

Topic Segmentation in Spoken Dialogue

James Ballantine
james at ballantine.com.au

Supervisor: Steve Cassidy

Department of Computing, Division of ICS, Macquarie University

Thesis submitted in partial fulfilment of the requirements for the
degree of Bachelor of Arts (Honours).

June 2004

Abstract

Topic segmentation is the division of linguistic data into semantically coherent blocks, based on the topics they cover. Traditional topic segmentation techniques focus on written text; spoken dialogue presents different challenges such as lack of paragraph markings, different lexical topic-change cues, and lack of organised structure.

This thesis first describes the implementation and evaluation of existing techniques for the special case of transcripts of spoken dialogue. Secondly, it describes the conduct and evaluation of experiments in improving topic segmentation results using spoken dialogue domain-specific cues and heuristic topic-change detection.

Contents

1	Introduction	4
2	Review of the Literature	6
2.1	Topic segmentation	6
2.1.1	<i>TextTiling</i> and Marti Hearst's work	6
2.1.2	Research extending and improving, and using <i>TextTiling</i>	9
2.1.3	Other approaches	12
2.2	Spoken dialogue and structure	18
2.2.1	Theories of structure	18
3	Software Implementation	21
3.1	Outline of functionality	21
3.2	Architecture	21
3.3	Implementation details	23
3.3.1	Ancillary modules	23
3.3.2	Parsing the data	24
3.3.3	Output	26
3.3.4	Visualisation	26
3.4	Summary of usage	28
3.5	Example session	29
3.5.1	Calculating similarities and detecting breaks	29
3.5.2	Creating annotated transcripts in HTML	29
3.5.3	Plotting graphs	29
3.6	Bugs and inefficiencies	30
4	Experiments	32
4.1	Outline	32
4.2	Methodology	33
4.2.1	Hand-annotated data collection	33
4.2.2	System parameter experiments	34
5	Results and Evaluation	38
5.1	Calibration data	38
5.2	Evaluations against human markup	39

5.2.1	Imprecision	40
5.2.2	Misses	41
5.2.3	Annotator normalisation	41
5.2.4	Sparseness agreement	42
5.2.5	Annotator differences	42
5.3	Parameter tuning	48
6	Conclusions	50
6.1	Observations	50
6.1.1	Implications of false positives	50
6.1.2	Inter-annotator agreement	51
6.2	Further work	51
A	Code listings	54
A.1	segmentation.py	54
A.2	micaseparse.py	56
A.3	cosinemeasure.py	64
A.4	peakpick.py	65
A.5	twoplot.R	66
A.6	newmicase-html-pseudosentces-breaks.xsl	67
A.7	makegraph.sh	68
A.8	buildgraph.sh	69
A.9	speakers.css	69
A.10	speakers-print.css	70

List of Figures

2.1	Three topic segments within a news feed	16
2.2	Ambiguous cue-phrases	18
5.1	Test word set for system evaluation	39
5.2	Full system graph output on the test word set	39
5.3	Honours advising session, $ps = 20$ $bs = 6$	40
5.4	Location of topic change 1 from figure 5.3, showing distance between detected and real topic boundaries ('break' tag is hand-marked, 'autobreak' is system-determined)	41
5.5	Context of missed topic change (manual topic change 5) in figure 5.3	42
5.6	American Culture advising session, $ps = 20$ $bs = 6$	43
5.7	CMU dialogue, $ps = 20$ $bs = 6$	43
5.8	CMU dialogue 2, $ps = 20$ $bs = 6$	44
5.9	ICSI dialogue, $ps = 20$ $bs = 6$	44
5.10	ICSI dialogue 2, $ps = 20$ $bs = 6$	45
5.11	LDC dialogue, $ps = 20$ $bs = 6$	45
5.12	LDC dialogue 2, $ps = 20$ $bs = 6$	46
5.13	NIST dialogue, $ps = 20$ $bs = 6$	46
5.14	NIST dialogue 2, $ps = 20$ $bs = 6$	47
5.15	A snippet from figure 5.13's source dialogue, showing unexpected topic-boundary marking style	47
5.16	Honours Advising dialogue: $ps = 10$ $bs = 12$	48
5.17	Honours Advising dialogue: $ps = 40$ $bs = 3$	49

Chapter 1

Introduction

Topic segmentation describes the automatic detection of meaningful blocks within a text or recording, each containing a distinct topic or subject. It has broad significance today, as we address the challenge of better organisation of overwhelmingly large amounts of data: Recording and storing large volumes of data has become much easier in recent years, but the problem of organising and dividing it sensibly still exists. Techniques such as topic segmentation can help to reduce collections of data into manageable chunks, divided into units in a meaningful way. Combined with a method for labeling each topic, these techniques can provide an automatic overview and index of documents of arbitrary size and complexity, removing some of the requirement to catalog and organise data manually.

While today data is most often stored as text, as storage capacity grows our ability to record and archive audio data increases. Thus techniques to order and manage data are becoming more applicable to recorded data—meeting room recordings, telephone conversation archives, broadcast news, and other recorded speech. Additionally, audio data is harder to navigate than written text—consider the difficulty experienced when attempting to find the start of a chapter or scene in a book-on-tape or other spoken recording. Audio recordings cannot be ‘scanned’ visually to locate an area of interest in the same way that even unformatted text can. Thus the ability to navigate into a large speech-based recording to find the area of interest would greatly improve access to this kind of data.

Natural spoken dialogue can occur with different purposes in mind—a dialogue can take place between a manager and an employee to communicate instructions for a particular task; it can also take place between friends, synchronising knowledge of interesting experiences and gossip. Regardless of the purpose of a dialogue, however, if meaning is imparted (as we would normally hope) then it always covers one or more *topics*. In the simplest case, a dialogue addresses a single topic, and this topic can be identified by a party following the conversation. For example, a conversation between a customer and a shopkeeper about the availability of a product may cover a single topic. A longer

conversation, however, will typically cover more than one topic in succession—it may carry a main topic to which it returns repeatedly, digressing to other topics at times; or it may follow a chain of related topics, moving away from the original subject completely.

Chapter 2

Review of the Literature

As background research for this work, a comprehensive review of the literature concerning textual topic segmentation, and concerning the nature of topics and topic change in spoken dialogue, was performed. Given its prominence within topic segmentation research, the *TextTiling* algorithm [6] is examined first, followed by a discussion of techniques and research derived from that algorithm. The section then reviews alternative approaches to topic segmentation.

The second half of the review discusses theories of spoken dialogue, with special focus on theories of topic, question-under-discussion, and topic change.

2.1 Topic segmentation

Topic segmentation is the division of language data into segments or chunks, based upon the topic or subject discussed. For example, a news broadcast which covers four different stories or articles clearly divides naturally into four different topics. Less clearly, a multi-page magazine article, while ostensibly covering a single broad topic, will usually cover a series of subtopics as it examines different aspects of its subject matter and explores the subject area.

2.1.1 *TextTiling* and Marti Hearst's work

The chief research into textual topic segmentation was performed by Marti Hearst in 1994, in the paper *Multi-Paragraph Topic Segmentation of Expository Text* [6]. In this paper, she described the *TextTiling* algorithm for locating topic changes within magazine articles based on a statistical word-frequency approach. The paper gave promising results for *TextTiling* in this domain, showing a high level of agreement with human annotators.

Hearst hypothesises that the higher frequency of certain words in a particular area of a document is an indicator of the location of a topic. When a word or group of words shows a change in local frequency at some location within a document, Hearst theorises that this may indicate that the current topic has

changed. This is based on the assumption that slightly different vocabulary sets are required for discussing different topics. For example, a document discussing the possibility of life on other planets (the example document Hearst discusses in [6]) may mention the word ‘water’ several times when discussing possible criteria for life, but few times or not at all in other sections, such as in areas discussing meteoric evidence of life.

However, there is no utility in comparing the frequency of non-topic words: Closed-class words and other ‘stop-words’ are unlikely to indicate topic in any scenario. However, some words which may be good topic indicators in one situation are poor indicators in another due to pervasive use (for example, the word ‘Mozart’ in an article on Mozart’s music).

In order to locate these few topic-indicative words and prioritise them, a cosine similarity measure is used. A cosine measure compares two vectors of the same length, returning a single value between zero and one which represents the ‘similarity’ of the two vectors. Visualising the vectors as n -dimensional vectors in n -space, the cosine measure gives the cosine of the angle between the two vectors. Thus, if the vectors have the same angle, the angle between them will be zero. This will result in a cosine measure of 1, since $\cos(0) = 1$. In contrast, vectors as dissimilar as possible will at maximum have an angular difference of $\frac{\pi}{2}$, and thus a cosine measure of 0.

This cosine similarity measure is used to compare groups of word frequency across the document. Each vector has a value for every word appearing in the document, with the value at that position containing the frequency of the word in the current section of the document. Thus, small segments of the document are compared based on their word frequency, and changes in frequency result in a low cosine measure score. This has the advantage of automatically removing words which are not good topic indicators: If a word is excessively frequent, it will have values of, for example, 14 and 23 in the two vectors under comparison. However, an uncommon word indicating a topic change may have values of 1 and 6, a proportionally much greater difference which will result in a much lower similarity score.

$$sim(b_1, b_2) = \frac{\sum_t w_{t,b_1} w_{t,b_2}}{\sqrt{\sum_t w_{t,b_1}^2 \sum_{t=1}^n w_{t,b_2}^2}} \quad (2.1)$$

The equation giving the cosine measure similarity score to two same-length vectors is shown in equation 2.1. The two vectors in question are represented by b_1 and b_2 ; w_t is the value at the current position in each vector (w standing for *weight*, in reference to the fact that the cosine similarity measure compares proportionally rather than by magnitude).

In order to compare different sections of a document using this cosine similarity measure, Hearst divides the document into *pseudosentences*, each containing the same number of words. The pseudosentences are grouped into logical units called *blocks*, comprising several pseudosentences; blocks are not fixed sets of pseudosentences, but rather a rolling window containing an exact number of pseudosentences.

To perform the similarity measure across the entire document, Hearst’s system serially compares two overlapping blocks, moving across the document one pseudosentence at a time. The blocks overlap by one pseudosentence, guaranteeing at least some similarity. At each step, a cosine measure is performed to determine the similarity of the vectors representing the frequencies of the words in the two blocks. Where a word does not appear in either block (one from elsewhere in the document), both the frequency counts for that word will both be zero. This does not affect the cosine measure.

Hearst set the size of each pseudosentence to be 20 words (not including stop words, which are removed from the comparison), a number intended approximately to reflect the length of a typical sentence in the target documents. She then determined experimentally that the optimal block size is 6 pseudosentences. This, Hearst estimates, corresponds to the typical size of a paragraph in the target documents.

The list of similarity values for the entire document falls into a range between zero and one. When plotted as a line graph, the troughs in the graph represent the algorithm’s detected areas of low topic cohesion—the locations most likely to contain topic breaks. Hearst’s system implements a trough detection algorithm—not described in detail—which is calibrated to locate the same number of topic changes as a human locates in a document of the same number of paragraphs. Thus, it finds the n -best topic change locations according to the cosine measure.

Having located the topic breaks as indicated by the cosine measure, they are then moved to the nearest real paragraph break. Hearst assumes that topic changes can only occur at paragraph boundaries (that is, that authors will always start a new conscious topic at the start of a new paragraph). Thus a degree of inaccuracy or ‘fuzziness’ to the results of the algorithm is not critical, as errors of less than half a paragraph-length will be corrected. At least, it will be corrected in terms of the results of comparison to human annotation, as it appears that Hearst told the annotators to mark topic changes at paragraph boundaries only.

Hearst’s evaluation was based on human annotation of the target documents. With a base-line precision and recall of 0.43 and 0.42, the system achieved 0.66 and 0.61, compared to the average human judge’s precision and recall of 0.81 and 0.71. The base-line values were measured from a random allocation of breaks on paragraph boundaries, at a rate consistent with the average human judgement of the number of breaks per paragraph (41%)—in this way similar to the rate-decision method for the *TextTiling* algorithm itself.

Hearst also commented on the lack of inter-annotator agreement in human tests of the documents: While some boundaries are strongly agreed upon by her seven annotators, others are disagreed upon, and no boundary markings were unanimous. Her evaluation, therefore, took an agreement of three or more annotators as a real topic break (she argues that overgeneration is more useful than undergeneration for the goals of the *TextTiling* algorithm). This is in contrast to [15], which required four annotators out of seven to agree on a topic break before it was accepted.

Hearst notes that attempts to provide the *TextTiling* algorithm with a thesaurus implementation to augment its word frequency calculations actually worsened the results of the system. While possibly an implementation problem, this is a positive reflection on the effectiveness of simple statistical implementations.

2.1.2 Research extending and improving, and using *TextTiling*

Marti Hearst’s paper describing *TextTiling* is widely cited both by those implementing and extending the algorithm, and those who have designed alternative algorithms and wish to demonstrate them against the de-facto standard. This section discusses implementations of the *TextTiling* algorithm for other domains, and modifications made to it to improve it or customize it for particular tasks.

TextTiling and document summarisation

Boguraev et al. [3] describe a system designed to meet the challenge of high-level information overview when handling large documents. It uses a modified form of *TextTiling* to organise automated document summaries into nested ‘point form’ documents. Topics are detected within the document, and subsequently the key phrases (salient expressions or ‘topic stamps’) are inserted as sub-points below each topic.

Boguraev’s research primarily concerns document summarisation, and not topic segmentation, but in implementing a version of *TextTiling* for the system, an important general extension is proposed to the topic segmentation algorithm: Automatic labeling. In many conceivable applications, topic labeling is an important aspect of topic segmentation, useful for such tasks as browsing quickly through a large document and automatically indexing subject matter.

Boguraev’s system makes use of ‘capsule overviews’ of documents, abstractions of document content designed to capture ‘aboutness’. The algorithm aims to collect for each document a set of “highly salient” phrases. Further, the highest-scored of these salient phrases is then labeled the “topic stamp”. The term *capsule overview* refers to the document’s collection of topic stamps.

The topic stamps are noun phrases, intended to be those relevant to the current topic. They are ideally *discourse referents* within the discourse of the document. Discourse referents are ranked according to their salience, which is judged by prominence of introduction into the discourse, the amount of discussion which involves it, and how often it is mentioned elsewhere in the document. These criteria are combined into a single score; those discourse referents with a high salience weight become the topic stamps for the document. The resulting set of topic stamps is highly compact, listing only the most important topic phrases in the document.

This list is integrated into a capsule overview by combining it with the result of a topic segmentation algorithm based upon *TextTiling*. The document is segmented as described in section 2.1.1 using *TextTiling*. Having determined

the location of (likely) topic breaks, the topic stamp algorithm is applied to the same document. The topic stamps are retrieved, and divided into groups according to the *TextTiling* topic under which they were found. It is important to note that this is different from performing a document summarisation task individually on each topic, as the algorithm analyses the whole document to learn the true importance of each topic stamp. The topic stamps are balanced representations of important subtopics, so a *TextTiling* topic which introduces few important topic stamps should receive fewer—this is only possible if the algorithm has access to the full document at topic stamp generation time.

The topic stamp strings, in order of appearance in the document, are superimposed on the overall layout provided by the *TextTiling* algorithm. This produces a numbered list (representing *TextTiling* topics), with sub-items representing an ordered list of subtopics (represented by topic stamps). In a full presentation of the system’s output, these bulleted lists have superimposed upon them “progressively more refined and more detailed discourse fragments” [3], meaning the system is capable of providing a top-down view of the document with the *TextTiling* topics serving as the highest possible level of structure.

