technische universität
dortmund

Bachelorarbeit

Experimentelle Analyse verschiedener linearer l_2-Einbettungen von dünn besetzten Eingabedaten

Jens Quedenfeld

14. Juli 2013

Betreuer:
Prof. Dr. Christian Sohler
Chris Schwiegelshohn

Fakultät für Informatik
Algorithmen und Komplexitätstheorie (Ls2)
Technische Universität Dortmund
http://ls2-www.cs.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und Hintergrund

Die Datenanalyse ist eine der ältesten und wichtigsten Aufgaben der Informatik. Vor der Analyse der Daten steht deren Erhebung. Dabei stellen wir für jedes Datum gewisse Merkmale fest, wobei jedes Merkmal eine Dimension genannt wird. Dies kann zum Beispiel die Häufigkeit eines bestimmten Wortes innerhalb eines Textes sein. Inzwischen sind die Datensätze allerdings häufig so groß, dass konventionelle Algorithmen und Hardware nicht für deren Analyse ausreichen. Nun gibt es zwei Möglichkeiten, die Datenmenge zu verkleinern: Entweder wird die Anzahl der Daten verringert oder deren Dimension. Welcher Ansatz der Richtige ist, hängt sowohl von der Anwendung als auch von der Datenmenge selbst ab. Liegen beispielsweise wenige hochdimensionale Daten vor, ist eine Dimensionsreduktion viel ergiebiger und sinnvoller als eine Verringerung der Anzahl der Daten. In beiden Fällen ist es jedoch entscheidend, dass bestimmte Eigenschaften der Daten beibehalten werden. Was diese Eigenschaften explizit sind, hängt dabei von der jeweiligen Anwendung ab.

Häufig (beispielsweise beim Clustering, der Nearest Neighbor Suche oder auch bei der Regression) werden die Daten als Punkte im Raum interpretiert. Wird nun die Dimension der Datenpunkte reduziert, sollte ihr Abstand zueinander annähernd erhalten bleiben. Der „Abstand“ kann dabei auf unterschiedliche Art und Weise definiert werden. Am naheliegendsten ist die Verwendung der euklidischen Distanz, also die Länge der direkten Verbindung zweier Punkte. Im Zweidimensionalem entspricht die euklidischer Distanz somit der Luftlinie zwischen den entsprechenden Punkten.

Neben der euklidischen Distanz gibt es noch andere Abstandsdefinitionen, wie zum Beispiel die so genannte Manhattan-Metrik: Der Name kommt daher, dass in Manhattan die meisten Straßen horizontal oder vertikal verlaufen, das heißt eine diagonale und damit direkte Bewegung von einem Punkt zu einem anderen ist nicht möglich. Im Zweidimensionalem ergibt sich der Abstand zwischen zwei Punkten dementsprechend aus der Differenz der horizontalen Komponenten plus der Differenz der vertikalen Komponenten.

Ein weiteres Distanzmaß ist die Maximumsmetrik: Hierbei ist der Abstand zweier Punkte gleich dem Maximum der Differenz der einzelnen Komponenten. Bildlich gesehen, ist die Distanz zweier Punkte bei der Maximumsmetrik die längste Strecke, die man in Manhattan gehen kann, ohne einen Richtungswechsel vorzunehmen oder von einer kürzesten Route abzukommen.

Das am häufigsten verwendete Abstandsmaß ist die euklidische Distanz. Sie wird beispielsweise bei der Clusteranalyse verwendet, bei der es darum geht, die Punkte des Datensatzes in mehrere Gruppen mit ähnlichen Eigenschaften aufzuteilen. Punkte, die nah beieinander liegen, sollen dabei der gleichen Gruppe zugeordnet werden. Da die Punkte des reduzierten Datensatzes möglichst zu den gleichen Gruppen gehören sollten wie die entsprechenden Originalpunkte, muss die Distanz zwischen ihnen bei der Datenverkleinerung ungefähr erhalten bleiben. Je genauer diese Abstände beibehalten werden, desto höher ist die Güte bzw. der Approximationsgrad der Datenreduktion.

Bisher wurde nur der Abstand zwischen den Punkten als Eigenschaft betrachtet, die bei der Dimensionsreduktion erhalten bleiben soll. Bei bestimmten Anwendungen sind jedoch ganz andere Kriterien entscheidend: Als Beispiel sei die lineare Regression genannt, bei der die optimale Lösung eines linearen Gleichungssystems gesucht wird. Die Punkte unseres Datensatzes entsprechen dabei den Koeffizienten der Gleichungen. Ist das Gleichungssystem zu groß, um die optimale Lösung direkt zu berechnen, kann entweder die Anzahl der Gleichungen oder die Anzahl der Unbekannten reduziert werden. In beiden Fällen ist es wichtig, dass das verkleinerte Gleichungssystem eine ähnliche optimale Lösung hat wie das ursprüngliche Gleichungssystem. „Ähnlich“ kann hier wieder auf verschiedene Weisen definiert werden: Beispielsweise kann es darauf ankommen, dass der euklidische Abstand der exakten und approximierten Lösung möglichst klein sein soll. Alternativ könnte man auch hier zum Vergleich der beiden Lösungen die Manhattan-Metrik oder ein anderes Distanzmaß verwenden.

Die beiden genannten Beispiele — das Clustering und die lineare Regression — zeigen, dass die Eigenschaften, die bei dem verkleinerten Datensatz erhalten bleiben sollen, recht unterschiedlich sein können.

Im Rahmen dieser Arbeit betrachten wir Verfahren, die sich zur Dimensionsreduktion von Punkten im Euklidischen Raum eignen. Als mögliche Anwendung kommt somit die bereits erwähnte Clusteranalyse in Frage oder beispielsweise auch die Nearest Neighbor Suche, bei der es darum geht, zu einem gegebenen Punkt den abstandsmäßig nächsten Punkt des Datensatzes herauszufinden. Bei der Datenverkleinerung werden die d-dimensionalen Punkte des Datensatzes auf Punkte im k-dimensionalen Raum abgebildet. Dieser Vorgang wird als Einbettung bezeichnet, das benutzte Verfahren als Einbettungsverfahren. An die Einbettungsverfahren werden dabei verschiedene Anforderungen gestellt: Die Dimension der Einbettung sollte möglichst klein sein, da die Laufzeit der nachfolgenden Algorithmen auf den Daten häufig stark von der Dimension abhängt. Genauso wichtig ist es, dass auch

die Einbettung selbst schnell berechnet werden kann. Um überhaupt einen Nutzen von der Einbettung zu haben, ist es notwendig, dass die Einbettung die ursprünglichen Daten gut repräsentiert. In unserem Fall bedeutet das, dass der euklidische Abstand aller Punktepaare annähernd erhalten bleiben soll. Wie groß die Abweichung zu den Originaldaten ist, wird dabei mit einer ϵ-Schranke beschrieben: Der Abstand eines Punktepaares darf sich beim Einbetten höchstens um den Faktor $1 \pm \epsilon$ ändern.

Bei den in dieser Arbeit betrachteten Methoden handelt es sich stets um lineare Einbettungen, d.h. die Transformation des Datensatzes lässt sich mit einer Matrix beschreiben. Jeder d-dimensionale Vektor wird dabei mit einer $(k \times d)$-Einbettungsmatrix multipliziert, sodass sich eine Einbettung mit k-dimensionalen Vektoren ergibt. Die Einbettungsmatrix ist dabei zumindest teilweise zufällig aufgebaut und kann durch Verwendung entsprechender Zufallsgeneratoren implizit abgespeichert werden. Die Voraussetzung für die implizite Speicherung ist, dass die zufälligen Elemente nicht vollständig unabhängig voneinander sind. Betrachtet man also eine hinreichend große Menge der zufälligen, aber nur beschränkt unabhängigen Elemente, so wird man bestimmte Muster oder Regelmäßigkeiten feststellen. Diese müssen jedoch in Kauf genommen werden, um die implizite Speicherung zu ermöglichen. Die implizite Speicherung ist nämlich sehr wichtig, da die Einbettungsmatrix unter Umständen sogar größer sein kann als die Datenmatrix selbst. Wie die Speicherung konkret aussieht, hängt von dem jeweiligen Einbettungsverfahren ab. Auch die Matrixmultiplikation wird bei den besseren Verfahren implizit berechnet, indem beispielsweise die Nullen im Eingabevektor oder besondere Eigenschaften der Einbettungsmatrix ausgenutzt werden.

Wir betrachten vor allem dünn besetzte Eingabedaten, das heißt, dass die meisten Merkmale eines Datums gleich null sind. Eine dünn besetzte Datenmenge erhält man zum Beispiel, wenn man sich die Worthäufigkeit in verschiedenen Texten anschaut: Jedes Datum repräsentiert dabei einen Text und jedes Merkmal ein bestimmtes Wort. Sind die Texte sehr unterschiedlich und möglicherweise auch noch sehr kurz, kommt nur ein geringer Anteil aller Wörter in einem Text vor. Ein Datum enthält somit sehr viele Nullen. Häufig können Operationen in der linearen Algebra schneller durchgeführt werden, wenn dünn besetzte Vektoren und Matrizen genutzt werden können. Es gibt Einbettungen, die die vielen Nullen eines Datums ausnutzen und so ein erhebliche bessere Laufzeit als konventionelle Verfahren haben, die eine Null wie jede andere Zahl interpretieren. Bestimmte Verfahren verwenden selbst dünn besetzte Matrizen, um die Eingabedaten einzubetten. Solche Einbettungsverfahren haben jedoch die Tendenz, Eingabevektoren mit sehr vielen Nullen stark zu verzerren und im Extremfall auf den 0-Vektor abzubilden. Daher kann es bei Verfahren, die dünn besetzten Matrizen im Rahmen einer Dimensionreduktion verwenden, einen Tradeoff zwischen Geschwindigkeit und Zieldimension geben.

1.2 Aufbau der Arbeit

Die Arbeit besteht im Wesentlichen aus zwei Teilen: Im ersten Teil werden verschiedene Einbettungsverfahren vorgestellt, die dann im zweiten Teil experimentell untersucht werden. Konkret geht es um die folgenden Fragestelllungen:

- In vielen Publikationen wird nur das asymptotische Wachstum der Zieldimension bewiesen, nicht aber, welcher konstante Faktor dahinter steckt. Um die Einbettungsverfahren experimentell besser analysieren und vergleichen zu können, wird zunächst in einem Experiment diese Konstante bestimmt. Zwar lässt sich diese auch theoretisch berechnen, allerdings ist dies meist mit sehr großzügigen Abschätzungen verbunden. Das empirische Vorgehen liefert daher für die Praxis genauere Werte.

- Welches Verfahren hat bei gegebenen Approximationsgrad die beste Laufzeit? Da auch Verfahren untersucht werden, die die dünne Besetzung der Eingabe explizit ausnutzen, wird auch die Anzahl der Nicht-Null-Einträge variiert. Außerdem wird die Frage beantwortet, ab welcher Anzahl diese Verfahren den besten konventionellen Verfahren überlegen sind.

- In manchen Anwendungen kann es wichtig sein, die Dimension der Datenpunkte möglichst stark zu reduzieren. Hierzu wird verglichen, welche Epsilon-Schranke die einzelnen Verfahren bei konstantem Speicherverbrauch bzw. konstanter Zieldimension einhalten.

- Einige Verfahren erfordern in der Theorie vollständige Unabhängigkeit, die allerdings nur mit hohem Rechenaufwand und Speicherbedarf erreicht werden kann. Wie stark weicht das Ergebnis bei verschiedenen Unabhängigkeitsgraden vom Sollwert ab?

- Gibt es Anwendungsszenarien bei denen eine Verkettung verschiedener Einbettungsverfahren lohnenswert ist? Vorstellbar wäre zum Beispiel zunächst ein schnelles Verfahren anzuwenden, welches aber keine hinreichend große Dimensionsreduktion zulässt. Anschließend wird das Ergebnis von einen anderen Verfahren eingebettet, das zwar langsamer ist, aber die Dimension nochmals reduzieren kann. Die hohe Laufzeit des zuletzt angewandten Verfahren würde aber nicht besonders stark ins Gewicht fallen, da bereits das erste Verfahren die Dimension wesentlich reduziert hat.

Kapitel 2

Theoretischer Teil

In diesem Kapitel wird die nötige Theorie erklärt, die zum Verständnis der Versuche im experimentellen Teil erforderlich ist. Zunächst wird dabei das Johnson-Lindenstrauss-Lemma erklärt, das einen zentralen Satz beim Thema Einbettungen darstellt. Anschließend werden die verschiedenen Einbettungsverfahren vorgestellt, die im zweiten Teil experimentell untersucht werden. Am Ende dieses Kapitels geht es um das Erstellen von Zufallszahlenfolgen mit wahlfreien Zugriff auf beliebige Elemente, die alle vorgestellten Verfahren benötigen.

2.1 Notation

Im Folgenden sollen kurz die verwendeten Variablen und Symbole sowie ihre Bedeutung vorgestellt werden. Mit x_i wird i-te Eintrag des Vektors x bezeichnet. $\|x\|_2$ sei die euklidische Länge von x, das heißt $\|x\|_2 \coloneqq \sqrt{\sum_{i=1}^{d} x_i^2}$, wobei d für die Anzahl der Dimensionen von x steht. Die Menge der natürlichen Zahlen von 1 bis m wird mit $[m]$ notiert. Der Datensatz, der analysiert werden soll, besteht aus n Punkten bzw. Vektoren im d-dimensionalen Raum. Die Menge aller Punkte bzw. Vektoren wird dabei mit N bezeichnet. Jeder Vektor enthält genau s Nicht-Null-Einträge; falls $d = s$ gilt, sind die Vektoren also voll besetzt. Die Einbettungsfunktion $\pi : \mathbb{R}^d \to \mathbb{R}^k$ bildet jeden Vektor des Datensatzes auf einen k-dimensionalen Vektor ab.

2.2 Johnson-Lindenstrauss-Lemma

Bei dem Johnson-Lindenstrauss-Lemma handelt es sich um einen sehr wichtigen Satz in der Theorie der Einbettungen, da es beweist, dass überhaupt Einbettungen mit den gewünschten Eigenschaften existieren. Das Lemma wurde bereits 1984 von Johnson und Lindenstrauss bewiesen und besagt formal Folgendes [9]:

2.2.1 Satz. *Gegeben sei eine Menge N von n Punkten in $\mathbb{R}^d$, ein festes $\epsilon > 0$ und eine hinreichend groß gewählte Konstante c. Dann existiert eine Einbettung $\pi : \mathbb{R}^d \to \mathbb{R}^k$ mit $k = c \cdot \ln n/\epsilon^2 < d$, sodass mit einer Wahrscheinlichkeit von $\frac{2}{3}$ für alle Punkte $p, q \in N$ gilt:*

$$(1-\epsilon)\|p-q\|_2 \leqslant \|\pi(p)-\pi(q)\|_2 \leqslant (1+\epsilon)\|p-q\|_2$$

Bemerkenswert ist, dass die Anzahl der Dimensionen der eingebetteten Vektoren unabhängig von der ursprünglichen Dimension ist. Im Folgenden werden nun verschiedene Einbettungen vorgestellt, die im zweiten Teil dieser Arbeit experimentell untersucht werden. Dabei bezeichnet n stets die Anzahl der Datenpunkte, d deren Dimension und k die Dimension der eingebetteten Daten. Die Menge aller ursprünglichen Punkte sei N.

