

# Bericht zum Hauptseminarprojekt „Textsegmentierung“

Armin Schmidt  
HS Automatisches Textverstehen (Dr. M. Strube), WS 2008/09  
Seminar für Computerlinguistik, Universität Heidelberg

10. Februar 2009

## Zusammenfassung

In diesem Projektbericht soll der Frage nachgegangen werden, wie gut sich etablierte Textsegmentierungsalgorithmen auf „echte“ Daten anwenden lassen. Hierzu wurde beispielhaft der Block-Comparison-Algorithmus (Hearst, 1997) nachimplementiert und auf Wikipediadaten ausgewertet. Die Implementierung sowie die Daten und deren Vorverarbeitung werden beschrieben. Wie zu sehen sein wird, fallen die Ergebnisse für das Experiment sehr schlecht aus. Es soll gezeigt werden, dass dies an den Besonderheiten der Testdaten, an bestimmten Parametern des Algorithmus und dessen Implementierung, als auch an den Unzulänglichkeiten des verwendeten Evaluierungsmaßes liegt.

## 1 Überblick über die Implementierung

Als Vorlage für die Implementierung wurde (Hearst, 1997) verwendet. Der dort beschriebene Block-Comparison-Algorithmus wurde im Detail nachvollzogen und in Java umgesetzt. BlockComparison teilt einen Text in Pseudosätze immer gleicher, festgelegter Länge, fasst eine vorher festgelegte Anzahl von Pseudosätzen in einen Block zusammen und berechnet einen Score, anhand dessen eine Aussage über die Kohäsion an der Grenze zweier Blöcke getroffen werden kann. Pseudosätze werden mit Hilfe einer Stoppwortliste von hochfrequenten Wörtern gereinigt, so dass sie möglichst nur noch Inhaltswörter (*closed-class items*, hauptsächlich Nomen, Verben und Adjektive bzw. Adverbien) enthalten. Zusätzlich merkt sich das Programm, welchem tatsächlichen Satz ein Pseudosatz entspricht<sup>1</sup>. Die Zuweisung eines Kohäsionsscores für die Grenze zwischen zwei Blöcken geschieht durch die Berechnung des normalisierten Skalarprodukts. Dabei wird ein Block als Vektor betrachtet, dessen Werte durch die Anzahl bestimmt wird, mit der jeweils eine bestimmte Wortform in ihm vorkommt. Nachdem die Kohäsionsscores für alle Blöcke berechnet worden sind, werden letztere um einen Pseudosatz verschoben und dieser Prozess iterativ durchgeführt, bis für alle Pseudosatzgrenzen bekannt ist, wie kohäsiv der Text an der entsprechenden Stelle ist. Das Ergebnis dieses Prozesses ist ein Vektor von Kohäsionsscores. Für alle Minima in diesem Vektor werden nun Tiefenscores (*depth scores*)

---

<sup>1</sup>Die Originalimplementierung von Hearst sieht vor, den Beginn eines Pseudosatzes der nächsten echten Satzgrenze zuzuordnen. Im Gegensatz dazu weist meine Umsetzung einen Pseudosatz demjenigen echten Satz zu, in dem er beginnt.

berechnet, die einer Art Wahrscheinlichkeit entsprechen, mit der an dieser Stelle ein neuer thematischer Abschnitt beginnt<sup>2</sup>. Wie auch in der Originalimplementierung wird auf dem Vektor von Kohäsionsscores Moving-Average-Smoothing<sup>3</sup> vor der Berechnung der Tiefenscores durchgeführt. Anhand letzterer können nun die Abschnittsgrenzen festgelegt werden. Entsprechend der Empfehlung von (Hearst, 1997) müssen dabei mindestens drei Pseudosätze zwischen zwei Abschnitten liegen. Ein Tiefenscore wird als Abschnittsgrenze gehandelt, wenn er über einem bestimmten Schwellwert (*cut-off*) liegt. Hearst (1997) empfiehlt zwei verschiedene Schwellwerte: den *Liberal Cutoff* (LC) und den *Conservative Cutoff* (HC), die sich beide aus Standardabweichung und Durchschnitt der Tiefenscores berechnen. LC büßt Präzision zugunsten besseren Recalls ein, während HC bessere Precisionwerte liefert, aber weniger Tiefenscores als echte Abschnittsgrenzen akzeptiert.

