Projektbericht

Semi-Automatische Generierung von Untertiteln

Jonathan Werner, Paul Bienkowski, Florian Stegen, Tim Dobert

14. April 2016

Inhaltsverzeichnis

1 Einleitung

In dem Projekt 'Semi-automatische Generierung von Untertiteln' an der Uni Hamburg im Wintersemester 14/15 war unser Ziel, eine Software zu entwickeln, mit der man mithilfe von gegebenen, manuell erstellten Transkriptionen von Videos/Audios automatisch Untertitel erstellen kann.

Ein konkreter Anwendungsfall, den wir als Materialbasis genommen haben ist der *Chaos Communication Congress*, den der Chaos Computer Club jährlich veranstaltet.

Der Chaos Computer Club veranstaltet einen jährlichen Kongress, den *Chaos Communication Congress*. Alle Vorträge werden dabei auf Video aufgezeichnet. Bereits während der Vorträge transkribiert ein Team freiwilliger Helfer gemeinschaftlich und live das Vorgetragene. Aus diesen Mitschriften und mit Hilfe der aufgezeichneten Videos werden dann Untertitel erstellt, die zusammen mit den Videos nach dem Kongress online veröffentlicht werden. Das Problem ist: nur bei einem kleinen Anteil dieser Videos wird dieser Prozess in Gänze durchgezogen, da er sehr aufwändig ist. Insbesondere das Ausrichten (Alignieren) der Transkriptionen auf die Videos ist sehr zeitintensiv.

Dieser konkrete Anlass fiel zusammen mit einem Feature-Release von *CMUSphinx* [Reference 1], das die Entwickler *Long Audio Alignment* nennen. Dieses Feature verbessert die Alignierungsqualität für längere Texte. Primär konnte erreicht werden, dass das häufig bei *forced alignment* auftretende Problem der 'globalen Verschiebung' von Audio und Transkription nicht mehr auftritt. Außerdem ist es in der Lage, fehlende Worte einzufügen und überflüssige zu überspringen.

Im Folgenden werden wir zuerst einen Überblick über die Pipeline geben, die wir implementiert haben. Danach werden wir jedes Modul der Pipeline genauer beleuchten und verschiedene Ansätze miteinander vergleichen. Abschließend werden wir die Ergebnisse analysieren und dann ein Fazit und Ausblick geben.

2 Infrastruktur

Wichtig bei einem komplexeren Projekt dieser Größenordnung ist eine einfache aber flexible Architektur bzw. Infrastruktur der Software. Unsere Software setzt sich daher aus vielen logischen Einzelteilen zusammen, im folgenden *Module* genannt. Jedes Modul erfüllt eine kleine, überschaubare Aufgabe, erhält gewisse Eingabedaten und berechnet daraus den nächsten Schritt. So ergibt sich eine *Pipeline*, in der Daten von einem Modul ans das nächste übergeben werden.

2.1 Das interne Datenformat

Um die generierten Daten zwischen den Modulen im Speicher zu behalten und nach Bedarf permanent abspeichern zu können, benötigten wir eine flexible Datenstruktur, die in Python unterstützt ist und sich leicht in ein menschenlesbares Format umwandeln lässt. Wir entschieden uns für eine verschachtelte Kombination aus Maps (Python `dicts`) und Listen (Python `lists`), welche sich sehr einfach von und nach JSON konvertieren lässt.

Der Vorteil an einer eigenen, internen Datenstruktur liegt auf der Hand: Die Pipeline kann nach jedem Schritt unterbrochen, die Daten in eine Datei serialisiert, und dann an dieser Stelle wieder fortgesetzt werden. Außerdem kann man allgemein nach jedem Schritt eine Ausgabedatei erzeugen und somit nachvollziehen, was in den einzelnen Schritten passiert. Dies ist besonders zu Debugging-Zwecken sinnvoll, aber auch um alternative Implementationen des gleichen Moduls vergleichen zu können.