2.3 Dimensionsreduktion mit Rademachermatrizen

Beim einfachsten Einbettungsverfahren (Achilioptas [1]) wird zur Einbettung eine $(k \times d)$-Rademacher-Matrix R verwendet. Die Einträge sind unabhängige Zufallsvariablen, die mit Wahrscheinlichkeit 1/2 gleich +1 und mit Wahrscheinlichkeit 1/2 gleich -1 sind. Die Einbettung eines Punktes p ist dann

$$\pi(p) = \frac{1}{\sqrt{k}} \cdot R \cdot p$$

Die Variable k wird dabei folgendermaßen gewählt:

2.3.1 Theorem. *Seien die Eingabeparameter $\beta, \epsilon > 0$ gegeben. Dann sei*

$$k \geqslant \frac{4+2\beta}{\epsilon^2/2 - \epsilon^3/3} \cdot \ln n$$

Mit Wahrscheinlichkeit $1 - n^{-\beta}$ gilt für alle Punkte $p, q \in N$:

$$(1-\epsilon)\|p-q\|_2 \leqslant \|\pi(p)-\pi(q)\|_2 \leqslant (1+\epsilon)\|p-q\|_2$$

Bei den Zufallsvariablen der Rademacher-Matrix R reicht dabei eine vierfache unabhängig aus [4]. Dies ist für die praktische Umsetzung von hoher Bedeutung, da nur bei beschränkter Unabhängigkeit die Einbettungsmatrix implizit gespeichert werden kann. Wie die implizite Speicherung von Zufallsvariablen realisiert werden kann, wird im Abschnitt 2.7 näher erläutert. Durch die beschränkte Unabhängigkeit können die einzelnen Elemente der Rademacher-Matrix in konstanter Zeit berechnet werden. Für die Einbettung eines Vektores ergibt sich somit eine Laufzeit von $\mathcal{O}(d \cdot k)$.

Es ist möglich, zufällig 2/3 der Einträge von R auf null zu setzen, ohne dafür „als Ausgleich“ k oder ϵ erhöhen zu müssen. Die Wahrscheinlichkeitsverteilung der Elemente von R ist dann gleich:

$$r_{ij} = \begin{cases} +1, & \text{mit Wahrscheinlichkeit } 1/6 \\ 0, & \text{mit Wahrscheinlichkeit } 2/3 \\ -1, & \text{mit Wahrscheinlichkeit } 1/6 \end{cases}$$

Da R nun weniger $+1$ bzw. -1 enthält, verändert sich die Abbildungsgleichung zu

$$\pi(p) = \sqrt{\frac{3}{k}} \cdot R \cdot p$$

Nutzt man bei der Berechnung die Nullen der Einbettungsmatrix explizit aus (indem die jeweilige Multiplikation nicht durchgeführt wird), verringert sich die Laufzeit um einen konstanten Faktor.

2.4 Schnelle Johnson-Lindenstrauss Transformation (SRHT)

Wenn man überlegt, wie man die Anzahl der Nullen in der Einbettungsmatrix weiter verringern kann, kommt man möglicherweise zu der Idee einfach k zufällige Einträge eines Eingabevektors auswählen, die dann den eingebetteten Vektor bilden. Bei sehr dünn besetzten Daten scheitert diese Methode jedoch:

2.4.1 Gegenbeispiel. Angenommen der d-dimensionale Eingabevektor p hat lediglich einen Nicht-Null-Eintrag mit dem Wert c. Wird dieses Element bei der Einbettung nicht ausgewählt, so hat der eingebettete Vektor $\pi(p)$ die Länge 0. Enthält der Datensatz den Nullvektor 0, der bei allen linearen Abbildung auf sich selbst abgebildet wird, so wird für jedes $\epsilon < 1$ die Ungleichung

$$(1-\epsilon)\|p-q\|_2 \leqslant \|\pi(p)-\pi(0)\|_2 \tag{2.1}$$

$$\Leftrightarrow (1-\epsilon)\cdot c \leqslant 0 \tag{2.2}$$

des Johnson-Lindenstrauss-Lemmas verletzt. Die Ungleichung 2.1 ist nur dann korrekt, wenn der Wert c des Eingabevektors auch im eingebetteten Vektor vorhanden ist. Da k Einträge des d-dimensionalen Vektors ausgewählt werden, beträgt die Wahrscheinlichkeit einen bestimmten Eintrag auszuwählen k/d. Im Johnson-Lindenstrauss-Lemma wird jedoch gefordert, dass die Ungleichung 2.1 mit einer Wahrscheinlichkeit von 2/3 erfüllt wird. Damit k/d gleich 2/3 ist, muss als Zieldimension k der Wert $\frac{2}{3}d$ gewählt werden. Allerdings hängt dann die Zieldimension k linear von d ab; die gewünschte Eigenschaft für Einbettungen war aber, dass k unabhängig von d ist. Das einfache Auswählen von k Einträgen bei beliebigen Datensätzen ist somit kein sinnvolles Einbettungsverfahren. □

Trotzdem bildet diese Methode die Grundidee der schnellen Johnson-Lindenstrauss Transformation, kurz FJLT (engl. *Fast Johnson-Lindenstrauss Transform*) [2] und deren Weiterentwicklung der SRHT (engl. *Subsampled Randomized Hadamard Transform*) [14]. Damit das gerade beschriebene Problem nicht auftritt, werden bei der SRHT die Elemente des Eingabevektors zunächst miteinander „verwischt“, sodass der Vektor höchst wahrscheinlich keine Nullen mehr enthält. Danach ist es unproblematisch, k Einträge für den eingebetteten Vektor auszuwählen. Bei dem „Verwischen“ handelt es sich um eine Art Fourier-Transformation. Diese kann (ähnlich wie bei der schnellen Fourier-Transformation) durch

einen Teile-und-Herrsche-Algorithmus (Divide and Conquer) effizient berechnet werden. Insbesondere geht beim „Verwischen“ keine Information verloren — ein erneutes Anwenden der Transformation liefert wieder die ursprünglichen Daten.

Das folgende Theorem beschreibt die Berechnung der SRHT:

2.4.2 Theorem. *Gegeben sei eine Menge N von Vektoren im $\mathbb{R}^d$, $\epsilon < 1$ und eine hinreichend groß gewählte Konstante c. Es wird im folgenden angenommen, dass d eine Zweierpotenz ist. Sollte dies nicht der Fall sein, werden die Vektoren implizit mit Nullen aufgefüllt. Die Dimension der eingebetteten Vektoren ist dann*

$$k = \frac{c \cdot \ln n}{\epsilon^2}$$

Die Einbettung eines Vektors p berechnet sich durch $\pi(p) = P \cdot H \cdot D \cdot p$:

- *P ist eine $(k \times d)$-Matrix. In jeder Zeile steht genau eine Eins, wobei die Positionen der Einsen unabhängig voneinander gewählt werden.*
- *H ist eine $(d \times d)$-Hadamard-Matrix. Es gilt:*

  $$H_{ij} = (-1)^{\langle i-1, j-1 \rangle}$$

  Dabei ist $\langle x, y \rangle$ das Skalarprodukt von x und y in Binärdarstellung.
- *D ist eine $(d \times d)$-Diagonal-Matrix. Jedes Element der Hauptdiagonalen ist mit Wahrscheinlichkeit 1/2 gleich +1 und mit Wahrscheinlichkeit 1/2 gleich -1. Alle anderen Einträge sind gleich null.*

Die Multiplikation $D \cdot p$ benötigt $\mathcal{O}(d)$ Zeit, da D eine Diagonalmatrix ist. Mit Hilfe der Walsh-Hadamard-Transformation lässt sich die Multiplikation mit H in $\mathcal{O}(d \log d)$ Operationen berechnen. Dabei handelt es sich um einen Teile-und-Herrsche-Algorithmus (Divide and Conquer), bei der eine besondere Eigenschaft der Hadamard-Matrix ausgenutzt wird [7]:

Sei H_q eine $(q \times q)$-Hadamard-Matrix. Dann gilt:

$$H_q = \begin{bmatrix} H_{q/2} & H_{q/2} \\ H_{q/2} & -H_{q/2} \end{bmatrix}$$

Die Multiplikation $H \cdot Dx$ ist somit gleich

$$\begin{bmatrix} H_{q/2} & H_{q/2} \\ H_{q/2} & -H_{q/2} \end{bmatrix} \cdot \begin{bmatrix} Dx_1 \\ Dx_2 \end{bmatrix}$$

wobei mit Dx_1 die obere und mit Dx_2 die untere Hälfte von Dx bezeichnet wird.

$$\begin{bmatrix} H_{q/2} & H_{q/2} \\ H_{q/2} & -H_{q/2} \end{bmatrix} \cdot \begin{bmatrix} Dx_1 \\ Dx_2 \end{bmatrix} = \begin{bmatrix} H_{q/2}Dx_1 + H_{q/2}Dx_2 \\ H_{q/2}Dx_1 - H_{q/2}Dx_2 \end{bmatrix}$$

Es reicht also aus, die Produkte $H_{q/2} \cdot Dx_1$ und $H_{q/2} \cdot Dx_2$ jeweils nur einmal zu berechnen. Für deren Berechnung wird der gerade beschriebene Rekursionsschritt erneut angewandt. Wenn $q = 1$ ist, wird die Rekursion abgebrochen:

$$H_1 \cdot Dx = Dx$$

Als Laufzeit ergibt sich beim Rekursionsschritt $T(d) = 2 * T(d/2) + \mathcal{O}(d)$ und beim Rekursionsabbruch $T(1) = 1$. Daraus folgt, dass $T(d) \in \mathcal{O}(d \log d)$ ist.

Schließlich fehlt noch die Multiplikation mit P. Werden von P die Positionen der Nicht-Null-Einträge gespeichert, ist eine Laufzeit von $\mathcal{O}(k)$ möglich. Die Gesamtlaufzeit beträgt somit $\mathcal{O}(d \log d + k)$.

2.5 Dünn besetzte Einbettungsmatrizen (CW)

Im vorherigen Abschnitt wurde gezeigt, dass es nicht ausreicht lediglich k zufällige Elemente des Eingabevektors auszuwählen: Bei dünn besetzten Daten könnte es sonst vorkommen, dass entscheidende Einträge nicht berücksichtigt werden. Das Einbettungsverfahren von Clarkson und Woodruff [5] liest jeden Eintrag der Eingabe und addiert und subtrahiert ihn auf einen zufälligen Eintrag der Einbettung. Die $(k \times d)$-Einbettungsmatrix S enthält somit in jeder Spalte an genau einer zufälligen Position entweder eine -1 (mit Wahrscheinlich $1/2$) oder $+1$ (mit Wahrscheinlich $1/2$). Die Einbettung eines Vektors p ist dann gleich $\pi(p) = S \cdot p$.

In der Publikation von Clarkson und Woodruff werden jedoch andere Gütekriterien betrachtet, die sich unter anderem auf lineare Regression beziehen. In wie weit bei Anwendung des Verfahrens auch der paarweise Abstand beibehalten wird, werden die experimentellen Analysen zeigen. Ebenso wird untersucht, wie k zur Einhaltung einer bestimmten ϵ-Schranke gewählt werden muss.

2.6 Kane und Nelson (KN)

Alle bisher beschriebenen Verfahren nutzen die dünne Besetzung der Eingabe nicht explizit aus. Es ist natürlich möglich, die Algorithmen effizient zu gestalten, indem nur die Nicht-Null-Einträge verarbeitet werden. Es gibt aber auch Verfahren, die speziell für dünn besetzte Eingabevektoren entwickelt wurden.

Das Einbettungsverfahren von Kane und Nelson [10] bildet jeden Nicht-Null-Eintrag des Eingabevektors auf genau b Elemente des Ausgabevektors ab. Dazu wird der Ausgabevektor vorab in b Bereiche unterteilt, die jeweils aus k/b Elementen bestehen. Wenn nun ein Eintrag p_i eines Eingabevektors p eingebettet wird, wird von jedem der b Bereiche genau ein Eintrag ausgewählt, auf dem dann der Wert p_i addiert und subtrahiert wird. Die Abbildung 2.1 veranschaulicht das Vorgehen.

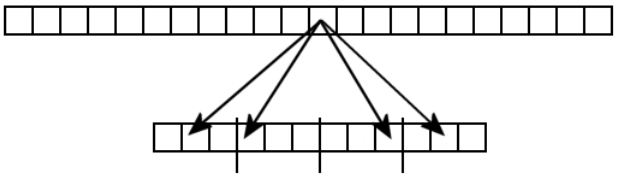

Abbildung 2.1: Einbettungsverfahren von Kane und Nelson [10]. Jede Eintrag p_i des Eingabevektors wird hier auf $b = 4$ Bereiche der Einbettung projiziert. Dabei wird p_i auf genau einen der $k/b = 3$ möglichen Einträge eines jeden Bereichs addiert oder von diesem subtrahiert.

Formal ist die Einbettung auf folgende Weise definiert:

2.6.1 Theorem. *Gegeben seien eine Menge N von n Vektoren im $\mathbb{R}^d$, $\epsilon < 1$, $\delta < 1/2$ und eine hinreichend groß gewählte Konstante C. Für die Zieldimension k gilt dann:*

$$k \geqslant \frac{C \cdot \ln n \cdot \ln(1/\delta)}{\epsilon^2}$$

Sei $b \geqslant \frac{2 \ln n \ln(1/\delta)}{\epsilon}$ eine ganze Zahl, sodass k durch b teilbar ist. Man wähle nun zwei Hashfunktionen $\sigma : [d] \times [b] \to \{-1, +1\}$ und $h : [d] \times [b] \to [k/b]$, wobei σ mindestens $2\ln(1/\delta)$-fache Unabhängigkeit ausweisen muss und h eine Unabhängigkeit vom Grad $\mathcal{O}(\ln(d/\delta))$ benötigt. Des Weiteren sollte es für jedes feste $i \in [d]$ und jedes feste $r \in [b]$ nur ein $j \in [k/b]$ geben, sodass $h(i, r) = j$ ist.

Die Einbettung eines Vektors p berechnet sich dann durch die Formel

$$\pi(p)_{r \cdot \frac{k}{b} + j} = \frac{1}{\sqrt{b}} \sum_{i=1}^{n} 1_{\{h(i,r)=j\}} \sigma(i, r) p_i$$

Mit einer Wahrscheinlichkeit von $1 - \delta$ gilt für alle Eingabevektoren $p, q \in N$:

$$(1 - \epsilon)\|p - q\|_2 \leqslant \|\pi(p) - \pi(q)\|_2 \leqslant (1 + \epsilon)\|p - q\|_2$$

2.7 Zufallszahlenfolgen

Für alle erwähnten Verfahren werden Zufallszahlen einer bestimmten Unabhängigkeit benötigt. Da die Einbettungsmatrix meist sogar größer ist als der Datensatz selbst, kommt eine explizite Speicherung nicht in Frage. Von daher müssen die Zufallsvariablen implizit gespeichert: Möchte man den Wert eines bestimmten Eintrags in der Einbettungsmatrix wissen, so wird eine (vom verwendeten Verfahren abhängige) Formel ausgewertet.

Es gibt zahlreiche Zufallsgeneratoren, die dies bewerkstelligen. Das BCH-Verfahren [3] ist dabei von der Laufzeit her eines der schnellsten Generatoren [13], von daher verwende ich BCH auch in meiner Implementierung.