This system provides a robust method for providing labels, or sets of labels, for *TextTiling* topics. In the examples given in [3], each *TextTiling* topic has two topic stamps (which are noun phrases as they are derived from discourse referents). The example document given is an editorial on Microsoft Corporation and Apple Computer: The discourse referents chosen for the first three topics were:

1. APPLE; MICROSOFT
2. DESKTOP MACHINES; OPERATING SYSTEMS
3. GILBERT AMELIO; NEW OPERATING SYSTEM

These topic stamps form an excellent label for the *TextTiling* topic. They also suggest a possible method for finding finer-grained, sentence-level topic changes within a *TextTiling* approach, based on individual topic stamps.

As described in [3], a significant addition to *TextTiling* is the system’s ability to “track” threads of topics. Because it follows topics from the highest level (*TextTiling*) to the sentence level, building a form of tree as it does so, it has a better and more complete picture of the layout and structure of the document.

This has many applications, of which here document summarisation is explored. Possibilities include the ability to tune a topic segmentation system to find broader or finer topics, in real time, according to a user request for more detail; one might imagine a slider or other widget allowing the user to expose more or less detail of a document (from two or three bullet points describing the subject of the document, to the entire document in its original format).

***TextTiling* for visual clarity**

In [17], Hearst works with Rao, et al., to implement an interactive query-based information retrieval system referred to as an “information workspace”. This

system makes use of *TextTiling* to make text easier to read at a glance.

The information workspace system is designed to be a workstation facilitating easy and fast access to heterogeneous data. It allows users to use iterative query refinement to locate pertinent data from multiple data sources. It allows better visualisation of interlinked documents, and supports parallel simultaneous tasks along with a framework for different kinds of activity (such as research and subsequent analysis of data located) without having to change the working environment.

Large bodies of text on a screen are intimidating and difficult to navigate quickly. Once the system has returned a document, the *TextTiling* algorithm is used to provide a visual indication of topics within the document by placing boxes around each topic on the page. This theoretically allows the user to skip ahead ‘topic at a time’ in trying to locate the section of the document he or she is interested in. Secondly, it also provides a visual aid when reading a particular topic of interest, to let the reader know how long the current topic is, and when it has ended.

It could be argued, however, that this application of *TextTiling* is inappropriate—if the author had intended there to be topic markings within the document purely for the purpose of clarity, would he or she not have inserted subject headings or section numbers initially? This manner of automatic insertion of metadata may possibly distort the meaning of certain documents: Topic segmentation is a subjective affair, with no absolute inter-annotator agreement [6], and as such the insertion of authoritative-seeming topic breaks within a document for the purpose of clarity may not be appropriate.

***TextTiling* in topic span detection**

Mittal et al. [13] performed an overview and evaluation of the state of document summarisation in 1999. They examined various approaches to developing “human quality” document summarisation methods, including the use of document level features during summarisation. In a similar way to [3], Mittal described research into making use of topic segmentation within a document to inform document summary.

Mittal states that the ultimate requirement in document summarisation is to find the discourse structure of the document, as a form of tree, at a fine-grained level. However, he regards this as a harder challenge than successful human-quality document summarisation, and so uses *TextTiling* as a broad, top-down approximation to discourse structure. While the provision of a broad linear chain of topics using *TextTiling* does not provide tree structure, Mittal notes that in some domains (such as news journalism), the tree of discourse structure is essentially “right branching” [13], in that it follows a linear approach, digressing into a series of discrete topics in turn. This means that the flow of the document is essentially linear, providing a significant possibility that *TextTiling* will discover most of the important interrelations in the discourse structure simply through its unbranching chain approach to topic flow.

Mittal notes that “in theory, sub-document segmentation can be carried out

recursively, yielding approximate boundaries for a hierarchical approximation to the discourse structure” [13]. Certain practical limitations seem to exist with respect to the *TextTiling* algorithm itself—its word frequency-based approach would find a data sparseness problem below the level it usually works at, and its aforementioned “fuzzy” nature in terms of its accuracy of topic break placement would present a challenge. However, the concept of using recursive topic segmentation to discover (an approximation to) the complete discourse structure of a document is intriguing. Perhaps *TextTiling* could still apply in very large documents, to provide a segmentation first into “chapters” and then recursively into the segments for which it is usually employed.

Mittal’s research showed that “in all cases, pre-processing for topic boundaries can significantly improve the quality of a summarization system, often by a factor of 2”¹. Essentially, *TextTiling* was used here to reduce redundancy: The system attempts to select summaries from different topics in the document, to avoid choosing two summary sentences which describe the same area (and presumably to avoid missing any topics altogether).

The document summarisation algorithms employed by Mittal, in conjunction with *TextTiling*, TF-IDF sentence-ranking and syntactic complexity/named-entity relationships, were also able to avoid using positional information typically required in document summarisation. Positional information can be seen as less important when working on a segmented document, as the requirement to locate potential topic changes positionally does not exist, and each segment can be assumed to be homogenous and monotopical.

2.1.3 Other approaches

TextTiling, while an important algorithm in topic segmentation as has been shown above, is not the only approach available for discerning or segmenting topics in data. Other statistical topic cohesion models exist, such as [1], [9], and [16]. Further, while not a focus of this literature review, segmentation approaches exist for other kinds of media such as audio data and multimedia ([14], [21]).

Audio topic segmentation (in the case of [14] working on broadcast news recordings) often makes use of information not available in a transcript, such as prosodic and pitch-change cues in the recorded voice signal. This is an alternative approach to topic segmentation in spoken dialogue, and while complementary to present research it is outside its scope.

Multimedia segmentation looks for different kinds of segments entirely; in [21], Wilcox and Boreczky work on audio and video data, using Hidden Markov Models to detect camera shot changes, transitions, fades, and dissolves. While useful, this is a different kind of segmentation, analogous to using paragraph breaks as topic markers in text: it relies on the original document containing markers hoped to coincide with topic change, and does not attempt to find topic change in the content itself.

¹“based on average 5-point F_1 score averaged across compression levels and normalized with the random sentence-selection baseline.” [13]

Lexical Cohesion Profile (LCP) text segmentation

Predating [6] by a year, Hideki Kozima published a paper describing an “indicator of text structure” called a lexical cohesion profile [9]. Its basis was a semantic network used to determine the similarity of words in a sentence. This similarity was then used to group areas of the text into topics for segmentation.

Kozima argues that words in fact only receive their meaning when placed in the context of the current topic; therefore, topic segmentation is fundamental to text understanding, and therefore is important for such tasks as resolution of anaphora. Kozima sees text segments as one of several constituents that make up the full structure of a text. In this respect, his idea of a text segment is not based on a human decision *per se*, as is assumed in [6] and [15], but is rather that a text segment is an *ideal* logical part of the structure of any discourse. He compares text segments to scenes in movies.

While Kozima’s idealised description of text segments is one of fundamental logical structure, the LCP system is, like [6], based essentially on a statistical approach: it uses “spreading activation on a semantic network” combined with trough detection upon the similarity output.

Kozima’s lexical cohesiveness metric is based on a semantic similarity between words, derived from [10], which functions as follows: A semantic network is systematically constructed from the LDOCE English dictionary. The similarity between two words is then given as a function of the activity of one of the words’ nodes in the semantic network after activation of the other node. The function brings into consideration the significance of the activated word, based on a normalisation metric to prevent common words such as “and” from being too significant.

The lexical cohesion profile is derived as follows: In a fixed-width window centered around the word currently under examination, take the mean of the current word’s similarity to the other words within the window. This number is calculated for each word in the document, and a line-graph of the similarity metric is constructed, in much the same manner as [6].

This is demonstrated well in two of Kozima’s examples in table 2.1.

Target sentence	LCP similarity
Molly saw a cat. It was her family pet. She wished to keep a lion.	0.403239
There is no one but me. Put on your clothes. I can not walk more.	0.235462

Table 2.1: Example of Kozima’s LCP metric on two ‘sentences’ or segments

Because the ‘important’ words *cat*, *pet*, and *lion* are strongly semantically related, the overall lexical cohesion profile of the ‘sentence’ (here a text segment comprising three sentences) is high due to strong activation of similar words in

the semantic network. In contrast, the second segment has a low overall topic cohesion. It is not necessarily an area of topic *change*, but it cannot be said to be a topic at all, rather it is three unrelated, *non sequitur* sentences.

Indeed, Kozima’s graphs of similarity throughout example documents do not show high plateaus with sudden drops, as is for example expected in ideal output from the *TextTiling* algorithm [6]. Rather, the majority of topic cohesion within the document is relatively low, with only a few short-duration peaks into an LCP value of above about 0.45. If the algorithm is effective, this suggests the possibility that a description of topic segmentation which must assign definite topic to all areas of a document may be attempting to label too much, and that some areas are essentially topic-free.

Kozima uses the lowest valleys of the LCP algorithm’s output, smoothed using a Hanning window, to mark possible locations of topic change.

In his conclusion Kozima suggests an interesting visualisation of topics within a document:

Segment boundaries can be considered as segment switching (push and pop) in hierarchical structure of text. [10]

This suggests a model of topic flow similar to grammar-based parsing of programming languages, where topics can be nested recursively, and each topic must be closed and ‘returned’ in order. This is perhaps most useful as a model for highly structured, well-laid-out documents, but does not seem appropriate for impromptu or “live” data.

Automatically learning topic-change cues for dialogue

Litman and Passonneau [11] present a method for topic segmentation based on machine learning of “discourse segment boundaries”, analysing both textual and prosodic information and using a corpus of spoken dialogue transcripts marked with prosodic features. They initially applied a process of hand-tuning the system based on errors it made in the corpus, and subsequently applied a machine learning technique to simulate the same process of guided improvement based on accuracy on a pre-tagged corpus. In subsequent evaluation, the machine learning algorithm performed slightly better than the hand-tuning approach.

The approach that Litman’s system uses to topic segmentation is to perform an analysis of segment boundaries. At each boundary in the training corpus, the prosodic and cue-phrase features are obtained. Each new utterance in turn is considered as a potential topic boundary: Using a static set of cue phrases derived from [7] (*also, and, anyway, basically, because, but finally, first, like, meanwhile, no, now, oh, ok, only, or, see, so, then, well, where . . .*), if a member of the cue-phrase list appears at the start of the sentence, and it is followed by another from a subset of secondary cue-phrase words, it is marked as a potential boundary.

Coreference resolution also plays a part in the detection of potential topic boundaries: If the first noun-phrase after the boundary is found to be *coreferent*

with a noun-phrase from before the boundary, it is marked as ‘+coref’, otherwise it is marked ‘-coref’. The influence of this marker on topic boundary detection is tuned later, either by hand or through machine learning, but it would be reasonable to hypothesize that coreference across a potential topic boundary would reduce its likelihood. In support of a hypothesis in [15], Passonneau notes that “adjacent utterances are more likely to contain expressions that corefer, or that are inferentially linked, if they occur within the same segment; and that a definite pronoun is more likely than a full NP to refer to an entity that was mentioned in the current segment, if not in the previous utterance.” [11]

Pause duration and other prosodic information are also introduced as separate metrics in the overall combination that makes up the system, but they are not the focus of this literature review.

The machine learning algorithm made use of the collection of linguistic features (anaphoric, cue-phrase and prosodic) at each potential boundary site. Two classes to be learned are defined: *boundary* and *non-boundary*. Using pre-marked training data consisting of 10 documents, comprising 1004 potential boundary sites, the algorithm produces a decision tree, predicting which of the two classes an unknown potential boundary site falls into.

Litman suggests that the success of this algorithm (which does not use any statistical word-frequency information such as the methods described in [6] and [1]) points to a correlation between specific linguistic devices and discourse structure. If this is true, a combination of word-frequency based statistical approach and one verifying potential topic changes using linguistic features may provide better results.

Segmentation where relatively few words are in common

Ponte and Croft [16], in 1997, approached textual topic segmentation with a divergent goal from previous work, especially from [6]: their aim was to research methods for detection of relatively small topic segments, which may not share many common words. This essentially discounts a *TextTiling* approach. Instead, they proposed a query expansion technique which identifies common features in the segments beside word frequency.

Of interest is that, unlike *TextTiling*, Ponte and Croft’s work does not make use of pre-defined boundaries in the source data such as paragraph breaks.

The research focuses on news-bite feeds, where individual articles can be as short as one or two sentences, expressing a fact and moving on—this discounts the possibility of using a fixed-window approach such as used by *TextTiling*. There is usually no topic word repetition at all. However, as the following example illustrates, some semantic relations exist, such as in figure 2.1. Here, the tokens “cyst” and “cancerous”, followed by “growth” and “surgical procedure” occur only once each, but are clearly semantically strongly related.

This problem is approached using Local Context Analysis (LCA): This essentially performs the work of a thesaurus in this context. When given a sentence, it returns a set of generalised “concepts” which allow words with similar conceptual meanings to be matched. For example, the second two-sentence topic

Police in Lebanon said that two Red Cross workers abducted last week are being held by Palestinian guerrillas led by terrorist Abu Nidal.

Meanwhile, thousands of students returned to classes as Lebanon's state schools, most private schools and the American University of Beirut reopened after being closed for six months.

The White House said that a cyst removed Friday from Bush's right middle finger wasn't cancerous.

A presidential spokesman said a routine pathological examination was performed on the half-inch growth following the 25-minute surgical procedure at Walter Reed Army Medical Hospital.

Singapore's Lee Kuan Yew said he would step down as prime minister by the end of next year, ending three decades of nearly one-man rule in the island nation.

The 66-year-old Lee, in an interview with the British Broadcasting Corp., said he would hand over power to his deputy, Goh Chok Tong.

Figure 2.1: Three topic segments within a news feed

shown in figure 2.1 (“The White House...”) has no words in common between its two sentences, but contains 11 counts of “concept” feature similarity. In contrast, there are zero similarities between either the first sentence and its preceding sentence in the news feed, or between the second sentence and the following sentence in the feed. Thus the LCA analysis seems to be a strong method for locating short topic segments.

The possible segments are then scored according to the sum of the pairwise similarities between adjacent sentences and the left- and right-external similarities. The segments are ranked according to the internal similarity minus the external similarities. This simply means, for example, that a pair which is self-similar (homogenous) but dissimilar to its neighbouring sentences will be scored highly as a candidate segment. This process is repeated for different sentence-count sizes of individual segments.

Ponte and Croft demonstrate an interesting case where the algorithm fails: Three consecutive news-bites, discussing a Conservative Party conference, the Yugoslavian Premier seeking financial assistance from the USA, and violence in Namibia respectively, were detected as two, with the division falling in the middle of the second article. This was a difficult case, as all of the articles have a similar theme (politics) and share concepts such as “president”, “party”, “economy”, “premier”, “political”, “administration”, and “leaders”. Ponte and Croft theorise that, given that the LCA database uses data from 1990–1992 and the problematic news-feed was produced in 1989, a training database from a closer time period would have improved matters. For example, it would allow

the use of time-specific data such as semantic relations between “Markovic”, “Yugoslavia” and “premier”.

Extracting features from areas around topic change

Beeferman, et al. [1], approach generalised textual topic segmentation (intended to be essentially domain-independent) using two metrics: *topicality* and cue words. Topicality refers here to the use of language models to detect changes of topic between one area and another, and this approach is combined with a more traditional list of specific cue-words, as seen in several examples above.

Beeferman motivates his work initially as a tool to improve Information Retrieval task output: When a user submits a query to a large corpus of documents, it would be preferable to return a snippet containing the requested information which is bounded by its nearest topic-change boundaries, rather than the more traditional character or word limit. This forms a simple but reasonable approach to the general question of how much surrounding material to return to the user.

Beeferman’s approach to topic segmentation is to assign a probability to the end of every sentence in the document, reflecting the likelihood that it is the end of a topic. This probability is determined by a model which weighs features in the surrounding text. Examples of features pertinent to the domain of multimedia include the presence of the phrase “coming up” in the last utterance.

