Unsupervised Hierarchical Probabilistic Segmentation Of Discrete Events*

Guy Shani†
Department of Information Systems Engineering
Ben-Gurion University, Israel
shanigu@bgu.ac.il

Asela Gunawardana and Christopher Meek
Microsoft Research, USA
aselag@microsoft.com, meek@microsoft.com

April 13, 2010

Abstract

Segmentation, the task of splitting a long sequence of symbols into chunks, can provide important information about the nature of the sequence that is understandable to humans. We focus on unsupervised segmentation, where the algorithm never sees examples of successful segmentation, but still needs to discover meaningful segments. In this paper we present an unsupervised learning algorithm for segmenting sequences of symbols or categorical events. Our algorithm hierarchically builds a lexicon of segments and computes a maximum likelihood segmentation given the current lexicon. Thus, our algorithm is most appropriate to hierarchical sequences, where smaller segments are grouped into larger segments. Our probabilistic approach also allows us to suggest conditional entropy as a measure of the quality of a segmentation in the absence of labeled data. We compare our algorithm to two previous approaches from the unsupervised segmentation literature, showing it to provide superior segmentation over a number of benchmarks. Our specific motivation for developing this general algorithm is to understand the behavior of software programs after deployment by analyzing their traces. We explain and motivate the importance of this problem, and present segmentation results from the interactions of a web service and its clients.

Keywords: Software Analysis, Sequence Segmentation, Probabilistic Segmentation, Multigram

*Parts of this paper appeared in the International Conference on Data Mining, ICDM, 2009 [18].
†Corresponding author.

1 Introduction

Data is often naturally expressed as a long sequence of discrete symbols or events. Typically, humans find it very difficult to identify interesting patterns or chunks within the long sequence. In such cases it may be beneficial to automatically segment the sequence into smaller chunks, resulting in a shorter sequence over a higher level lexicon [16, 6].

For example, many modern software developers add trace messages to their software, allowing them to collect logs of the execution of the software after deployment. When problems arise after deployment, the developer can look at these logs to try and understand the behavior that caused the problem. Typically, these logs can become very large, and understanding the behavior of the software by looking at the recorded events becomes very difficult. However, if we can automatically identify that certain subsequences (or chunks) of these sequences originate from certain procedures, replacing these chunks by a single procedure name can make tracking these logs more manageable.

In many cases the developer can construct some model of the software, such as a Control Flow Graph (CFG, see e.g. [1]) that is generated by a compiler, making the task easier by providing a lexicon to the segmentation algorithm. However, in two important examples such a model is not available. In user driven applications, such as word processors, the logs capture the sequence of actions taken by the user, which is not completely determined by the program. In web services that provide services for clients, the logs capture the sequence of actions taken by the clients, which are typically programs that were written by a second or third party developer. In both cases, the developer has no access to a model of the behavior of the software, and cannot provide the segmentation algorithm with an existing lexicon.

The unsupervised segmentation problem is naturally decomposed into two phases — the identification of a lexicon of meaningful segments, or chunks, and the seg-

mentation of the sequence given the lexicon of chunks. In this paper we suggest a hierarchical, agglomerative approach, where these two phases are repeated iteratively, incrementally building a hierarchical lexicon and creating a hierarchical segmentation of the sequence.

We begin with a lexicon that contains all unique symbols and segment the sequence into minimal single-symbol chunks. At each iteration, we first select successive chunks to concatenate and add to the lexicon based on the mutual information between them. We then compute a probabilistic model known as a multigram, which learns the probabilities of the 'words' of the lexicon. Finally, this model is used to re-segment the sequence, disambiguating between multiple segmentations that may be allowed by the lexicon by choosing the segmentation with maximal likelihood.

Thus, our approach constructs a hierarchical lexicon and a hierarchical segmentation of the sequence. This is natural in many problems, including the software traces problem, as software programming is typically done using hierarchical structures, where procedures execute sub-procedures.

We compare the performance of our method to two previous algorithms, Sequitur [16] and Voting Experts [6], over standard benchmarks from the literature — an English and Chinese word identification tasks. In both cases our method outperform both Sequitur and Voting Experts.

Finally, we present some results on lexicon acquisition and segmentation over a real dataset of service requests from different clients accessing a Microsoft Exchange server. Even though we do not have labeled data for this domain, we show that our hierarchical lexicon acquisition method generates segments that capture much of the structure present in the logs. We measure this by estimating how much of the Shannon entropy of the service request sequences is captured by the segmentations generated by our algorithm.

2 Background and Previous Approaches

We begin by formally defining the problem of segmenting sequences of discrete events. We then present in detail two competing approaches — the Sequitur algorithm that uses a computational linguistic approach, and the Voting Experts algorithm that uses a probabilistic approach. We end the section describing additional related work.

2.1 Segmentation of Sequences of Discrete Events

Let $\vec{e} =< e_1, e_1, .., e_n >$ be a sequence of discrete events ("letters") where each e_i belongs to a finite, known, alphabet Σ. Let $\eta = \{\vec{e}_1, ..., \vec{e}_m\}$ be a set of m such sequences.

A *segmentation* of $\vec{e}$ will be denoted by $\vec{i}$, which is a sequence of indexes $i_1, ..., i_k$ s.t. $i_1 = 0$, $i_k = |\vec{e}|$ and $i_j < i_{j+1}$. We say that s_j is the j^{th} segment in the sequence $\vec{e}$ with segmentation $\vec{i}$ if $s_j =< e_{i_j+1}, \ldots, e_{i_{j+1}} >$. That is, the segment boundary is placed after the segmentation index.An event sequence $\vec{e}$ and its segmentation $\vec{i}$ generate a *segment sequence* $\vec{s}$ by $\vec{s} =< s_1, \ldots, s_{k-1} >$.

By extension, we say that s is a segment in η if there exists $\vec{e} \in \eta$ with segmentation $\vec{i}$ such that s is a segment in $\vec{e}$ with segmentation $\vec{i}$. The set of all segments found in η will be called the lexicon, denoted by S.

Example 2.1 *Consider the sequence of characters* $\vec{e} =< callmeishmael >$ *over the English alphabet* $\Sigma = \{a, ..., z\}$. *A possible segmentation of this sequence is* $\vec{i} = 0, 4, 6, 13$*, denoting the segmentation "call me ishmael", the first line of the novel "Moby Dick". The segments of this segmentation are* $\vec{s} =< call, me, ishmael >$. *For "call", e.g.,* $j = 1$, $i_j = 0$*, and* $i_{j+1} = 4$*, and the symbols are* $e_1 = c, e_2 = a, e_3 = l, e_4 = l$.

A hierarchical lexicon is a set of segments such that if s is in the lexicon and s contains more than a single event, then there exist segments $s_1, .., s_k$ in the lexicon

such that $s = s_1 + ... + s_k$, where $+$ is the concatenation operator. $s_1, ..., s_k$ are the sub-segments of s.