Wir zeigen ein Beispiel für einen Datensatz, der den Zustand nach Beendigung aller Schritte enthält. Man kann hierin den Originaltext (`raw.normalized`), die Words/Tokens (`words.original`/`words.normalized`) mit den zugeordneten Zeiten, sowie die Unterteilung in Sätze und Chunks erkennen.


```
{
    "words": [
        { "stop": 1080, "start": 850, "original": "one ", "normalized": "one" },
        { "stop": 1500, "start": 1080, "original": "zero ", "normalized": "zero" },
        { "stop": 1920, "start": 1500, "original": "zero ", "normalized": "zero" },
        { "stop": 1920, "start": 1500, "original": "zero ", "normalized": "zero" },
        { "original": "one ", "normalized": "one" },
        { "stop": 4130, "start": 3910, "original": "nine ", "normalized": "nine" },
        { "stop": 4250, "start": 4130, "original": "oh ", "normalized": "oh" },
        { "stop": 4500, "start": 4250, "original": "two ", "normalized": "two" },
        { "stop": 4640, "start": 4500, "original": "one ", "normalized": "one" },
        { "stop": 4880, "start": 4640, "original": "oh ", "normalized": "oh" },
        { "stop": 6620, "start": 6250, "original": "zero ", "normalized": "zero" },
        { "stop": 6900, "start": 6620, "original": "one ", "normalized": "one" },
        { "stop": 7100, "start": 6900, "original": "eight ", "normalized": "eight" },
        { "stop": 7460, "start": 7100, "original": "zero ", "normalized": "zero" },
        { "stop": 7940, "start": 7460, "original": "three. ", "normalized": "three" },
```

```
        { "original": "One ", "normalized": "one" },
        { "original": "zero? ", "normalized": "zero" },
        { "original": "Too ", "normalized": "too" },
        { "original": "three. ", "normalized": "three" }
    ],
    "sentences": [
        { "start_word": 0, "end_word": 15 },
        { "start_word": 15, "end_word": 17 },
        { "start_word": 17, "end_word": 18 }
    ],
    "chunks": [
        { "start_word": 0, "end_word": 10 },
        { "start_word": 10, "end_word": 17 },
        { "start_word": 17, "end_word": 18 }
    ],
    "raw": {
        "normalized": "one\nzero\nzero\nzero\none\nnine\noh\ntwo\none\noh\nzero\one\n
            eight\nzero\nthree\none\nzero\ntoo\nthree"
    }
}
```

2.2 Das Konfigurations-System

Um einfach verschiedene Konfigurationen der Pipeline testen zu können, brauchten wir außerdem ein flexibles und mächtiges Konfigurations-System. Dieses implementierten wir ebenfalls als verschachtelte Maps (`dict`), um den Pipeline-Modulen jeweils ein eigenen Abschnitt zuordnen zu können.

Die Konfiguration wird dann über Konfigurationsdateien gelöst, die jeweils die verschachtelte Map in JSON enthalten. Somit muss man nur die Konfigurationsdatei beim Programmaufruf übergeben, der gesamte Programmablauf inklusive Ein- und Ausgabedateien ist damit festgelegt. Das macht es besonders einfach, immer wieder gleiche Konfigurationen zu testen.

Ebenfalls möglich ist die Vererbung von Konfigurationen, sodass man leicht veränderte Programmabläufe von einer gemeinsamen Konfiguration erstellen kann. Außerdem kann man jede Option von der Kommandozeile überschreiben, um schnell Varianten auszuprobieren.