These features are selected by incrementally building an exponential model (see [2]), and then combined into a predictive model using an algorithm “akin to growing a decision tree”. Each potential feature in the source stream is added to the exponential model, and the *gain* (the largest possible reduction in divergence between the model and the pre-marked text) achieved from this addition is recorded. The candidate feature yielding the largest gain is then added to the model. This process is described in [2].

Each word is examined in the context of approximately 500 words surrounding it, effectively guaranteeing that the context is unique. Using Kullback-Leibler divergence, the degree to which the current context approximates each model in the training corpus is calculated. To avoid overfitting, it must be determined which are the *salient* features present in the training corpus. This is done by creating a *linear exponential family* distribution, which ensures that the features are salient.

Topicality is defined as a function combining a ‘myopic’ trained trigram model for topic change combined with the longer-range model. This is because the long-range model is prone to confusion immediately after topic change: its probability assignments to various words will represent the wrong domain for several sentences after the change, which will confuse it.

Beeferman’s exponential model for text segmentation achieved better results than *TextTiling* for the domain of newspaper text: An error rate of 0.19 compared to *TextTiling*’s 0.296, a miss probability of 0.24 (*TextTiling*: 0.457) and a false-positive probability of 0.1575 (*TextTiling*: 0.191). The algorithm is, however, greatly more complex, both in implementation and in computa-

tional efficiency. Also, Beeferman’s evaluation environment is disparate chunks of newspaper text, a more easily defined but different challenge to the one for which *TextTiling* was designed: ‘Subjective’ segmentation within a single document.

2.2 Spoken dialogue and structure

This section explores theories and models of dialogue structure, with particular emphasis on theories of topic flow and change.

2.2.1 Theories of structure

Cue phrases

Within the literature on automatic topic segmentation, the most widely cited research is Hirschberg and Litman’s *Empirical Studies on the disambiguation of cue phrases* [7].

Cue phrases are words and phrases such as *now* and *well* (along with many others described in section 2.1.3) which serve primarily to indicate document structure or flow, rather than to impart semantic information about the current topic. Of particular interest to this literature review is their capacity to indicate imminent or recent topic change, especially in dialogue where speakers may use cue phrases to indicate desire to change topic.

Hirschberg’s research aims to address the problem that most cue phrases are ambiguous; depending on context, they may be a true cue phrase, or may have a purely sentence-semantic role. Consider the two examples in figure 2.2:

Nobody really expects it to work. Incidentally, the last time I felt this way was...
Inspiration only ever arrives incidentally. It’s not the sort of thing you can plan.

Figure 2.2: Ambiguous cue-phrases

In The first sentence, the phrase “incidentally” is used to introduce a digression in the story—clearly, here it’s used as a cue phrase. In contrast, the second sentence uses “incidentally” as an adverb; in Hirschberg’s terminology, this use is *sentential*, meaning it has only semantic information and no structural relevance.

Another example, provided by Hirschberg, shows the word “now” used as a cue phrase and sententially in the same utterance:

Now now that we have all been welcomed here it’s time to get on with the business of the conference.

Hirschberg performs an empirical analysis of the appearance of both cue-phrase and sentential forms of ambiguous phrases to determine methods of disambiguation. Her conclusion is that prosodic information—pitch curvature and

pause duration—is the most important available feature to disambiguate cue phrases, where this information is available. However, of more relevance to this literature review, she considers the case of distinguishing cue phrase use in transcriptions of speech. Using her extensive collection of phrases disambiguated using her optimal prosodic analysis technique (which will not be examined here) to inform the investigation, she examines the textual transcriptions of prosodically analysed speech to discover orthographic disambiguation cues.

Making use of simple orthographic information—the presence of punctuation immediately before the phrase, punctuation immediately after it, a turn-change indicator immediately before the phrase (that is, the phrase occurring at the start of a turn), or a combination of these—Hirschberg correlates these specific orthographic features to disambiguated phrases from predetermined prosodic information. Using the phrase “now”, of the occurrences where it was in the first position intonationally, it was preceded by punctuation 56.7% of the time, and preceded by a speaker turn 28.3% of the time. This means that 85% of cases where prosodic information determines that “now” is introducing an intonational phrase, it is indicated by punctuation or other information available in a (properly punctuated) transcript. More interestingly, in the test set there was a 100% precision rate—no instances of “now” were preceded by punctuation or a speaker turn that were not prosodically marked as the start of an intonational phrase. Given that in Hirschberg’s corpus, 93.7% of true cue-phrase (discourse) occurrences of “now” were in fact first in their intonational phrase, this simple metric seems highly effective in disambiguating at least the cue-phrase “now” using only transcription. Hirschberg deduces that in total, 80% of discourse uses of “now” may be determined through orthographic means alone. Again, this assumes a punctuationally correct transcript—a feature lacking in many transcripts in practice.

Secondarily, Hirschberg examined the correlation of pairs of potential cue phrases occurring adjacent to one another and their status as true cue phrases. Although she admits the data is sparse, out of 26 *discourse* uses of cue phrases preceded by other cue phrases, 20 (76.9%) were also discourse uses. Correspondingly, 21 out of 29 sentential uses preceded by a potential cue phrase (72.4%) were in fact preceded by a sentential use.

A small boost to accuracy is given by Hirschberg’s use of a part-of-speech tagger: Based on a human judged corpus, potential cue-phrases taking particular parts of speech which are usually tagged as sentential are designated sentential by the algorithm, and vice versa. This simple metric alone can tag 63.9% of potential cue phrases as sentential or discourse correctly.

Hirschberg’s research shows that *certain* cue phrases can be relatively accurately determined from transcriptions of speech alone. This does not directly translate to a topic segmentation solution, but it means that certain aspects of discourse structure, a necessary element of true topic understanding as shown in [9], can be determined from transcripts.

Purpose within discourse

Grosz and Litman [5] describe a model for discourse structure as a whole comprising three aspects: The linguistic structure, the intentional structure (structure of purposes), and the attentional state (state of attention focus).

In this model, linguistic structure corresponds roughly to the kind of structure that *TextTiling* attempts to discern in [6], and that Kozima [9] describes—The topic-based grouping of sentences into contiguous chunks. The intentional structure contains the *purposes* of the discourse which are expressed by the content of the linguistic structure. The attentional structure is about *things*—it records objects, properties and relations currently under discussion within the discourse.

Grosz and Litman hope to provide the foundation of an account of discourse meaning, based on this account of discourse structure.

Of direct relevance to topic segmentation, Grosz and Litman address the issue of what it is that makes a group of sentences constitute a single discourse, or several. They define two kinds of speaker within any ‘segment’ of discourse: The Initiating Conversational Participant, and the Other Conversational Participants (ICP and OCP). Each segment is by definition initiated by the ICP, and then subsequently discussed by the OCPs (and potentially the ICP as well). This model introduces the intentional model described above: The ICP is spearheading a topic of his or her own choosing each time—flow through multiple topics does not occur directionlessly.

Grosz and Litman reinforce the assertion in [7] that there is a relationship between the utterances constituting a discourse and the discourse structure. They show that it is a *two way* relationship: Discourse structure is indicated and controlled by cue phrases (here also referred to as “clue words”), but the meaning of cue phrases is determined in part by their context—that is, the discourse structure they inhabit. Again, this strongly recalls [7].

It is theorised that intentional structure is important to understanding the coherence of a discourse; specifically, whether a series of utterances comprises more than one discourse (for which we may wish to substitute the word “topic”). The theory states that those participating in a conversation have an aim in mind: Their Discourse Purpose (DP). The DP motivates both the act of speaking at all, and the particular message conveyed by that speech.

Finally, attentional state is given by a *focus space*. There is a focus space for every discourse segment, containing the properties, objects and relations pertaining to that segment. The focus space does not contain the Discourse Purpose as it is not intentional—it merely describes the state of affairs in terms of objects.

A discourse is said to be coherent when all of its Conversational Participants share a Discourse Purpose, and each utterance of the discourse contributes in some way to achieving this purpose.

Chapter 3

Software Implementation

3.1 Outline of functionality

The system takes as input a transcribed dialogue, and outputs topic change information for that dialogue. This information is in the form of a machine-readable (XML) file, a human-readable (HTML) transcript of the dialogue with topic changes marked, and a graph showing the continuous topic-cohesion metric for the dialogue, superimposed with marks indicating the locations of the detected topic breaks. Additionally, the system accepts human-marked topic changes, plotting them alongside its own detected changes: This allows all experiments to be evaluated easily for precision and recall against human judgement, which is the primary method of evaluation.¹

3.2 Architecture

The system consists of the following modules:

- Text segmentation
- Cosine measure calculation
- Trough detection
- XML generation
- HTML generation
- Graph generation
- Automation and batch processing

¹Hearst notes, however, that there is a high level of disagreement between human judges in a topic segmentation task, and thus this method is not infallible [6]. See Chapter 5 for a discussion of this issue.

Each module is individually usable as a standalone unit: Complete generation of all data for an individual transcription requires a sequence of separate executions. This allows individual aspects of the system to be tested, debugged and evaluated more quickly, as the execution time of each component is kept to a minimum. For example, some experiments may require fully-marked-up HTML files, but no graphs.

Because the system consists of multiple modules which must work in concert to produce a ‘complete’ result set, scripts have been included to automate the process.

A full run of the system produces a number of intermediate files containing the data points of the cosine similarity measure, the smoothed similarity measure values, a list of detected topic changes, and a list of hand-marked topic changes. These data files can be cached to save processing time when multiple runs of ‘downstream’ modules are required, or used immediately and deleted. They are required for the generation of graphs and for the generation of marked-up conversations in XML and HTML.

The central module is the text segmentation unit itself. It is responsible for reading the original transcript file and applying the segmentation algorithm to it.

The cosine measure module has a single purpose: It calculates the value of the cosine measure for two vectors currently under examination. For details of the cosine measure formula and the contents of the vectors, see section 3.3. It is called repeatedly by the central module during calculation of the continuous values similarity across each transcript.

The trough detection module is called by the text segmentation unit: it detects troughs in a continuous set of similarity values, which will form the putative topic breaks output by the text segmentation unit.

The XML generation module uses the internal (memory-only) structure created by the text segmentation module to interleave markers for topic change into a simplified version of the original input file. It must therefore be called simultaneously with the text segmentation module, and its input cannot be cached.

The HTML generation module uses the cacheable output of the XML module to create the human-readable transcript of the data. The output of the HTML module is used in conjunction with a style sheet to allow convenient viewing and printing.

The graph generation module uses the cacheable output of the text segmentation module - the listings of similarity measure, detected topic changes, and hand-marked topic changes. It is capable of working with partial input, so will generate graphs containing whatever input is provided.

Additionally, all modules will accept as optional input a pre-generated list of detected topic breaks in order to reduce redundancy.

3.3 Implementation details

The system is implemented using the following languages and technologies:

- Python
- XSLT
- CSS
- R (data visualisation)
- Bash

The main body of the system (the text segmentation module) is implemented as a collection of Python modules:

- `cosinemeasure.py`
- `peakpick.py`
- `micaseparse.py`
- `segmentation.py`

`segmentation.py` is a front-end to the text segmentation module itself, which allows convenient coordination and manipulation of the text segmentation system from the command line or from a batch script. It accepts command-line arguments which select the type of output information required and specify the location of the input file (or files, if a pre-processed topic-break location list is available) - see section 3.4 for a list of command line options accepted.

3.3.1 Ancillary modules

`segmentation.py` contains two variable constants relevant to the tuning of the system: `smooth_size`, which controls the size of the window over which the smoothing algorithm will function (details below) and `percent_detection`, the minimum size of a detected trough as a percentage of the total range of the cosine measure over the current document.

The `cosinemeasure.py` module is implemented as follows: It takes as input two equal-length vectors of integers. It computes a similarity value as a single real number, n , $0 \leq n \leq 1$. This similarity metric is computed using equation 3.1:

$$sim(b_1, b_2) = \frac{\sum_t w_{t,b_1} w_{t,b_2}}{\sqrt{\sum_t w_{t,b_1}^2 \sum_{t=1}^n w_{t,b_2}^2}} \quad (3.1)$$

Where b_1 and b_2 are the vectors under comparison, t ranges across the values in each vector, and w_{t,b_n} is the value in vector n at position t . The result of this calculation is the cosine of the angle between the hyperdimensional lines represented by each vector. For example, imagine two vectors of length two: In

this case, the angle between the lines they represent is easy to visualise - the first number represents an arbitrary rotation in the horizontal plane, and the second a rotation in the vertical plane. This system is also immune to magnitude: The vectors [1, 2, 3, 4] and [2, 4, 6, 8] represent the same line, and thus have an angle between them of 0 radians, and a corresponding cosine measure value of 1.

In a language technology context, the cosine measure is usually used to determine the similarity of two documents based on word frequency. Thus, the vectors represent every unique word present in the entire document under segmentation. Each value is the frequency of the corresponding word within the current window. At any time, most of the values in the vector are zero, but this has no effect on the cosine measure, and only nonzero numbers impact the similarity measure.

The `peakpick.py` module locates significant local minima (troughs) within a vector of real numbers. It must in addition be passed a percentage value representing its sensitivity as a percentage of the total variation.

The module first determines the range within the vector, and calculates the absolute change in value that the percentage argument represents. It then steps through the values in the vector, remembering the highest value. When a value is located that has dropped more than the threshold amount from the current highest value, this value is watched. If the value rises above the current lowest value by a greater amount than the threshold, the current lowest value is marked as a trough, and current highest and lowest values are reset.

3.3.2 Parsing the data

The `micaseparse.py` module, called by `segmentation.py`, coordinates the core of the system: It is concerned with reading the dialogue transcript from an XML source file, storing the relevant data internally, coordinating its processing through the various segmentation metrics (including `cosinemeasure.py` and `peakpick.py`), and printing the resulting data. These functions are achieved in several ‘passes’ over the internal data, implemented as follows:

In its first pass, the module handles reading and chunking of the input XML files and coordinates the internal processing of the data. It makes use of the SAX module from the PyXML package to parse the input files easily. It creates a `SaxUtils` default handler for the XML document, calling it to parse the document specified by the calling module (`segmentation.py`). SAX then handles the task of traversing the input file, passing XML entity tokens to the handler. The handler itself acts only when it receives a *turn* event—turns contain the utterances of the speakers in the conversation, whereas all information outside those turns is metadata not required by the algorithm.

The data received by the handler is then stored in a complex set of data structures: At parse-time, the following structures are filled in parallel (from a single parse of the input file):

- The dialogue stripped of tags, speakers, and stop-words, as a list.

- The dialogue in its original format, marked with pseudosentence markers, and optionally with automatically detected topic breaks, as a string.

During the first pass, the handler repeatedly calls the private method `chunkBody()`, passing in the current string of data, without XML tags, if it is inside an utterance.

`chunkBody()` tokenises the strings it receives into individual words, strips them of punctuation and extraneous characters, and removes words in the `stop_words` list. Secondly, it copies the dialogue it receives directly into a string representing the original dialogue. Pseudosentence numbers are inserted in XML tags, allowing later visualisations of the dialogue to display them for correlation against graphs during evaluation and experimentation. Lastly, the system takes note of any `<break />` markings within the text: These are indicators of where humans have located topic change, and are manually inserted. The locations of these breaks are recorded by the index number of the pseudosentence they appear in, and built into a list.

After parsing the file itself, the following additional data structures are created (dependent on the specific calling options):

- A list of all unique words
- A list of pseudosentence objects
- A list of any human-annotated topic breaks

The second pass of the module, working on the in-memory representation of the dialogue, builds a list of all unique words within the dialogue. This structure will be used by the `cosinemeasure.py` module to construct a list of the frequencies of all words within the current windows.