2.2 Sequitur

Context free grammar (CFG) is a tool for describing formal languages. A CFG is a set of rules of the form $X_i \rightarrow X_{i1}, ..., X_{ik}$, where X_{ij} is either a rule or a symbol. An expansion of a rule is the replacement of the left hand side of the rule with the right hand side repeatedly, until all rules have been replaced by symbols.

The Sequitur algorithm [16] constructs a CFG over a sequence of symbols by repeatedly identifying subsequences of symbols that appear together more than once.

For each such subsequence, Sequitur creates a rule that expands to that subsequence. Then, all the occurrences of the subsequence are replaced by the rule symbol. The operation is repeated until the following two properties are met: First, each rule, except for the root rule,is used more than once. Second, each subsequence of symbols appears only once. Hence, Sequitur builds a complete parse tree of the sequence. The resulting set of rules is not necessarily unique. Sequitur does not prefer one CFG to another as long as both CFGs obey the properties above.

Sequitur was designed to learn the hierarchical structure of sequences of discrete events, not for segmentation, but this hierarchical structure can be used for segmenting the sequence [6].

The rules that Sequitur finds can be thought of as a hierarchical lexicon. Now, we can segment a sequence, by applying these rules (i.e. parsing the sequence), and treating each rule expansion as a segment. In practice, this amounts to expanding the rules from the root rule to some depth,which may vary at every branch of the tree. Segment boundaries are placed after the rule expansion at that depth. As Sequitur does not provide tools for deciding where to stop the expansion, stopping at a fixed depth seems reasonable. For example, we can decide that segment boundaries are placed

after the expansion of rules of a fixed depth d.

Example 2.2 *Let us consider the sequence "abcdbcabcdbc" over the finite alphabet* $\Sigma = \{a, b, c, d\}$[1]. *We can construct the following rules:*

- $S \rightarrow AA$
- $A \rightarrow aBdB$
- $B \rightarrow bc$

that maintain the two properties. A possible segmentation is to stop at depth 2, resulting in expanding the rules S *and* A *but not the rule* B*, and in the following segmentation "a bc d bc a bc d bc".*

2.3 Voting Experts

A second, probabilistic approach, was suggested by Cohen et al [6] in their Voting Experts algorithm. The algorithm uses a set of 'experts' — rules for deciding where to place segment boundaries. The algorithm moves a sliding window of a fixed size over the sequence of symbols. At each position of the window, each expert votes on the most likely segment boundary within the current window. Then, we traverse the sequence of symbols, and introduce segment boundaries where the sum of votes for the next position is smaller than the sum of votes for the current position.

As opposed to Sequitur, the experts can use probabilistic information, making the resulting segmentation a maximum likelihood segmentation of the sequence. Specifically, Cohen et al. use two experts — one that minimizes the internal entropy of the chunk, and one that maximizes the frequency of the chunk among chunks of the same length. Voting Experts was demonstrated to outperform Sequitur on a word boundary discovery from different languages, and in identifying robot motion sequences.

[1] This example is taken from [16]

Example 2.3 *Consider the sequence of symbols "itwascold" over the English alphabet*[2]*. We slide a window of length 3 over the sequence. The first window we consider covers the symbols "itw". The entropy expert votes on placing a boundary after the* t*, while the frequency expert votes on placing the boundary after the* w*. The window slides to the next position, covering the symbols "twa". Now, both experts vote on placing the boundary between* t*, and* w*. The window moves again, covering "was", and both experts vote on a boundary after the* s*. The votes for the first 3 positions are hence 0 (no one voted for a boundary after the* i*), 3 (1 vote from the first window and two from the second window), and 1 (a vote from the first window). As the third symbol got a smaller vote than the second symbol, a boundary is placed after the* t*. Hence, the word "it" becomes the first segment of this sequence.*

2.4 Related Work

The related problem of time series segmentation [3] has been thoroughly studied in the past [10], and applied to numerous real world problems. Time series are sequences of continuous events usually modeling some sensor outputs. The continuous nature of the data can be used in order to define a similarity between events and to compute values such as the mean of the events in a segment. Therefore, most of these methods are not directly applicable to discrete events.

Another similar problem is DNA sequence segmentation. In this domain, Bussemaker et al. [4] suggest an algorithm that uses a statistical test to establish whether two adjacent events are related. Related events are unified and the algorithm is executed again, using the aggregated events instead of the low-level one. This lexicon acquisition is similar to the method that we suggest below, only training the word probabilities using a method specialized for the domain. Bussemaker et al. are interested in finding interesting segments only, not in the full segmentation of the data. They are also not

[2]This example is taken from [6]

interested in a hierarchical lexicon.

Gwadera et al. [11] suggest the use of context trees for segmentation. They efficiently implement the Bellman dynamic programming solution by evaluating the score of segments using a function computed using the context trees, such as the Bayesian Information Criterion. To achieve that, they construct a different prediction tree for each step of the dynamic program, resulting in $O(n)$ trees. Their method is very slow, due to the large number of trees, and does not scale up to the size of data sets we are interested in.

Hierarchical lexicons have been discussed in the context of Natural Language Processing (e.g. [21]). However, in this context, learning is augmented by labeled data and by the use of domain knowledge. As such, state of the art methods developed for that domain cannot be directly adapted to generic unsupervised problems.

Finally, Lo et al. [15, 14] mine program specification from a set of low level program traces. They create a complex procedure that learns a Finite State Machine (FSM) from traces. The states of the FSM correspond to high level methods. Unfortunately, their method has multiple layers of enhancements that are specific to the domain of program specifications. As such, it is difficult to tell whether their approach is applicable to other, even closely related, problems.

3 Hierarchical Segmentation Using Multigrams

We now turn to our approach — building a hierarchical segmentation using multigrams. We begin by describing the multigram model[8, 9]. We then explain how to construct a hierarchical lexicon and use it to train a multigram.

3.1 Multigram

A multigram [8, 9] is a model originating from the language modeling community, designed to estimate the probabilities of sentences, given a lexicon of words. A sentence

is modeled as a concatenation of independently drawn words. Here, we will model event sequences as "sentences" which are concatenation of independently drawn segments ("words"). A multigram Θ defines a distribution over a lexicon of segments S as $\Theta = \{\theta_s : s \in S\} : \theta_s = p(s)$.

The likelihood of a sequence of segments $\vec{s} = s_1, \ldots, s_k$ where each $s_k \in S$ is defined by $\prod_{j=1..k} p(s_j) = \prod_{j=1..k} \theta_{s_j}$. The likelihood of all possible segment sequences $\{\vec{s}\}$ consistent with a sequence of events $\vec{e}$ is computed by summing the probabilities of the different segment sequences.

Given a lexicon and a sequence we can learn the segment probabilities using the a dynamic programming procedure derived by specializing the Forward Backward algorithm for HMMs [2], also known as the Baum-Welch algorithm, to the case of multigrams[3]. We define three variables α, β, and γ over the event sequence $\vec{e} =< e_1, ..., e_n >$:

- α_t is the likelihood of all segment sequences that end at index t given the current θ. That is, α_t is the forward likelihood of a segment ending at t.
- β_t is the likelihood of all segment sequences that begin at index $t + 1$ given θ. That is, β_t is the backward likelihood of a segment beginning at $t + 1$.
- γ_t is the average number of subsequences (words) in a segment sequence ending at t.