Hier zeigen wir die Standardkonfiguration, die vor der eigentlichen Konfigurationsdatei geladen wird und gleichzeitig als Referenz für die bestehenden Optionen der einzelnen Module gilt.

```
{
    "output": "gen",
    "language": "en_US",
    "loglevel": "info",
    "save-steps": true,
    "input": {
        "transcript": "resources/{name}.txt",
        "audio": "resources/{name}.wav"
```

```
    },
    "steps": [
        "input",
        "normalize",
        "align",
        "syntax",
        "chunkify",
        "playback",
        "analyze",
        "output"
    ],
    "modules": {
        "align": {
            "memory": "4G",
            "jar": "sphinx4-samples-mod2.jar",
            "cp": "",
            "acousticModel": "resource:/edu/cmu/sphinx/models/en-us/en-us",
            "dictionary": "resource:/edu/cmu/sphinx/models/en-us/cmudict-en-us.dict",
            "g2p": "data/aligner/data/us/g2p/en_us_nostress/model.fst.ser",
            "skip": false,
            "gzip-log": true
        },
        "analyze": {},
        "chunkify": {},
        "chunkalign": {},
        "input": {
            "import-chunking": true,
            "format": "auto",
            "mary-url": "http://localhost:59125/process"
        },
        "normalize": {
            "mary-url": "http://localhost:59125/process"
        },
        "output": {
            "file": "{output}/{name}.{format}",
            "formats": ["txt", "srt"]
        },
        "playback": {},
        "spellcheck": {
            "ignore": "resources/{name}.ignore.txt"
        },
        "syntax": {
            "tagger-model": "resources/syntax/en_universal_tagger.model",
            "parser-model": "resources/syntax/en_universal_parser.model",
            "skip": false,
            "skip-tagger": false,
            "skip-parser": false
        }
    }
}
```

3 Pipeline-Übersicht

Es folgt eine kleine Übersicht über die implementierten Module und ihre Aufgaben, sowie die Daten, die sie verarbeiten und generieren:

- **Eingabe** (`input`) Das Transkript wird aus einer Textdatei geladen und in Sätze und Wörter zerlegt. Verschiedene Eingabeformate werden unterstützt.
- **Normalisierung** (`normalize`) Vorverarbeitung für programmatische Benutzung der Eingabedaten.
- **Align** (`align`) Analyse der Audiodatei um jedem Wort Zeitinformationen zuzuweisen. Dies ist der Kern der Anwendung.
- **Syntax Parsing** (`syntax`) Für weitere Schritte werden die Sätze syntaktisch analysiert, ein Syntax Tree wird erstellt.
- **Zerteilung in Chunks** (`chunkify`) Anhand der generierten Informationen und eines Regelsatzes werden zu lange Sätze in Abschnitte (Chunks) zerlegt, welche einzeln angezeigt werden können.
- **Chunk Aligning** (`chunkalign`) Anhand der Zeitinformationen der enthalten Wörter werden die Chunks ausgerichtet.
- **Ausgabe** (`output`) Eine Ausgabedatei mit Zeitinformationen wird erstellt. Verschiedene Ausgabeformate werden unterstützt.
- **Playback** (`playback`) Die Daten werden für einen selbstgeschriebenen Player aufbereitet und ausgegeben.
- **Analyse** (`analyze`) Bei der Berechnung angefallene Daten werden analysiert und ggf. mit einem Referenzdatensatz verglichen.

3.1 Module

3.1.1 Import

Das Importmodul unterstützt die zwei Formate: Plaintext (TXT) und das standardisierte Untertitelformat *SRT*. Aus den Dateien wird zuerst der Text extrahiert. Beim *SRT* wird zusätzlich noch das originale Chunking aus der Eingabedatei übernommen. Danach wird der Eingabetext in Tokens unterteilt. Dies geschieht entweder mit Hilfe des Programms MaryTTS oder mit Hilfe der Pythonbibliothek *NLTK*. Dabei werden außerdem die Satzgrenzen bestimmt. Falls vorhanden, wird danach das originale Chunking auf die neuen Tokens gemappt, da das Chunking intern tokenweise abgespeichert wird. Das Chunking aus *SRT*-Dateien importieren zu können erleichtert es, später Vergleiche zwischem dem altem und neuem Chunking aufzustellen.