The system then begins to construct a list of objects of type `pseudoSentence`. This type is defined inside the `micaseparse.py` module, and exists to perform a number of important tasks related to each pseudosentence: Foremost, it contains the pseudosentence itself as a list of string tokens. Its main method tallies the frequencies of the words in the pseudosentence and returns a list of those frequencies. It also performs a variety of functions upon the pseudosentence, mostly related to reporting for statistical and debugging purposes.

Once the list of pseudosentences has been populated and their frequency lists calculated, a loop iterates across the list of pseudosentences, applying the TextTiling algorithm: Starting from position $blocksize/2$ in the list, the algorithm compares overlapping $blocksize$ groups of pseudosentences using the `cosinemeasure.py` module. For each comparison, a value between zero and one is returned. These ‘continuous’ similarity values are collected in a list.

The system then creates a smoothed copy of the similarity list using a simple smoothing algorithm: Each value in the smoothed list is given as the mean of itself and its nearest neighbours. The number of neighbours compared can be specified in the `segmentation.py` module, but the default is to use three values: $n - 1$, n , and $n + 1$.

The trough detection module `peakpick.py` is then used upon the smoothed similarity values list: It returns a list of integers corresponding to the pseudosentences detected as being significant troughs; that is, candidates for topic breaks.

3.3.3 Output

Having completed the population of all internal data structures, the system is capable of outputting the following information:

- Similarity values ($0 \leq n \leq 1$)
- Smoothed similarity values ($0 \leq n \leq 1$)
- Detected topic breaks (integers representing pseudosentence numbers)
- Human-annotated topic breaks
- Dialogue in XML format, with pseudosentence numbers marked, and optionally with automatic or hand-transcribed topic breaks marked

Each data type (except for the XML format) is output in a tab-separated flat-file, with a comment (signified by “#”) naming the columns. For example, a smoothed-similarity-values output begins:

```
#Pseudosentence SmoothSimilarity
5                0.622067
6                0.634649
7                0.615261
8                0.566583
9                0.501763
```

...

3.3.4 Visualisation

The system makes use of two visualisation methods for the data it produces: Graphs and marked-up transcriptions.

The graphs are produced using *R*. A collection of simple *R* scripts reads as many input files as are available (at a maximum the similarity data, smoothed similarity data, detected topic breaks and human-annotated topic breaks files) and plots the similarity values as a continuous line graph, and the detected breaks and human-annotated breaks as clear, coloured vertical lines. These can be combined flexibly to show as much data as is necessary: The full functionality of the scripts (in file `twoplot.R`) overplots the raw similarity data in grey, the smoothed similarity data in black, the automatically detected breaks in broken green lines, and the human-marked data in red. *R* outputs its graphs in Encapsulated Postscript (EPS) format, suitable for printing as a full page.

The marked-up transcriptions are produced using XSLT, an XML transformation language, processed by the program `xsltproc`. The XML transcript files produced by the python modules are processed by an XSLT ‘stylesheet’ which transforms the XML elements such as `<S name="x"/>` (speaker), `<BREAK />` (human annotated topic break), and `<AUTOBREAK />` (detected topic break) into valid HTML4.0 for viewing in a web browser. Each speaker turn is marked as a separate `<P />` (paragraph), with special features such as speaker names, pseudosentence numbers, and topic breaks marked as a `` with a special style name.

Three XSLT scripts exist; each includes a different level of detail in the output HTML:

- `newmicase-html.xml`
- `newmicase-html-pseudosentences.xml`
- `newmicase-html-pseudosentences-breaks.xml`

These scripts allows custom CSS (Cascading Style Sheet) files to be written for the transcript HTML easily, changing its layout and colouring to suit the medium. Two CSS files have been created:

- `speakers.css`
- `speakers-print.css`

They are optimised for screen-viewing and (black and white) print viewing respectively. `speakers.css` uses different colours to represent each speaker, allowing easier following of in-line interruptions and overlaps between two speakers (though this feature only applies to older variants of the MICASE XML format). Topic breaks and pseudosentence numbers are printed in red, with breaks marked by two vertical bars, “||”. In both CSS files, speaker names are printed using a larger font, and interrupting and overlapping text is printed in italics.

Finally, batch processing of input dialogue files for graphing purposes is managed by shell scripts:

- `makegraph.sh`
- `buildgraph.sh`

These coordinate the processing of dialogue files into a series of graphs to show human-annotated and detected topic breaks: `makegraph.sh` calls the python subsystem repeatedly on all XML files in the current working directory, producing four files per dialogue: Raw and smoothed similarity values, and the lists of hand-annotated breaks and detected breaks.

`buildgraph.sh` then uses these output files to build four progressive graphs, from the raw similarity data plotted in grey, to an overplot of all four data sources, as described above. These progressive graphs are useful for examining the success rate of the automatic detection without it crowding the graph.

3.4 Summary of usage

The usage summary printed by the dialogue segmentation system's interactive (i.e. non-batch) front-end, `segmentation.py`, is as follows:

```
usage: segmentation.py [options]

options:
-h, --help            show this help message and exit
-n, --numbers          print out the unsmoothed (raw) similarity data for file source_file
-b, --handbreaks      print the human-annotated breaks (if any) found in the file.
-m, --markuppseudo    print the original XML file (a subset of it not containing metadata) with pseudosentence tags.
-iBREAKSFILE, --insertbreaks=BREAKSFILE
-s, --smoothdata      print out the smoothed similarity data for file source_file.
-d, --detectbreaks    print the detected topic breaks.
-fFILENAME, --file=FILENAME
```

The following are common ways to call the system:

`segmentation.py -f source_file --smoothdata` – print out the smoothed similarity data for file `source_file`.

`segmentation.py -f source_file --numbers` – print out the unsmoothed (raw) similarity data for file `source_file`

`segmentation.py -f source_file --handbreaks` – print the human-annotated breaks (if any) found in the file.

`segmentation.py -f source_file --detectbreaks` – print the detected topic breaks.

`segmentation.py -f source_file --markuppseudo` – print the original XML file (a subset of it not containing metadata) with pseudosentence tags.

`segmentation.py -f source_file --markuppseudo --insertbreaks=breaks_file` – print the XML dialogue as above, additionally inserting topic-break tags from the file specified in `breaks_file`

The R scripts may be called in the following manner:

```
R BATCH twoplot.R
```

This will attempt to perform graph generation on every file with the extension `.txt` in the current working directory. Optional files with the same name ending with `.txt.auto`, `.txt.breaks`, or `.txt.spiky` will be found and used to plot detected topic breaks, hand-transcribed topic breaks, and unsmoothed similarity data respectively.

The XSLT script may be called as follows:

```
xsltproc newmicase-html.xsl input_file.xml
```

This will print well-formed HTML on standard out, which can be piped to a file.

`makegraph.sh` must be called as follows:

`makegraph.sh input_file output_name` Where `output_name` is the base name of the output files to be created. They will be created with the extensions `.txt.auto`, `.txt.breaks`, and `.txt.spiky`.

`buildgraph.sh` must simply be called with no arguments, with their input files in the current working directory. It processes every file with the extension `.txt`, leaving four EPS files with the same base-name.

3.5 Example session

This shows an example session producing a complete set of output data for a single input dialogue.

3.5.1 Calculating similarities and detecting breaks

```
$ mkdir examplesession
$ makegraph.sh dialogue.xml examplesession/dialogue
Informational: No breaksfile name provided, none used.
Number of pseudosentences: 274
Informational: No breaksfile name provided, none used.
Number of pseudosentences: 274
Informational: No breaksfile name provided, none used.
Number of pseudosentences: 274
Informational: No breaksfile name provided, none used.
Number of pseudosentences: 274
```

3.5.2 Creating annotated transcripts in HTML

```
$ segmentation.py -f dialogue.xml -i examplesession/dialogue.txt.breaks
-m > dialogue-hand-annotation.xml
Number of pseudosentences: 274
$ segmentation.py -f dialogue.xml -i examplesession/dialogue.txt.auto
-m > dialogue-detected-breaks.xml
Number of pseudosentences: 274
$ xsltproc newmicase-html-pseudosentences-breaks.xsl dialogue-hand-
annotation.xml > dialogue-hand-annotation.html
$ xsltproc newmicase-html-pseudosentences-breaks.xsl dialogue-detected-
breaks.xml > dialogue-detected-breaks.html
```

3.5.3 Plotting graphs

```
$ cd examplesession/
$ buildgraph.sh
spiky:
spiky+smooth:
spiky+smooth+auto:
(spiky)+smooth+auto+breaks:
```

3.6 Bugs and inefficiencies

The software is suboptimally architected in a number of ways: Due to its nature as an experimental system designed for testing various theories, its functionality was not well-defined initially. While the core of the algorithm is implemented as a clean Python module, it contains several different methods for passing in options and variables: Some are hard-coded in the `micaseparse.py` file (pseudosentence length, block size, and whether to use stop words); some are hard-coded in `segmentation.py` (smoothing size); and some are passed in from the command line interface to `segmentation.py`.

The architecture of `micaseparse.py` means that it essentially can only perform one pass over the input XML data. Because it extracts only the information it needs from the file, it is sometimes necessary to perform two passes over the data to produce certain kinds of output. For example, two passes are required to produce automatic-break-annotated XML output: The first pass calculates the similarity values and performs trough detection, printing a list of breaks on standard out. The second pass reads in a list in the same format, printing out an XML file with breaks inserted at the appropriate locations. This is very inefficient, as each run of the software performs a complete pseudosentence chunking and similarity value calculation, which in this case is completely redundant. An identical inefficiency occurs if an experiment requires both smoothed and unsmoothed similarity data: There is currently no way to print both kinds of data simultaneously, so two full runs of the system are required, with different options each time.

On a 1 Gigahertz CPU, using a pseudosentence length of 20 and a block size of six, single-run execution time for a 5480 word dialogue (excluding stop-words) is only 25.264 seconds. However, settings involving a greater number of comparisons take significantly longer: Using a pseudosentence size of 1 and a block size of 120, a single run takes over three hours. Therefore, when experimenting with pseudosentence length it is clearly important to streamline the execution time as much as possible, and to remove redundancy.

The batch scripts written to automate the creation of the various output files, and to produce graphs, all use hard-coded paths to the scripts they execute. This is not a problem for the test environment, but represents a hazard to portability.

A bug exists in the annotated XML generation functionality: When reading an outside breaks list file, it incorrectly prints the name of the file it reads on standard out. This means that piping its output to a file includes this informational message, which confuses strict XML parsers such as XSLT, and it must be manually removed.

During the course of experimentation and evaluation, a new set of MICASE dialogue XML documents was acquired. Unfortunately, the XML format had changed (for example, the standard ‘speaker’ tags, `<S1 />` and `<S2 />`, representing the original and overlapping speaker respectively, were changed to the unified `<S />`). This introduced compatibility issues as scripts were modified to handle both new MICASE dialogues and existing ones which had already received much attention.

While the Python scripts were successfully modified to read both formats, forks were required in the creation of XSLT documents to handle each. As a result, the newer MICASE XML documents cannot be displayed using a different colour for each speaker. Additionally, due to differences between the original XML documents from micase and the system's output XML, further versions of the XSLT scripts were required to handle these cases.

Chapter 4

Experiments

4.1 Outline

The aim of this research was to investigate methods of topic segmentation for the special case of spoken dialogue transcripts. To facilitate this research, the differences between traditional written text (‘expository text’ [6]) and spoken dialogue were explored. Subsequently, a series of tests were performed to evaluate the *TextTiling* algorithm, designed for textual topic segmentation, in this domain: Using human annotation as a comparison and reference, evaluations were performed on a test-suite of 10 diverse transcripts from different domains of spoken dialogue. Additionally, parameters for the *TextTiling* algorithm were modified to evaluate the effect of the spoken dialogue domain upon the optimal parameters derived by Hearst [6] for the domain of magazine articles. Finally, results were examined at the level of individual topic breaks to investigate patterns in the successes and failures of *TextTiling* in the spoken dialogue domain.

To this end, a system using *TextTiling* was implemented to test upon transcripts of spoken dialogue (see chapter 3 for implementation details), and the Michigan Corpus of Academic Spoken English (MICASE) was chosen as a source for spoken dialogue transcripts. These transcripts are manually transcribed (not a product of speech-recognition) so represent a relatively ‘clean’ source for spoken dialogue transcriptions. However, it is important to note that they contain little punctuation within speaker turns—see section 2.2.1 for a discussion of how this would affect heuristic detection of cue-phrases.

Human-annotated topic breaks within MICASE documents were then used to evaluate automatically detected topic breaks by the system for a variety of different system parameters.

4.2 Methodology

4.2.1 Hand-annotated data collection

Volunteers were provided with a printed dialogue transcript marked only with the spoken text itself, and codenames for the speakers of the form $s_1, s_2, \dots s_n$ identifying the speaker of each utterance. The volunteers were asked to draw a line on the page wherever they felt the topic of the dialogue had changed, dividing the text of one topic from the text of the other. This line was allowed to be placed anywhere, but had to divide the text unambiguously at that location, even if this was in the middle of a speaker turn or sentence (early trials with insufficient instructions resulted in some volunteers marking a line indicating an approximate area of topic change (such as within a turn) without placing its location exactly). These hand-annotated breaks were then re-integrated into a version of the original dialogue XML file for automatic processing as described in chapter 3.

The volunteers were given no more information than this—specifically, they were not told how many topic changes they should find in the document, whether the pattern of their appearance would be regular or irregular, or indeed whether any topic changes existed in the document they had received. They were also given no more information on the definition of a topic, leaving the interpretation to them. This was considered to be the best way to avoid the problem of ‘self-fulfilling prophecy’, in which the subjects provide answers they expect the system to agree with. In this way, the purpose of the research—to investigate methods of segmenting topics in a *meaningful* way—is better preserved.

The ten test dialogues are as follows (with the exception of the MICASE dialogues, they are not explicitly described, and their content is described informally for the convenience of the reader):

- MICASE (Michigan Corpus of Academic Spoken English)
 - Honours Advising session
 - American Culture advising session
- CMU (Carnegie Mellon University)
 - CMU_20020320-1500: Students discussing UNIX
 - CMU_20020419-1400: Student discussion, general
- ICSI (International Computer Science Institute)
 - ICSI_20010208-1430: Researchers, speech recognition meeting
 - ICSI_20010322-1450: Researchers, audio recording meeting
- LDC (Linguistic Data Consortium)
 - LDC_20011116-1400: Discussing television programs
 - LDC_20011116-1500: Discussing examinations

- NIST (National Institute of Standards and Technology)
 - NIST_20020214-1148: Discussing wood-working
 - NIST_20020305-1007: Discussing orientation day

This sample of dialogues is biased towards university recording environments (students and staff) but is otherwise fairly broad. The MICASE dialogues take place between students and advisers. The rest occur between two or more peers in a somewhat ‘guided’ (that is, purposeful) meeting-type discussion.

Each of these dialogues was marked up by one of the group of volunteers to show topic changes. Where a volunteer marked up multiple dialogues, this fact was recorded to allow analysis of commonalities in personal style. These data were entered into the graphing system to produce plots showing the correlation between the hand-marked topic breaks and the system’s automatically detected breaks (these graphs can be seen in section 5).

4.2.2 System parameter experiments

Three parameters of the software system underwent experimentation to determine optimal settings: Pseudosentence length, block (or ‘pseudoparagraph’) size, and trough threshold.

Pseudosentence length refers to the number of words (tokens) in each pseudosentence in the system. As the system compares two overlapping blocks, each containing `block_size` pseudosentences, effectively the pseudosentence length only controls the size of the overlap between the blocks, and the step size taken by the ‘rolling window’ functionality of the system. For example, a pseudosentence of length 1 has an overlap between the two blocks of only one word, and will step its current window forward a single word at a time.