We can compute these quantities using a dynamic programming approach:

$$\alpha_t = \sum_{i=1\ldots l} \alpha_{t-i} p(< e_{t-i+1}, \ldots, e_t >) \quad (1)$$

$$\beta_t = \sum_{i=1\ldots l} p(< e_{t+1}, \ldots, e_{t+i} >) \beta_{t+i} \quad (2)$$

$$\gamma_t = 1 + \sum_{i=1\ldots l} \gamma_{t-i} \frac{\alpha_{t-i} p(< e_{t-i+1} \ldots, e_t >)}{\alpha_t} \quad (3)$$

[3]For a detailed description of multigram training see [8]

where l is the length of the longest word in S,

$$p(< e_1, \ldots, e_i >) = \begin{cases} \theta_{<e_1,\ldots,e_i>} & < e_1, \ldots, e_i > \in S \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

and $\alpha_0 = 1$, $\beta_n = 1$, and $\gamma_0 = 0$.

After computing the three variables we can update the estimate of the probability of a segment s that appears in the event sequence $\vec{e}$. Let $i_1, .., i_k$ be the set of indexes where s appears in $\vec{e}$. Then

$$\theta_s = \frac{\sum_{j=1..k} \alpha_{i_j - |s|} p(s) \beta_{i_j}}{\beta_0 \gamma_n} \tag{5}$$

The process of estimating α, β, and γ given θ, and then updating θ using α, β, and γ is executed repeatedly until θ has converged. This procedure finds a local but not necessarily global optimum of the likelihood, [2]. Given a multigram we can find the maximum likelihood segmentation of a sequence of letters, using a specialization of the Viterbi search. This proceeds by starting at the end of the sequence and finding the most likely boundary for the last word, and recursively proceeding backwards. Thus, the most likely boundary defining the word ending at t is given by

$$i_t^{max} = \text{argmax}_{j=1\ldots l} \, \alpha_{t-j} p(< e_{t-j+1}, ..., e_t >) \tag{6}$$

The segmentation $\vec{i}(< e_1, \ldots, e_t >)$ of the sequence $< e_1, \ldots, e_t >$ up to t is given recursively by

$$\vec{i}(< e_1, \ldots, e_t >) = \vec{i}(< e_1, \ldots, e_{t - i_t^{max}} >) \cup \{i_t^{max}\} \tag{7}$$

In terms of the most likely boundary of the word ending at t and the segmentation of the event sequence up to that boundary.

3.2 Hierarchical Lexicon Acquisition and Segmentation

As we will later show in our experiments, the lexicon given to the multigram has a significant impact on the multigram accuracy. A lexicon can be defined by selecting a maximal length l and adding each sequence of length l or less that was observed in the data. However, this approach results in a huge lexicon, a long training time, and possible loss of accuracy, due to local minima.

It is therefore better to filter out some of the observed sequences using some function of the sequence probability such as its mutual information . We suggest here an iterative method for hierarchical lexicon acquisition. We begin with a lexicon containing all low level events (Σ), and the trivial segmentation defined by this lexicon.

During each iteration we add concatenations of existing segments, where pairs of segments to concatenate are chosen using their mutual information. The mutual information of the concatenation xy of segments x and y is estimated using[5]

$$I(x;y) = p(xy) \log \frac{p(xy)}{p(x)p(y)} \tag{8}$$

where the probabilities are empirical probabilities computed over the observed data.

For each concatenation whose mutual information exceeds a predefined threshold, we create a new segment, and add it to the lexicon. Then we train a multigram over the expanded lexicon, produce a new segmentation and a new iteration begins. If no sequences passes the mutual information threshold, the threshold is reduced.

Example 3.1 *Let us look at the sequence "howthenisthisarethegreenfieldsgonewhatdotheyhere" from Moby Dick. At the first iteration the subsequences "ow", "th", "is", "ar", "re", "ld", "on", "wh", "at", and "he" have a mutual information value that meets the threshold, and the resulting most likely segmentation is "h ow th e n is th is ar e th e g re e n f i e ld s g on e wh at d o th e y he re". After the second pass, words such as "how", "this", "the", and "what" are identified, and the best segmentation is*

"how the n is this ar e the gre en fi e ld s g one what do the y here". After the third pass, more words are added to the lexicon, and the resulting segmentation is "how then is this ar e the gre en fi e ld s g one what do they here", and so forth[4].

The process can be stopped when the threshold is less than ϵ and we can then select the lexicon that maximized the likelihood of the data. As this lexicon is created by always joining together sequences of segments from the lexicon, it is hierarchical.

Algorithm 1 The Hierarchical Multigram Learning.

input $\eta = \{\vec{e}_0, ..., \vec{e}_m\}$
input δ — MI threshold
  $i \leftarrow 0$
  $\eta_0 \leftarrow \eta$ *//The first segmentation is the raw text*
  $L_0 = \Sigma_\eta$ *//The initial lexicon contains the letters of the alphabet*
  while $\delta > \epsilon$ **do**
    $i \leftarrow i + 1$
    $L_i \leftarrow L_{i-1}$
    //Add to the lexicon all pairs whose mutual information exceeds δ
    for each consecutive pair of events e_1, e_2 in η_i **do**
      if $MI(e_1 e_2) > \delta$ **then**
        Add $e_1 e_2$ to L_i
      end if
    end for
    //Create the new multigram over the extended lexicon
    Initialize a multigram M using lexicon L_i
    Train M on η_i using the Baum-Welch algorithm (Equations 1-5)
    //Segment the sequence using the new multigram
    for $j = 0$ to m **do**
      Add the most likely segmentation of $\vec{e}_j$ given M to η_{i+1}
    end for
    $\delta \leftarrow \frac{\delta}{2}$
  end while
output η_{i-1}

When creating a new segment to augment the lexicon, we remember the segments that were concatenated to create it. We hence maintain for each segment s an ordered list of segments $L_s = s_1, ..., s_k$ such that $s = s_1 + ... + s_k$, and each s_i is in the lexicon. Each segment can participate in the creation of many longer segments.

Our implementation uses mutual information as the criterion for creating new words,

[4]The correct segmentation is "How then is this? Are the green fields gone? What do they here?"

but other measurements such as the conditional probability or the joint probability can also be used. The power of the method comes from collecting term frequencies from already segmented data. Using the segmented data to compute term frequencies guarantees that the resulting frequency counts are based on a consistent segmentation. Simply counting substrings that match the lexicon would not do this. Term frequencies computed from a segmentation are thus more accurate than frequencies computed from unsegmented data.

Consider, for example, the input string $abcdbcdab$, segmented into $ab|cd|b|cd|ab$. In the raw data, the term bc appears twice, while in the segmented data the term does not appear at all. The term bcd appears twice in the raw data but only once in the segmented data.

