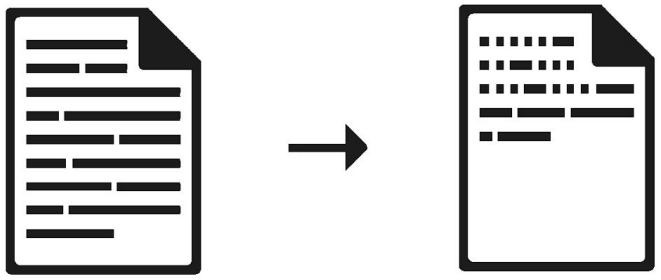
Grundlagenforschung zur Komprimierungsrate von Texten verschiedener Art

Seminarfacharbeit Informatik, Linguistik

Fabian Keßler & Moritz Decker



Inhaltsverzeichnis

Inhaltsverzeichnis **1**

Einleitung & Ziel der Arbeit (Verfasser: Fabian Keßler & Moritz Decker) **2**

Der Kompressions- & Dekompressionsalgorithmus (Verfasser: Moritz Decker) **3**

Definition der Anforderungen 3

Formale Definition der Anforderungen 3

Implementierung der Kompression 4

Implementierung der Dekompression 5

Laufzeit der Kompression 5

Optimierung der Laufzeit 6

Analyse und Interpretation der Metadaten (Verfasser: Fabian Keßler) **7**

Erhebung der Daten 7

Auswertung der Daten 8

Fazit (Verfasser: Fabian Keßler & Moritz Decker) **10**

Schlussaussage 10

Fehlerbetrachtung 10

Danksagung 10

Einleitung & Ziel der Arbeit (Verfasser: Fabian Keßler & Moritz Decker)

Bei dieser Seminarfacharbeit handelt es sich um eine Grundlagenforschung. Es wird desweiteren auch keine konkrete Fragestellung aufgestellt. Viel mehr wird offen untersucht, ob interessante Muster in den Metadaten beim Komprimieren von Texten gefunden werden können. Beispielsweise fiele darunter die Frage, ob aus Metadaten der Kompression Rückschlüsse auf den Inhalt des komprimierten Textes getroffen werden können. Als Datensatz wird sich hierbei des "Gutenberg-DE"-Projektes[1] bedient, welches rund 10.000 deutschsprachige Texte verschiedener Autoren und Genres umfasst. Die gesamte Software, die im Zusammenhang mit dieser Seminarfacharbeit von uns entwickelt wurde, ist quelloffen auf GitHub[2]. Alle in dieser Arbeit aufgeführten Diagramme können eigenständig auf seminarfach.zapto.org reproduziert werden.

[1] Erreichbar unter https://www.projekt-gutenberg.org/ (Stand 16. Dezember 2020)
[2] Erreichbar unter https://github.com/Antricks/seminarfacharbeit-10 (Stand 16. Dezember 2020)

Der Kompressions- & Dekompressionsalgorithmus (Verfasser: Moritz Decker)

Definition der Anforderungen

Das Ziel des Kompressionsalgorithmus ist es, den eingegebenen Text in einer verarbeiteten Form auszugeben, die im Optimalfall weniger Textlänge in Zeichen benötigt, als der ursprüngliche Text, jedoch durch einen entsprechenden Algorithmus eindeutig in die ursprüngliche Form zurückgebracht werden kann. Dabei kann von einem Text aus Wörtern ausgegangen werden, die sich aus Zeichen des UTF-8 Zeichensatzes zusammensetzen und durch gängige Satzzeichen, Leerzeichen oder Kontrollzeichen wiederum des UTF-8 Zeichensatzes von einander getrennt werden. Ein alleinstehendes Wort oder eine alleinstehende Zahl zeichnet sich dadurch aus, dass davor entweder der Anfang der Zeichenkette oder ein Trennzeichen ist und, dass dahinter entweder das Ende der Zeichenkette oder ebenfalls ein Trennzeichen ist.