Block size refers to the number of pseudosentences to compare during each comparison: It affects the total size of the current window (that is, the scope inside which words within the dialogue have any effect) - this window is always $(2 * block_size) - 1$ pseudosentences in length.

By modifying block size and pseudosentence length, the proportion of overlap between the comparison blocks, the size of the step forwards between each comparison, and the total size of the window under comparison can all be modified.

Trough threshold refers to the sensitivity of the through detection algorithm used to locate potential topic breaks. It dictates a percentage value indicating how deep a trough must be before it is marked. This percentage is a proportion of the total range of the continuous similarity data, from lowest value to highest, across the current document. A trough will be detected if its lowest point has peaks to both sides higher than the specified percentage of the total height of the graph.

TextTiling parameters

Primary system parameter experimentation involved varying pseudosentence length and block size. The initial values are shown in table 4.1:

Pseudosentence length	20
Block size	6

Table 4.1: Experiment 1

These values were chosen as a starting point, as the original *TextTiling* system decided on these values as best for expository text [6]. It was theorised that these settings would be suboptimal for spoken dialogue.

It was noted that more data points were acquired when the pseudosentence size was reduced. Given the data sparsity observed in some evaluation dialogues (which were shorter than the magazine article examined by Hearst) which would not have allowed many troughs to appear due to a lack of data points, an experiment was performed using a pseudosentence of length 1 (table 4.2). To avoid altering more than one parameter, the effective total block length (in words) was kept the same, by changing `block_size` to 120. These single-word-pseudosentence experiments were time intensive, and only a limited number were performed.

Pseudosentence length	1
Block size	120

Table 4.2: Experiment 2

The experiments in tables 4.3 and 4.4 were then performed, modifying Hearst’s settings more conservatively:

Pseudosentence length	40
Block size	3

Table 4.3: Experiment 3

While still maintaining an overall block-size of 120 words (as with experiments 1 and 2), experiments 3 and 4 respectively double and halve the pseudosentence length, effectively modifying the step-size and overlap size of the rolling window.

Because experiment 2 (table 4.2) showed interesting results, a further two experiments were performed modifying the size of the window, shown in tables 4.5 and 4.6.

This modification has the result of changing the total size of the area of the dialogue currently under consideration by the system, without affecting

Pseudosentence length	10
Block size	12

Table 4.4: Experiment 4

Pseudosentence length	1
Block size	240

Table 4.5: Experiment 5

the step size or the overlap size. With a larger window, more words form the two blocks under comparison, increasing the sample size over which the word frequency table is constructed. Therefore, random frequency fluctuation would be expected to be reduced by a larger window, while resolution, as well as the ability to perform comparisons near the beginning and end of the document due to the requirement to have a whole block to the left and right of each measurement point, might suffer.

Trough detection parameters

The trough detection algorithm’s parameter, a percentage value, dictates its sensitivity. A trough will be marked as a potential topic break if it contains ‘hills’ on both sides which rise more than this value as a percentage of the total range of the graph.

The challenge is to tune its sensitivity sufficiently to locate as many topics as possible without detecting false positives (or without detecting too many). Due to the nature of the input data, for a given sensitivity setting some dialogues produce a large number of detected topic breaks, while others produce none at all (see chapter 5, including figures 5.9 and 5.8). While the precise reasons for this will be explored in chapters 5 and 6, a guiding assumption during the design of the experiments presented here was that high variation in topic change frequency is acceptable: Some dialogues may change topic more frequently than others.

Experiments were performed using the following sensitivity settings:

- 5%
- 10%
- 15%
- 25%
- 30%

The resulting topic detections were compared to the relevant human-annotated documents to determine an optimal level to produce a ‘clean’ conservative set

Pseudosentence length	1
Block size	60

Table 4.6: Experiment 6

of potential topic breaks. See chapter 6 for a discussion of what false positives from this algorithm mean, and what it cannot detect.

Chapter 5

Results and Evaluation

This chapter presents the graphical results obtained from the experiments described in chapter 4, and commentary on the successful and unsuccessful topic detections by the system (when compared to a human annotator). The graphs show human-annotated and automatically detected topic breaks, superimposed over a representation of the cosine measure function from which the automatic topic breaks derive. This allows ‘near misses’ to be examined, as well as a more complete view of topic cohesion trends beyond a thresholded binary view of topic change.

In each graph presented in this section, the black line represents the smoothed similarity value according to the cosine measure; the red lines represent the human-evaluated topic boundaries; and the dashed green lines represent the system’s topic change location theories. The grey line represents the unsmoothed cosine similarity metric: its purpose in the graph is to allow visual detection of spikes which may have been removed by the smoothing algorithm in error. While the trough detection algorithm works using the smoothed data only, it is useful to see the cosine measure’s original output for evaluation purposes.

5.1 Calibration data

In order to verify the correct functioning of the topic detection algorithm as a whole, a test data set was created, representing the ideal topic segmentation task for the TextTiling algorithm: A 100% frequency of one word followed by a 100% frequency of another, and so on. A snippet of this data can be seen in figure 5.1. Because this data set involves a complete change in word frequency (zero cohesion between the ‘topics’), the algorithm should work optimally on it. The text was ‘evaluated’ manually and the results, presented in graph form, can be seen in figure 5.2. The red lines represent the hand-marked topic changes (and thus, in this case, the actual boundary between one repeated word and the next) and the green lines represent the system’s automatic location of the breaks. Of interest is the peak between marked breaks 2 and 3, representing

```

...one one one one one one one one one one one one one one one
one one one one one one one one one one one one one one one...one one
one two two two two two two two two two two two two two two two
two two two two two two two two two two...two two two three three
three three three three...three three four four four four four
four four four four four four four four four four four...four
four four five five five five five five five five five five five
five five five five five five five five five...

```

Figure 5.1: Test word set for system evaluation

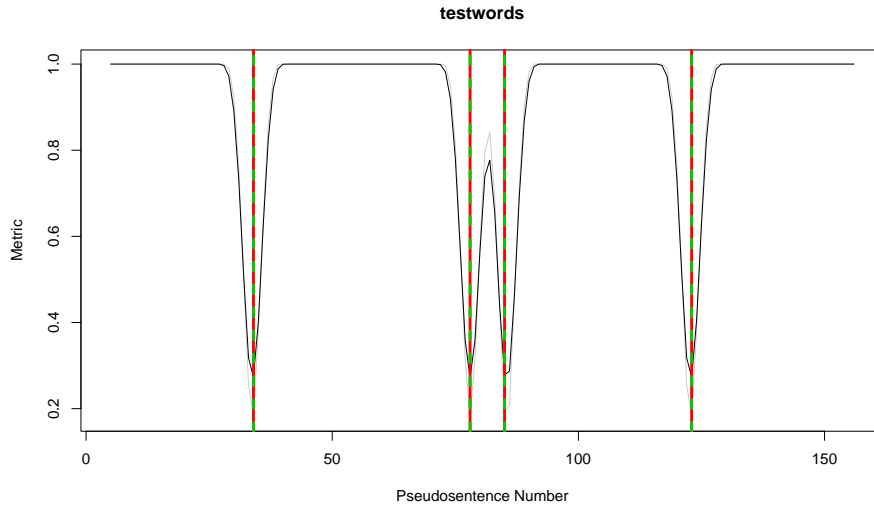


Figure 5.2: Full system graph output on the test word set

the third located topic. It is lower than the other peaks in this example because it is not long enough (in terms of word count) to register fully according to the TextTiling algorithm; that is, the whole topic is somewhat smaller than the rolling window used in the algorithm.

5.2 Evaluations against human markup

The graphs in figures 5.3, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11, 5.12, 5.13 and 5.14 show the final output of the tuned system on the test dialogue set described in chapter 4. Interesting features in these graphs will be discussed below.

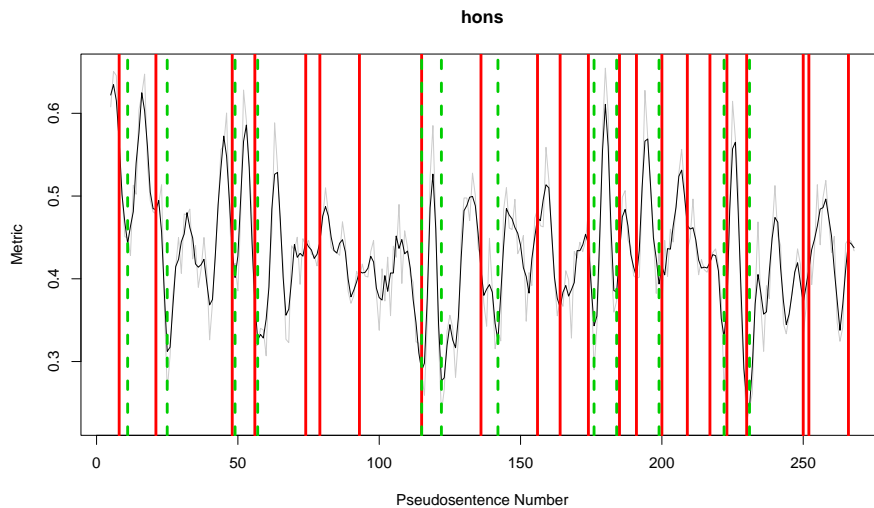


Figure 5.3: Honours advising session, $ps = 20$ $bs = 6$

5.2.1 Imprecision

In the original domain of TextTiling (magazine articles) [6], precision (meaning finding the exact location of a topic change, instead of its general area) was not a priority. The algorithm moved each detected topic break to the nearest paragraph boundary within the text. However, an equivalent of this guideline does not exist in spoken dialogue: Turn-changes are essentially the only semantic divisions provided by data, and these correspond more closely to sentences in prose—they are too small to provide a similar indicator of potential topic change. Examining the graph provided by Hearst in [6] showing detected topic boundaries superimposed over the cosine measure line-graph, it is possible to see *significant* movement away from the actual troughs in the graph in order to match to the nearest paragraph boundary.

Examining figure 5.3, then, we observe this phenomenon in action: The first four detected topics are clearly successful (in terms of agreement with the human subject), but not perfectly aligned. In text with the equivalent of paragraph boundaries available, clearly this problem would not exist, and the first four topic detections in this instance would be perfect.

To demonstrate this issue more clearly, figure 5.4 shows the site of the first topic change from figure 5.3. While the real and detected breaks are ‘close’ and clearly related in the graph, they are nearly two full speaker turns apart in the document itself. This is too large a gap to be closed merely by moving to the nearest speaker change; clearly, in fact, in this case such a decision would make the segmentation precision worse.

S2: ...international religion or uh not religion international relations, so, those things i wanna - i think i'm gonna concentrate more on, i don't think i wanna do the business.
 <break />
 S1: have you done any digs or anything like that ?
 S2: no i'm really like dinosaurs like fascinate me like that stuff fascinates me but i don't know if that's like a career choice yet but i, was looking through the course book and i know they offer like a w- half a term class or something, that if i had space that i could, like, take and see if i, if it was worth it that i should go into, you know more depth or if that was just sort of like okay, i l- i like it but i don't wanna, like study
 <autobreak /> that so i don't know.
 S1: both geology and biological anthropology, will lead you, that way.

Figure 5.4: Location of topic change 1 from figure 5.3, showing distance between detected and real topic boundaries ('break' tag is hand-marked, 'autobreak' is system-determined)

5.2.2 Misses

With the current settings, the system is rather conservative in its detection of potential topics. In most cases, the human annotator has identified significantly more topics than the system. This is partially a matter of trough-detection sensitivity—clearly, in some cases (such as the first topic break in figure 5.6) a higher sensitivity to troughs would cause higher agreement. This, however, is not the most interesting case of a 'miss'; consider instead, for example, the fifth hand-annotated topic break in figure 5.3: It is essentially not inside a trough of any description, but resting on a plateau. The textual context of this case is shown in figure 5.5.

The implication of cases such as this is that there are some topic changes identified as such by human subjects, but which do not register at all using the cosine measure. This suggests that TextTiling, while effective, does not encompass a complete account of the process of topic change.

5.2.3 Annotator normalisation

An interesting aspect of human annotations, of which figure 5.7 is a possible example, is that they appear to be *self-normalising*. In other words, when an annotator is assigned the task of dividing a dialogue “into topics”, he or she has a preconception about how frequently topics might change. Reading a monotypical stretch of a document, it is theorised that the annotator begins to begin to feel that a topic change is due, and becomes more sensitive to possible changes. In this way, a text comprising regular, obvious changes will be marked with those changes, whereas a text with only subtle changes (if any)

S1: ...um, and then tomorrow morning when you come in to talk to me, you'll have pretty much a solid idea of what you're gonna be taking. i advise you actually to look at the introductory geology courses, not just the ones that are in the freshman seminar, but the ones that are in the

S2: like the bulletin that we had

S1: yeah, well, have you been on the online online course guide yet?

<break />

S2: no huh'uh

S1: okay it's really easy i'll show you. you know you know what the web is ...

Figure 5.5: Context of missed topic change (manual topic change 5) in figure 5.3

seems to be marked more sensitively by human annotators. When told to find topic changes, the annotators usually do. Examining the second half of 5.7, a very regular pattern of topic changes emerges after a gap (with, incidentally, no support from the TextTiling metric). In consulting with test subjects, it becomes clear that if they fail to find 'enough' topics they feel they must be mistaken and begin to look more closely. This is in contrast with the system itself, which uses an absolute metric and is capable of assigning no topic breaks at all to some documents (for example figure 5.8).

5.2.4 Sparseness agreement

In contrast to section 5.2.3, figure 5.11 demonstrates a high level of agreement between the system and the annotator about the atypically sparse topic breaks. While the system only located half of the topic breaks determined by the annotator, they were clustered in the same area around the center of the document. The annotator professed concern at her inability to locate more topic changes (reinforcing the theory that users expect a certain rate of topic change *a priori*).

5.2.5 Annotator differences

As discussed in chapter 4, annotators were intentionally given only the instruction to “mark wherever the topic changes” within documents. As a result, different annotators marked documents in slightly different ways: At one extreme, annotators marked only the start or end of large, clear, coherent topics. At the other extreme, some annotators made a mark every time a topic drifted in or out of immediate focus. For instance, in figure 5.13, all ‘disjoint’ discussion is partitioned individually, such as the *non-sequitur* utterance “I can go home now”, seen in figure 5.2.5. We can see here that the topic boundaries are used to *constrain* a topic amid a sea of noise—an assertion that the change from topic to no-topic is a topic change.