Um $(2k+1)$-fach unabhängige Zufallsvariablen zu erzeugen, wird ein zufälliger Seed-Vektor $S = [s_0, S_0, ..., S_{k-1}]$ verwendet, wobei s_0 ein Bit groß ist und jedes S_i aus n Bits besteht. Der i-te Wert der Zufallszahlenfolge wird dann durch folgende Funktion berechnet:

$$f(S, i) = S \cdot [1, i, ..., i^{2k-1}]$$

Dabei wird das Skalarprodukt über dem Körper $\mathbb{Z}/2\mathbb{Z}$ berechnet und die Potenzen von i über der Körpererweiterung $GF(2^n)$. Als Ergebnis erhält man somit den Wert 0 oder 1. Auf diese Weise kann eine Folge von 2^n Zufallsvariablen erzeugt werden. Reichen $2k$-fach unabhängige Zufallszahlen aus, kann das Bit s_0 weggelassen werden.

Bei einer gewöhnlichen Implementierung wählt man für n den Wert 32: Die Potenzen können dann auf folgende Weise berechnet werden: i wird in ein 64-bit langes Wort geschrieben und anschließend mit sich selbst multipliziert. Von dem Ergebnis werden nur die unteren 32 Bits behalten. Anschließend wird erneut mit i multipliziert. Nach erneutem Löschen der oberen 32 Bits erhält man das Ergebnis von i^3. Für die höheren Potenzen wiederholt man die beschriebenen Schritte.

Bei der Berechnung des Skalarproduktes werden zunächst die 32-bit großen Potenzen von i mit den jeweiligen 32-bit großen Teilvektoren des Seed-Wertes „ver-UND-et“(Operator & in C++). Anschließend wird die Parität der Anzahl der Einsen in den jeweils 32-bit großen Teilergebnissen bestimmt. Die Gesamtparität ist dann die berechnete Zufallszahl.

Kapitel 3

Experimenteller Teil

3.1 Implementierung und Durchführung

3.1.1 Verwendete Abkürzungen

Für die im vorherigen Kapitel beschriebenen Einbettungsverfahren werden nun die folgenden Abkürzungen benutzt:

BCH-Sketch	Voll besetzte Rademachermatrizen	[1]
BCH-Null	Zu einem Drittel besetzte Rademachermatrizen	[1]
SRHT	Schnelle Johnson-Lindenstrauss Transformation	[2]
CW	Dünn besetzte Einbettungsmatrix von Clarkson und Woodruff	[5]
KN	Einbettungsverfahren von Kane und Nelson	[10]

3.1.2 Implementierung von BCH-Null

Bei der Implementierung von BCH-Null besteht das Problem, dass Zufallswerte erzeugt werden müssen, dessen Wahrscheinlichkeiten keine Zweierpotenzen sind. Konkret sieht die Wahrscheinlichkeitsverteilung folgendermaßen aus:

$$r_{ij} = \begin{cases} +1, & \text{mit Wahrscheinlichkeit } 1/6 \\ 0, & \text{mit Wahrscheinlichkeit } 2/3 \\ -1, & \text{mit Wahrscheinlichkeit } 1/6 \end{cases}$$

Der verwendete Zufallsgenerator BCH (siehe Abschnitt 2.7) erzeugt jedoch einzelne Bits, sodass zunächst nur Wahrscheinlichkeiten der Form $1/2^n$ mit $n \in \mathbb{N}$ generiert werden können. Dieses Problem kann umgangen werden, indem in bestimmten Fällen einfach eine neue Zufallszahl erzeugt wird. Beispielweise könnte man drei Zufallsbits erstellen, aber bei Zweien der insgesamt acht möglichen Folgen, werden erneut drei Zufallsbits erzeugt und zwar so lange, bis eine der sechs „gewünschten“ Folgen generiert wird. Solch eine Funktion kann sechs verschiedene Ergebnisse mit einer Wahrscheinlichkeit von jeweils 1/6 liefern.

Genau diese Idee wird in meiner Implementierung realisiert: Und zwar wird zunächst ein Bit erzeugt; sollte dieses gleich null sein, wird 0 zurückgegeben. Andernfalls wird ein weiteres Bit erzeugt; ist dieses ebenfalls gleich null, wird noch ein Bit erzeugt, das entscheidet, ob $+1$ oder -1 als Ergebnis zurückgegeben wird. Sollte das zweite Bit jedoch gleich eins sein, beginnt der Verfahren erneut, das heißt, es wird eine anderes „erstes“ Bit generiert. Der Algorithmus sieht in Pseudocode also wie folgt aus:

Algorithmus 1 CreateEntryForBCHNull($index$)

```
1: if bch(3 · index) == 0 then
2:   return 0
3: else
4:   if bch(3 · index + 1) == 0 then
5:     return 2 · bch(3 · index + 2) − 1
6:   else
7:     return CreateEntryForBCHNull(index + d · k)
8:   end if
9: end if
```

Die Funktion $\mathrm{bch}(x)$ generiert dabei das x-te Zufallsbit einer durch das BCH-Verfahren erzeugten Zufallszahlenfolge. Für den rekursiven Aufruf wird das Produkt $d \cdot k$ addiert, da dies der Anzahl der Einträge der implizit gespeicherten Rademachermatrix R entspricht. So wird sichergestellt, dass jedes Bit der Zufallszahlenfolge nur für höchstens einen Eintrag von R verwendet wird.

Die erwartete Anzahl an erstellten Bits pro Eintrag in der Rademachermatrix ist:

$$\mathbb{E}[Bits] = \frac{1}{2} \cdot 1 + \frac{1}{8} \cdot 3 + \frac{1}{8} \cdot 3 + \frac{1}{4} \cdot (2 + \mathbb{E}[Bits])$$

Diese Gleichung lässt sich umformen in:

$$\mathbb{E}[Bits] = 3{,}5$$

Somit müssen im Durchschnitt 3,5 Zufallsbits berechnet werden. Dies ist natürlich erheblich mehr als bei BCH-Sketch, welches nur 1 Zufallsbit benötigt. Von daher kann man damit rechnen, dass BCH-Null nicht dreimal so schnell wie BCH-Sketch sein wird.

3.1.3 Implementierung von SRHT

Im theoretischen Teil wurde bereits erwähnt, wie man die Multiplikation mit einer Hadamard-Matrix effizient programmieren kann. In diesem Abschnitt geht es darum, wie man das Auswählen der k Einträge durch die Matrix P in die rekursive Berechnung integriert und somit die Laufzeit nochmals beschleunigt. Dafür wird P implizit in einem Feld p der

Länge k gespeichert, sodass p die Spaltenindizes der Einsen in P aufsteigend sortiert enthält. Bei der Einbettung werden also von dem Vektor $H \cdot D \cdot x$ die $p[i]$-ten Einträge (für alle $i \in 1, ..., k$) ausgewählt. Vor Beginn der Rekursion wird die Multiplikation $D \cdot x$ explizit berechnet und in einem Vektor $\tilde{x}$ gespeichert.

Für den Rekursionsschritt betrachten wir nochmal die Ausgangsgleichung des SRHT-Verfahrens:

$$\begin{aligned} P \cdot H \cdot D \cdot x &= P \cdot H \cdot \tilde{x} \\ &= \begin{bmatrix} P_1 & P_2 \end{bmatrix} \cdot \begin{bmatrix} H_{q/2} & H_{q/2} \\ H_{q/2} & -H_{q/2} \end{bmatrix} \cdot \begin{bmatrix} \tilde{x}_1 \\ \tilde{x}_2 \end{bmatrix} \\ &= \begin{bmatrix} P_1 & P_2 \end{bmatrix} \cdot \begin{bmatrix} H_{q/2}\tilde{x}_1 + H_{q/2}\tilde{x}_2 \\ H_{q/2}\tilde{x}_1 - H_{q/2}\tilde{x}_2 \end{bmatrix} \\ &= P_1 H_{q/2} \cdot (\tilde{x}_1 + \tilde{x}_2) + P_2 H_{q/2} \cdot (\tilde{x}_1 - \tilde{x}_2) \end{aligned}$$

In jedem Rekursionsschritt wird die Summe $(\tilde{x}_1 + \tilde{x}_2)$ und die Differenz $(\tilde{x}_1 - \tilde{x}_2)$ berechnet. Außerdem wird das Array p in zwei Teile geteilt, sodass der Erste die Werte von 1 bis $q/2$ enthält und der Zweite die Werte von $q/2+1$ bis q. Da p sortiert vorliegt, kann man den mittleren Wert $q/2$ effizient durch binäre Suche finden. Um die Multiplikationen $P_1 H_{q/2} \cdot (\tilde{x}_1 + \tilde{x}_2)$ bzw. $P_2 H_{q/2} \cdot (\tilde{x}_1 - \tilde{x}_2)$ zu berechnen, werden die beteiligten Matrizen erneut geteilt, das heißt die gerade beschriebene Methode wird erneut angewandt. Sobald das übergebene Feld p_i keinen oder nur noch einen Eintrag besitzt, wird die Rekursion abgebrochen: Im erstgenannten Fall ist nichts weiter zu tun, denn die zu p_i zugehörige Teilmatrix P_i enthält nur Nullen. Wenn p_i noch genau einen Wert a enthält (zweiter Fall), wird das Skalarprodukt der a-ten Zeile von $H_{q/2}$ mit dem Vektor $\tilde{x}_i$ berechnet und auf den entsprechenden Eintrag des Ergebnisvektors addiert.

3.1.4 Implementierung von KN

Bei dem Einbettungsverfahren KN muss neben der Zieldimension k eine Anzahl an Bereichen b gewählt werden, sodass k/b eine ganze Zahl ist. Außerdem wird eine Hashfunktion benötigt, die auf die Menge $\{1, ..., k/b\}$ abbildet. Um ähnliche Probleme wie bei BCH-Null (siehe Abschnitt 3.1.2) zu umgehen, wird in meiner Implementierung k/b stets als Zweierpotenz gewählt, sodass für jeden Hash-Wert genau $\log_2(k/b)$ viele Zufallsbits mit dem BCH-Generator erzeugt werden. In der Theorie [10] sind bereits untere Schranken für b und k in Abhängigkeit von dem gewählten ϵ und δ angegeben, das heißt, k muss gegebenenfalls um bis zu einem Faktor von 2 größer gewählt werden, damit k/b eine Zweierpotenz ist. Konkret wurde die Wahl von b und k wie folgt implementiert, wobei C eine hinreichend groß gewählte Konstante ist:

1. $b := \left\lceil \frac{2\ln(1/\delta)}{\epsilon} \right\rceil$

2. $k := \left\lceil \frac{C \cdot \ln(1/\delta) \cdot \ln(n)}{\epsilon^2} \right\rceil$

3. Finde das kleinste m für das gilt: $2^m \geqslant k/b$

4. $k := b \cdot 2^m$

Das beschriebene Vorgehen hat jedoch einen Nachteil, nämlich dass auch bei einer deutlich höheren Zahl n an Eingabevektoren die Zieldimension unter Umständen konstant bleibt. Dies erschwert den Vergleich mit anderen Einbettungsverfahren, wenn man beispielsweise experimentell untersuchen möchte, wie die Laufzeit der Verfahren von n abhängt. Zum Beispiel ergibt sich bei $\epsilon = 0{,}2$, $\delta = 1/3$, $C = 1$ und $n = 700$ dieselbe Zieldimension ($k = 352$) wie bei $n = 70.000$. Erst bei $n = 368.122$ verdoppelt sich die Zieldimension auf $k = 704$. Wird aber möglicherweise nur der Bereich zwischen $n = 700$ und $n = 350.000$ analysiert, sieht es so aus, als ob die Laufzeit des KN-Verfahrens linear von n abhängen würde. Vergleicht man dagegen die Laufzeiten bei $n = 368.121$ und $n = 368.122$, wird man durch die doppelt so große Zieldimension einen deutlichen Anstieg der Laufzeit feststellen. Da das Generieren der Hash-Werte jedoch bei Zweierpotenzen einfacher und effizienter ist, wurde dieser Nachteil in Kauf genommen.

3.1.5 Bewertung der Einbettungen

Bei vielen Experimenten soll bestimmt werden, wie gut die Einbettung die Originaldaten repräsentiert. Dazu muss der paarweise Abstand aller eingebetteten Vektoren mit den Abständen der Eingabevektoren verglichen werden. Aus der größten relativen Abweichung ergibt sich dann die tatsächliche Güte der Einbettung:

$$\epsilon_{\text{out}} \coloneqq \max \left\{ \left| \frac{\|\pi(p) - \pi(q)\|_2}{\|p - q\|_2} - 1 \right| \right\}_{p,q \in N}$$

Dieses Vorgehen ist jedoch mit hohem Rechenaufwand verbunden: Bei n gegebenen Vektoren müssen $\frac{n \cdot (n-1)}{2}$ Abstände berechnet und gespeichert werden. Zum Bestimmen eines Abstandes müssen bei voll besetzten Daten d Differenzen gebildet werden. Insgesamt sind somit $\mathcal{O}(d \cdot n^2)$ viele Operationen nötig. Das Einbetten der Daten kostet selbst bei den langsamen Verfahren BCH-Sketch und BCH-Null nur $\mathcal{O}(n \cdot d \cdot k)$ viel Zeit. Daraus folgt, dass für $k < n$ das Analysieren der Einbettung länger dauert als das Einbetten der Daten selbst. Da für zuverlässige Ergebnisse immer mehrere Datensätze eingebettet wurden, erhöht sich die Laufzeit nochmals um einen konstanten Faktor. Bei Experimenten, bei denen also die Güte der Einbettung bestimmt werden sollte (und nicht nur die Laufzeit), musste aus diesem Grund auf große Werte für n verzichtet werden. Aus diesem Grund wurden für n bei solchen Versuchen meist nur Werte bis 1.000 verwendet, höchstens jedoch bis 10.000.

3.1.6 Durchführung

Die Experimente wurden auf einem Linux-Rechner durchgeführt. Dieser hatte 8 GB Arbeitsspeicher und als Prozessor einen *Intel E7400* mit 2,8 GHz. Als Programmiersprache wurde C++ verwendet, als Compiler der g++ der *GNU Compiler Collection*, Version 4.7.3, wobei stets mit der höchst möglichen Optimierung (Option -O3) kompiliert wurde.

3.2 Untersuchung der theoretischen Epsilon-Schranken

3.2.1 Konstanter Faktor der Zieldimension

Fragestellung Bei diesem Experiment soll der konstante Faktor für die Zieldimension bei den verschiedenen Einbettungsverfahren empirisch bestimmt werden. Dazu wurde auf folgende Weise vorgegangen:

Durchführung Zunächst wurden verschiedene Datensätze generiert: Darunter waren sowohl vollständig zufällige, voll besetzte Daten (Typ „zufällig“ in der Tabelle 3.1), als auch sehr dünn besetzte Eingaben (Typ „dünn“) sowie Vektoren mit ausschließlich $+1$ und -1 Einträgen (Typ „worst“). Die Größe der $(d \times n)$-Matrizen betrug $(5 \cdot 10^5 \times 2)$, $(2 \cdot 10^4 \times 50)$ und $(10^4 \times 1.000)$, wobei in den ersten beiden Fällen jeweils drei Datensätze der Typen „zufällig“ und „worst“ erstellt wurden. Im letzten Fall wurde wegen der großen Vektorenanzahl (siehe Abschnitt 3.1.5) nur ein vollständig zufälliger Datensatz erstellt. Die dünn besetzten Eingabedaten hatten die Größe von $(2 \cdot 10^4 \times 50)$ und enthielten einen bzw. 100 Nicht-Null-Einträge, wobei von jeder der beiden Konfigurationen drei Datensätze generiert wurden.