Formale Definition der Anforderungen

Um später Klarheit bei der Beweisführung zu haben ist ebenfalls eine formale Definition unserer Anforderungen an den Kompressions- sowie Dekompressions-algorithmus angebracht:

Es seien die Kompressionsfunktion f und die Dekompressionsfunktion g.

Die Kompressionsfunktion $f(m)$ bilde von der Menge M aller Nachrichten m auf die Menge N aller komprimierten Nachrichten $n = f(m)$ ab. Es gelte $f : M \rightarrow N$.

Die Dekompressionsfunktion $g(n)$ sei die Umkehrfunktion von $f(m)$.

Es gelte dementsprechend natürlich: $g : N \rightarrow M$

Beide Abbildungen $f : M \rightarrow N$ und $g : N \rightarrow M$ seien bijektiv. Es gebe also für jede Nachricht $m \in M$ nur eine einzige, eindeutig auf die Ursprungsnachricht $m \in M$ rückführbare komprimierte Nachricht $n \in N$ und andersherum.

Für eine erfolgreiche Kompression sollte des Weiteren für jede Nachricht $m \in M$ gelten: $|m| \leq |f(m)|$

Dies folgt daraus, dass die Kompression in der praktischen Anwendung zum Ziel hat, die gleiche Information in weniger Speicherplatz zu enthalten.

Implementierung der Kompression

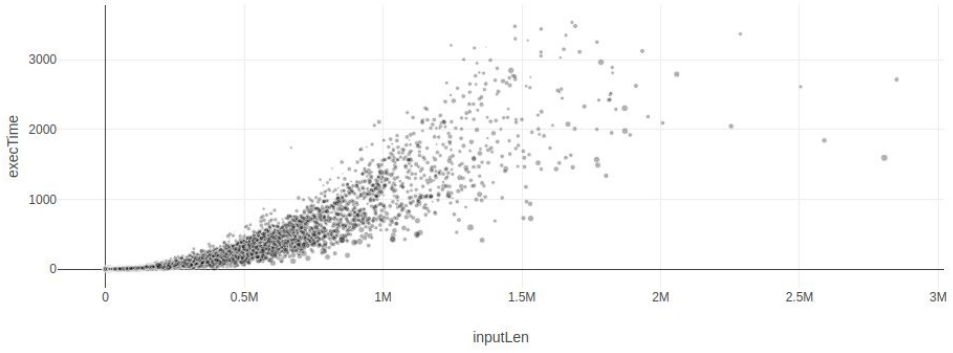
Auf einer Definition von Ansprüchen kann nun also ein Algorithmenpaar aufgebaut werden, das eben beschriebene Funktionalität implementiert. Der von uns gewählte Ansatz zur Kompression trennt Wörter aus dem Eingabetext durch genannte Trennzeichen auf und bildet eine Liste, die alle Wörter des Textes in der Originalreihenfolge beinhaltet. Diese Liste aller Wörter ignoriert dementsprechend alle Trennzeichen. Der Eingabetext wird weiterhin als Zeichenkette gespeichert, um in den folgenden Schritten modifiziert zu werden. Daraufhin wird aufbauend auf der Liste aller Wörter eine wiederum neue Liste aller zu ersetzenden Worte gebildet, die alle alleinstehenden Zahlen beinhaltet und alle Wörter beinhaltet, die doppelt vorkommen. Der Grund dafür, dass Zahlen kategorisch zu dieser neuen Liste zählen, wird durch den nächsten Schritt anschaulich. Nun wird durch jedes Wort der Liste der zu ersetzenden Worte iteriert und in der Originalzeichenkette durch den Index ersetzt, an dem es sich in der neuen Liste befindet. Dieser Index wird als Zahl in ASCII-Repräsentation in der Basis 10 in die Zeichenkette eingesetzt. Wären nun also alleinstehende Zahlen aus dem Originaltext nicht in die Liste der zu ersetzenden Worte aufgenommen worden, würden diese nun auf ungewollte Weise mit den vom Algorithmus eingesetzten Zahlen interferieren bzw. von der Dekompression fehlinterpretiert werden. Dies würde der Anforderung der Bijektivität des Algorithmus widersprechen, weshalb dieser potentielle Verlust an Kompressionsvolumen hier in Kauf genommen wird. Im letzten Schritt wird die Liste aller zu ersetzenden Wörter und aller Zahlen durch eine Zeichenkette ausgedrückt. Dabei werden die Objekte der Liste durch Leerzeichen getrennt und die dadurch entstandene Zeichenkette durch einen Zeilenumbruch vom Text getrennt an das Resultat angehängt. Die Verkürzung des Textes wird hierbei dadurch erreicht, dass die Länge des Index in ASCII-Repräsentation kürzer ist, als das eigentliche Wort. So muss eine mehr oder weniger lange Zeichenkette nur ein Mal gespeichert werden und ausschließlich richtig referenziert werden. Der Grund, dass Wörter nicht ersetzt werden, die nur ein einziges Mal im Text vorkommen ist, dass die Zeichenkette sowieso gespeichert werden müsste, der Text würde bei einer solchen Ersetzung deshalb verlängert werden, da nun statt der einfachen Verwendung des Wortes die exakt gleich lange Zeichenkette des Wortes an einer anderen Stelle gespeichert werden müsste und zusätzlich referenziert werden müsste. Da Referenzen jedoch ebenfalls Speicherplatz benötigen, würde sich das Resultat mit diesem Schritt verlängern, was offensichtlich und laut Definition ungewollt ist.