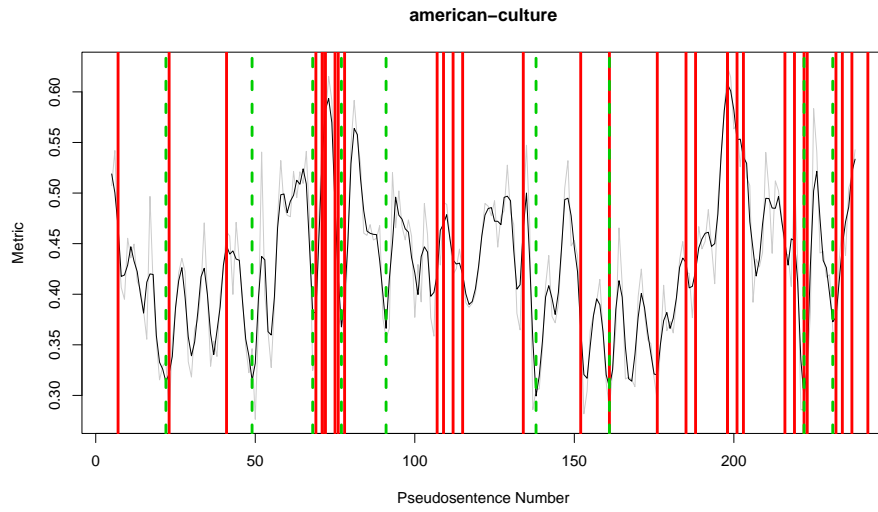


Figure 5.6: American Culture advising session, $ps = 20$ $bs = 6$

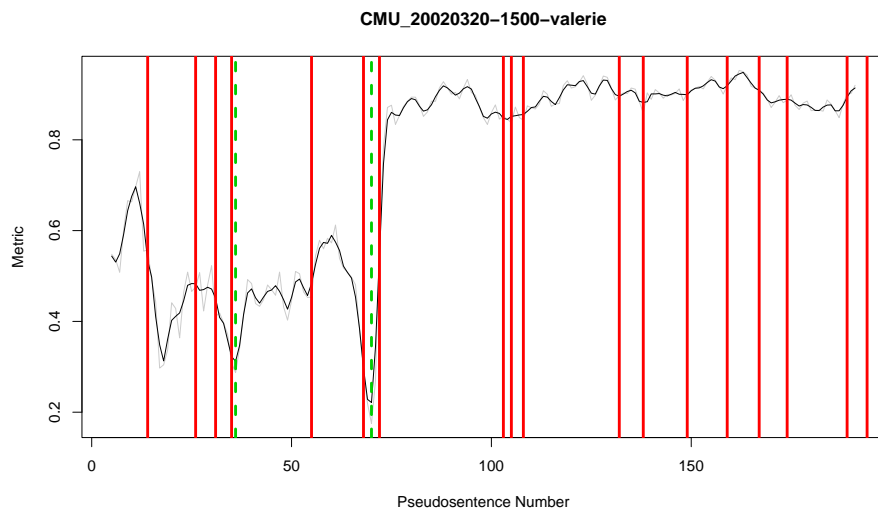


Figure 5.7: CMU dialogue, $ps = 20$ $bs = 6$

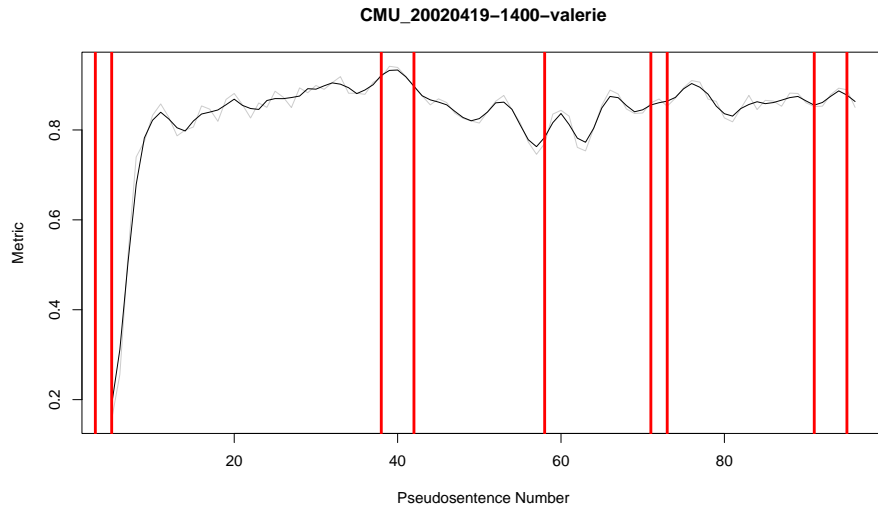


Figure 5.8: CMU dialogue 2, $ps = 20$ $bs = 6$

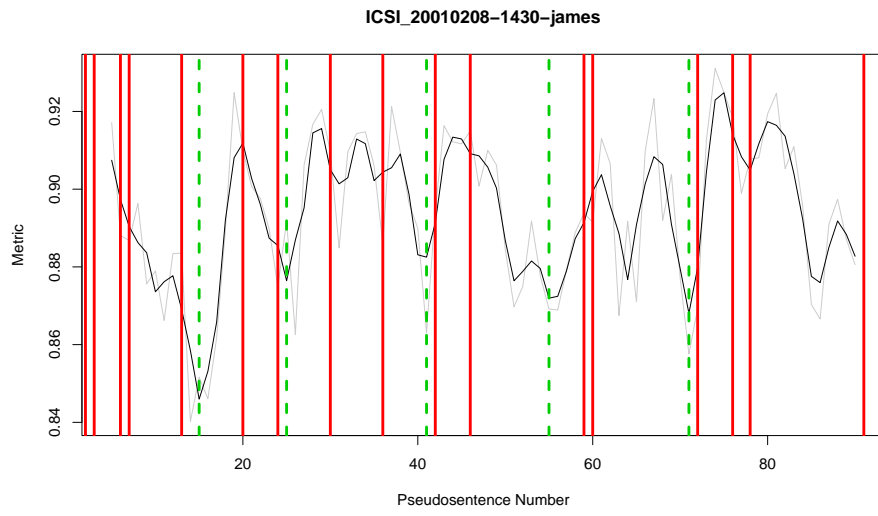


Figure 5.9: ICSI dialogue, $ps = 20$ $bs = 6$

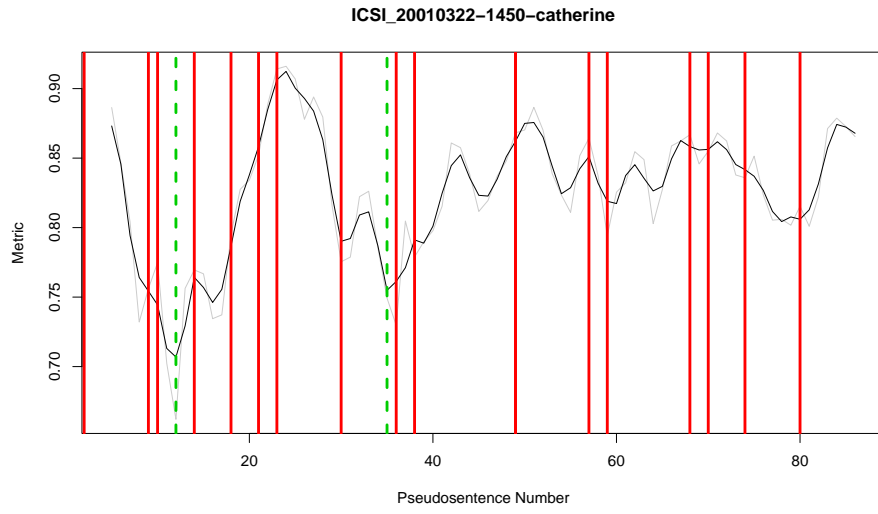


Figure 5.10: ICSI dialogue 2, $ps = 20$ $bs = 6$

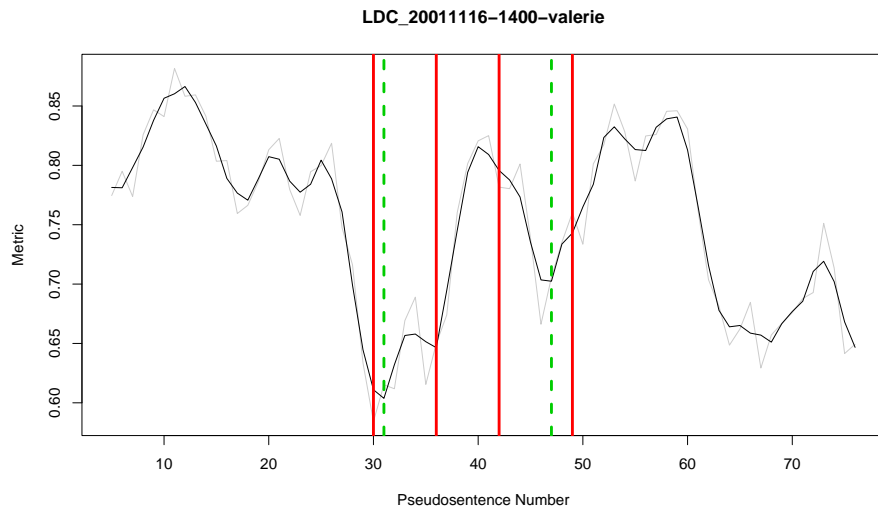


Figure 5.11: LDC dialogue, $ps = 20$ $bs = 6$

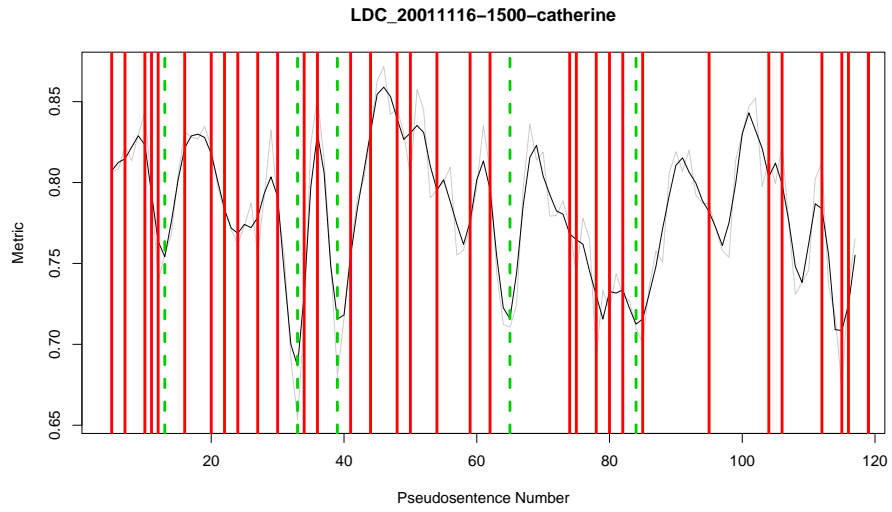


Figure 5.12: LDC dialogue 2, $ps = 20$ $bs = 6$

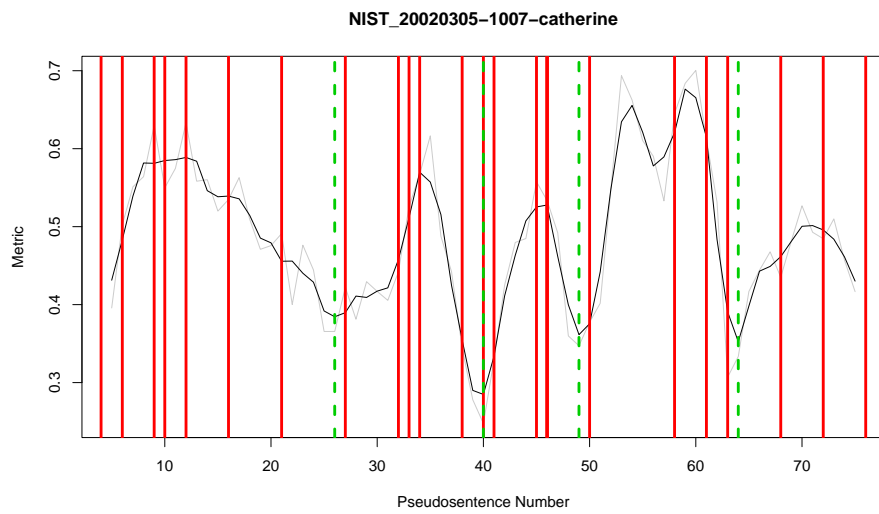


Figure 5.13: NIST dialogue, $ps = 20$ $bs = 6$

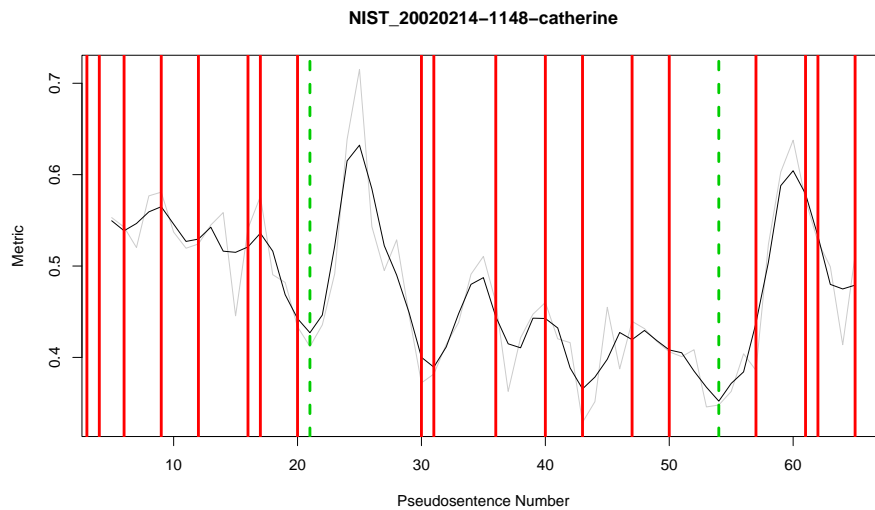


Figure 5.14: NIST dialogue 2, $ps = 20$ $bs = 6$

```

019: oh, that's a good idea
005: that's
003: you learn something new everyday
<break />
006: {mmm}
029: {breath} inhale
006: {laughing}
028: {laugh}
005: {laugh}
<break />
003: I can go home now
<break />
002: {laugh}
005: {exhale}
029: {laugh}
003: {laugh}
029: {breath}
006: {vocalization}
<break />

```

Figure 5.15: A snippet from figure 5.13's source dialogue, showing unexpected topic-boundary marking style

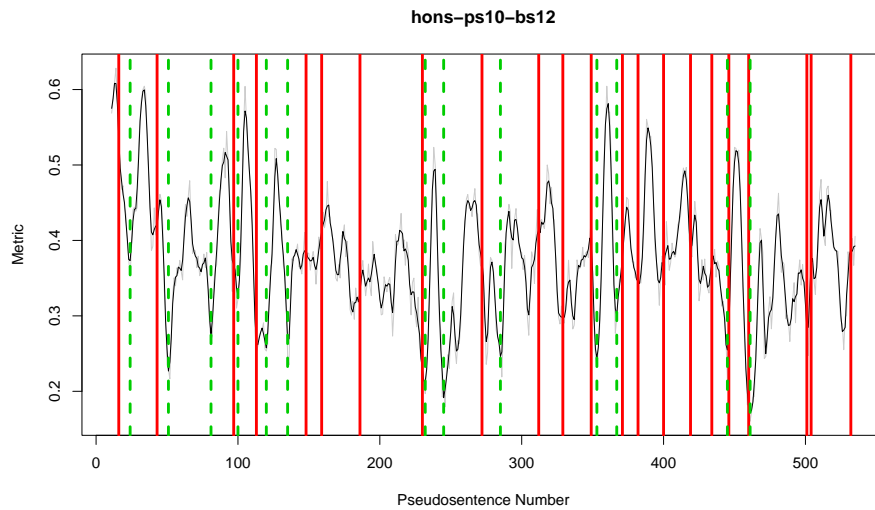


Figure 5.16: Honours Advising dialogue: $ps = 10$ $bs = 12$

5.3 Parameter tuning

Using the Honours Advising dialogue due to its interesting features (the existence of several convincing ‘hits’, and some human-annotated topic changes not even hinted at by the TextTiling metric), and using as a starting point Hearst’s default settings [6] of pseudosentence length=20, block size=6, experiments were performed to determine the optimal settings for these two parameters. Maintaining the total window size as an approximate constant, the pseudosentence length was first halved, as seen in figure 5.16. Then the same run was performed with the pseudosentence length doubled from its original size of 20, as seen in figure 5.17. While interesting, these results seemed to be universally worse than Hearst’s original settings, and so it was decided that the domain in this respect had no effect on parameter requirements.