3.1.2 Normalisierung

Das Normalisierungsmodul dient dazu, die Eingabedaten in eine Form zu bringen, welche dem tatsächlich gesprochenen Text möglichst nahe kommt. Hierzu werden eine Reihe von Ersetzungen und Normalisierungen vorgenommen. Hauptsächlich müssen Zahlen, Jahreszahlen, Abkürzungen und bei englischen Texten auch Kontaktionen expandiert werden. Die gleichen Transformationen müssen ebenfalls ausgeführt werden,wenn Text automatisch gesprochen werden soll. Deshalb bot sich das Einbinden eines **T**ext **T**o **S**peech Programms an, welches diese Transformationen bereits implementiert hat. Danach werden außerdem werden Punktuationen und andere Zeichen, welche nicht mitgesprochen werden, entfernt.

3.1.3 Alignment

Das Alignmentmodul dient dazu, den Text mit Zeitinformationen zu versehen. Dies geschieht mittels automatischer Spracherkennung. Das eigentliche Alignment, also die Zuordnung zwischen Text und Audio, wird durch das in Java geschriebene Programm *CMU-Sphinx* durchgeführt. Sphinx wurde ausgewählt, weil es eines der wenigen quelloffenen Spracherkennungsprogramme ist und das Alignment langer Texte zu Audiodaten bereits implementiert ist. Um Sphinx verwenden zu können, muss der normalisierte Text in eine Datei geschrieben werden. Die Textdatei wird dann zusammen mit dem Audio im 16Khz WAV-Format an Sphinx übergeben. Das Ausgabeformat unterscheidet sich allerdings von im Projekt intern verwendeten Datenformat und Sphinx führt zusätzlich selbst noch ein wenig Normalisierung durch und verschiebt manchmal einzelne Wörter. Um die Zeitinformationen in das projekteigene Format einzutragen, müssen die zusätzlich normalisierten Wörter auf die Originale zurückzuführt werden. Dazu wird mittels der Python-Bibliothek `difflib` eine Liste mit Änderungen und Ersetzungen erstellt und dann mit Hilfe dieser Liste Wort für Wort die Zeitinformation übernommen.

3.1.4 Syntax-Parsing

Für das Aufteilen von langen Sätzen in einzelne Untertitel (Chunks) sind systanktische Informationen hilfreich. Daher wurde ein Parser benötigt. Wir benutzen den Turboparser, wegen seinen Python-Bindings. Er wurde auf der Universal Dependency Annotation trainiert, weil sie für mehrere Sprachen funktioniert. Das macht es leichter, später Unterstützung für andere Sprachen einzubauen, da dann nur einige Bewertungskriterien und sprachspezifische Wortfolgen angepasst werden müssen. Die relevanten vom Parser ermittelten Eigenschaften werden in den Wortobjekten gespeichert.

3.1.5 Chunking

Als Chunking bezeichnen wir das Aufteilen des Textes in einzelne Untertitel, das festlegen von Zeilenumbrüchen, sowie das Festlegen der Zeit zu der der Untertitel angezeigt wird. Diese Aufteilung wird anhand der vorher gesammelten Syntaxinformationen sowie Zeitinformationen berechnet.

Zeitinformationen alleine reichen nicht, um gut lesbare Untertitel zu erzeugen. Für längere Sätze müssen Entscheidungen getroffen werden, welche Wörter gleichzeitig auf dem Bildschirm erscheinen. Das alleine von der Länge abhängig zu machen, kann zu unangenehmen Umbrüchen führen. Es mussten Regeln recherchiert und entwickelt werden, die beschreiben, an welchen Stellen Unterteilungen sinnvoll sind. Um diese Regeln anzuwenden werden die Syntaxinformationen des Parsers benutzt. Zwischen allen Wörtern in einem Satz werden Penalties berechnet, die angeben, wie schlimm es ist an dieser Stelle einen Umbruch zu machen. Mit dieser Grundlage wurden verschiedene Strategien getestet.

Verfahren zur Aufteilung in Chunks

Greedy Search Um die Ergebnisse zu verbessern, haben wir ein regelbasiertes System umgesetzt, welches jede mögliche Aufteilung bewertet und mit einer Penalty belegt – einer Art Score, die angibt, wie ungünstig eine Unterteilung vor dem ausgewählten Wort wäre. Diese Penalty-Regeln werden im nächsten Abschnitt detailiert beschrieben.