Implementierung der Dekompression

Die Dekompression rekonstruiert den Originaltext nun dadurch, dass sie zuerst die Liste aller ersetzten Wörter aus der letzten Zeile der Datei einliest und jede alleinstehende Zahl im komprimierten Text nun als einen Index der Liste interpretiert. Diese Zahl wird dann durch das Wort oder die Zahl an der jeweiligen Stelle der Liste ersetzt und die Schleife läuft so lange durch, bis sie das letzte Wort des Textes erreicht hat. Die Zeichenkette, die bei der dieser Ersetzung entsteht ist nun wieder der rekonstruierte Originaltext.

Laufzeit der Kompression

Die Laufzeit der Komprimierung spielt in der praktischen Anwendung ebenfalls eine wichtige Rolle, da optimalerweise nicht nur Platz, sondern auch Zeit zu sparen ist. Im Fall unseres Algorithmus ist bei ansteigender Textlänge ebenfalls auch eine Form quadratischen Wachstums der Laufzeit zu erwarten, da es verschachtelte Schleifen im Programm gibt. Zudem wurde die Software zur Kompression und Dekompression in der interpretierten Programmiersprache Python verfasst, was sich ebenfalls negativ auf die Laufzeit auswirkt. Die Zeiten zum Komprimieren variierten schon beim Testen sehr stark abhängig davon, wie lang die eingegebenen Texte waren. Dies zeigt sich ebenfalls in der folgenden Grafik, wobei execTime die Laufzeit in Sekunden angibt und inputLen die Länge des Eingabetextes in Zeichen angibt:



Wie zu sehen ist, streuen mit zunehmender Eingabelänge auch die Werte der Laufzeit zunehmend. So gibt es Texte, die eine sehr ähnliche Länge in Zeichen haben, jedoch massive Unterschiede in der Laufzeit aufweisen. Dies folgt daraus, dass die Längen der Wörter und die Anzahl der mehrfach verwendeten Wörter von Text zu Text variieren.

Optimierung der Laufzeit

Im Fokus der Laufzeitoptimierung stand vor allem die Wortersetzung, die von uns händisch implementiert wurde, was daraus folgt, dass ausschließlich Wörter ersetzt werden sollten, die - wie bereits in den Anforderungen definiert - durch Trennzeichen oder Start bzw. Ende des Textes gekennzeichnet sind. Hierbei sind zwei Stellen im Code wiederum besonders wichtig: die Erkennung, ob ein Wort überhaupt ersetzt werden sollte und die Ersetzung selbst.