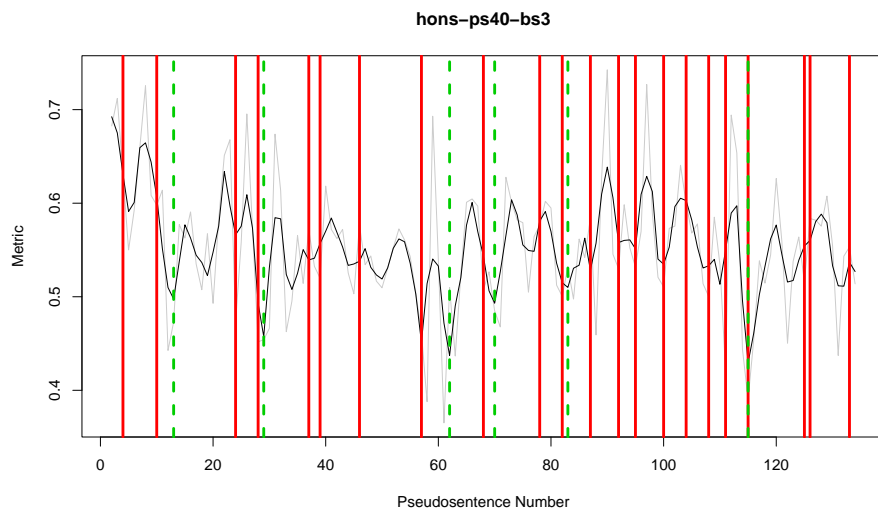


Figure 5.17: Honours Advising dialogue: $ps = 40$ $bs = 3$

Chapter 6

Conclusions

In the preceding chapters, it has been shown that a textual topic segmentation algorithm designed to work on structured, paragraph-segmented text can be made to work effectively for the domain of transcripts of spoken dialogue.

An exploration of the background research of topic segmentation and of spoken dialogue structure has been undertaken; and a series of controlled tests have been performed to examine the effectiveness of a customised version of the TextTiling [6] algorithm on spoken dialogue transcripts, using various kinds of spoken dialogue data and various algorithm-tuning parameters. The results of these experiments have been plotted and analysed

6.1 Observations

Given the difficulty encountered by human evaluators in determining topic changes within the evaluation documents, it is possible that it is not sensible to talk about ‘topics’ within a spoken dialogue as it is within written text. Because written text is laid out in a logical progression or exploration of ideas, its topics are *intentional* - the author wishes to convey several points, and addresses them in a sensible order. In contrast, while some structured meeting dialogues contain a series of ‘action items’, many spoken dialogues are unstructured, and comprise largely unplanned discussions concerning topics not critical (or even related) to the agenda of the speakers. It is possible, therefore, that spoken dialogue contains many sections which cannot reasonably be classified as ‘topics’ at all, at least in the same sense that Hearst [6] and Beeferman [1] use. When a conversation can be said to have ‘digressed’ from its original topic, it is not always clear that it has entered another.

6.1.1 Implications of false positives

When the system locates a potential topic change which is not supported by the human annotator, two possibilities occur: Either there is no break in topic,

in which case the system is wrong; or there is indeed a change in topic, in which case the human judge failed to identify it. Given Hearst’s experience [6] with lack of inter-annotator agreement, it is certainly conceivable that in these experiments, which used only a single human annotator for each of the test documents, human error exists. Hearst notes an interesting phenomenon during the course of her experiment: The final paragraph of her test document, marked as a new topic by the system, was a summary of the whole article—a fair place to mark a new topic, she argues. However, only two of her seven judges chose to mark it themselves.

Assuming Hearst’s assertion has merit and that the system was correct, against the collective weight of the human judges, this raises interesting questions about the false positives located in this research: It is quite likely that at least *some* of them represent valid topic changes not located by the human annotators.

6.1.2 Inter-annotator agreement

Of the four annotators who volunteered for this project, each seemed to have a slightly different pattern of topic marking. Number of topics located per page varied, as did philosophies of where exactly a fading topic ends. However, as each document was segmented only once, this cannot be analysed.

One annotator (not a computational linguist) stated that she began to develop a pattern for locating topic change: Essentially using cue-phrases such as “so” and “anyway” as a way to add accuracy to her location of topic changes. This is interesting as it recalls strongly some computational methods for topic segmentation, and also highlights the subjectivity and difficulty of detecting topic change by true understanding of the discourse flow.

6.2 Further work

This project opens a large number of possibilities both for further research and for applications:

Perhaps the most important next stage of evaluation is to expand the number of human evaluators in order to perform inter-annotator agreement tests on the scale of [6] and [15].

Expansion of the system from a word-frequency-only approach to a combined approach making use of cue-phrases would also be useful: Unlike TextTiling’s native domain, cue-phrases are ubiquitous in spoken dialogue, and as shown in [7] can be effective in locating potential topic changes.

A useful expansion of the current system in terms of applicability to real-world situations would be the ability to label detected topics by their content, as performed in [3]. This would allow such applications as web-based meeting transcript browsers, allowing users to scan down a list of discussed topics to index directly to the one they seek.

Bibliography

- [1] BEEFERMAN, D., BERGER, A., AND LAFFERTY, J. D. Statistical models for text segmentation. *Machine Learning* 34, 1-3 (1999), 177-210.
- [2] BERGER, A. L., PIETRA, S. D., AND PIETRA, V. J. D. A maximum entropy approach to natural language processing. *Computational Linguistics* 22, 1 (1996), 39-71.
- [3] BOGURAEV, B., KENNEDY, C., BELLAMY, R., BRAWER, S., WONG, Y., AND SWARTZ, J. Dynamic presentation of document content for rapid on-line skimming, 1998.
- [4] CORE, M. G., AND ALLEN, J. F. Coding dialogues with the DAMSL annotation scheme. In *Working Notes: AAAI Fall Symposium on Communicative Action in Humans and Machines* (Menlo Park, California, 1997), D. Traum, Ed., American Association for Artificial Intelligence, pp. 28-35.
- [5] GROSZ, B. J., AND SIDNER, C. L. Attention, intentions, and the structure of discourse. *Computational Linguistics* 12, 3 (1986), 175-204.
- [6] HEARST, M. Multi-paragraph segmentation of expository text. In *32nd. Annual Meeting of the Association for Computational Linguistics* (New Mexico State University, Las Cruces, New Mexico, 1994), pp. 9-16.
- [7] HIRSCHBERG, J., AND LITMAN, D. J. Empirical studies on the disambiguation of cue phrases. *Computational Linguistics* 19, 3 (1993), 501-530.
- [8] KASPER, R. T., DAVIS, P. C., AND ROBERTS, C. An integrated approach to reference and presupposition resolution.
- [9] KOZIMA, H. Text segmentation based on similarity between words. In *Meeting of the Association for Computational Linguistics* (1993), pp. 286-288.
- [10] KOZIMA, H., KOZIMA, A., AND TEIJI, H. Similarity between words computed by spreading activation on an english dictionary, 1993.
- [11] LITMAN, D. J., AND PASSONNEAU, R. J. Combining multiple knowledge sources for discourse segmentation. In *Meeting of the Association for Computational Linguistics* (1995), pp. 108-115.

- [12] MIDGLEY, T. D., AND MACNISH, C. Automatic dialogue segmentation using discourse chunking. In *16th Australian Joint Conference on Artificial Intelligence (AI'03)* (2003), pp. 772–782.
- [13] MITTAL, V. O., KANTROWITZ, M., GOLDSTEIN, J., AND CARBONELL, J. G. Selecting text spans for document summaries: Heuristics and metrics. In *AAAI/IAAI* (1999), pp. 467–473.
- [14] MUEHLER, G., AND MAYER, J. A method for the analysis of prosodic registers.
- [15] PASSONNEAU, R. J., AND LITMAN, D. J. Intention-based segmentation: Human reliability and correlation with linguistic cues. In *Meeting of the Association for Computational Linguistics* (1993), pp. 148–155.
- [16] PONTE, J. M., AND CROFT, W. B. Text segmentation by topic. In *European Conference on Digital Libraries* (1997), pp. 113–125.
- [17] RAO, R., PEDERSEN, J. O., HEARST, M. A., MACKINLAY, J. D., CARD, S. K., MASINTER, L., HALVORSEN, P.-K., AND ROBERTSON, G. G. Rich interaction in the digital library. *Communications of the ACM* 38, 4 (1995), 29–39.
- [18] STARK, H. A. What do paragraph markings do? *Discourse Processes* 11 (1988), 275–303.
- [19] STOLCKE, A., AND SHRIBERG, E. Automatic linguistic segmentation of conversational speech. In *Proc. ICSLP '96* (Philadelphia, PA, 1996), vol. 2, pp. 1005–1008.
- [20] WAIBEL, A., BETT, M., FINKE, M., AND STIEFELHAGEN, R. Meeting browser: Tracking and summarizing meetings. In *Proceedings of the Broadcast News Transcription and Understanding Workshop* (Lansdowne, Virginia, February 1998), D. E. M. Penrose, Ed., Morgan Kaufmann, pp. 281–286.
- [21] WILCOX, L., AND BORECZKY, J. S. Annotation and segmentation for multimedia indexing and retrieval. In *HICSS (2)* (1998), pp. 259–266.

Appendix A

Code listings

A.1 segmentation.py

```
#!/usr/bin/python

from optparse import OptionParser
import micaseparse

#this stuff all for micaseparse - should be imported inside
#micaseparse?
from xml.sax import make_parser
from xml.sax.handler import feature_namespaces
import micaseparse

#set up the command-line arguments:
argparser = OptionParser()

#default value for 'breaksfile' is empty - most of the time we
#don't have one.
breaksfile = ""

argparser.add_option("-n", "--numbers", action="store_true",
    dest="printsimmums", default=False)
argparser.add_option("-b", "--handbreaks", action="store_true",
    dest="printheadbreaks", default=False)
argparser.add_option("-m", "--markuppseudo", action="store_true",
    dest="printpseudomarked", default=False)
argparser.add_option("-i", "--insertbreaks", action="store",
    type="string", dest="breaksfile")
argparser.add_option("-s", "--smoothdata", action="store_true",
```

```

    dest="smoothdata", default=False)
argparser.add_option("-d", "--detectbreaks", action="store_true",
    dest="detectbreaks", default=False)
argparser.add_option("-f", "--file", action="store", type="string",
    dest="filename")

(options, args) = argparser.parse_args()

#globals:
smooth_size = 3
percent_detection = 35

parser = make_parser()

#"not interested in namespaces"
parser.setFeature(feature_namespaces, 0)

#create the handler:
dh = micaseparse.FindUtterances(options.breaksfile)

#register the handler with the parser:
parser.setContentHandler(dh)

parser.parse(options.filename)
dh.cleanupParse()

dh.getWords()
dh.getFrequencies()

#If you want graphable output:
if options.printsimnums:
    dh.recordBlockSim()

#If you want to see the entirety of all words nabbed, one per line:
#dh.printAllWords()

#if you want to print self.words for debugging:
#dh.printWords()

```

```

#If you want all the pseudosentences with numbering:
#dh.printWithNumbers()

#If you want a plottable file containing all the pseudosentences
#with manual breaks in them:
if options.printhandbreaks:
    dh.recordManualBreaks()

#If you want a sorted list of the most frequent words:
#dh.getFrequencies()
#dh.printFrequencies()

#If you want the original conversation XML (more or less) with
#<pseudosentence number=n /> tags:
#If you want the same conversation XML with breaks in it, use -i.
if options.printpseudomarked:
    dh.printNumberedDialogue()

if options.smoothdata:
    dh.recordBlockSimSmoothed(smooth_size)

if options.detectbreaks:
    #options are percentage detection, smoothing value<must be odd>
    dh.recordAutoBreaks(percent_detection, smooth_size)

```

A.2 micaseparse.py

```

#!/usr/bin/python

from xml.sax import saxutils
import re
import sys
import cosinemeasure
import peakpick

#CONSTANTS:
# words that aren't used:
stop_words = ['and', 'the', 'but', 'a', 'an', 'how', 'why',
              'be', 'uh', 'um', 'mhm', 'i', 'you', 'we',
              'they', 'that', 'how', 'what', 'which', 'how',
              'of', 'is', 'to', 'in', 'it', 'you\'re', 'for',
              'if', 'have', 'like', 'really', 'are', 'mhm',
              'so', 'about']

```

```

# size of the chunks of words compared ('pseudosentences'):
pseudosentence_size = 20

# 'blocksize' - number of pseudosentences in a pseudoparagraph;
# This is a moving window sort of affair:
block_size = 6

use_stop_word_list = 1

# how many values over which to smooth:
#smooth_size = 30

#a print function for mapping across lists of tuples:
def printtup(a):
    print a[1], ":",
    print a[0]

#a sorting function to compare tuples:
def sortFreqs(a, b):
    if a[1] < b[1]:
        return 1
    elif a[1] > b[1]:
        return -1
    else:
        return 0

class PseudoSentence:
    def __init__(self):
        self.words = []
        self.freqslist = []

    def add(self, singleWord):
        self.words.append(singleWord)

    def clear(self):
        self.words = []

    def size(self):
        return len(self.words)

    def report(self):
        print "Pseudosentence \"", self.words[0], "...\" - ", \
              self.size(), " words."
        print "Freqslist size is:", len(self.freqslist), \
              ", and its sum is", reduce(lambda x, y: x + y,

```

```

        self.freqslist)

def calculateFreqs(self, allwords):
    #print "trying to calculate freqs, size of allwords is", \
        len(allwords)
    for word in allwords:
        self.freqslist.append(self.words.count(word))
        #print "frequency of", word, "is", \
            self.words.count(word)
    #for ftup in zip(allwords, self.freqslist): print ftup[0], \
        ":", ftup[1]

def prettyprint(self):
    print '''
    for word in self.words:
        print word,
    print '''

class FindUtterances(saxutils.DefaultHandler):
    def __init__(self, breaksfile):
        self.inUtterance = 0
        self.body = ""
        self.chunks = []
        self.wordFreqs = {}
        self.words = []
        self.utterance = ""

        #It may occur that the input file has manually marked-up breaks
        #for learning and evaluation. If so, their pseudosentence numbers
        #will be in this list.
        self.manualBreaks = []

        #Holder for the possible input from 'breaksfile', filled
        #below in this method.
        self.autoBreaks = []

        #The original dialogue, plus PS breaks.
        self.dialogueXML = ""

        #The current, not-yet-full pseudosentence
        self.currentChunk = PseudoSentence()

        #remove useless attached punctuation

```

```

self.punc = re.compile('[.,\(\)]')

#philosophical assumption that questionmarks can indicate topic.
self.question = re.compile('\?')

#removing "so then-" without removing "lexis-nexus". Not sure
#about @eol.
self.interrupt = re.compile('(?!<=\\S)[-_](?!\\S)')

# Read in the breaks file (-i option) if it exists.
if breaksfile:
    print "breaksfile is", breaksfile, "."
    f=open(breaksfile, 'r')
    self.autoBreaks = f.readlines()
    #remove the head, which is a comment
    self.autoBreaks.pop(0)
    #lose trailing \n:
    self.autoBreaks = map(lambda x: x.strip(), self.autoBreaks)
else:
    sys.stderr.write("Informational: No breaksfile name
        provided, none used.\n")

def startElement(self, name, attrs):
    self.utterance = self.utterance + " "
    if name == "break":
        #put the current pseudosentence number (first==0)
        #into self.manualBreaks:
        self.manualBreaks.append(len(self.chunks))
    if name == "U1" or name == "U2" or name == "U":
        self.dialogueXML += "<" + name + " WHO=\"" + \
            attrs.get('WHO') + "\">"
    #for things that have an opening and closing tag:
    if name == "OVERLAP1" or name == "OVERLAP2" or name == "OVERLAP":
        self.dialogueXML += "<" + name + ">"
    #for monads (single tags with no corresponding closer):
    if name == "break":
        self.dialogueXML += "<" + name + " />"
    #we accept every element, regardless of speaker:
    #is this a bug? how many Un are there?
    #Update: Seems only two, from the DTD.
    if name != "U1" and name != "U2" and name != "U": return
    self.inUtterance = self.inUtterance + 1

def endElement(self, name):
    self.utterance = self.utterance + " "