Die einfachste Art, die Penalties in Chunkings umzusetzen, ist mit einer Greedy-Strategie. Zunächst wird ein Zeichenlimit pro Chunk festgelegt, dann wird satzweise über den Text iteriert. Zu einem Chunk werden so lange Wörter aus dem Satz hinzugefügt, bis das Zeichenlimit erreicht ist. Dann wird die Aufteilung mit der geringsten Penalty für den aktuellen Chunk ausgewählt und ausgeführt. Dies wird so lange wiederholt, bis der gesamte Text in Chunks aufgeteilt ist.

Dieses Verfahren führte schon zu wesentlich besseren Ergebnissen. Da ein gute Auftrennung früh im Satz allerdings dazu führen kann, dass später nur noch schlechte übrig bleiben, ist es allerdings keine optimale Strategie. Um ein globales Optimum in annehmbarer Zeit zu finden, haben wir das Problem auf ein Problem des kürzesten Weges zwischen zwei Punkten (*point to point shortest path*) reduziert.

Optimale Lösung mit einer Dijkstra-Variante Um die Auteilung in Chunks nach den Penalty-Regeln zu optimieren, muss jeweils satzweise die Kombination aus Splits gebildet werden, die die geringste Gesamtpenalty aufweist. Leider ist die Penalty eines Chunks auch von seiner Länge abhänging, daher können nicht einfach alle Splits berechnet werden. Stattdessen haben wir einen Suchbaum aufgebaut, der jeweils von vorne

anfängt den Satz aufzuteilen, die partiell berechneten Gesamtpenalties zusammen mit den dafür benutzten Splits in einer Priority-Queue speichert, und nur bei der geringsten Gesamtpenalty weiterrechnet. Der Algorithmus terminiert, sobald der Satz vollständig in Chunks aufgetrennt wurde. Dies ist eine Variation von Dijsktras Algorithmus.

Penalty p abhängig von der Anzahl der Buchstaben x:

$$p(x) = \begin{cases} \left(\frac{x - c_{\max}}{2}\right)^{c_{\text{pow}}}, & \text{wenn } x > c_{\max} \\ (x - c_{\min})^{c_{\text{pow}}}, & \text{wenn } x < c_{\min} \end{cases}$$

Mit folgenden Konstanten:

$$\begin{aligned} c_{\min} &= 42 \\ c_{\max} &= 68 \\ c_{\text{pow}} &= 1.2 \end{aligned}$$

Verfahren zur Festlegung der Penalties:

Einfaches Chunking Wenn keine Syntaxinformationen vorliegen, lässt sich ein einfaches Chunking mittels Wortvergleichen erreichen. Wir haben folgende Regeln verwendet:

Regel	Penalty
vor *and* und *or*	6
nach *the*, *a*, *an*, *on*, *of*, *in*	15
nach Satzzeichen	5
nach Zeilenumbrüchen in der Eingabe	6
bei nicht durch Leerzeichen getrennten Worten (z.B. Bindestrich)	20
wenn keine andere Regel greift	10

Syntaxbasiertes Chunking Das einfache Chunking führt teilweise zu unangenehmen Brüchen, bei denen Wörter getrennt werden, unabhängig davon wie sie syntaktisch zusammenhängen. Um das zu verbessern, haben wir nach Regeln für gute Untertitel gesucht. Das intelligente Verfahren basiert auf Richtlinien vom BBC [Reference 2] und ESIST [Reference 3]. Diese geben an, wie lang einzelne Zeilen sein sollten und welche Wörter möglichst nicht voneinander getrennt werden sollten. Die wichtigsten davon sind:

- Artikel und Nomen
- Präposition und Nomen
- Pronomen und Verb

- Teile von komplexen Verben

Zusätzlich benutzen wir die Zeitdaten und einige andere Heuristiken:

- Pausen zwischen Wörtern
- aufeinanderfolgende Wörter mit Großbuchstaben sollten nicht getrennt werden, da sie im Englischen auf Eigennamen hindeuten (z.B. "*Electronic Frontier Foundation*")
- spezielle zusammengehörige Wortpaare (z.B. "*kind of*")

Um diese Richtlinien umzusetzen, müssen zuerst mit einem Parser die syntaktischen Funktionen und die Beziehnung zwischen Wörtern ermittelt werden.