Es wurden außerdem zwei Baseline-Algorithmen implementiert: *SameSizeSegments* teilt einen Text in eine gegebene Anzahl gleichgroßer Abschnitte ein. *RandomBoundaries* weist eine vorgegebene Anzahl von Segmentgrenzen zufällig zu. Alle drei Algorithmen erwarten ein Java-Objekt vom Typ *Article*, welches mit einem Multi-Abschnitt-Dokument initialisiert wird. Abschnittsgrenzen sind im Dokument gekennzeichnet und werden von der *Article*-Klasse gespeichert, von den Algorithmen jedoch nicht genutzt. Die Klasse *Evaluation* erwartet eine Liste mit den vom jeweiligen Algorithmus gelieferten und eine Liste mit den tatsächlichen Abschnittsgrenzen und ermittelt entsprechend Precision, Recall und F-Maß. Das Programm kann demnächst als quelloffener Code unter <http://armin.diotavelli.net> heruntergeladen werden.

## 2 Wikipedia-Artikel als Testdaten

### 2.1 Motivation

Die Auswertung von Textsegmentierungsalgorithmen wird oft auf eher künstlichen Daten vorgenommen. Choi (2000), beispielsweise, extrahiert eine bestimmte Anzahl von sogenannten „Segments“, d.h. Abschnitten festgelegter Länge am Anfang eines Dokuments, aus dem Brown-Korpus und konkateniert diese. Dabei ignoriert er sowohl, dass der thematische Unterschied zwischen zwei Artikeln wesentlich größer sein kann als der zwischen den topikalen Abschnitten innerhalb *eines* Textes, als auch den Umstand, dass die Anfänge von Dokumente oft aus Einleitung bestehen, welche aufgrund ihrer überblicksartigen Natur weniger kohäsiv sind als Abschnitte weiter hinten im Dokument. Auch wenn sich mehrere verschiedene Ansätze anhand dieses künstlichen Testkorpus vergleichen lassen, ist es schwer, eine Voraussage für ihr Verhalten auf anderen, natürlichen Texten zu treffen.

Artikel aus der Wikipedia scheinen sich hervorragend für die Aufgabe zu eignen, Textsegmentierungsalgorithmen auszuwerten: Wikipedia ist groß genug, um genügend brauchbares Datenmaterial herzugeben, sie enthält „echte“, natürliche Sprache und ihre Artikel sind oft in Abschnitte, d.h. innerthematische, kohäsive Einheiten untergliedert, deren Anfänge von Überschriften gekennzeichnet sind. Abschnitte und Unterabschnitte sind dabei anhand des jeweiligen WikiMedia-Markups unter-

---

<sup>2</sup>Intuitiv liegt eine Abschnittsgrenze dort vor, wo die Kohäsion am geringsten ist.

<sup>3</sup>In meiner Implementierung beträgt die Weite der Average-Smoothing-Methode  $s = 2$ .

scheidbar: ein String umgeben von '==' bezeichnet einen Abschnittsüberschrift, ein solcher umgeben von '===' eine Unterabschnittsüberschrift, usw.

## 2.2 Vorverarbeitung

Für die Experimente habe ich mithilfe der Java-Bibliothek JWPL (Zesch et al., 2008) zufällig 1000 Artikel aus einem MySQL-Dump der englischen Wikipedia extrahiert, die nach der Reinigungsphase 1. mindestens 1000 Zeichen enthalten und 2. aus mindestens zwei nicht-leeren Abschnitten bestehen. Die englische Version der Wikipedia wurde benutzt, da die für den BlockComparison-Algorithmus notwendigen Satzsplitter nur für die englische Sprache vorlagen. In der Reinigungsphase wurden folgende Schritte durchgeführt:

1. Entfernen von Hyperlinks. Wikipedia-interne Links wurden durch ihren Ankertext ersetzt.
2. Entfernen von Boxen und Tabellen.
3. Entfernen von wikipediaspezifischem XML-Markup.
4. Entfernen von Zeilen, die der wikipediainternen Weiterverarbeitung dienen.
5. Entfernen von Abschnitten, die normalerweise keine natürlichsprachigen Sätze enthalten, d.h. solche überschrieben mit z. B. „References“, „Links“, „See also“, etc.