Die erste Veränderung die hier ins Spiel kommt ist das Eingrenzen der Laufzeit der Ersetzung, was hauptsächlich dadurch von uns erreicht wurde, dass wir nur noch in dem Bereich des Textes Worte ersetzen, der hinter dem ersten Auftreten des Wortes anfängt, da offensichtlicherweise dadurch weniger Text nach passenden Zeichenketten durchsucht werden muss.

Die zweite Veränderung war eher das Beheben eines Denkfehlers. Zuvor wurde der gesamte Text durchsucht und gezählt, wie oft das gesuchte Wort vorkam. Für den Fall, dass diese Anzahl größer oder gleich 2 sein sollte, würde dann eine Ersetzung stattfinden, was prinzipiell nicht falsch war, jedoch unnötig Laufzeit beansprucht hat. Die Optimierung wurde hierbei dadurch geschaffen, statt eine Anzahl zu suchen, durch das Python-Schlüsselwort “in“ zu entscheiden, ob das gesuchte Wort im Teil des Textes enthalten war, der hinter dem ersten Auftreten des Wortes anfing.

Analyse und Interpretation der Metadaten (Verfasser: Fabian Keßler)

Erhebung der Daten

Um die Statistiken der Komprimierbarkeit von Texten verschiedener Art untersuchen zu können, mussten wir erst einmal eine Website finden, von der wir eine große Anzahl solcher Texte herunterladen konnten. Diese Website musste allerdings einige Kriterien erfüllen, um von uns als Quelle genutzt werden zu können. Sie musste eine möglichst große Anzahl an verschiedenen Texten bieten, welche unter einer konstanten Seitenstruktur mit ausreichender Menge von Zusatzinformationen zum Text wie Titel, Autor und Genre aufgelistet werden. Außerdem war wichtig, dass die dortigen Texte dauerhaft komplett frei zur Verfügung stehen. Die Seite, die wir uns ausgesucht haben, ist projekt-gutenberg.org. Sie gehört zu dem "Projekt Gutenberg-DE", das sich als Ziel gesetzt hat, kapitelweise gemeinfreie[3] Werke deutscher Sprache frei für die Allgemeinheit zur Verfügung zu stellen.

Nachdem eine Textquelle gefunden worden war, mussten diese Texte noch komprimiert und deren Daten gespeichert werden. Für diese Aufgabe wurde von uns ein sogenannter "Webcrawler" entwickelt, also ein Programm, das alle Unterseiten des Projekt Gutenbergs untersucht und die geforderten Informationen wie Autor, Text und Genre extrahiert. Anschließend wird der Text komprimiert und zusammen mit den Daten von der Website und den Daten, die durch die Komprimierung gewonnen werden konnten, in unserer eigenen Datenbank abspeichert. Schlussendlich hatten wir dann einen großen Datensatz mit allen Daten der komprimierten Texte. Jeder Text wurde zusammen mit einer eindeutigen, automatisch verliehenen ID und seinem Titel sowie dem Link zum Text gespeichert. Dazu kamen noch die Eingabelänge und Ausgabelänge in Zeichen, aus denen wir einen Wert errechneten, welchen wir "charRate" nannten. Diesen Wert haben wir folgendermaßen definiert:

charRate = round(100 - len(output_text)/len(input_text)*100, 2)

Dieser Wert beschreibt die Komprimierbarkeit des Textes als prozentualen Wert.