Von den Informationen, die der Parser liefert, benutzen wir den *Part-Of-Speech-Tag* und die *Parent-Child-Relation*. Aus diesen Informationen werden auch die Übergänge zwischen Haupt- und Nebensätzen berechnet. Basierend darauf wird zu jedem Wort eine Penalty berechet, die angibt wie schlimm es ist, davor einen Split zu machen, bzw. damit einen neuen Chunk damit anzufangen.

Zeilenumbrüche Die Richtlinien erlauben bis zu zwei Zeilen pro Chunk. Schlechte Zeilenumbrüche können die Lesbarkeit ebenfalls beinträchtigen. Deshalb wird die Penalty des besten Zeilenumbruchs mit einem Faktor verrechnet und auf die des Chunks addiert. Diese wird größtenteils mit den gleichen Regeln berechnet. Zwei zusätzliche Regeln werden aber beachtet:

- beide Zeilen sollten ungefähr gleich lang sein
- die erste Zeile sollte kürzer sein als die zweite

Die Zeilenumbrüche werden auf diese Art bei der Suche nach dem globalen Optimum berücksichtigt.

3.1.6 Export

Es wird die Ausgabe in den Formaten `.txt`, `.srt`, `.ass` und `.usf` unterstützt. Bei `.ass` wird zusätzlich Zeitinformationen für jedes einzelne Wort unterstützt (Karaoke-Lyrics).

3.1.7 Frontend/Player

Die Debug UI dient dazu, die algorithmischen Entscheidungen, die in den einzelnen Modulen getroffen werden, zu visualisieren. Die grundlegende Aufgabe sollte sein, das Audio synchronisiert mit dem Text und den korrespondierenden Chunks wiederzugeben.

Dazu wurde eine Web-Oberfläche gestaltet, die gegen Ende des Projektes folgendermaßen aussah:

TODO: screenshot-debug-ui

Der obere schwarze Bereich zeigt einen kompletten Untertitel an. Ziel ist hier, ein unmittelbaren visuellen Eindruck von der Güte eines Untertitels zu bekommen.

In den grauen Bereichen werden die Penalties, die bei der Auswahl des Chunks zutrafen, angezeigt. Hier wird sowohl die *length penalty*, also die Penalty die es für das Überschreiten der Richtlänge gab, als auch *single penalties*, also die Regeln, die verletzt wurden, angezeigt. Diese *single penalties* werden sowohl für das Ende des Chunks als auch für das Ende der *line* im Chunk angezeigt.

Zwischen den grauen Bereichen werden die Chunks selber angezeigt. Nicht von Sphinx zugeordnete Wörter sind rot. Ein hovern über einem Wort zeigt die Detailinformationen, die im Datenformat zu diesem Wort abgespeichert sind an, also zB. die Syntax-Parse-Informationen.

4 Ergebnisse

Nach manueller Analyse der SRT-Dateien blieben 8 englische Vorträge übrig, welche für die Auswertung infrage kamen. Es folgt eine Statistik, wie erfolgreich das Alignment bei diesen Dateien war:

Erkennungsrate	Vortragtitel
22.3 %	5311 - Lasers in Space
81.0 %	5405 - Data Mining for Good
79.1 %	5423 - The Tor Network
82.1 %	5449 - Mobile network attack evolution
84.6 %	5476 - Electronic Bank Robberies
76.9 %	5477 - An introduction to Firmware Analysis
82.9 %	5605 - Opening Event
79.0 %	5622 - 30c3 Keynote
75.6 %	**Gesamterkennungsrate**

4.1 Vergleich mit Referenz-SRT

Zum Abschluss liefern in Abbildung 1 wir einen Vergleich von unseren Ergebnissen mit einem Referenz-SRT. Es handelt sich um ein Exzerpt aus dem dem Keynote Video [Reference 4].