## 3 Experimente und Auswertung

Sowohl BlockComparison als auch die beiden Baselines wurden jeweils auf alle 1000 Dokumente angewendet. Die Evaluierung fand mit Bezug auf zwei Vergleichssegmentierungen statt:

1. Unterteilung in alle Arten von Abschnitten und Unterabschnitten.
2. Unterteilung nur in Hauptabschnitte.

BlockComparison wurde außerdem einmal mit dem *Liberal Measure* (LC) und einmal mit dem *Conservative Measure* (HC) durchgeführt. In allen Durchläufen verwendete BlockComparison die Defaultparameter (Blockgröße: 10, Pseudosatzlänge: 20). Beide Baselinealgorithmen trennen anhand einer vorher festgelegten Anzahl von erwarteten Abschnitten. Um hier eine sinnvolle Wahl zu treffen, habe ich die durchschnittliche Anzahl von Abschnitten in allen 1000 Artikeln berechnet. Demnach besteht ein Artikel aus durchschnittlich vier Abschnitten unabhängig davon, ob es sich bei ihnen um Haupt- oder Unterabschnitte handelt, und aus durchschnittlich drei Hauptabschnitten.

	Presicion	Recall	F-Measure
alle Abschnitte	0.068	0.039	0.044
Hauptabschnitte	0.068	0.038	0.044

Tabelle 1: Ergebnisse BlockComparison (Conservative Measure)

	Presicion	Recall	F-Measure
alle Abschnitte	0.067	0.039	0.044
Hauptabschnitte	0.067	0.039	0.044

Tabelle 2: Ergebnisse BlockComparison (Liberal Measure)

	Presicion	Recall	F-Measure
alle Abschnitte	0.11	0.14	0.11
Hauptabschnitte	0.12	0.1	0.1

Tabelle 3: Ergebnisse SameSizeSegment

	Presicion	Recall	F-Measure
alle Abschnitte	0.11	0.14	0.11
Hauptabschnitte	0.12	0.1	0.1

Tabelle 4: Ergebnisse RandomBoundary

## 3.1 Fehlerquellen

### 3.1.1 Beinah-Verfehlungen

Ein näherer Blick auf die Ergebnisse enthüllt einige Ursachen für die schlechten Ergebnisse. Einerseits erzeugt der Algorithmus viele Beinah-Verfehlungen (*near-misses*), d. h. die errechnete Segmentgrenze hat die tatsächliche Grenze um einen oder wenige Sätze verfehlt. Hier zeigen sich die Schwächen der Metriken Precision und Recall (siehe Pevzner & Hearst (2002)). Gerade Probleme wie Beinah-Verfehlungen können von Maßen wie WindowDiff (Pevzner & Hearst, 2002) angemessener behandelt werden<sup>4</sup>(siehe dazu auch die Erläuterungen in Abschnitt 3.1.4).

### 3.1.2 Kurze Abschnitte

Ein weiteres Problem liegt in den Daten selbst. Bei der Vorverarbeitung wurden zwar nach der Reinigungsphase leere Abschnitte sowie allzu kurze Artikel aussortiert. Nicht geprüft wurde jedoch auf die Frage, ob Abschnitte ebenfalls eine Mindestgröße besitzen. Oftmals bestehen gerade Einleitungsabschnitte (sog. *lead sections*) aus nur einem oder zwei Sätzen. Auch können Abschnitte nach der Reinigung noch wenig Text enthalten, beispielsweise dann, wenn die den Abschnitt füllende Tabelle kurz erläutert werden soll.