Zu jedem Datensatz wurden mit jedem Verfahren 15 Einbettungen mit $\epsilon_{in} = 0{,}1$ und $\delta = 1/3$ erstellt. Anschließend wurde die Einbettung mit den Originaldaten verglichen und das sich daraus ergebende ϵ berechnet. Da in der Theorie nur ein Anteil von $(1 - \delta)$ der Einbettungen erfolgreich sind, wurden die $15 \cdot \delta = 5$ größten ϵ-Werte ignoriert. Das unter den restlichen Werten größte ϵ wurde mit ϵ_{in} verglichen und daraus berechnet, um welchen Faktor die Zieldimension k der Einbettung hätte größer sein müssen, damit das ϵ_{in} der Eingabe mit dem ϵ aus dem Datenvergleich übereinstimmt. Der jeweils größte Faktor eines Datensatzes bildet das Ergebnis dieses Tests. Durch Wiederholen dieser Methode, konnte der Faktor approximiert werden.

Prinzipiell ist es egal, ob m Einbettungen auf einem Datensatz berechnet werden oder jeweils eine Einbettung auf m Datensätzen der gleichen Konfiguration. Dies liegt daran, dass sowohl die Einträge der Einbettungen als auch die Einträge der Datensätze zufällig gewählt werden. Somit ist gerechtfertigt, dass für $n = 1.000$ lediglich ein Datensatz generiert wurde. Bei $n = 50$ und $n = 2$ geht das Berechnen der paarweisen Abstände zwischen den

Datenpunkten erheblich schneller als bei $n = 1.000$. Von daher war es unproblematisch, hier trotzdem mehrere Datensätze zu erzeugen.

Ergebnisse Tabelle 3.1 zeigt die errechneten Konstanten. Bei BCH-Sketch und BCH-Null wurde als Grundlage jeweils die Formel aus [1] genommen:[1]

$$k \geqslant \frac{4 + 2 \cdot \frac{\ln(1/\delta)}{\ln n}}{\epsilon^2/2 - \epsilon^3/3} \cdot \ln n$$

Da bei SRHT und KN die genaue Gleichung nicht bekannt war, wurde jeweils die bekannte Formel des Johnson-Lindenstrauss-Lemmas mit einem Koeffizienten von 1 verwendet:

$$k \geqslant \frac{\ln(1/\delta)}{\epsilon^2} \cdot \ln n$$

Datensatz	1	2	3	4	5	6	7	Max
Typ	zufällig	zufällig	worst	worst	dünn	dünn	zufällig	
$n =$	2	50	2	50	50	50	1.000	
$d =$	500.000	20.000	500.000	20.000	20.000	20.000	10.000	
$s =$					1	100		
BCH-Sketch	0,076	0,168	0,307	0,174	0,071	0,163	0,176	0,307
BCH-Null	0,042	0,179	0,079	0,166	0,167	0,188	0,199	0,199
SRHT	0,431	1,497	1,096	1,585	0,758	1,504	1,596	1,596
KN	0,626	0,990	0,747	0,916	0,624	0,988	0,858	0,990

Tabelle 3.1: Gezeigt werden die Maxima der berechneten Koeffizienten für die Zieldimension der einzelnen Einbettungsverfahren bei verschiedenen Datensätzen. n bezeichnet die Anzahl der Datenpunkte und d deren Dimension. Beim Typ „zufällig“ wurden alle Einträge zufällig gewählt. Beim Typ „worst“ ist die erste Spalte der Nullvektor, die darauf folgende Spalte enthält nur Einsen und die weiteren Spalten (sofern vorhanden) enthalten als Einträge entweder $+1$ oder -1. Beim Typ „dünn“ gibt s die Anzahl der Nicht-Null-Einträge pro Vektor an. Die letzte Spalte zeigt den größten Koeffizienten des jeweiligen Verfahrens.

Auswertung Bei BCH-Sketch fällt der (im Vergleich zu den anderen Datensätzen) hohe Wert von 0,307 beim Datensatz 3 auf. Fraglich ist auch, warum bei dem sehr ähnlichen Verfahren BCH-Null dieser nur 0,079 beträgt. Aus diesem Grund wurde speziell für diese Anomalie ein weiterer Test durchgeführt: Als Grundlage wurde die Matrix aus Datensatz 3 genommen und abwechselnd mit BCH-Sketch und BCH-Null Einbettungen erstellt. Die

[1] Die originale Formel aus [1] verwendet statt des Terms $\frac{\ln(1/\delta)}{\ln n}$ die Variable β. Die Versagenswahrscheinlichkeit ist dementsprechend $n^{-\beta}$. Beide Varianten sind äquivalent, wie einfaches nachrechnen zeigt: $n^{-\beta} = n^{-\frac{\ln(1/\delta)}{\ln n}} = \exp\left(\ln n \cdot \frac{-\ln(1/\delta)}{\ln n}\right) = \exp\left(\ln\left(\left(\frac{1}{\delta}\right)^{-1}\right)\right) = \exp\left(\ln\left(\delta\right)\right) = \delta$

Einbettungen wurde mit der Originalmatrix verglichen und daraus das sich jeweils ergebende ϵ berechnet. Die Differenzen $\epsilon_{\text{BCH-Sketch}} - \epsilon_{\text{BCH-Null}}$ wurden addiert. Positive Werte bedeuten somit, dass BCH-Sketch besser ist, negative, dass BCH-Null besser ist. Lässt man dieses Experiment eine Zeit lang laufen, so stellt man fest, dass die Summe mal positiv und mal negativ ist. Eine klare Tendenz ist nicht feststellbar. Somit kann das Ergebnis 0,307 als Ausreißer betrachtet werden.

Für den Laufzeit-Vergleich (siehe 3.3) wurden nun die folgenden Koeffizienten gewählt: Bei BCH-Sketch und BCH-Null jeweils 0,2 (der Ausreißer bei BCH-Sketch wurde ignoriert), bei SRHT 1,6 und bei KN 1,0. Man beachte, dass in der bei BCH-Sketch und -Null zugrunde gelegten Formel bereits implizit der Faktor 8 steckt. Multipliziert mit dem Faktor 0,2 ergibt sich somit der selbe Koeffizient wie bei SRHT, nämlich 1,6. Bezüglich SRHT sollte noch erwähnt werden, dass in der empirischen Untersuchung von Venkatasubramanian und Wang [15] ein Koeffizient zwischen 1,5 und 1,7 bestimmt. Dies passt also sehr gut zu den Ergebnissen dieser Arbeit.

Bei BCH-Sketch und BCH-Null wurde in der Theorie [1] eine konkrete Formel für die Zieldimension k angegeben. Trotzdem zeigten die empirischen Tests, dass in der Praxis eine um Faktor 5 kleinere Zieldimension zur Einhaltung der ϵ-Schranke ausreicht. Aus diesem Grund ist verständlich, warum in der Literatur bei SRHT, CW und KN nur noch das asymptotische Wachstum der Zieldimension betrachtet wurde.

3.2.2 Asymptotisches Wachstum bei CW

Fragestellung und Durchführung Bei dem Einbettungsverfahren von Clarkson und Woodruff [5] war zunächst unklar, wie das asymptotische Wachstum der Zieldimension aussieht bzw. ob es sich überhaupt zur Dimensionsreduktion eignet. Testweise wurde für die Zieldimension dieselbe Formel wie bei SRHT und KN gewählt, also $k \geqslant \frac{\ln(1/\delta)}{\epsilon^2} \cdot \ln n$. Der Rest des Versuchs lief genauso wie bei der Untersuchung der anderen Einbettungsverfahren (Abschnitt 3.2.1). Tabelle 3.2 zeigt die sich ergebenden Faktoren.

Datensatz	1	2	3	4	5	6	7	Max
Typ	zufällig	zufällig	worst	worst	dünn	dünn	zufällig	
$n =$	2	50	2	50	50	50	1.000	
$d =$	500.000	20.000	500.000	20.000	20.000	20.000	10.000	
$s =$					1	100		
CW	0,757	1,333	0,746	1,296	57,925	1,452	1,095	57,925

Tabelle 3.2: Gezeigt werden die Maxima der berechneten Koeffizienten für CW. Die Datensätze wurden wie bei Tabelle 3.1 erstellt.

Auswertung Der sehr große Koeffizienten von fast 60 bei dem Datensatz 5 legt nahe, dass die benötigte Zieldimension bei CW nicht wie bei den anderen Verfahren in $\mathcal{O}\left(\frac{logn}{\epsilon^2}\right)$ liegt. Eine theoretische Überlegung zu sehr dünn besetzten Daten (wie bei Datensatz 5) kann das Scheitern von CW erklären:

Theoretische Betrachtung Gegeben seien n Vektoren, die jeweils an (paarweise) unterschiedlichen Positionen genau eine Eins und sonst nur Nullen enthalten. Somit haben alle Vektoren den Abstand $\sqrt{2}$ voneinander. O. B. d. A. werde die Eins im ersten Vektor p_1 beim Einbetten durch die CW-Methode auf die erste Koordinate im Zielraum projiziert und dabei das Vorzeichen beibehalten. Wird die Eins des zweiten Vektors p_2 auf dieselbe Koordinate ohne Vorzeichenwechsel projiziert, ist der Abstand der eingebetteten Vektoren $\|\pi(p_1) - \pi(p_2)\|_2$ gleich 0. Damit wäre die Epsilon-Schranke

$$(1-\epsilon)\cdot\sqrt{(2)} = (1-\epsilon)\|p-q\|_2 \leqslant \|\pi(p)-\pi(q)\|_2 = 0$$

für jedes $\epsilon < 1$ verletzt. Für jeden weiteren Vektor, der eingebettet wird, sinkt die Wahrscheinlichkeit, dass er auf einer Position projiziert wird, bei der die Epsilon-Schranke erfüllt bleibt: Bei der Einbettung des i-ten Vektors beträgt diese Wahrscheinlichkeit $\frac{2k-i+1}{2k}$, wobei k gleich der Zieldimension ist. Bei n Vektoren beträgt somit die Gesamtwahrscheinlichkeit einer erfolgreichen Einbettung höchstens

$$P_{Erfolg} \leqslant \prod_{i=1}^{n} \frac{2k-i+1}{2k}$$

Dieses Produkt kann man grob abschätzen, indem man die erste Hälfe der Faktoren auf (den größeren Wert) 1 setzt und die restlichen Faktoren auf den ebenfalls größeren Wert $\frac{2k-n/2}{2k}$:

$$P_{Erfolg} \leqslant \prod_{i=1}^{n} \frac{2k-i+1}{2k} \leqslant \left(\frac{2k-\frac{n}{2}}{2k}\right)^{\frac{n}{2}}$$

Dies lässt sich umformen zu:

$$\left(\frac{2k-\frac{n}{2}}{2k}\right)^{\frac{n}{2}} = \left(1-\frac{1}{\frac{4k}{n}}\right)^{\frac{4k}{n}\cdot\frac{n^2}{8k}}$$

Wegen

$$\lim_{n\to\infty}\left(1-\frac{1}{n}\right)^{n} = \frac{1}{e}$$

geht der obige Term für große n gegen den Wert $\left(\frac{1}{e}\right)^{\frac{n^2}{8k}}$. Das heißt

$$\lim_{n\to\infty} P_{Erfolg} \leqslant \left(\frac{1}{e}\right)^{\frac{n^2}{8k}}$$

Nach Logarithmieren beider Seiten und anschließendem Umformen nach k erhält man (für n geht gegen unendlich):

$$k \geqslant \frac{n^2 \cdot \ln(P_{Erfolg}^{-1})}{8}$$

Somit liegt k in $\Omega(n^2)$.[2] Die eingebettete Matrix hat somit $\Omega(n^3)$ Elemente. Mit einem Speicherverbrauch $\frac{n(n-1)}{2} \in \mathcal{O}(n^2)$ lassen sich jedoch die paarweisen Abstände aller Vektoren des Datensatzes auch exakt speichern. Von daher ist die CW-Methode zur Dimensionsreduktion extrem dünn besetzter Daten unter näherungsweiser Beibehaltung der l_2-Norm aufgrund des hohen Speicherbedarfs eher weniger geeignet.

Da bei allen anderen getesteten Datensätzen die CW-Methode Erfolg hatte, wird in Abschnitt 3.6 die Fragestellung untersucht, wie viele Nicht-Null-Einträge die Eingabevektoren mindestens haben müssen, damit die benötigte Zieldimension k in $\mathcal{O}\left(\frac{\log(1/\delta)}{\epsilon^2} \cdot \log n\right)$ liegt.

3.3 Vergleich der Laufzeit

3.3.1 Laufzeit in Abhängigkeit von der Anzahl der Nicht-Null-Einträge

Fragestellung Bei diesem Experiment wurde die Laufzeit der einzelnen Einbettungsverfahren getestet. Dabei ging es insbesondere um die Abhängigkeit der Laufzeit von der Güte ϵ sowie von der Anzahl der Nicht-Null-Einträge s.

Durchführung Die Eingabedaten enthielten $n = 100$ Vektoren mit je $d = 100.000$ Dimensionen. Die Anzahl der Nicht-Null-Einträge variierte in Zehnpotenzen beginnend von 1 bis zur vollen Besetzung ($s = d = 10^5$). Die dünn besetzten Datensätze ($s < d$) wurden dabei elementeweise eingebettet, während die voll besetzte Matrix zeilenweise bearbeitet wurde (also gleichzeitig die i-te Dimension aller Vektoren). Der Parameter ϵ wurde einmal auf 0,2 und einmal auf 0,05 gesetzt. Die Zieldimension wurde mit den im vorherigen Abschnitt bestimmten Konstanten berechnet. Jede Einbettung wurde dabei 10 mal erstellt, um die Messgenauigkeit der Laufzeiten zu erhöhen. Tabelle 3.3 zeigt die Mittelwerte der gemessenen Zeiten.

Auswertung Besonders in Abbildung 3.1 fällt auf, dass die Laufzeit bei dem voll besetzten Datensatz nicht wie erwartet ansteigt, sondern sogar teilweise geringer ist als bei $s = 10.000$. Die Ursache dafür ist das unterschiedliche Vorgehen beim Einbetten: Wie bereits im Teil *Durchführung* erwähnt, werden bei der voll besetzten Eingabematrix ganze Zeilen gleichzeitig eingebettet. Von daher mussten die entsprechenden Zufallszahlen der Einbettungsmatrix nur einmal berechnet werden. Dies führt letztendlich zu einer geringen

[2] Zu dem Ergebnis, dass k in $\Omega(n^2)$ liegt, kamen auch Nelson und Nguyen [12]. Darüber hinaus konnten sie zeigen, dass eine Zieldimension von $\mathcal{O}(n^2/\epsilon^2)$ ausreicht [11].

$\epsilon = 0,2$ // $s =$	1	10	100	1.000	10.000	100.000
BCH-Sketch	0,00	0,01	0,10	0,97	9,62	3,70
BCH-Null	0,00	0,02	0,19	2,02	18,27	10,80
SRHT	1,90	1,91	1,91	1,92	1,97	1,84
KN	0,00	0,00	0,03	0,27	2,56	1,80
$\epsilon = 0,05$ // $s =$	1	10	100	1.000	10.000	100.000
BCH-Sketch	0,03	0,16	1,46	15,38	133,87	53,83
BCH-Null	0,04	0,30	2,87	28,58	268,98	286,15
SRHT	2,44	2,45	2,45	2,44	2,49	2,37
KN	0,01	0,03	0,16	1,46	14,41	11,05

Tabelle 3.3: Laufzeit bei verschieden dichter Besetzung der Datenmatrix in Sekunden. Der Datensatz enthielt $n = 100$ Vektoren im $d = 100.000$ dimensionalen Raum, wobei jeder Vektor s Nicht-Null-Einträge hatte. δ ist überall gleich 1/3. Es wurden jeweils 10 Einbettungen berechnet und die Laufzeiten gemittelt.

