also used.

F.2 Additional Packages

Unit testing, a standard software engineering practice, is done in the suffix tree language model by *cpptest*[16], a simple unit testing framework designed for unit testing in C++.

The suffix tree language model also has a server mode, which is made possible by *XmlRpc++*[17]. XMLRPC is a standard way of creating services. It even allows

[13] http://www.statmt.org/moses/?n=Moses.CodeStyle
[14] http://eclipse.org
[15] http://valgrind.org/
[16] http://cpptest.sourceforge.net/
[17] http://xmlrpcpp.sourceforge.net/

communication to and from various programming languages. For example, one can set the suffix tree language model in server mode and use the Python programming language to send sentences and receive log probabilities.

A string compression tool known as *zlib*[18] is also included. It could potentially save space when using very large vocabularies. For small vocabularies it makes little difference.

F.3 Adaptation to Moses SMT

Several changes needed to be made to some of the source files in order to make the suffix tree language model compile and be usable by moses. Moses uses unique identifiers with language models, I set STLM to have an ID of 10.

configure.in

```
if test "x$with_stlm" != 'xno'
then
  SAVE_CPPFLAGS="$CPPFLAGS"
  CPPFLAGS="$CPPFLAGS -I${with_stlm}/inc"
  AC_CHECK_HEADER(STLanguageModel.h,
                  [AC_DEFINE([HAVE_STLM], [], [flag for STLM])],
                  [AC_MSG_ERROR([Cannot find STLM!])])
  LDFLAGS="$LDFLAGS -L${with_stlm}/lib -L${with_stlm}/obj"
  LIBS="$LIBS"
  #AC_CHECK_LIB([], [], [], [AC_MSG_ERROR([Cannot find STLM's library in ${with_stlm}/l
  AM_CONDITIONAL([ST_LM], true)
```

moses/Makefile.am

```
if ST_LM
libmoses_la_SOURCES += LanguageModelSTLM.cpp
endif

if ST_LM
libmoses_la_HEADERS += LanguageModelSTLM.h
endif
```

moses/src/TypeDef.h

```
#  ifdef HAVE_STLM
#    define LM_ST 1
#  else
#    undef LM_ST
#  endif
(added to ID list) ,STLM = 10
```

moses/src/LanguageModelFactory.cpp

[18] http://www.zlib.net/

```
Header:
#ifdef LM_ST
# include "LanguageModelSTLM.h"
#endif

Switch:
case STLM:
#ifdef LM_ST
lm = new LanguageModelSTLM();
#endif
```

There needs to be a class and corresponding header for LanguageModelSTLM which implements LanguageModelPointerState. In this class, the Load method will be where the path to the tokenized corpus is passed to the suffix tree language model class. It is then built and made ready to use. The GetValue method creates a sentence object usable by the suffix tree language model which is then passed to the language model to obtain the log probability. As it is in log base 10, it is converted before being returned.

The final thing was to change was in the LanguageModel class. Here, in the CalcScore method it takes a sentence and creates n-grams based on the order of the language model. It also pads the sentence according to the order. I wanted to be able to control padding, but always have the full sentence sent to my language model instead of n-grams. To this end, I took out several assert statements and moved the code that evaluates the ngram outside of the loop. This makes one full-sentence "n-gram" and the padding is set by the order, which is typically 1 or 0 for the suffix tree language model.

Finally, sometimes moses wasn't able to use the suffix tree language model even though the path to the object files is set at compilation. This is remedied by compiling a shared library file of the suffix tree language model and putting it in a place where moses can access it. This library file creation is included in the Makefile under *lib*.

There are some parts of code that are implemented, but don't work very well. For example, because of the nature of the language model, it didn't make sense to create a standard ARPA formatted file as that would limit the depth of the data, and construction from an ARPA formatted file would not be enough to reconstruct the suffix tree. I therefore made a custom language model file. It works, but somehow reading in the file takes longer than training from the original. For now, I removed the command line flag which tells the software where to create the language model file.