### 3.1.3 Fragliche Annotation

Wikipedianer sind keine Annotierer und setzen andere Maßstäbe an eine Gliederung. So werden beispielsweise längere Artikel in Abschnitte untergliedert um die Lesbarkeit zu erhöhen. Bei kurzen Artikeln wird oftmals keine Untergliederung vorgenommen, obwohl diese aus Sicht einer thematischen Segmentierung sinnvoll wäre. Die Frage danach, was einen „Abschnitt“ ausmacht, kommt hier voll zum Tragen. Auf die Schwierigkeit der Aufgabe, ausgedrückt durch den Kappa-Koeffizienten über mehrere manuelle Annotationen, hat bereits Hearst (1997) hingewiesen. Schaut man sich einige Stichproben der Segmentierungen des BlockComparison-Algorithmus an, stellt man fest, dass die Einteilungen durchaus sinnvoll erscheinen, aber nicht mit der Annotierung übereinstimmen.

### 3.1.4 Besonderheiten der Implementierung

BlockComparison verwendet einige Parameter, die variabel sind, z.B. Blockgröße, Pseudosatzlänge, etc. Ich bin in meiner Implementierung den Empfehlungen von Hearst gefolgt und habe die Versuche lediglich mit den Default-Parametern durchgeführt. In der Nachbearbeitung dieses Projekts könnte versucht werden, verschiedene Parameter anzupassen, um bessere Ergebnisse zu erzielen.

Für die Größe des Smoothingfensters  $s$ (Abschnitt 1) gibt es auch bei Hearst keine Empfehlung, so dass ich hier eine eher zufällige Auswahl treffen musste. Auch hier könnte durch experimentelle Anpassung des Parameters eine bessere Performanz erzielt werden. Der einzige Punkt, in dem meine

---

<sup>4</sup>Aufgrund fehlender Zeit habe ich WindowDiff nicht implementieren können. Dies, wie auch die Berechnung des Anteils von Fehlern wie Beinahverfehlungen kann und sollte in der Nachbearbeitung dieser Arbeit nachgeholt werden.

Umsetzung von BlockComparison von der Beschreibung in Hearst (1997) abweicht, liegt darin, dass eine Pseudosatzgrenze nicht der am nächsten gelegenen echten Satzgrenze zugewiesen wird, sondern immer der vorherigen. Der per Augenmerk festgestellte große Anteil von Beinahverfehlungen legt nahe, dass hier eine potentielle Fehlerquelle liegen könnte.

Schlussendlich ist festzustellen, dass der verwendete Satzsplitter (LingPipe<sup>5</sup>) nicht perfekt ist und bei Sätzen, die ungewöhnliche Punktierung verwenden, wie sie in Wikipediaartikeln jedoch oft genug vorkommt, eine niedrige Akkuratheit zu haben scheint. Auch die von mir verwendete Stoppwortliste (ca. 500 Einträge) ist sicherlich nicht vollständig und weist einige Fehler auf.

## Literatur

- Choi, Freddy Y. Y. (2000). Advances in domain independent linear text segmentation. In *Proceedings of the 1st Conference of the North American Chapter of the Association for Computational Linguistics*, Seattle, Wash., 29 April – 3 May 2000, pp. 26–33.
- Hearst, Marti A. (1997). TextTiling: Segmenting text into multi-paragraph subtopic passages. *Computational Linguistics*, 23(1):33–64.
- Pevzner, Lev & Marti Hearst (2002). A critique and improvement of an evaluation metric for text segmentation. *Computational Linguistics*, 28(1):19–36.
- Zesch, Torsten, Christof Müller & Iryna Gurevych (2008). Extracting lexical semantic knowledge from wikipedia and wiktionary. In *Proceedings of the 6th International Conference on Language Resources and Evaluation*, Marrakech, Morocco, 26 May – 1 June 2008.

---

<sup>5</sup>Alias-i. 2008. LingPipe 3.7.0. <http://alias-i.com/lingpipe>.