As mentioned above, we use a hard assignment of segment boundaries to generate term frequencies. Another alternative is to use a soft assignment of segment boundaries to collect term frequencies from all possible segmentations weighted by their probability as predicted by the multigram.

4 Evaluating Unsupervised Segmentation

In this paper we focus on unsupervised segmentation of discrete event sequences. The typical method for evaluating the quality of a segmentation is to use a labeled dataset, such as a set of sequences that were already segmented by an expert, and see whether the unsupervised algorithm recovers those segments. Alternatively, an expert can observe the resulting segmentation of several algorithms and decide which algorithm produced the best segmentation.

However, in many cases obtaining even a relatively small labeled dataset, or manually analyzing the output of an algorithm, can be very expensive. In such cases, we cannot evaluate the accuracy of a segmentation algorithm with respect to a true segmentation. We therefore suggest a novel measure to evaluate how well a segmentation

algorithm captures the underlying statistical structure of the low level event sequences $\vec{e}$.

A natural candidate for estimating the captured structure is the mutual information $I(\vec{E};\vec{I})$ which is an information theoretic measurement of how much information a segmentation $\vec{i}$ provides about the event sequence $\vec{e}$ [7]. We suggest that a segmentation algorithm whose output contains as much information about the event sequence, and hence the highest mutual information with respect to the event sequence should be preferred among the candidate segmentations. We note that this mutual information between the event sequence and its segmentation should not be confused with the mutual information between adjacent segments that was used in constructing the hierarchical lexicon.

The mutual information can be written as

$$I(\vec{E};\vec{I}) = H(\vec{E}) - H(\vec{E}|\vec{I})$$

where $H(\vec{E})$ is the entropy of the event sequence and $H(\vec{E}|\vec{I})$ is the entropy of the event sequence given the segmentation [7]. Since only the second term depends on the segmentation, choosing the segmentation algorithm with the highest mutual information is equivalent to choosing the one with the lowest conditional entropy. The per-event conditional entropy can be estimated as:

$$\hat{H}(\vec{E}|\vec{I}) = -\frac{1}{|\eta|}\sum_{\vec{e}\in\eta}\frac{\log p(\vec{e}|\vec{i}(e))}{|\vec{e}|} \tag{9}$$

$$p(\vec{e}|\vec{i}) = \prod_{(i_j,i_{j+1})\in\vec{i}(e)}\frac{p(< e_{i_j},...,e_{i_{j+1}} >)}{\sum_{s\in S,|s|=i_{j+1}-i_j}p(s)} \tag{10}$$

where $p(s)$ is the estimated probability of a segment s in the lexicon S.

In order to estimate the conditional entropy of a segmentation, we divide the set of event sequences into a train and test set. We train a multigram over the train set

to learn the lexicon probabilities, initializing the lexicon to the words (segments) that were observed in the segmentation. Then, we estimate the conditional entropy of the given segmentation on the test set. It is important that the conditional entropy be estimated on a test set distinct from the lexicon training set in order to measure how well a segmentation captures the underlying structure of the event sequences, rather than measuring how close a segmentation comes to memorizing the event sequences.

5 Process Recovery

In this section we present a new and challenging problem that can be tackled using a hierarchical segmentation of discrete events — the problem of software process recovery from software traces. We begin with two motivating examples that illustrate the importance of this problem. We then discuss the general problem of process recovery and why hierarchical segmentation is an appropriate tool for this task.

5.1 Motivating Examples

We begin by an overview of two motivating examples of real world domains where software traces exist, but are currently analyzed only using statistics over the low level events. We explain for both domains why a better understanding of the high level process is needed.

5.1.1 Microsoft Exchange Server

The Microsoft Exchange Server is a messaging product, supporting e-mail, calendaring, contacts and tasks. Exchange servers can be accessed by many different clients. An instrumented Exchange server keeps a log of all the requests issued by the clients. These logs contain information such as user id, client application id and requested operation. As such, we can create sequences of operations corresponding to a single user interaction with a specific client application.

These sequences contain only low level operations. A server administrator can use these low level operations to compute statistics, such as the frequency of an operation, or the average number of operations per session. Afterwards, these statistics can be used, for example, to execute simulation interactions with the server to test the server performance in extreme conditions.

However, it was noticed by Exchange administrators that these simulations poorly imitate true user sessions. This is because in a true user session low level operations occur within the context of the high level process of the client application. Therefore, understanding the high level behavior of a client can improve these simulations, and allow Exchange to better optimize clients requests.

5.1.2 User Interface Driven Applications

In many cases an instrumentation of a user interface application, such as Microsoft Word or Excel, captures the events the user initiates, such as clicks on toolbars or menus. These low level events can be later collected and sent to the application producer through tools such as Microsoft Customer Experience Improvement Program (CEIP)[5]. For example, when a Microsoft Office product experiences a failure, a user can allow a report to be sent back to Microsoft for analysis.

Using these traces, software engineers can reproduce problems locally, thus making it possible to better understand and fix the problem. However, Microsoft, like other producers of widely used software, receive many such reports. Each report then needs to be transferred to a specific group that handles a specific failure. This classification process is very difficult when the only available data is statistics over the low level operations.

Inferring the high level process that generated a trace can help both the classification process of reports, as well as the reproduction of problems in a controlled environment.

[5]http://www.microsoft.com/products/ceip/EN-US/default.mspx

5.2 Software Traces

Software applications have become an essential part of our modern society, yet many deployed software applications perform below their pre-deployment expected behavior. While responsible software development companies do their best to understand post-deployment performance prior to deployment, the problem of predicting post-deployment usage is challenging. For instance, applications that are event driven such as user interfaces and web services that are activated by unknown clients allow immense flexibility in the way that the software is used. This flexibility makes it challenging to foresee likely use scenarios, let alone all possible usage scenarios. An understanding of this behavior can usually only be obtained after their deployment. In this paper we focus on understanding the behavior of deployed software applications. Such an understanding can then be used by software developers to fix failures and improve the performance of the deployed application.

The software industry is well familiar with the need to monitor an application after its deployment. Therefore, techniques have been developed to gather execution information that can later be analyzed. It is common practice for programmers to add execution traces to their code. These traces are created by recording low level events such as procedure activations, and can provide valuable data for debugging and for performance evaluation. Traces are logged and can later be recovered and sent to the producer of the application for analysis. The process of adding such traces is known as an instrumentation of the code (e.g. [22]).

5.3 From Low-Level Traces to High-Level Behavior

Our goal is to use traces of low level events to provide insights as to the application behavior. However, using low level events to understand the high level process that the user or the web client are performing is in many cases difficult. Traces can be long and may contain much irrelevant information, making their understanding challenging.

The traditional approaches to software optimization and bug fixing require a model of the program such as a Control Flow Graph (CFG, see e.g. [1]) that is generated by a compiler. However, in the cases that motivate us, such as when unknown clients execute requests over a web service, we cannot assume the existence of a CFG.