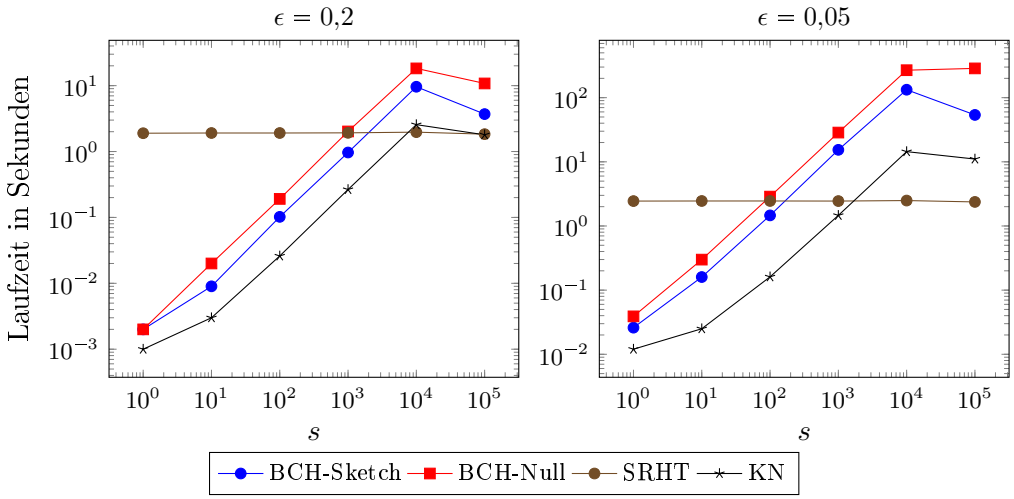

Abbildung 3.1: Abhängigkeit der Laufzeit (y-Achse) von der Anzahl der Nicht-Null-Einträge s (x-Achse). Die Graphen zeigen die Daten aus Tabelle 3.3. Das „Abknicken“ der Graphen bei voller Besetzung ($s = 10^5$) liegt an der zeilenweisen Einbettung.

Laufzeit als bei den dünn besetzten Matrizen, bei denen die Zufallszahlen für jeden Vektor erneut berechnet werden mussten.

Interessanterweise ist BCH-Null langsamer als BCH-Sketch. Dies hat wahrscheinlich folgenden Grund: Bei BCH-Sketch müssen bei der Einbettung eines Wertes k Zufallsbits erzeugt und k Additionen bzw. Subtraktionen durchgeführt werden. Bei BCH-Null dagegen werden $3{,}5 \cdot k$ Zufallsbits generiert (siehe Abschnitt 3.1.2) und $k/3$ Additionen bzw. Subtraktionen durchgeführt. Die Generierung eines Zufallsbits mit der BCH-Methode kostet jedoch deutlich mehr Zeit als eine Addition oder Subtraktion. Aus diesem Grund ist BCH-Null deutlich langsamer. Sind mehrere Einträge einer Zeile der Datenmatrix bekannt (also die i-te Dimension mehrerer Vektoren), kann man die berechneten Zufallsbits mehrfach nutzen, sodass der Vorteil von BCH-Null (nämlich die Nullen in der Einbettungsmatrix) stärker ins Gewicht fällt. Aber auch BCH-Sketch kann optimiert werden: Kommen die Daten zeilenweise, kann man ein Paket von beispielsweise $a = 1000$ Zeilen zwischenpuffern. Der dazugehörige Teil der Einbettungsmatrix wird ebenfalls berechnet und explizit abgespeichert. Anschließend wird das Produkt aus der $(k \times a)$-Einbettungsmatrix und der $(a \times n)$-Datenmatrix mit der schnellen Matrixmultiplikation[3] bestimmt und das Ergebnis auf die Einbettung addiert. Wenn man bei BCH-Null die Nullen der Einbettungsmatrix ausnutzen möchte, ist die Benutzung der schnellen Matrixmultiplikation nicht ohne Weiteres möglich: Man könnte natürlich Teile der Einbettungsmatrix wie bei BCH-Sketch explizit zwischenspeichern, um danach die schnelle Matrixmultiplikation anwenden zu können, allerdings geht dann der Vorteil von BCH-Null (nämlich die Nullen in der Einbettungsmatrix) verloren.

Auch bei SRHT ist eine Optimierung für dünn besetzte Daten nicht direkt möglich. Von daher bleiben die Laufzeiten unabhängig davon, wie dünn die Eingabe besetzt ist, immer ungefähr gleich. Bei dicht besetzen Daten ist SRHT das beste Verfahren. Sind die Daten weniger stark besetzt, wird KN besser.

In einem weiteren Experiment soll nun untersucht werden, wann SRHT und wann KN bei einer gegebenen Kombination von n, d, s und ϵ schneller ist. Die Einbettungsverfahren BCH-Sketch und BCH-Null werden dabei nicht näher untersucht, da in allen untersuchten Fällen entweder SRHT oder KN eine geringere Laufzeit hatten.

3.3.2 Vergleich zwischen SRHT und KN

Vorüberlegungen Vor dem experimentellen Vergleich beider Verfahren, soll zunächst die asymptotische Laufzeit theoretisch betrachtet werden. Wie bereits in Abschnitt 2.4 erwähnt, benötigt bei SRHT die Multiplikation der Datenmatrix mit einer Hadamard-Matrix $\mathcal{O}(d \cdot n \cdot \log d)$ Zeit. Wird bei der Implementierung das Auswählen der k Einträge

[3] Für die schnelle Matrixmultiplikation wurde der Algorithmus von Strassen verwendet, siehe [8]. Theoretisch möglich wäre auch die Benutzung des asymptotisch schnelleren Coppersmith–Winograd Algorithmus [6] oder dessen Verbesserungen [16].

nicht erst nach sondern schon während der Multiplikation durchgeführt (siehe Abschnitt 3.1.3), kann die Laufzeit auf $\mathcal{O}(d \cdot n \cdot k)$ gesenkt werden. Da k in $\mathcal{O}\left(\frac{\log(1/\delta)\cdot\log(n)}{\epsilon^2}\right)$ liegt, ergibt sich eine Laufzeit von

$$T_{SRHT} \in \mathcal{O}\left(n \cdot d \cdot \log \frac{\log(1/\delta) \cdot \log(n)}{\epsilon^2}\right)$$

Bei KN müssen n Vektoren mit je s Nicht-Null-Einträgen eingebettet werden. Jeder Eintrag wird auf b Bereiche projiziert, wobei für jede Projektion $\log_2(k/b)$ Zufallsbits berechnet werden müssen. Als Laufzeit ergibt sich somit $\mathcal{O}(s \cdot n \cdot b \cdot \log(k/b))$. Setzt man die entsprechenden Formeln für b und k ein, erhält man $\mathcal{O}(s \cdot n \cdot \frac{\log(1/\delta)}{\epsilon} \cdot \log \frac{\log n}{\epsilon})$. Da die Unabhängigkeit der Hash-Funktion h nicht konstant, sondern gleich $\ln(n/\delta)$ ist, kommt dieser Faktor noch zur Laufzeit hinzu. Zusammen ergibt sich somit

$$T_{KN} \in \mathcal{O}\left(s \cdot n \cdot \frac{\log(1/\delta)}{\epsilon} \cdot \log \frac{\log n}{\epsilon} \cdot \log \frac{n}{\delta}\right)$$

In den nun folgenden experimentellen Untersuchungen soll überprüft werden, in wie weit die theoretischen Überlegungen mit den tatsächlichen Laufzeiten übereinstimmen.

$n =$	100	300	1.000	3.000	10.000	20.000
SRHT	0,56	1,71	5,74	17,63	80,51	358,38
KN	0,65	1,87	9,05	32,70	104,85	247,12

Tabelle 3.4: Abhängigkeit der Laufzeit von der Anzahl der Vektoren. Die gezeigten Laufzeiten sind Mittelwerte in Sekunden. Die anderen Parameter wurden wie folgt gewählt: $d = 20.000$, $s = 1000$, $\epsilon = 0{,}1$, $\delta = 1/3$.

Abhängigkeit von n Tabelle 3.4 zeigt die Abhängigkeit der Laufzeit von der Vektorenanzahl n. Hier bestätigt sich nun, dass in Abschnitt 3.1.4 beschriebene Problem der KN-Implementierung: Bei $n = 300$ und $n = 10.000$ Vektoren sind beide Verfahren etwa gleich schnell, dazwischen ($n = 3.000$) hat jedoch SRHT eine nur halb so große Laufzeit wie KN. Die Ursache für den hohen Laufzeitanstieg bei KN zwischen $n = 300$ und $n = 1000$ ist wahrscheinlich, dass sich die Anzahl der zu erzeugenden Zufallsbits für die Hash-Funktion h von 5 auf 6 erhöht. Insgesamt ist es äußerst schwierig, Schlüsse aus diesem Experiment zu ziehen: Beispielsweise ist unklar, ob für alle $n \geqslant 20.000$ KN schneller ist oder ob für größere n SRHT und KN etwa gleich schnell bleiben. Aus diesem Grund wurde in einem weiteren Experiment ein größerer Bereich von Werten für n mit mehr Zwischenwerten untersucht.

Der linke Graph von Abbildung 3.2 zeigt die Ergebnisse dieses Tests. Die asymptotische Laufzeitanalyse besagt, dass die Laufzeit beider Verfahren mit $\mathcal{O}(n \cdot \log\log n)$ von n abhängen. Demnach müsste der Quotient $\frac{T(n)}{n\cdot\log\log n}$ konstant sein, wobei $T(n)$ die Laufzeit des jeweiligen Verfahrens bezeichnet. Der rechte Graph zeigt diesen Quotienten. Bis

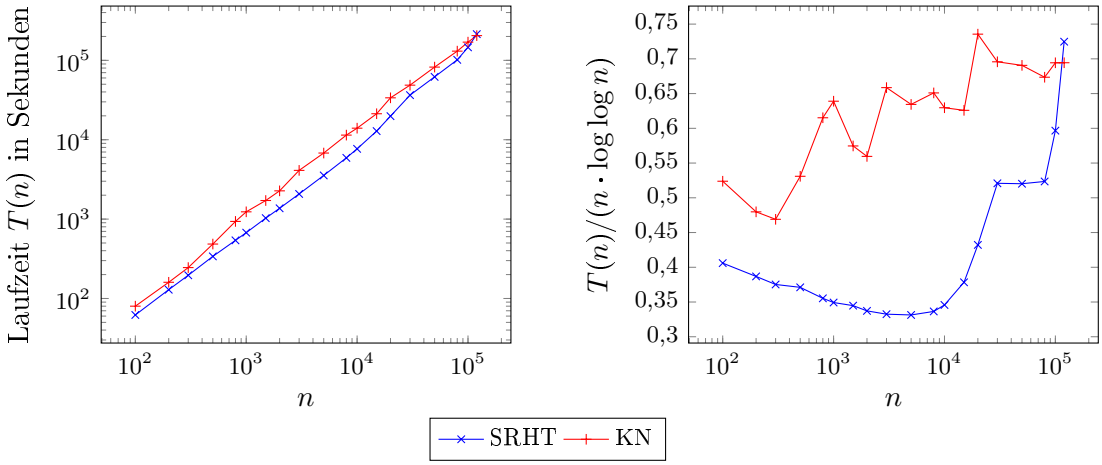

Abbildung 3.2: Der linke Graph zeigt die Laufzeit in Abhängigkeit von der Anzahl der Datenpunkte. Der rechte Graph zeigt die Laufzeit dividiert durch $n \cdot \log\log n$. Die sich ergebenden Quotienten sollten konstant sein. Die restlichen Parameter wurden wie folgt gewählt: $d = 2500$, $s = 125$, $\epsilon = 0{,}1$, $\delta = 1/3$.

$n = 10^4$ sind beide Kurven auch mehr oder weniger konstant, danach steigt die Konstante von SRHT jedoch stark an. In Tabelle 3.4 ist dieses Verhalten bei den letzten beiden Datenpunkten ähnlich: Obwohl sich die Vektoranzahl nur von $n = 10.000$ auf $n = 20.000$ verdoppelt, erhöht sich die Laufzeit von SRHT um mehr als das Vierfache. Dies könnte eventuell damit zusammenhängen, dass sowohl in der Tabelle ($d = 20.000$) als auch im Graphen ($d = 2.500$) der Arbeitsspeicher fast vollgeschrieben wird: Die Eingabematrix selbst benötigt explizit gespeichert $sizeof(\text{double}) \cdot n \cdot d = 8 \cdot n \cdot d$ Bytes an Speicher. Im ersten Fall entspricht dies 2,98 GB[4], im zweiten Fall 2,24 GB. Da in meiner Implementierung alle Vektoren gleichzeitig eingebettet werden, kommt nochmal so viel für die einzelnen Rekursionsstufen hinzu, da in jedem Rekursionsschritt eine halb so große Matrix erstellt wird. Weiterer Speicher wird für die implizit gespeicherte Eingabematrix benötigt, die für das KN-Verfahren benutzt wird, wobei man beachten sollte, dass hier ein großer Overhead durch die Speicherung als binärer Suchbaum entsteht: Für Speicherung eines Wertes (8 Byte) fallen drei Speicheradressen (jeweils 8 Byte) für den linken und rechten Teilbaum sowie für den Vaterknoten an, d.h. die implizite Eingabematrix benötigt nochmals $32 \cdot n \cdot s$ Bytes, also hier 611 MB bzw. 457 MB. Des Weiteren fällt noch Speicher für die eingebettete Matrix selbst an und zwar $8 \cdot n \cdot k$ Bytes: Im ersten Fall sind dies 266 MB, im zweiten Fall 1882 MB. Zusammen kommt man so dem verfügbarem Arbeitsspeicher von 8 GB des Linux-Rechners so schon recht nahe: In beiden Fällen verbrauchen die betrachteten Objekte jeweils zusammen etwa 6,7 GB. Dabei sollte man beachten, dass nur die größten

[4] 1 GB entsprechen in dieser Arbeit stets $1024^3 \approx 1{,}074 \cdot 10^9$ Bytes. Dementsprechend sind 1 MB gleich $1024^2 = 1{,}049 \cdot 10^6$ Bytes

Objekte betrachtet wurden und weitere Quellen mit Overhead außer Acht gelassen wurden. Noch größere Eingabedaten, wie z.B. $n = 30.000$ bei $d = 20.000$ oder $n = 150.000$ bei $d = 2.500$, führten zum Abbruch des Programms wegen fehlendem Speicher. Da in der Implementierung von SRHT sehr oft Speicher angefordert und wieder freigegeben wird, ist der Speicher insgesamt sehr fragmentiert. Die erneute Speicherallokation könnte dann länger dauern als gewöhnlich, sodass die Laufzeit bei größerem n stärker ansteigt als man zunächst annehmen würde.

Letztendlich ändert dies aber nichts daran, dass die asymptotischen Laufzeitformeln für n korrekt sind. Eine sorgfältigere Implementierung von SRHT, in der Speicher nicht mehrfach allokiert und freigegeben wird, würde diese Laufzeitanomalie wahrscheinlich lösen.