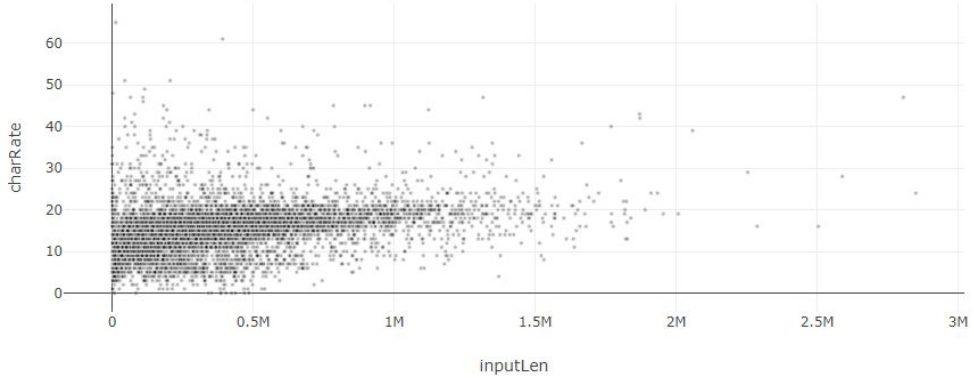
Zudem wurden auch noch das Genre, die Anzahl der Duplikatswörter[4] zusammen mit einer Liste dieser und ihrer Häufigkeit sowie die Zeit, die benötigt wurde, um den Text zu komprimieren abgespeichert.

[3] Texte, die von Autoren geschrieben wurden, die schon mehr als 70 Jahre tot sind
[4] Wörter, welche mehrmals im Text vorkommen

Auswertung der Daten

Um die in einer Datenbank gespeicherten Daten der Komprimierungen verwerten zu können, haben wir ein Hilfsmittel programmiert, das uns nach bestimmten Angaben automatisch mit den gegebenen Daten anschauliche Diagramme erstellt.[5]

Zuallererst haben wir uns die Werke nach Genre sortieren lassen, sodass jede Farbe in der Legende ein Genre darstellt. Dabei erwartete uns schon direkt das erste Problem. Die insgesamt 9.804 Artikel, die wir vom Projekt Gutenberg-DE runtergeladen hatten, wurden mit gesamt 624 Genres gekennzeichnet. Es gab teilweise nicht erklärbare Dopplungen, sowie ähnliche oder mit Kommata separierte Genre. Um ein paar Beispiele zu nennen gab es das Genre “Reisen”, aber auch “Reisen: Asien” oder “Humor” und “Humor, Satire, Kabarett”. Rein logisch betrachtet müssten auch die Werke, welche im Genre “Humor, Satire, Kabarett” sind, dem Genre “Humor” zugewiesen werden. Dies oder das Zusammenlegen dieser ähnlichen Genre war uns jedoch aus zeitlichen Gründen nicht möglich. Daher werten wir nur signifikante Genre aus, also solche, die mehrere Werke beinhalteten ohne ähnliche Genre mit mehr Werken.



Dieses Diagramm zeigt alle von uns gesammelten Werke als Punkte, dessen Position auf der X-Achse die Eingabelänge und die Position auf der Y-Achse die charRate, also die Komprimierbarkeit der Texte in “%” beschreibt. Allgemein betrachtet erkennt man, dass der Großteil aller Werke unter einer Million Zeichen lang ist. Wie erwartet ist bei größerer Eingabelänge auch die Komprimierbarkeit der Texte höher. Unerwartet war dabei jedoch der Unterschied der Steigungen der unteren und oberen Grenze des Konzentrationsbereichs. Während die untere Grenze des Bereichs eine langsam abflachende Steigung aufweist, bleibt die obere Grenze nahezu konstant bei 20%. Dieser Wert stellt die Zeichenrate dar, bei der die meisten Texte aufhören, bei zunehmender Länge zu komprimieren.