Most programming languages have an inherent hierarchical structure. Specifically, procedural languages have a hierarchical structure of procedures where high level procedures call lower level procedures. We therefore model the behavior of a software application in terms of the hierarchical procedural structure of a program. In the domains that interest us, this hierarchical structure is unknown, and we only have event traces for some of the procedures. We attempt to recover the hierarchical process that generated the low level events. The value of a hierarchical flow of procedures has already been acknowledged [17] using a predefined CFG. Our problem is more difficult, since we need to uncover both the hierarchical structure and the flow of procedures. This learning task is unsupervised, in that we do not have any set of labeled data to help us learn the structure.

5.4 Hierarchical Segmentation of Software Traces

We suggest that process recovery can be achieved through a hierarchical segmentation of the low level traces. Such a segmentation can capture the true execution of software procedures that resulted in the sequence of low level events. A segmentation that properly identifies various procedures in the program execution allows the user to browse the high level procedures rather than the long low level event sequences. The hierarchical segmentation allows us to drill down into interesting procedures (segments) while observing the rest of the sequence at a high level.

Given a sequence of low level software traces, we attempt to reconstruct the program execution that generated these traces. We consider program execution in terms of the hierarchical execution of procedures.

A *procedure* is a cohesive unit of a software program. Each procedure is a sequence of commands, which can be either programming language commands, or procedures. Some of the commands may evaluate conditions or cause loops and thus a procedure may have several possible executions. A *program* is a finite DAG of procedures, where a procedure p is the parent of a procedure p' if p directly calls p' in one of its possible executions. The leaves of the DAG are procedures that call only uninstrumented procedures. All leaves must be instrumented, i.e. log some event, but other nodes in the DAG can also be instrumented. The sources of the DAG correspond to the various program *tasks*.

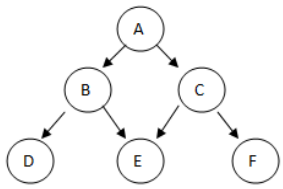
The definition of a program as a DAG does not elegantly capture some procedural patterns such as recursion. Recursion can be modeled by attaching a level to the procedure name. For example, instead of saying that a procedure f recursively calls itself we can say that a procedure f_0 calls a procedure f_1. Since we are interested only in a finite set of executions, this definition can capture all the observed recursions. In the domains that interest us, such as UI driven applications, it is unclear how many recursions are used. Nevertheless, other models may avoid this redundancy.

For the Exchange server example the program models a client application that connects to the service. The leaves of the program DAG in this case are the service requests and these are the only instrumented nodes. For the user interface example a program models the entire application, where various sources correspond to various UI tasks such as 'Create Table' or 'Find and Replace'. In this scenario many nodes may be instrumented.

A *program execution* is a recursive traversal through the DAG, starting at a source and terminating at the leaves. Through this traversal a procedure can call its sub-procedures any number of times. A traversal is generated by executing a task (a source procedure) with a set of parameters. Given the parameters the procedure calls a sequence of its sub-procedures with specific parameters and so forth. We represent a

```
int A(int x)
{
    if (x < 3)
        return B(x - 1);
    else
        return B(x) + C(x + 1);
}
int B(int x)
{
    int y = D(x);
    return E(x);
}
int C(int x)
{
    if (x == 5)
        return E(x) + F(x);
    else
        return F(x) + E(x);
}
```

```
int D(int x)
{
    if (x < 0)
        return x + 1;
    else
        return x - 2;
}
int E(int x)
{
    return x * x;
}
int F(int x)
{
    return x / 2;
}
```



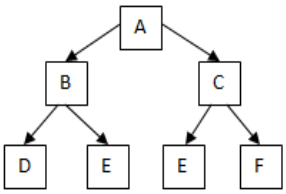


Figure 1: A program, its corresponding program DAG, and an execution tree for $x = 5$ where children are ordered left to right.

traversal P_E by an ordered tree — a tree with an ordering over children. The nodes of the tree are program procedures. A procedure p' can be a child of a procedure p in the tree only if p' is a child of p in the DAG. The child procedures are ordered in the tree by order of execution. The tree leaves are leaves of the DAG.

In the Exchange server scenario a possible program execution is searching and retrieving a specific e-mail. The parameters in this case could be the user id, a specific string that appears in the e-mail and a specific folder to search.

An *execution trace* is a sequence of low level trace events generated by an execution. Given an execution tree P_E we generate the execution trace by walking the ordered tree, emitting a trace event each time an instrumented procedure was entered. As all the leaves are instrumented, the execution trace is a super-sequence of the ordered set of leaves. An execution trace can be considered to be a projection of the execution onto the space of event sequences. For the example in Figure 1 a possible execution trace can be D, E, E, F.

Clearly, other models of instrumentation can exist, such as adding more traces within an instrumented procedure, or using predicate traces. In this paper, however, we limit ourselves to the simple case described above.

Given a trace sequence, generated by a program P and a program execution P_E we define two goals:

- Find a program P' that is as close as possible to P.
- Find a program execution P'_E that is as close as possible to P_E, such that P'_E emits the same execution trace as P_E.

In practice, many programs and many executions can generate identical sequences of low level events. We hence need a method to discriminate between various programs given a set of event sequences. As we can think of a program execution as a generative process, we argue that the likelihood of a set of event sequences given a program and a set of executions is an appropriate measurement for its quality.

Our definition of a program can be viewed as a limited Control Flaw Graph (CFG) — a directed graph that captures call dependencies between procedures (e.g. [1]). Most analysis tools take advantage of the existence of a CFG that was generated by the compiler.

In the web service scenario the required CFG is the CFG of the client application, which is unknown to us. In the user driven scenario we are modeling the user behavior as a part of the program. Therefore, not only that a CFG is unavailable in both scenarios — we do not even know the set of procedures of the program.

Process recovery can be achieved by various methods (e.g. Finite State Automaton [15]). In this paper we suggest modeling process recovery through a hierarchical segmentation, where the hierarchical lexicon is a manifestation of a program. The sub-segments of a segment correspond to the sub-procedures of a procedure. A segmentation of a sequence corresponds to a task execution of a program.

6 Empirical Evaluations

In this section we provide an empirical comparison of three unsupervised algorithms for segmentation — our hierarchical multigram approach, Sequitur [16], and Voting Experts [6]. Our results demonstrate the superiority of the segmentation that we produce both on standard benchmarks and over a set of real world event sequences from an Exchange server.

All our experiments were executed on a standard dual-core PC with 2GB RAM. Algorithms, including the Voting Experts and Sequitur approaches, were implemented in $C\#$. The Sequitur code was translated from a Java implementation available on `http://sequitur.info/java/`.

Table 1: Properties of the textual datasets

Dataset	$\|\Sigma\|$	#characters	#words	Longest word
MobyDick	26	155,976	37,054	17
Sinica Corpus	4635	1,839,868	1,120,639	16

6.1 Segmenting Text

We begin by a traditional evaluation of the quality of the segmentation produced by the algorithms on a pre-segmented dataset, as done by most researchers. The resulting scores may be somewhat misleading, as a supervised algorithm that is trained on the segmented dataset can produce much better results. Still, such an evaluation is useful in providing an understanding of the relative performance of the unsupervised algorithms.