$d =$	3.000	10.000	30.000	100.000	300.000
$s =$	150	500	1.500	5.000	15.000
SRHT	0,71	2,80	11,37	41,15	161,98
KN	1,35	4,47	13,44	44,14	137,52

Tabelle 3.5: Abhängigkeit der Laufzeit von der Dimension der Vektoren, wobei 5% der Einträge ungleich null sind. Die gezeigten Laufzeiten sind Mittelwerte in Sekunden. Die anderen Parameter wurden wie folgt gewählt: $n = 1.000$, $\epsilon = 0,1$, $\delta = 1/3$.

Abhängigkeit von d Wie die Laufzeit von der Dimensionsanzahl abhängt, zeigt Tabelle 3.5. Die Vektoren waren dabei jeweils zu 5% gefüllt, damit die beiden Verfahren vergleichbar bleiben. Nach der theoretischen Betrachtung sollten die Laufzeiten beider Methoden linear von d abhängen. In dem oberen Graphen der Abbildung 3.3 sieht man jedoch, dass die Laufzeit von SRHT anscheinend schneller ansteigt als bei KN. In dem linken Graphen wurde ein größerer Bereich von Werten für d getestet. Des Weiteren wurde d immer als Zweierpotenz gewählt, da SRHT die Dimensionsanzahl sowieso implizit auf die nächste Zweierpotenz aufrundet. Während bei $d \leqslant 8192$ noch SRHT fast doppelt so schnell eingebettet wie KN, braucht SRHT bei $d = 16384$ bereits länger als KN. Danach steigt die Laufzeit von SRHT wieder ähnlich an wie bei KN. Genauer kann man dies in dem rechten Graphen sehen: Hier wurde die Laufzeit jeweils durch d geteilt. Die sich ergebende normierten Zeiten sollte nach der asymptotischen Laufzeitanalyse jeweils konstant sein. Bei KN ist dies ziemlich genau der Fall, bei SRHT wird dagegen die Anomalie zwischen $d = 8192$ und $d = 16384$ noch deutlicher. Die Ursache könnte möglicherweise sein, dass die Daten, auf die der Algorithmus zugreift, zusammen noch klein genug, um in den Cache des Prozessors zu passen. Dadurch werden viel weniger Hauptspeicherzugriffe benötigt als bei $d = 16384$, wo viel häufiger Daten aus dem Cache verdrängt werden. Dagegen spricht jedoch, dass die

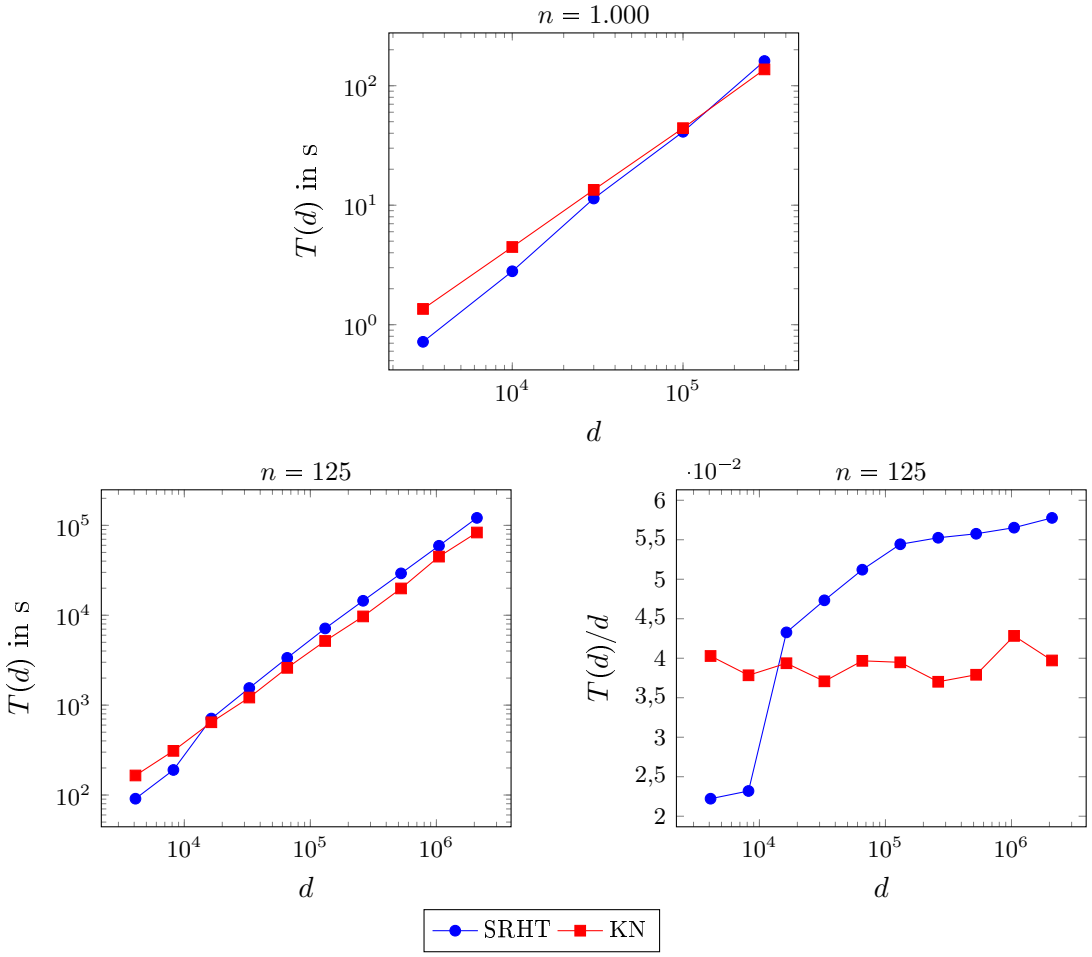

Abbildung 3.3: Abhängigkeit der Laufzeit von d. Der obere Graph zeigt die Daten aus Tabelle 3.5. Bei dem Graphen unten links wurde $n = 125$ gewählt, um größere Werte für d zu ermöglichen. Der Graph rechts daneben zeigt die normierte Laufzeit $T(d)/d$ für $n = 125$. Die Laufzeit ist jeweils in Sekunden angegeben.

Datenmatrix bei $d = 8192$ selbst etwa 8 MB benötigt und somit nicht mehr in den 3 MB großen L2-Cache der CPU passt.

Neben dem starken Anstieg zwischen $d = 8192$ und $d = 16384$, fällt bei SRHT im rechten Graphen auf, dass der Quotient mit zunehmendem d weiter ansteigt. Der Grund für diese Anomalie ist jedoch auch unklar. Letztendlich kann man festhalten, dass bei hinreichend großen d-Werten (d.h. $d \geqslant 10^6$) das SRHT-Verfahren eine um ca. 50% höhere Laufzeit als KN hat.

$\epsilon =$	0,4	0,2	0,1	0,05	0,025
SRHT	4,15	4,91	5,51	6,18	7,22
KN	1,42	3,79	8,99	20,69	45,58

Tabelle 3.6: Abhängigkeit der Laufzeit von dem Genauigkeitsparameter ϵ. Die gezeigten Laufzeiten sind Mittelwerte in Sekunden. Die anderen Parameter wurden wie folgt gewählt: $n = 1.000$, $d = 20.000$, $s = 1.000$, $\delta = 1/3$.

Abhängigkeit von ϵ Tabelle 3.6 zeigt das Laufzeitverhalten bei Variation des ϵ-Parameters. Hier ist das Ergebnis eindeutig: Bei einem großen ϵ ist KN schneller, bei einem kleinen ϵ dagegen SRHT. Dies liegt daran, dass die Laufzeit bei SRHT in $\mathcal{O}\left(\log\log\frac{1}{\epsilon^2}\right)$ liegt und bei KN in $\mathcal{O}\left(\log\frac{1}{\epsilon\cdot\log(1/\epsilon)}\right)$. Wenn also eine sehr exakte Einbettung erwünscht ist, sollte das SRHT-Verfahren angewendet werden, auch wenn die Eingabedaten nur dünn besetzt sind. Abbildung 3.4 zeigt sowohl die Daten aus der Tabelle (links oben), als auch einen größeren Datensatz (links unten) sowie die normierte Laufzeit (jeweils rechts). Letztere ist bei KN insgesamt sehr konstant (man beachte, dass die y-Achse nicht bei null beginnt). Bei SRHT fällt die Kurve mit sinkendem ϵ, allerdings nicht so signifikant, dass es hier weiter untersucht.

$\delta =$	0,5	0,25	0,1	0,01	0,001
SRHT	5,33	5,60	5,85	6,21	6,48
KN	5,51	11,90	21,61	56,18	99,41

Tabelle 3.7: Abhängigkeit der Laufzeit von der Versagenswahrscheinlichkeit δ. Die gezeigten Laufzeiten sind Mittelwerte in Sekunden. Die anderen Parameter wurden wie folgt gewählt: $n = 1.000$, $d = 20.000$, $s = 1.000$, $\epsilon = 0{,}1$.

Abhängigkeit von δ In Tabelle 3.7 wird die Laufzeit bei verschiedenen Versagenswahrscheinlichkeiten δ gezeigt. Das Ergebnis ist hier ähnlich wie bei der Variation von ϵ: Die Laufzeit von SRHT erhöht sich bei Verringerung von δ kaum, KN benötigt jedoch bei hohen δ-Werten erheblich länger. Dies liegt daran, dass die Laufzeit von KN quadratisch von $\log(1/\delta)$ abhängt, die von SRHT aber nur logarithmisch. Betrachtet man die normierten

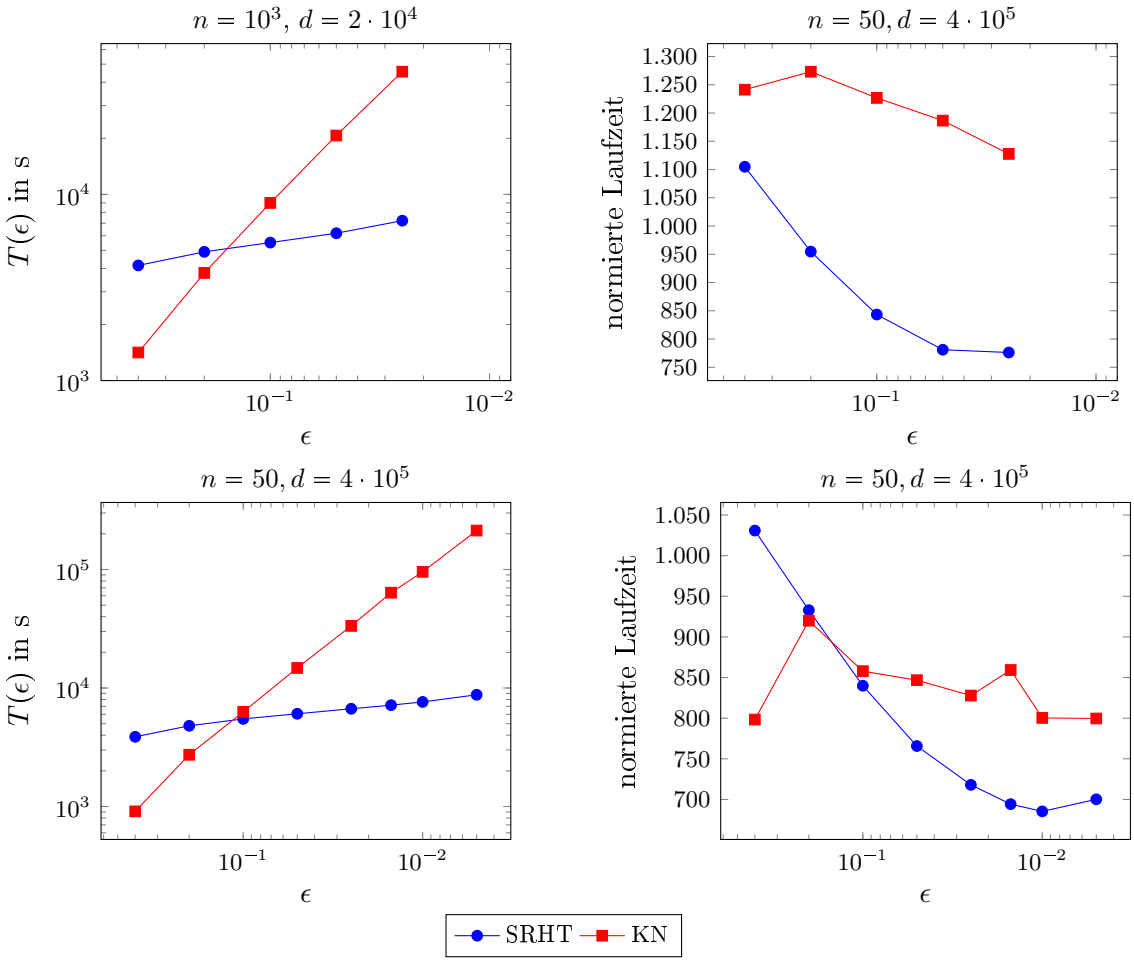

Abbildung 3.4: Abhängigkeit der Laufzeit von ϵ. Der Graph links oben zeigt die Daten aus der Tabelle 3.6. Bei dem Graph links unten wurde $n = 50$ und $d = 400.000$ gewählt, um kleinere Werte für ϵ zu ermöglichen. Die rechten Graphen zeigen jeweils die normierten Laufzeiten (für SRHT: $T(\epsilon)/\ln \frac{1{,}6 \cdot \ln n \cdot \ln(1/\delta)}{\epsilon^2}$, für KN: $5 \cdot T(\epsilon) \cdot \epsilon \cdot \ln\left(\frac{\ln n}{\epsilon}\right)$). Die Laufzeit ist jeweils in Sekunden angegeben.

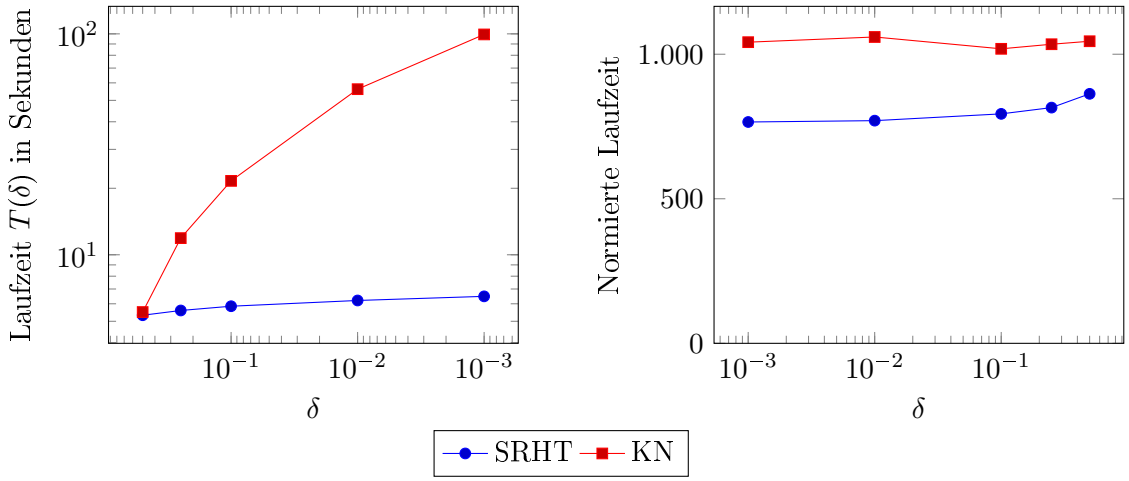

Abbildung 3.5: Der linke Graph zeigt die Laufzeit in Abhängigkeit von der Versagenswahrscheinlichkeit δ. Die Daten entsprechen denen aus Tabelle 3.7. Der rechte Graph zeigt normalisierte Laufzeiten, das heißt, die Laufzeit von SRHT wurde durch $\ln \frac{1{,}6 \cdot \ln(1/\delta) \cdot \ln n}{\epsilon^2}$ dividiert und die Laufzeit von KN durch $\ln(1/\delta) \cdot \ln(n/\delta)$. Somit zeigt der Graph, in wie weit die Laufzeit von dem theoretischen Laufzeitschranke abweicht.