[5] Erreichbar unter http://seminarfach.zapto.org/stats.php (Stand 16. Dezember 2020)

Den Ursprung dieser Rate vermute ich in der Funktionsweise unseres Kompressions-Algorithmus. Da die Wörter, welche oft komprimiert, also durch eine Zahl ersetzt werden, tendenziell eher aus drei oder weniger Zeichen bestehen, wird ab einer gewissen Stellenlänge die Zahl selber genau so groß, oder sogar größer als das ursprüngliche Wort. Dies hat zur Folge, dass die eigentlich ansteigende Komprimierbarkeit durch häufigere Nutzung gleicher Wörter, welche wiederum durch die steigende Textlänge entsteht, mit zunehmender Anzahl der Duplikatswörter wieder abnimmt. Die über der 20%-Schwelle liegenden Werke haben alle ein relativ oft benutztes Duplikatswort, das überdurchschnittlich, also länger als drei Zeichen lang ist. Diese langen Duplikatswörter ziehen dann die Charrate hoch, da die Zahl, mit der sie ersetzt wurden, maßgeblich weniger Stellen vorweist als das Ursprungswort. Aus diesem Grund müsste der optimale Text zur Kompression für unseren Algorithmus auch möglichst wenige Duplikatswörter in der Anzahl, aber lange Duplikatswörter in der Zeichenlänge aufweisen.

Das Werk, das diese Voraussetzungen am besten erfüllt, ist “Der Struwwelpeter” von Heinrich Hoffmann. Allerdings wird bei den am meisten komprimierten Worten “Die”, “Geschichte” und “vom” relativ schnell klar, dass es sich hierbei nicht um die Kompression der gesamten Geschichte handelt, sondern lediglich der Überschriften. Diese fangen zum Großteil mit den Wörtern “Die Geschichte von/vom ” an, und sind die einzigen in Textform vorhandenen Wörter im Artikel. Die Geschichten sind hier nur in Form von Bildern enthalten und wurden daher von unserem Crawler auch nicht eingelesen.

Ein vollständig eingelesenes Beispiel mit hoher Komprimierbarkeit ist “Hyperbeln auf Wahls große Nase” von Friedrich Haug aus dem Genre “Humor, Satire, Kabarett”. Auch hier ist die hohe Komprimierbarkeit von 51% den im Vergleich zum Durchschnitt etwas längeren Worten wie “Wahls”, “Nase” und “Wahl” verschuldet. Auffällig bei diesem Werk ist auch, dass es nur 195 Duplikatswörter gibt, was für eine Textlänge von rund 45.000 Zeichen außerordentlich wenig ist. Ähnlich lange Werke haben um die 700 Duplikatswörter und somit fast das Vierfache. Dies kann man einerseits auf die Art des Textes als Gedicht, andererseits auch auf einen unterdurchschnittlich geringen Wortschatz zurückführen. Zudem sind die sehr abwechslungsreichen und kurzen Verse oft auch voll mit nur einmal vorkommenden Wörtern. Insofern zeigt uns dieses Werk sehr eindrucksvoll, welch große Anzahl von Faktoren die Anzahl der Duplikatswörter, welche für die Komprimierung essentiell sind, beeinflussen können.

Fazit (Verfasser: Fabian Keßler & Moritz Decker)

Schlussaussage

Hier würden noch mal alle gewonnenen Erkenntnisse zusammengefasst werden, die besonders signifikant sind.

Fehlerbetrachtung

Hier würden sämtliche Fehler erklärt werden, die im Rahmen der Seminarfacharbeit gemacht wurden und dazu passende Lösungen vorgetragen werden. Darunter fallen vor allem Denkfehler und Bugs bei der Programmierung von Kompression, Dekompression und des Webcrawlers. Ein Beispiel von vielen wäre eine Race Condition, die dazu geführt hat, dass mehrere Instanzen des Crawlers gleichzeitig an der gleichen Datei gearbeitet und dabei fehlerhafte Daten erzeugt haben.

Danksagung

Unser Dank richtet sich an die Software-Abteilung der Bruker Daltonik GmbH Bremen, da uns für eine Nacht der Rechenserver “C3PO” mit 96 CPU-Kernen und 384 GB RAM zur Verfügung gestellt wurde, um die benötigte Zeit zur Datenerhebung von rechnerisch ungefähr 5 Tagen auf rund 7 Stunden zu reduzieren.