We evaluate the three algorithms over two tasks — the identification of word boundaries in English and in Chinese texts [4, 6, 20]. For the English text, we took the first 10 chapters of "Moby Dick"[6] and transformed the text into unlabeled data by removing all the characters that are not letters (spaces, punctuation marks, etc.) and transformed all letters to lower case. For the Chinese text we used the labeled Academia Sinica corpus[7].

All algorithms have a tunable parameter for the decision on segment boundaries. Voting Expert builds a tree of symbol sequences, and the depth of the tree affects the segment boundary identification. Sequitur identifies a set of rules, and in order to generate a segmentation from these rules, we place word boundaries after expanding rules up to a finite depth, after which rules are expanded into words. Our lexicon creation procedure is affected by the threshold on the required mutual information for concatenating chunks.

A typical measurement for the success of a segmentation is a precision-recall trade-off. When identifying segment (or word) boundaries we can either correctly identify a boundary (TP — true positive), fail to identify a segment boundary (FN — false nega-

[6] http://www.physics.rockefeller.edu/~siggia/Projects/mobydick_novel_files/Chap1-10.txt

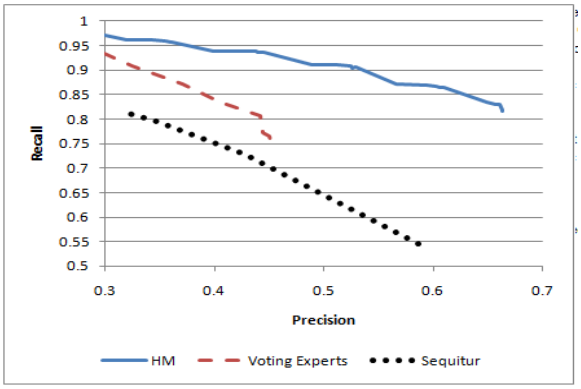
[7] www.sighan.org/bakeoff2003/bakeoff_instr.html

tive), or predict a boundary within a segment (FP — false positive). Precision is defined as the portion of true positives among the guessed boundaries — $\frac{\#TP}{\#TP+\#FP}$, while recall is the portion of true positives among all the segment boundaries — $\frac{\#TP}{\#TP+\#FN}$. There is a clear tradeoff between precision and recall. At one extreme we can predict no boundary, making no mistake and getting a precision of 1, but identifying no boundaries and getting a recall of 0. At the other extreme we can identify a boundary after each symbol, getting a recall of 1 but a relatively low precision.

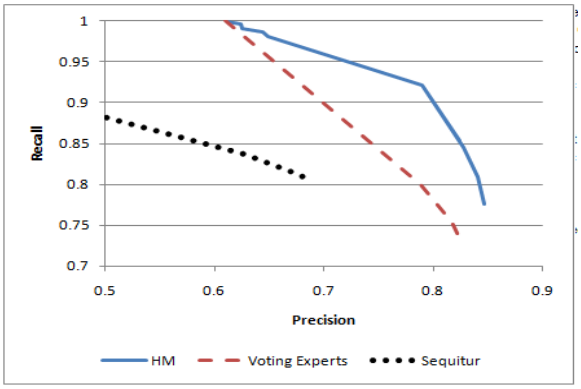
Figure 2 shows the precision recall curves resulting from tuning the parameters. As we can see, our method dominates the two other on both datasets. These tasks have limited hierarchical structure. For example, we can identify chunks of English words that occur repeatedly, such as "th", "ing", and "tion". Identifying these chunks early in the process can help us to construct the words afterwards. However, the hierarchical structure is limited, and we expect that in tasks that have a deeper hierarchical structure, the advantages of our method will be even more pronounced.

As we are suggesting here a method for evaluating the segmentation over an unlabeled dataset, it is informative to compute the conditional entropies of the three algorithms for the text segmentation problem. Table 6.1 shows the conditional entropies of our approach, and the Sequitur and Voting Experts algorithms on the English and Chinese word segmentation tasks. For each algorithm we picked the segmentation that provided the best F score, defined as $F = \frac{2 \cdot precision \cdot recall}{precision + recall}$. We then trained a multigram using the lexicon derived by the segmentation on a train set (80% of the data) and computed the conditional entropy of the given segmentation over the test set (20% of the data). In addition, we also evaluated the conditional entropy of the true segmentation using the same procedure.

Looking at Table 6.1, we see that the true segmentation gives the most information about the text. This provides evidence that the structure captured by words is highly useful. The true segmentation is followed by our approach, Sequitur, and then Voting



(a) Moby Dick (English text)



(b) 1998-01-qiefen (Chinese text)

Figure 2: Precision-Recall curves on the two word segmentation problems.

Experts for the English text. On the Chinese text Voting Expert outperformed the Sequitur approach. This corresponds well with the precision-recall curves in Figure 2. The ability of Sequitur to achieve higher precision at the expanse of lower recall, resulted in a higher conditional entropy.

Shannon estimated that English had an entropy of about 2.3 bits per letter when only effects spanning 8 or fewer letters are considered, and on the order of a bit per letter when longer range effects spanning up to 100 letters are considered [19]. We estimate the conditional entropy given word segmentations but not taking into account long range effects. Thus, we would expect the conditional entropy to be lower than the

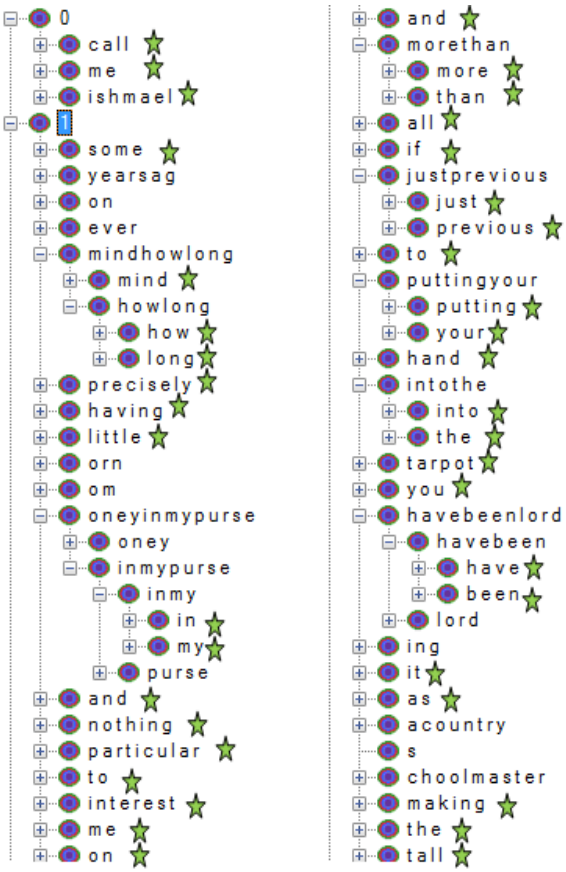


Figure 3: Hierarchical segmentation of the Moby Dick text. Showing the first two sentences (left side) and another random sentence (right side). Correctly identified words are annotated by a star.

entropy taking into account only short range effects, though perhaps not as low as the entropy given longer range effects. Our results using the true segmentations as well as our results using the hieratical multigram are consistent with this expectation.