Laufzeiten im rechten Graphen von Abbildung 3.5 zeigt sich (im Vergleich zu den vorherigen Graphen) ein erstaunlich gutes Ergebnis: Beide Kurven sind nahezu konstant, d.h. die asymptotische Laufzeit passt sehr gut zur gemessenen Laufzeit.

Auswertung Zusammenfassend kann man nun konkrete Formeln für beide Verfahren aufstellen. Dazu werden die sich ergebenden Konstanten für verschiedene Eingabedaten berechnet und anschließend gemittelt (siehe Tabelle 3.8). Verwendet man für die Berechnung der Einbettung einen Computer mit den gleichen Komponenten wie unter Abschnitt 3.1.6 beschrieben, sollten die Formeln die Laufzeit sehr gut vorhersagen können. Für das SRHT-Verfahren ergibt eine Berechnungsdauer von

$$T_{\text{SRHT}}(n, d, s, \epsilon, \delta) = 4{,}09 \cdot 10^{-8} \cdot n \cdot d \cdot \log_2 \frac{1{,}6 \cdot \ln n \cdot \ln(1/\delta)}{\epsilon^2}$$

Sekunden. Das Einbettungsverfahren KN hat eine Laufzeit von

$$T_{\text{KN}}(n, d, s, \epsilon, \delta) = 7{,}08 \cdot 10^{-9} \cdot n \cdot s \cdot \frac{2 \cdot ln(1/\delta)}{\epsilon} \cdot \log_2 \frac{2 \ln n}{\epsilon} \cdot \ln \frac{n}{\delta}$$

Neben dem Mittelwert für die Konstanten, der in die obigen Formeln eingesetzt wurde, ist noch die Standardabweichung interessant. Die Standardabweichung ist ein Maß dafür, wie stark die Konstante vom Mittelwert abweichen kann, und wird mit der folgenden Formel berechnet:

$$\sqrt{\frac{\sum_{i=1}^{n} (\bar{x} - x_i)^2}{n - 1}}$$

n	d	s	ϵ	δ	T_{SRHT}	T_{KN}	C_{SRHT}	C_{KN}
20.000	20.000	1.000	0,1	0,333	358,38	247,12	$8{,}32 \cdot 10^{-8}$	$6{,}70 \cdot 10^{-9}$
120.000	2.500	125	0,1	0,333	213,88	204,92	$6{,}48 \cdot 10^{-8}$	$6{,}18 \cdot 10^{-9}$
1.000	300.000	15.000	0,1	0,333	161,98	137,52	$5{,}27 \cdot 10^{-8}$	$7{,}33 \cdot 10^{-9}$
125	2.097.152	104.857	0,1	0,333	121,15	83,30	$4{,}75 \cdot 10^{-8}$	$7{,}40 \cdot 10^{-9}$
1.000	20.000	1.000	0,025	0,333	7,22	45,58	$2{,}53 \cdot 10^{-8}$	$7{,}11 \cdot 10^{-9}$
50	400.000	20.000	0,005	0,333	8,77	213,08	$2{,}43 \cdot 10^{-8}$	$9{,}12 \cdot 10^{-9}$
1.000	20.000	1.000	0,1	0,001	6,48	99,41	$2{,}51 \cdot 10^{-8}$	$7{,}33 \cdot 10^{-9}$
50	400.000	20.000	0,1	10^{-100}	8,16	33355,00	$2{,}38 \cdot 10^{-8}$	$4{,}92 \cdot 10^{-9}$
100	100.000	10.000	0,05	0,333	2,49	14,41	$2{,}14 \cdot 10^{-8}$	$7{,}64 \cdot 10^{-9}$

Tabelle 3.8: Verwendete Daten für die Berechnung der Konstanten der Laufzeitformeln. Die Spalten T_{SRHT} und T_{KN} zeigen jeweils die Laufzeit der Einbettung des jeweiligen Datensatzes in Sekunden. In den Spalten C_{SRHT} und C_{KN} stehen die sich ergebenden Konstanten für die Laufzeitformel.

Für SRHT ergibt sich $C_{\text{SRHT}} = (4{,}09 \pm 2{,}23) \cdot 10^{-8}$ und für KN $C_{\text{KN}} = (7{,}08 \pm 1{,}14 \cdot 10^{-9})$. Bei KN konnte die Laufzeit also recht genau bestimmt, bei SRHT ist die Abweichung jedoch recht groß. Dies zeigte sich bereits bei der Betrachtung der normierten Laufzeiten.

Überprüfung der Ergebnisse In einem abschließendem Test soll nun die Laufzeit für die Einbettung eines großes Datensatzes gemessen und mit den beiden obigen Formeln verglichen werden. Der Datensatz enthielt $n = 50.000$ Vektoren im $d = 10^6$ dimensionalen Raum mit je $s = 10.000$ Nicht-Null-Einträgen (d.h. 99% der Werte waren Nullen). Als Eingabeparameter für die Einbettung wurde eine Güte von $\epsilon = 0{,}05$ gewählt und eine Fehlerwahrscheinlichkeit von $\delta = 1/10$. Da der Datensatz nicht vollständig in den Speicher passt, wurden immer Pakete aus 50 Vektoren erstellt und eingebettet.

Verfahren	prognostizierte Laufzeit			gemessene Laufzeit
	Minimum	Erwartungswert	Maximum	
SRHT	$1{,}30 \cdot 10^4$ s	$2{,}85 \cdot 10^4$ s	$4{,}41 \cdot 10^4$ s	$1{,}35 \cdot 10^4$ s
KN	$3{,}14 \cdot 10^4$ s	$3{,}74 \cdot 10^4$ s	$4{,}35 \cdot 10^4$ s	$4{,}23 \cdot 10^4$ s

Tabelle 3.9: Vergleich zwischen der gemessenen und berechneten Laufzeit für einen großen Datensatz. Die Parameter wurden wie folgt gewählt: $n = 50.000$, $d = 1.000.000$, $s = 10.000$, $\epsilon = 0{,}05$, $\delta = 1/10$. Die Spalten „Minimum“ und „Maximum“ zeigen die berechnete Laufzeit für die um eine Standardabweichung kleinere bzw. größere Konstante in der Laufzeitformel.

Tabelle 3.9 zeigt die gemessenen und prognostizierten Laufzeiten. Bei KN ist die tatsächliche Laufzeit um lediglich 13% größer als die Erwartete. Die obige Formel kann die reale Laufzeit also recht genau vorhersagen. Bei SRHT ist das Ergebnis deutlich schlechter:

Die Einbettung des Datensatzes dauerte weniger als halb so lange wie vermutet. Allerdings liegt die Laufzeit noch in der 1σ-Umgebung, das heißt, die tatsächliche Laufzeit ist größer als die berechnete Laufzeit bei Verwendung der um eine Standardabweichung kleineren Konstanten (also $C = (4{,}09 - 2{,}23) \cdot 10^{-8} = 1{,}86 \cdot 10^{-8}$). Die Laufzeitformel für SRHT ist somit nicht falsch, sondern sie liefert insgesamt recht ungenaue Prognosen. Zusammenfassend kann man sagen, dass sich die Laufzeit von KN besser vorhersagen lässt als von SRHT.

3.4 Vergleich der Dimensionsreduktion

Fragestellung In diesem Abschnitt geht es darum, welche Epsilon-Schranke die einzelnen Einbettungsverfahren bei gegebenem Speicherbedarf bzw. bei gegebener Zieldimension einhalten.

Durchführung Dazu wurden verschiedene Datensätze untersucht, deren Dimension jeweils auf $k = 4.096$ reduziert wurde. Die Zweierpotenz wurde gewählt, da SRHT und CW sowieso die Zieldimension k auf eine Zweierpotenz aufrunden würden. Jeder Datensatz wurde mit jedem Verfahren 15 mal eingebettet. Die schlechtesten 5 Einbettungen wurden ignoriert ($\delta = 1/3$). Tabelle 3.10 zeigt das arithmetische Mittel der ϵ-Werte der restlichen 10 Einbettungen.

Datensatz	1	2	3	4	5	6	7
Typ	dünn	dünn	zufällig	worst	dünn	zufällig	worst
$n =$	200	200	200	200	500	2	2
$d =$	100.000	100.000	100.000	100.000	500.000	100.000	100.000
$s =$	1	2.000			2.000		
BCH-Sketch	0,0425	0,0444	0,0453	0,0635	0,0497	0,0036	0,1141
BCH-Null	0,0450	0,0439	0,0429	0,0439	0,0490	0,0052	0,0065
SRHT	0,0275	0,0432	0,0446	0,0441	0,0481	0,0058	0,0052
CW	0,7212	0,0425	0,0425	0,0436	0,0485	0,0057	0,0068
KN	0,0280	0,0318	0,0329	0,0314	0,0377	0,0039	0,0048

Tabelle 3.10: Vergleich der Qualität der Einbettungen bei gegebener Zieldimension von $k = 4.096$. Gezeigt sind die erreichten ϵ-Werte der einzelnen Einbettungen bei verschiedenen Datensätzen. Erklärung der *Typ*-Zeile: siehe Tabelle 3.1.

Auswertung Bei den meisten Datensätzen (2 bis 5 sowie 7) liefert KN die besten Ergebnisse. Bei Datensatz 1 ist SRHT geringfügig besser, bei Datensatz 6 liegt BCH-Sketch vorne. Wie schon in Tabelle 3.1 gesehen, versagt CW bei sehr dünn besetzten Daten (Datensatz 1). Weiß man jedoch, dass die Daten nicht diese Form haben, kann das schnelle

CW-Verfahren durchaus genutzt werden: Bei Datensatz 2 bis 7 werden ähnlich gute Ergebnisse wie bei den anderen Einbettungsverfahren erzielt. Trotzdem ist Vorsicht geboten, denn es könnte noch andere strukturierte Daten geben, bei denen CW versagt. Bei den Datensätzen des Typs „worst“ (Datensatz 4 und 7) schneidet BCH-Sketch im Vergleich zu den anderen Verfahren recht schlecht ab, vor allem bei Datensatz 7: Hier ist das erzielte ϵ_{out} um mehr als Faktor 16 größer als bei dem zweit schlechtesten Verfahren CW. Dies könnte auf eine unzureichende Unabhängigkeit hinweisen, bei BCH-Sketch wurde jedoch bewiesen, dass vierfache Unabhängigkeit ausreicht [4]. Da Datensatz 7 keine zufälligen Elemente enthält[5] sind Abhängigkeiten zwischen den Eingabedaten und den Zufallsvariablen der Einbettung ausgeschlossen. Warum BCH-Sketch bei jener Eingabe dennoch so schlecht abschneidet, bleibt damit unklar.

Insgesamt schneidet das KN-Verfahren am Besten ab. Vor allem bei den praxisrelevanten[6] Datensätzen (2 bis 5) erzeugt es die besten Einbettungen. Es folgen BCH-Sketch, BCH-Null, SRHT und CW mit jeweils etwa gleich guten Ergebnissen.

3.5 Untersuchung der benötigten Unabhängigkeit

Fragestellung In den vorherigen Tests wurde bei allen Verfahren (außer KN) eine Unabhängigkeit von 4 verwendet. Bei BCH-Sketch und BCH-Null konnte auch bewiesen werden, dass vierfache Unabhängigkeit ausreicht [4]. Bei SRHT wird jedoch von Ailon und Chazelle [2] vollständige Unabhängigkeit für die Einträge der Diagonalmatrix D gefordert. In wie weit dies gerechtfertigt ist, werden die folgenden Experimente zeigen. Bei KN wurden die in der Publikation [10] genannten Unabhängigkeiten benutzt, nämlich $deg_\sigma = \lceil 2\log(1/\delta)\rceil$ und $deg_h = \lceil \log(n/\delta)\rceil$. Es wurde jedoch immer mindestens eine vierfache Unabhängigkeit benutzt, auch wenn der berechnete Wert kleiner war.

Im folgenden soll nun untersucht werden, wie sich die Unabhängigkeit der Zufallsvariablen auf die Qualität der Ergebnisse und auf die Laufzeit auswirkt.

3.5.1 Unabhängigkeit bei SRHT

Durchführung In diesem Experiment wurde der Einfluss der Unabhängigkeit der Zufallsvariablen in der Diagonalmatrix D beim Einbettungsverfahren SRHT untersucht. Dazu wurden drei verschiedene Datensätze (ein vollständig zufälliger, ein dünn besetzter und ein Datensatz mit ausschließlich $+1$ und -1 Einträgen) eingebettet. Der Unabhängigkeitsgrad variierte beginnend bei 1 in Zweierpotenzen bis 32, wobei pro Konfiguration 15 Einbettungen mit $\epsilon_{\text{in}} = 0{,}1$ und $\delta = 1/3$ erstellt wurden. Die tatsächliche Güte ϵ_{out} der 10 besten

[5] Datensatz 7 enthält einen Nullvektor und einen Vektor mit ausschließlich Einsen als Elemente

[6] Datensatz 1 wird wegen seiner extrem dünnen Besetzung als nicht praxisrelevant bezeichnet. Datensatz 6 und 7 sind nicht praxisrelevant, da sie nur zwei Vektoren enthalten.

Einbettungen wurde gemittelt und bildete zusammen mit der gemessenen Laufzeit das Ergebnis des Tests.

Die Unabhängigkeit der Einträge in der Auswahlmatrix P konnte nicht näher untersucht werden, da hierbei nicht auf das BCH-Verfahren zurückgegriffen wurde, sondern auf einen Zufallsgenerator der C++-Standardbibliothek.

deg	dünn	zufällig	worst	Laufzeit (in s)
1	0,096	0,089	0,880	2,093
2	0,089	0,088	0,863	2,094
4	0,090	0,092	0,093	2,104
8	0,090	0,089	0,091	2,103
16	0,088	0,091	0,089	2,105
32	0,091	0,091	0,090	2,126

Tabelle 3.11: Laufzeit und Güte bei verschiedenen Unabhängigkeitsgraden für SRHT. Die Datensätze enthielten jeweils $n = 100$ Vektoren im $d = 100.000$ dimensionalen Raum. Die Vektoren des „dünnen“ Datensatzes hatten $s = 2.000$ Nicht-Null-Einträge. Die angestrebte Güte betrug $\epsilon_{in} = 0,1$ mit $\delta = 1/3$. Die Laufzeit war bei allen Datensätzen ungefähr gleich, von daher wird hier nur die Durchschnittslaufzeit der drei Datensätze gezeigt.

Auswertung In Tabelle 3.11 fällt zunächst auf, dass bei zufälligen Daten sogar eine Unabhängigkeit von $deg = 1$ ausreicht. Dies ist erstaunlich, denn $deg = 1$ bedeutet bei dem verwendeten Zufallsgenerator, dass alle erzeugten Zufallszahlen den gleichen Wert haben. Bei entsprechenden Daten (Datensatz „worst“) führt jedoch eine zu geringe Unabhängigkeit ($deg \leqslant 2$) zu falschen Ergebnissen (das berechnete $\epsilon_{out} = 0,88$ ist deutlich größer als das angestrebte $\epsilon_{in} = 0,1$). Es gibt jedoch keinen Grund, eine zu kleine Unabhängigkeit zu wählen, denn der Grad der Unabhängigkeit erhöht die Laufzeit nur marginal. Dies liegt daran, dass die Gesamtlaufzeit in $\mathcal{O}(n \cdot d \cdot \log k)$ liegt und lediglich d Zufallswerte generiert werden müssen.