Auffällig ist hier auf jeden Fall, dass die Referenz-SRT sich nicht mit den Richtlinien vereinen lässt, die wir gefunden hatten: es gibt keine Zeilenumbrüche und die Subtitles sind zu lang. Allerdings sind auch unsere Chunks nicht optimal. Beispielweise ist die letzte Trennung zwischen *the* und *night* nicht korrekt, dies lässt sich jedoch auf die Ausgaben des Syntax Tree Parsers zurückführen, welche teilweise fehlerhaft sind. Besonders mit langen ,,Bandwurmsätzen", wie sie häufiger in Vorträgen auftreten, kommt der Parser nicht zurecht.

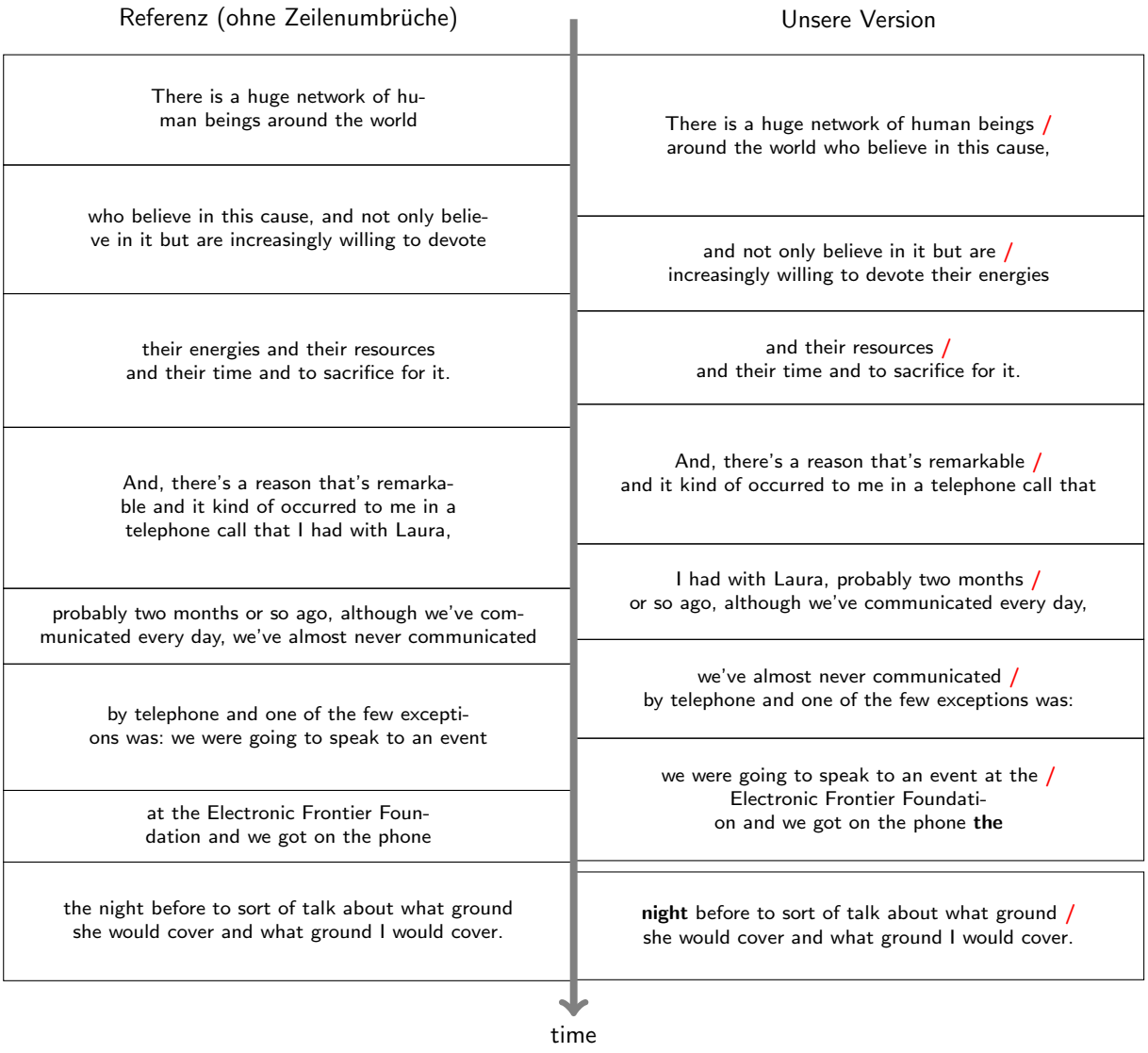

Abbildung 1: Vergleich unserer generierten Untertitel zu manuell erstellter Referenz. Die Zeilenumbrüche der Referenz wurden nicht manuell eingegeben, daher müssen die Chunks vom darstellenden Programm automatisch umgebrochen werden.

5 Fazit und Ausblick

Zusammenfassend kann man folgende Ergebnisse festhalten: Die Software produziert brauchbare Ergebnisse, sofern die Qualität der Eingabedaten gut ist. Leider gilt dies nur für die allerwenigsten Transkriptionen, sodass nur bei einem Bruchteil überhaupt Resultate heraus kamen.

Ein Vergleich unserer Ergebnisse mit manuell erstellten Chunks ist schwierig objektiv zu führen, da die manuellen Chunks sich kaum an Richtlinien halten. Es gibt daher keine geeigneten Metriken, mithilfe deren man eine objektive Güte messen könnte. Hier wäre eine subjektive Auswertung nötig – diese müsste experimentell durchgeführt werden, was außerhalb des Umfangs dieser Arbeit lag.