```

```

if name == "U1" or name == "U2" or name == "U" or name == \
    "OVERLAP1" or name == "OVERLAP2" or name == "OVERLAP":
    self.dialogueXML += "</"+name+">"
if name == "U1" or name == "U2" or name == "U":
    self.inUtterance = self.inUtterance - 1;

def characters(self, ch):
    if self.inUtterance>0:
        #Run regular expression(s) to clean input -
        #if you do it here, it affects the readable output as well.
        #Consider doing it in chunkBody.
        ch = self.question.sub(" ? ", ch)
        #Then split() and chunk current stream and put it
        #into self.chunks[]:
        self.chunkBody(ch)

def cleanupParse(self):
    if self.currentChunk.size() > 0:
        #self.currentChunk.calculateFreqs(self.words)
        #self.currentChunk.report()
        self.chunks.append(self.currentChunk)
        ##map(printtup, tempchunk)
    sys.stderr.write("Number of pseudosentences: %d\n" %
        len(self.chunks))

def printBody(self):
    print "self.body contains: ", self.body

def printAllWords(self):
    print "Words are:"
    for pseudosentence in self.chunks:
        for word in pseudosentence.words:
            print word

#create a collection of pseudosentences:
def chunkBody(self, currString):
    for word in currString.split():
        #Throw the stream onto our internal representation of the XML:
        self.dialogueXML += word + ' '
        #Run the regular expressions to clean the input:
        word = self.punc.sub("", word)
        word = self.interrupt.sub(" ", word)
        word = word.strip()
        if not word in stop_words or not use_stop_word_list:
            self.currentChunk.add(word)
            if self.currentChunk.size() == pseudosentence_size:

```

```

        self.chunks.append(self.currentChunk)
        self.currentChunk = PseudoSentence()
        self.dialogueXML += "<pseudosentence number='" + \
            repr(len(self.chunks)) + "' />"
        if len(self.autoBreaks) and repr(len(self.chunks)) \
            == self.autoBreaks[0]:
            self.dialogueXML += "<autobreak />"
            self.autoBreaks.pop(0)

#Build the frequency table (just the words
#that need to be in it, not their actual freqs)
def getWords(self):
    for pseudosentence in self.chunks:
        for word in pseudosentence.words:
            if self.words.count(word) == 0:
                self.words.append(word)
    #print "self.words size is", len(self.words)

#Build the lists of frequencies for each chunk/pseudosentence:
def getFrequencies(self):
    for pseudosentence in self.chunks:
        pseudosentence.calculateFreqs(self.words)
        #pseudosentence.report()

def printFrequencies(self):
    sortedFreqs = self.wordFreqs.items()
    sortedFreqs.sort(sortFreqs)
    map(printtup, sortedFreqs)

def square(self, d):
    return d**2

#same as printblocksim, printing every actual sentence comparison.
def printEverySim(self):
    print "#Blockgap\tSimilarity"
    for startchunk in range(len(self.chunks)-(block_size)):
        print "S1:",
        self.chunks[startchunk].prettyprint()
        print "S2:",
        self.chunks[startchunk+block_size].prettyprint()
        print "%d\t\t%f" % (startchunk,
            cosinemeasure.sim(self.getBlockFreqs(startchunk),
                self.getBlockFreqs(startchunk+1)))
    #print startchunk + "\t" + \
        cosinemeasure.sim(self.getBlockFreqs(startchunk), \

```

```

        self.getBlockFreqs(startchunk+1))

def printBlockSim(self):
    #per-block similarity:
    print "starting block similarity"
    for startchunk in range(len(self.chunks)-((block_size-1)*2)):
        print "Comparing sentences starting at chunk", startchunk, \
              "and", startchunk, ":"
        print "Similarity score is:", \
              cosinemeasure.sim(self.getBlockFreqs(startchunk), \
                                self.getBlockFreqs(startchunk+1))
        print ""

#Print out pseudosentence number of center of current value, smoothed:
def recordBlockSim(self):
    sims = self.getBlockSim()
    print "#Pseudosentence\tSmoothSimilarity"
    for value in sims:
        print "%d\t\t%f" % (value[0], value[1])
        #print startchunk + "\t" + \
              cosinemeasure.sim(self.getBlockFreqs(startchunk), \
                                self.getBlockFreqs(startchunk+1))

#Print out pseudosentence number of center of current value, smoothed:
def recordBlockSimSmoothed(self, smooth_size):
    sims = self.smooth(self.getBlockSim(), smooth_size)
    print "#Pseudosentence\tSmoothSimilarity"
    for value in sims:
        print "%d\t\t%f" % (value[0], value[1])
        #print startchunk + "\t" + \
              cosinemeasure.sim(self.getBlockFreqs(startchunk), \
                                self.getBlockFreqs(startchunk+1))

def getBlockSim(self):
    sims = []
    for startchunk in range(len(self.chunks)-((block_size-1)*2)):
        sims.append([startchunk+(block_size-1),
                    cosinemeasure.sim(self.getBlockFreqs(startchunk),
                                        self.getBlockFreqs(startchunk+(block_size-1)))]
    return sims

def getBlockSimSmoothed(self, smooth_size):
    return(self.smooth(self.getBlockSim(), smooth_size))

def smooth(self, pairs, smooth_size):
    smoothed = []

```

```

if smooth_size == 0:
    return pairs
for i in range(len(pairs)):
    total = 0.0
    for j in range(i-(smooth_size/2), i+(smooth_size/2)+1):
        if j<0:
            total += pairs[0][1]
        elif j>len(pairs)-1:
            total += pairs[len(pairs)-1][1]
        else:
            total += pairs[j][1]
    smoothval = total/smooth_size
    smoothed.append([pairs[i][0], smoothval])
return smoothed

#Print out gap number of current value, not pseudosentence position:
def recordBlockSimGapNumber(self):
    print "#Blockgap\tSimilarity"
    for startchunk in range(len(self.chunks)-((block_size-1)*2)):
        print "%d\t\t%f" % (startchunk, \
            cosinemeasure.sim(self.getBlockFreqs(startchunk), \
                self.getBlockFreqs(startchunk+(block_size-1))))
        #print startchunk + "\t" + \
            cosinemeasure.sim(self.getBlockFreqs(startchunk), \
                self.getBlockFreqs(startchunk+1))

def recordManualBreaks(self):
    print "#Pseudosentence-number"
    listfilled = 0
    for number in self.manualBreaks:
        print number
        listfilled = 1
    if not listfilled:
        print "0"

def recordAutoBreaks(self, percentage, smooth_size):
    #employ the peak detection algorithm
    sims = self.getBlockSimSmoothed(smooth_size)
    print "#PseudosentenceNumber"
    listfilled = 0
    for i in peakpick.pick(sims, percentage):
        print i
        listfilled = 1
    if not listfilled:
        print "0"

```

```

def getBlockFreqs(self, startpos):
    blockfreqs = self.chunks[startpos].freqslist
    for i in range(startpos+1, startpos + block_size):
        blockfreqs = map(lambda x: x[0]+x[1], zip(blockfreqs,
            self.chunks[i].freqslist))
    return blockfreqs

def printWithNumbers(self):
    #simply print each pseudosentence in turn:
    #maybe this should output an XML of sorts, for easier viewing?
    for i in range(len(self.chunks)):
        print "Pseudosentence", i, ":"
        self.chunks[i].prettyprint()

def printNumberedDialogue(self):
    #print out an XML-formatted document with turns in it,
    #with pseudosentence breaks marked:
    print "<TEXT>"
    print self.dialogueXML
    print "</TEXT>"

def printWords(self):
    #print self.words, one per line:
    for word in self.words:
        print word

```

A.3 cosinemeasure.py

```

#!/usr/bin/python

def dotproduct(a, b):
    return sum(map(product, zip(a, b)))

def product(a):
    return a[0]*a[1]

def square(d):
    return d**2

# compute similarity between lists of weightings:
def sim(b1, b2):
    # sigma t wt,b1 wt,b2:
    a = dotproduct(b1, b2)

```

```

# sigma t w2t,b1 sigmafrom t=1 to n w2t,b2:
b = ( (sum(map(square, b1))) * (sum(map(square, b2))) )**\
(1.0/2.0)
return a/b

```

A.4 peakpick.py

```

#!/usr/bin/python

#Python Peak Picker module
#james ballantine
#12/05/04

def simplepick(data):
    #takes in a list of values.
    #returns a list containing the indices where peaks exist.
    peaks = []
    for i in range(0+1, len(data)-1):
        if data[i]>data[i-1] and data[i] > data[i+1]:
            peaks.append(i)
    return peaks

def pick(datatuples, percentage):
    #refuses to notice peaks of less than a given percentage.
    #data is a list of tuples.
    index, data = map(lambda t: list(t), zip(*datatuples))
    peaks = []
    size = max(data) - min(data)
    threshold = size * (percentage * 0.01)
    highpos = 0
    lowpos = 0
    islive = 0
    for i in range(0, len(data)):
        if data[i] > data[highpos]:
            highpos = i
        if islive:
            if data[i] < data[lowpos]:
                lowpos = i
            elif data[i] > (data[lowpos]+threshold):
                #we have a peak (umm... trough)
                #print "peak:",lowpos, "(", data[lowpos], ")"
                peaks.append(index[lowpos])
                islive = 0 #?
                highpos = i
                lowpos = i

```

```

    elif data[i] <= (data[highpos]-threshold):
        islive = 1
        lowpos = i
    return peaks

```

A.5 twoplots.R

```

# generate a line plot for all .txt files in a directory
# files are two columns suitable for reading with read.table
# Batch file to be called with:
# R BATCH plotindir.R output.Rout

files <- dir(".", "\.txt$")

for( f in files ) {
  # generate filename.ps
  name <- substring(f, 0, nchar(f)-4)
  psfilename <- paste(name, "ps", sep=".")

  cat("Processing ", f, " to ", psfilename, "\n")
  # open a postscript file
  postscript(file=psfilename, title=f, horizontal=FALSE,
    paper="special", width=10, height=6)

  data <- read.table(f)
  # do any work on data here, eg remove values == 1
  # Remove ceiling values:
  #tmp <- data[,2] == 1.0
  #data <- data[!tmp,]

  #smooth stuff:
  #data[,2] <- smooth(data[,2])
  if( file.exists(paste(f, "spiky", sep="."))){
    spikydata <- read.table(paste(f, "spiky", sep="."))
    plot(spikydata, type="l", xlab="Pseudosentence Number",
      ylab="Metric",
      main=name, col=grey(0.8))
    lines(data)
  }else{
    plot(data, type="l", xlab="Pseudosentence Number",
      ylab="Metric",
      main=name, col=1)
  }
}

```

```

if( file.exists(paste(f, "breaks", sep="."))){
  breaks <- read.table(paste(f, "breaks", sep="."))
  for( b in breaks ) {
    abline(v=b, col=2, lwd=3)
  }
}

if( file.exists(paste(f, "auto", sep="."))){
  auto <- read.table(paste(f, "auto", sep="."))
  for( b in auto ) {
    abline(v=b, col=3, lty=2, lwd=3)
  }
}

# close the postscript file
dev.off()
}

```

A.6 newmicase-html-pseudosentences-breaks.xsl

```

<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:fo="http://www.w3.org/1999/XSL/Format"
xmlns="http://www.w3.org/1999/xhtml">
  <!--This stylesheet is for identifying speakers in the
micase corpus -->
  <xsl:output method="xml" encoding="ISO-8859-1" indent="yes"
doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
version="1.0" />
  <xsl:template match="TEXT">
    <html>
      <head>
        <title>A MICASE Conversation</title>
        <link rel="stylesheet" href="speakers.css"
type="text/css" title="Colours"/>
        <link rel="stylesheet" href="speakers-print.css"
type="text/css" title="Printable"/>
      </head>

```

```

        <body>
            <h1>A MICASE Conversation</h1>
            <xsl:apply-templates />
        </body>
    </html>
</xsl:template>
<xsl:template match="pseudosentence">
    <span class="newsentence">
        <xsl:value-of select="@number"/>
    </span>
</xsl:template>
<xsl:template match="autobreak">
    <span class="autobreak">|</span>
</xsl:template>
<xsl:template match="U">
    <p>
        <xsl:attribute name="class" >
            <xsl:value-of select="@WHO"/>
        </xsl:attribute>
        <strong><xsl:value-of select="@WHO"/>: </strong>
        <xsl:apply-templates />
    </p>
</xsl:template>
<xsl:template match="U/U">
    <span>
        <xsl:attribute name="class" >
            <xsl:value-of select="@WHO"/>
        </xsl:attribute>
        <xsl:apply-templates />
    </span>&#160;
</xsl:template>
<xsl:template match="OVERLAP">
    <span class="overlap">
        <xsl:apply-templates />
    </span>&#160;
</xsl:template>
<xsl:template match="TEIHEADER" />
</xsl:stylesheet>

```

A.7 makegraph.sh

```

#!/bin/bash
#1: file to work on
#2: name for outputs

```

```

#unsmoothed output:
~/workspace/thesis-all/code/python/segmentation.py \
  -f $1 -n > $2.txt.spiky
#smoothed output:
~/workspace/thesis-all/code/python/segmentation.py \
  -f $1 -s > $2.txt
#hand breaks output:
~/workspace/thesis-all/code/python/segmentation.py \
  -f $1 -b > $2.txt.breaks
#autodetected breaks output:
~/workspace/thesis-all/code/python/segmentation.py \
  -f $1 -d > $2.txt.auto

```

A.8 buildgraph.sh

```

#!/bin/bash

echo "spiky:"
R BATCH ~/workspace/thesis-all/code/r/spiky.R
echo "spiky+smooth:"
R BATCH ~/workspace/thesis-all/code/r/spikysmooth.R
echo "spiky+smooth+auto:"
R BATCH ~/workspace/thesis-all/code/r/spikysmoothauto.R
echo "(spiky)+smooth+auto+breaks:"
R BATCH ~/workspace/thesis-all/code/r/twoplot.R

```

A.9 speakers.css

```

p {color: rgb(127,127,127); border-style: solid}
S1 {color: rgb(0,0,100)}
S2 {color: rgb(0,100,0)}
S3 {color: rgb(100,0,0)}
S4 {color: rgb(80,80,0)}
S5 {color: rgb(0,80,80)}
S6 {color: rgb(80,0,80)}

p.S1 {color: rgb(0,0,100)}
p.S2 {color: rgb(0,100,0)}
p.S3 {color: rgb(100,0,0)}
p.S4 {color: rgb(80,80,0)}
p.S5 {color: rgb(0,80,80)}
p.S6 {color: rgb(80,0,80)}

span.S1 {color: rgb(0,0,100); font-style: italic}

```

```
span.S2 {color: rgb(0,100,0); font-style: italic}
span.S3 {color: rgb(100,0,0); font-style: italic}
span.S4 {color: rgb(80,80,0); font-style: italic}
span.S5 {color: rgb(0,80,80); font-style: italic}
span.S6 {color: rgb(80,0,80); font-style: italic}
/*span.overlap {color: rgb(50,0,0)}
   overlaps not really interesting.*/
span.newsentence {color: red}
span.autobreak {color: red; font-size:275%}

strong {font-size:175%}
```

A.10 speakers-print.css

```
span.S1 { font-style: italic}
span.S2 { font-style: italic}
span.S3 { font-style: italic}
span.S4 { font-style: italic}
span.S5 { font-style: italic}
span.S6 { font-style: italic}
/*span.overlap {color: rgb(50,0,0)}
   overlaps not really interesting.*/

strong {font-size:175%}
```