6.2 Segmenting Exchange Sequences

We now move to evaluating the performance of the three algorithms over event sequences gathered from a Microsoft Exchange server. We obtained logs from the interactions of the server with 5 different clients. As we can see in Table 3 the properties

Algorithm	English		Chinese	
	CE	Time	CE	Time
True	1.56	N/A	5.73	N/A
HM	2.01	633 sec.	6.01	2518 sec.
Voting Experts	3.38	11 sec.	6.87	196 sec.
Sequitur	3.00	$<$ 1 sec.	9.25	3 sec.

Table 2: Conditional entropy and runtime of the various segmentations of the three algorithms and the true segmentation on the two text segmentation tasks.

of the clients different significantly in terms of the functionality that they require from the server (Σ) and the length of a single interaction.

We segmented all the domains using Voting Experts and the hierarchical multigram algorithms, varying the tunable parameters. The Sequitur algorithm failed completely on these datasets. We speculate that this is because context-free grammars are inappropriate for capturing segmentations that contain even relatively limited loops in the client algorithm. We report in Table 4 the conditional entropy of the best segmentation that was achieved by each algorithm.

Client	$\lvert\Sigma\rvert$	Sessions	Avg. Session Length
AS	22	24,630	19.5
ASOOF	21	17,833	17
OWA	56	9,689	14.7
Airsync	54	27,115	25.9

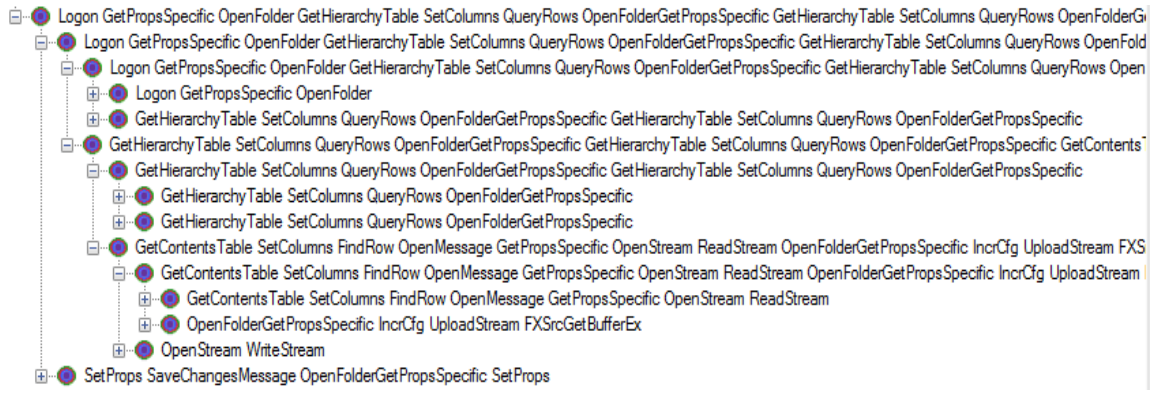
Table 3: Properties of the Exchange client applications.

Client	HM	Voting Experts
AS	0.11	0.66
ASOOF	0.069	0.067
OWA	0.42	1.22
Airsync	0.48	1.36

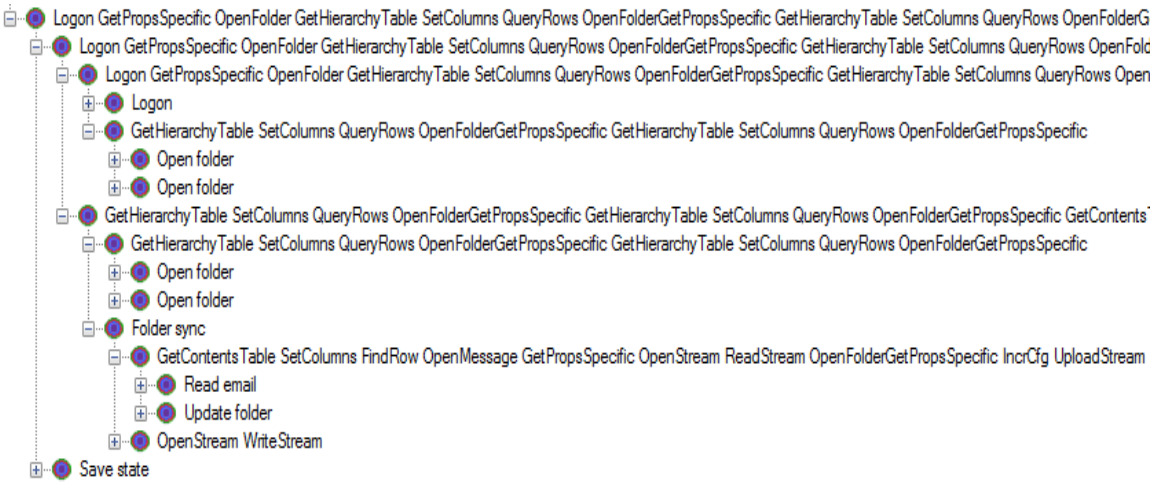
Table 4: Comparing the conditional entropy of the hierarchical multigram and the Voting Experts over the various Exchange client applications.

The hierarchical multigram generated segmentations with considerably better con-

ditional entropy in all domains, except for the ASOOF client, which is by far the simplest and most structured domain, as we can see from the very low conditional entropy of both segmentations.



(a) Raw events



(b) Annotated

Figure 4: A session from the Airsync client interactions with the Exchange server, before and after annotation by a domain expert.

7 Discussion

In this paper we presented a new method for the unsupervised hierarchical segmentation of sequences of discrete symbols, that we call Hierarchical Multigram (HM). Here, we discuss the major differences between HM and the two other segmentation algorithms that we discuss in this paper, Voting Experts and Sequitur. Afterwards, we discuss the relationships of our algorithm and other approaches.

Sequitur is very different from the two other approaches; First, it was not created to build a segmentation, but rather to learn a hierarchical structure of the sequence. Second, it does not use any statistical or probabilistic information about the sequence. For example, when two possible CFGs are available, Sequitur cannot prefer the CFG that has a higher likelihood of generating the data. Sequitur doesn't even produce the shortest description of the data, which would have been tightly related to the Shannon Entropy. Finally, for the software traces dataset, the Sequitur approach was unable to construct a reasonable CFG.