Der vorliegende Stand der Pipeline hat noch Potential für Verbesserung. Primär wäre die Frage interessant, inwiefern eine dynamische Anpassung der Penalty-Werte zu qualitativ besseren Resultaten führen könnte. Im momentanen Stand wird jeder Regel ein statischer Penalty-Wert zugewiesen - die Festlegung dieses Wertes erfolgte aber relativ willkürlich und experimentell. Es wäre wüschenswert, einen lernenden Algorithmus zu implementieren, der diese Penalties nach einer qualitäts-optimierenden Heuristik optimiert.

Außerdem würde im Hinblick auf die deutsche Sprache die Zuhilfenahme eines Sprachmodells, das Applaus und Stille erkennt, die Erkennungsgenauigkeit verbessern. Hier ist der momentan vorhandene Stand an Modellen, die frei verfügbar sind, nicht ausreichend.

5.1 Glossar

- **Chunk** Den Begriff Chunk verwenden wir projektintern für ein oder mehrere Zeilen Untertitel, welche gleichzeitig dargestellt werden.
- **Alignment** Die Zuordnung von Text zu Zeitabschnitten in der Sprachdatei
- **SRT** (**S**ub**R**ip**T**ext) ist ein sehr eifach aufgebautes, textbasiertes Untertitelformat.

Bibliographie

[1]N. V. Shmyrev, "Long Audio Aligner Landed In Trunk." http://cmusphinx.sourceforge.net/2014/07/long-audio-aligner-landed-in-trunk/, 2014.

[2]G. F. Williams, "bbc. co. uk Online Subtitling Editorial Guidelines V 1.1." http://www.bbc.co.uk/guidelines/futuremedia/accessibility/subtitling_guides/online_sub_editorial_guidelines/vs1_1.pdf, 2009.

[3]M. Carroll and J. Ivarsson, "Code of Good Subtitling Practice." http://www.esist.org/ESIST%20Subtitling%20code_files/Code%20of%20Good%20Subtitling%20Practice_en.pdf, 1998.

[4]CCC, "30C3 Keynote." http://www.amara.org/en/videos/eBoWzGWkTn2Z/info/30c3-keynote/, 2013.