Like HM, the Voting Expert approach is a probabilistic approach. It tries to optimize measures such as the entropy of sequences. The approach was originally designed for the segmentation of sequences of discrete events. Indeed, for this reason Voting Experts indeed produced reasonable results on the software traces datasets. Voting Experts , unlike Sequitur and HM, was not designed to handle hierarchical structures. We suspect that explicitly handling the hierarchical structure of the data is one of the key reasons why HM outperforms Voting Experts.

In terms of complexity, HM is substantially slower than the two other algorithms. Sequitur makes a single pass over the sequence, but for each new symbol it must check the entire sequence up to the current position to check for reoccurrences. A simple implementation is quadratic in the length of the sequence, but using smart data structures the runtime is linear. Voting Experts, on the other hand, makes a constant number of passes over the sequence; The first pass collects statistics that will be used by the

experts, such as the number of appearances of any subsequence of length k. The second pass computes the votes for segment boundaries. The third pass uses the votes to segment the sequence. Hence, Voting Experts is linear in the size of the input.

Our HM approach, on the other hand, requires many passes over the input sequence. Each iteration of the Baum-Welch algorithm is linear in the size of the sequence, but Baum-Welch must make a number of iterations before it converges. After converging, we compute a segmentation, which is also linear in the size of the sequence. However, for constructing the complete hierarchy, we repeat the above process for each level of the hierarchy. Indeed, as our results show, HM requires in practice much more time than the two other approaches.

That being said, the segmentation of software traces task in which we are interested is an offline process and as such, it is reasonable to spend hours parsing sequences. The results that we provide for the Exchange datasets were all computed in less than two hours on an average PC.

In the context of time series, some probabilistic segmentation models were suggested, mostly built upon Hidden Markov Models (HMMs) [12, 13]. HMMs assume that events are generated by a set of hidden variables. In our case, the hidden variables can reflect the hierarchical structure of a program — each variable modeling the state of the procedure at a single procedural level. Indeed, multigrams can be viewed as a special case of HMMs, where each state emits a deterministic sequence of observations, and state transitions are independent. The probability training method we describe above are an adaptation of the forward-backward method for training HMMs.

In the paper that originally introduced the multigram model, Deligne and Bimbot suggested using a lexicon consisting of all combinations of characters of bounded length appearing in the text when a lexicon is otherwise unavailable[9]. As the training of the multigram depends on the size of the lexicon, this approach results in very long training time. We tried training a multigram over such a naive lexicon, and observed

that segmentations with comparable quality could only be generated by lexicons which were from 20 times to 1000 times larger. For example, in the Airsync domain the naive lexicon had $91,248$ words, while the hierarchical lexicon had only 81 words. In the relatively simple ASOOF domain the naive lexicon had $1,085$ words while the hierarchical lexicon had 65 words. The performance that the Airsync hierarchical lexicon achieved in less than 45 minutes was not achieved by the naive lexicon even after 48 hours of training.

8 Conclusions

In this paper we proposed a hierarchical probabilistic segmentation method based on a multigram. The performance of using multigrams for segmentation as measured both in terms of quality and computational effort is highly dependent on the input lexicon that is provided. We propose a method for iteratively building a hierarchical lexicon. Our method computes the criteria for joining segments based on the current segmentation of the data. As such, the generated term frequencies are more accurate. In addition, the lexicon generated with this method is smaller than previous approaches. This reduces the computational cost of using a multigram segmentation and seems to yield more meaningful lexicons.

We experimented with a text segmentation problem, showing our method to produce superior accuracy, and over real data sets gathered from an Exchange server, showing our method to provide models with higher conditional entropy than previous algorithms.

Our hierarchical segmentation results in a tree of segments. Potentially fruitful directions for future research include the investigation of alternative methods for identifying the best cut through the tree for segmentingthe data and developing measures for the quality of a hierarchical segmentation. We also intend to apply our techniques to the segmentation of user interactions with applications, allowing us to understand

the behavior of users when accomplishing complicated tasks.

References

[1] T. J. Ball and S. Horwitz. Constructing control flow from control dependence. Technical Report CS-TR-1992-1091, 1992.

[2] L. E. Baum, , T. Petrie, G. Soules, and N. Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *Ann. Math. Statist.*, 41(1):164–171, 1970.

[3] R. Bellman. On the approximation of curves by line segments using dynamic programming. *Communications of the ACM*, 4(6):284, 1961.

[4] H. J. Bussemaker, H. Li, and E. D. Siggia. Regulatory element detection using a probabilistic segmentation model. In *ISMB*, pages 67–74, 2000.

[5] K. W. Church and P. Hanks. Word association norms, mutual information, and lexicography. *Computational Linguistics*, 16(1):22–29, 1990.

[6] P. Cohen, N. Adams, and B. Heeringa. Voting experts: An unsupervised algorithm for segmenting sequences. *Intell. Data Anal.*, 11(6):607–625, 2007.

[7] T. M. Cover and J. A. Thomas. *Elements of information theory*. John Wiley and Sons, Inc., 1991.

[8] S. Deligne and F. Bimbot. Language modeling by variable length sequences: Theoretical formulation and evaluation of multigrams. In *Proc. ICASSP '95*, pages 169–172, 1995.

[9] S. Deligne and F. Bimbot. Inference of variable-length linguistic and acoustic units by multigrams. *Speech Commun.*, 23(3):223–241, 1997.

[10] A. Gionis and H. Mannila. Tutorial on segmentation algorithms for time series and sequence data, 2005. SDM.

[11] R. Gwadera, A. Gionis, and H. Mannila. Optimal segmentation using tree models. In *ICDM*, pages 244–253, 2006.

[12] A. Kehagias. A hidden markov model segmentation procedure for hydrological and environmental time series. *STOCHASTIC ENVIRONMENTAL RESEARCH AND RISK ASSESSMENT*, 18(2):117–130, 2004.

[13] Y. Kim and A. Conkie. Automatic segmentation combining an hmm-based approach and spectral boundary correction. In *ICSLP*, pages 145–148, 2002.

[14] D. Lo and S. cheng Khoo. Smartic: Toward building an accurate, robust and scalable specification miner. In *In SIGSOFT FSE*, pages 265–275, 2006.

[15] D. Lo, S. C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. In *KDD ́07*, pages 460–469, 2007.

[16] C. Nevill-Manning, , and I. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[17] E. Perelman, T. M. Chilimbi, and B. Calder. Variational path profiling. In *IEEE PACT*, pages 7–16, 2005.

[18] G. Shani, C. Meek, and A. Gunawardana. Hierarchical probabilistic segmentation of discrete events. In *ICDM '09: Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, pages 974–979, 2009.

[19] C. E. Shannon. Prediction and entropy of printed english. *The Bell System Technical Journal*, pages 50–64, January 1951.

[20] R. Sproat and T. Emerson. The first international chinese word segmentation bakeoff. In *The Second SIGHAN Workshop on Chinese Language Processing*, 2003.

[21] E.-J. van der Linden. Incremental processing and the hierarchical lexicon. *Comput. Linguist.*, 18(2):219–238, 1992.

[22] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *ICML '06*, pages 1105–1112, 2006.