Weiterführende Tests Mehr als 4-fache Unabhängigkeit bringt bei den getesteten Daten keine weitere Verbesserung der Ergebnisse. Um zu überprüfen, ob 4-fache Unabhängigkeit immer ausreicht, wurde in einem weiteren Experiment eine Hadamard-Matrix als Eingabematrix verwendet. Das SRHT-Verfahren verwendet bei der Einbettung selbst eine solche, von daher ist hier am ehesten davon auszugehen, dass 4-fache Unabhängigkeit eventuell nicht ausreichen könnte: Werden zwei $(q \times q)$-Hadamard-Matrizen miteinander multipliziert, ist das Ergebnis das q-fache der Identitätsmatrix. Dies ist quasi das Schlimmste, was bei SRHT passieren kann, denn die Multiplikation mit der Hadamard-Matrix soll eigentlich dafür sorgen, dass ein gegebenenfalls dünn besetzter Eingabevektor in einen dicht besetzten umgewandelt wird (und nicht andersrum).

Der Test zeigte jedoch, dass auch bei einer Hadamard-Matrix als Eingabe eine Unabhängigkeit von $deg = 4$ ausreicht. Bei geringeren Unabhängigkeiten schlug die Einbettung fehl, höhere Unabhängigkeiten führten zu keiner Verbesserung. Dass bei den Experimenten der vorherigen Kapitel 4-fache Unabhängigkeit benutzt wurde, war also genau die richtige Wahl.

3.5.2 Unabhängigkeit bei KN

Fragestellung Bei dem Einbettungsverfahren von Kane und Nelson gibt es zwei Hash-Funktionen, dessen Unabhängigkeiten variiert werden: Zum einem $\sigma : [1...d] \times [1...s] \rightarrow \{-1, +1\}$ und zum anderen $h : [1...d] \times [1...s] \rightarrow [1...k/s]$.

Zunächst soll untersucht werden, wie sich die Unabhängigkeiten der Hash-Funktionen auf die Qualität der Einbettung auswirkt. Anschließend wird überprüft, wie stark sich die Laufzeit mit zunehmender Unabhängigkeit erhöht.

Durchführung (Messung der Güte) Tabelle 3.12 zeigt die erreichten ϵ-Schranken bei drei verschiedenen Datensätzen. Als angestrebte Güte wurde stets $\epsilon_{in} = 0{,}1$ verwendet. Somit zeigen Tabellenwerte, die wesentlich größer als 0,1 sind, an, dass die Einbettung nicht erfolgreich war. Die Versagenswahrscheinlichkeit δ betrug bei dem ersten Datensatz 1/3, bei dem zweiten und dritten jeweils 1/100. Entsprechend wurden bei Datensatz 1 für jeden Wert in der Tabelle jeweils 15 Einbettungen berechnet, die schlechtesten 5 ignoriert und aus den restlichen 10 das arithmetische Mittel gebildet. Die Datensätze 2 und 3 wurde pro Tabellenwert 100 mal eingebettet, wobei die besten 99 besten ϵ-Werte gemittelt wurden.

Auswertung (Güte) Zunächst zur Unabhängigkeit der Hash-Funktion h: Betrachtet man den ersten Datensatz, zeigt sich deutlich, dass einfache und zweifache Unabhängigkeit nicht ausreichen: Selbst wenn man $deg_\sigma = 32$ wählt, erreicht die Einbettung nur eine Güte von $\epsilon_{out} = 0{,}542$ bzw. $\epsilon_{out} = 0{,}248$, was jeweils größer als das angestrebte ϵ_{in} von 0,1 ist. Eine Unabhängigkeit von $deg_h = 4$ bringt bereits die optimalen Ergebnisse. Höhere Unabhängigkeiten wie $deg_h = 8$ oder $deg_h = 16$ bringen zwar leicht bessere Ergebnisse, dies sind aber übliche Schwankungen. Dies kann man auch daran erkennen, dass die Qualität der Einbettung bei $deg_h = 32$ wieder geringfügig schlechter wird.

Bei der Unabhängigkeit der σ-Funktion sieht die Lage etwas weniger eindeutig aus. Klar ist, dass einfache Unabhängigkeit definitiv nicht ausreicht, wie man leicht am ersten Datensatz erkennen kann ($\epsilon_{out} = 54 \gg 0{,}1$). Bei zweifacher Unabhängigkeit und $deg_h = 4$ wird das angestrebte $\epsilon_{in} = 0{,}1$ mit $\epsilon_{out} = 0{,}114$ jedoch fast erreicht. Dass es nicht ganz erreicht wird, könnte zum einen an den bei Zufallsexperimenten typischen Schwankungen liegen, aber auch daran, dass $deg_\sigma = 2$ tatsächlich nicht ausreicht. Betrachtet man den zweiten Datensatz zeigt sich ein ähnliches Ergebnis: Hier erfüllt die Einbettung bei Verwendung von zweifacher Unabhängigkeit sogar die angestrebte Güte von $\epsilon_{in} = 0{,}1$ mit

worst, $n = 100$	$deg_h = 1$	$deg_h = 2$	$deg_h = 4$	$deg_h = 8$	$deg_h = 16$	$deg_h = 32$
$deg_\sigma = 1$	315,228	138,482	54,910	54,910	54,910	54,910
$deg_\sigma = 2$	1,266	0,597	0,114	0,102	0,104	0,113
$deg_\sigma = 4$	0,538	0,243	0,097	0,098	0,096	0,097
$deg_\sigma = 8$	0,543	0,236	0,099	0,097	0,095	0,100
$deg_\sigma = 16$	0,534	0,253	0,095	0,098	0,095	0,099
$deg_\sigma = 32$	0,542	0,248	0,099	0,095	0,096	0,102
zufällig, $n = 100$	$deg_h = 1$	$deg_h = 2$	$deg_h = 4$	$deg_h = 8$	$deg_h = 16$	$deg_h = 32$
$deg_\sigma = 1$	3,421	0,980	0,256	0,256	0,258	0,256
$deg_\sigma = 2$	0,966	0,360	0,067	0,062	0,061	0,064
$deg_\sigma = 4$	0,276	0,109	0,050	0,049	0,049	0,050
$deg_\sigma = 8$	0,280	0,111	0,049	0,049	0,049	0,050
$deg_\sigma = 16$	0,278	0,108	0,049	0,050	0,049	0,049
$deg_\sigma = 32$	0,274	0,105	0,048	0,049	0,049	0,049
worst, $n = 2$	$deg_h = 1$	$deg_h = 2$	$deg_h = 4$	$deg_h = 8$	$deg_h = 16$	$deg_h = 32$
$deg_\sigma = 1$	315,228	222,607	157,116	157,116	157,116	157,116
$deg_\sigma = 2$	0,774	0,656	0,126	0,125	0,156	0,116
$deg_\sigma = 4$	0,059	0,040	0,029	0,028	0,030	0,028
$deg_\sigma = 8$	0,055	0,042	0,028	0,031	0,028	0,029
$deg_\sigma = 16$	0,066	0,043	0,028	0,031	0,030	0,030
$deg_\sigma = 32$	0,062	0,039	0,029	0,026	0,033	0,033

Tabelle 3.12: Qualität der Einbettung bei verschiedenen Unabhängigkeitsgraden. Die Parameter wurden in den drei Tabellen wie folgt gewählt: $d = 100.000$; $\epsilon = 0{,}1$, $\delta = 1/3$ (erster Datensatz) bzw. $\delta = 1/100$ (zweiter und dritter Datensatz). In der Tabelle sind alle Werte, die größer als $1{,}5\epsilon$ sind (und damit auf eine nicht ausreichende Unabhängigkeit hinweisen), rot hinterlegt. Werte größer als ϵ sind orange hinterlegt.

$\epsilon_{out} = 0{,}067$ (bei $deg_h = 4$). Eine Erhöhung auf $deg_\sigma = 4$ verbessert die Ergebnisse ähnlich wie bei Datensatz 1 geringfügig, nämlich auf 0,050.

Ein eindeutiges Resultat zeigt sich bei Datensatz 3: Erhöht man die Unabhängigkeit von σ von 2 auf 4, verbessern sich die Ergebnisse von $\epsilon_{out} = 0{,}126$ auf $\epsilon_{out} = 0{,}029$ (bei Verwendung von $deg_h = 4$). Höhere Unabhängigkeiten als 4 bringen jedoch keine weitere Verbesserung: Bei dem ersten Datensatz liegt ϵ_{out} jeweils bei ungefähr 0,1, beim zweiten bei etwa 0,05. Beim letzten Datensatz schwankt ϵ_{out} leicht: So verbessert sich die Güte bei $deg_h = 16$ und $deg_\sigma = 8$ auf 0,028, bei $deg_\sigma = 32$ sinkt sie jedoch wieder auf 0,033 ab, das heißt, es handelt sich hierbei um gewöhnliche statistische Schwankungen. Ein Blick auf die Laufzeit soll nun die Frage beantworten, ob es sich „lohnt“ nur zwei- statt vierfache Unabhängigkeit bei σ zu verwenden:

Durchführung (Messung der Laufzeit) Tabelle 3.13 zeigt die Laufzeiten bei zwei verschiedenen Datensätzen mit je $n = 100$ Vektoren und $d = 100.000$ Dimensionen. Der erste war voll besetzt und wurde zeilenweise eingebettet, das heißt, dass die berechneten Hash-Werte n-fach genutzt werden konnten. Beim zweiten sind nur 2% der Werte ungleich null ($s = 2.000$), sodass jeder Eintrag der Eingabematrix einzeln eingebettet wurde. Somit mussten bei der Einbettung der ersten Matrix $m = d \cdot b = 2{,}2 \cdot 10^7$ Werte der Hash-Funktion berechnet werden und bei der zweiten $m = s \cdot n \cdot b = 4{,}4 \cdot 10^7$ (dabei bezeichnet b die Anzahl der Bereiche, auf die ein Wert abgebildet wird (siehe auch Abschnitt 2.6)). Insgesamt werden beim ersten Datensatz $n \cdot d = 10^7$ Werte eingebettet, bei zweiten Datensatz dagegen nur $n \cdot s = 2 \cdot 10^5$.

Auswertung (Laufzeit) Aus der Tabelle wird deutlich, dass der Einfluss der verwendeten Unabhängigkeiten auf die Laufzeit sehr unterschiedlich sein kann: Bei der zeilenweisen Einbettung (oberer Teil der Tabelle 3.13) erhöht sich die Laufzeit von der minimalen Unabhängigkeit ($deg_h = 1$ und $deg_\sigma = 1$) auf die maximale betrachtete Unabhängigkeit ($deg_h = 32$ und $deg_\sigma = 32$) um nur etwa 130%. Bei dem dünn besetzten Datensatz (unterer Teil der Tabelle 3.13) steigt die Laufzeit dagegen um mehr als das 40-fache. Das heißt, vor allem bei dünn besetzten Daten, die nicht zeilenweise eingebettet werden können, sollte eine möglichst geringe Unabhängigkeit verwendet werden. Durch Tabelle 3.12 wissen wir, dass $deg_h = 4$ die richtige Wahl ist. Bei deg_σ stellt sich die Frage, wie stark sich die Laufzeit bei Verdopplung der Unabhängigkeit von zweifach auf vierfach erhöht: Bei dem ersten Datensatz sind dies gerade einmal 1,1%, bei dem zweiten 9%. Ein Blick in Tabelle 3.12 zeigt, dass sich die Güte um mindestens 4% erhöht (erster Datensatz, $deg_h = 8$), im Durchschnitt um 12% (erster Datensatz, $deg_h \geqslant 4$), teilweise aber auch um 80% (dritter Datensatz, $deg_h = 16$). Somit verbessert sich die Qualität der Einbettung um mehr Prozent als die Laufzeit ansteigt. Hinzukommt die Tatsache, dass es durchaus Datensätze

Typ: „zufällig“	$deg_h = 1$	$deg_h = 2$	$deg_h = 4$	$deg_h = 8$	$deg_h = 16$	$deg_h = 32$
$deg_\sigma = 1$	16,03	23,54	26,56	28,13	30,68	35,08
$deg_\sigma = 2$	16,13	23,34	26,38	28,12	30,90	35,22
$deg_\sigma = 4$	16,30	23,84	26,67	28,11	31,23	35,76
$deg_\sigma = 8$	16,60	24,25	27,10	28,86	31,51	35,78
$deg_\sigma = 16$	17,09	24,99	27,60	29,08	31,99	36,40
$deg_\sigma = 32$	17,89	25,84	28,48	29,91	32,74	37,27
Typ: „dünn“	$deg_h = 1$	$deg_h = 2$	$deg_h = 4$	$deg_h = 8$	$deg_h = 16$	$deg_h = 32$
$deg_\sigma = 1$	0,52	1,88	3,36	6,07	10,75	18,60
$deg_\sigma = 2$	0,75	2,06	3,56	6,33	11,02	18,77
$deg_\sigma = 4$	1,04	2,38	3,88	6,63	11,35	19,16
$deg_\sigma = 8$	1,63	2,97	4,46	7,26	11,95	19,76
$deg_\sigma = 16$	2,57	3,87	5,45	8,19	12,95	20,72
$deg_\sigma = 32$	4,15	5,42	7,02	9,80	14,50	22,33

Tabelle 3.13: Laufzeit in Sekunden bei verschiedenen Unabhängigkeitsgraden. Die Parameter wurden in beiden Tabellen wie folgt gewählt: $n = 100$, $d = 100.000$; $\epsilon = 0.1$, $\delta = 1/100$. Der erste Datensatz („zufällig“) war voll besetzt und wurde zeilenweise eingebettet. Der „dünne“Datensatz enthielt $s = 2.000$ Nicht-Null-Einträge pro Vektor, die Einbettung geschah elementeweise.

geben könnte, bei denen $deg_\sigma = 2$ nicht ausreicht. Aus diesen Gründen sollte auch bei der Funktion σ vierfache Unabhängigkeit verwendet werden.

3.6 Genauere Untersuchung von CW

Da das Verfahren von Clarkson und Woodruff bei extrem dünn besetzten Daten versagt, wurde es bei den bisherigen Analysen meistens ignoriert. Im folgenden geht es um die nähere Untersuchung der Einbettungsmethode CW, das heißt, wann es anwendbar ist und wie das Laufzeitverhalten aussieht.

3.6.1 Abhängigkeit der Zieldimension von s und n

Fragestellung In Abschnitt 3.2.2 zeigte sich bereits, dass bei hinreichend dicht besetzten Eingabedaten das CW-Verfahren eine ähnlich gute Dimensionsreduktion wie die anderen in dieser Arbeit untersuchten Verfahren erreicht.

Durchführung Bei dem folgenden Versuch wurde deshalb das ϵ_{out} bei verschiedenen Kombinationen von s, n und k bestimmt. Da die Zieldimension k in meiner CW-Implementierung stets auf eine Zweierpotenz aufgerundet wird, wurde die Werte für k direkt als Zweierpotenz gewählt. Jeder Datensatz wurde insgesamt neun mal eingebettet. Die Einbettungen

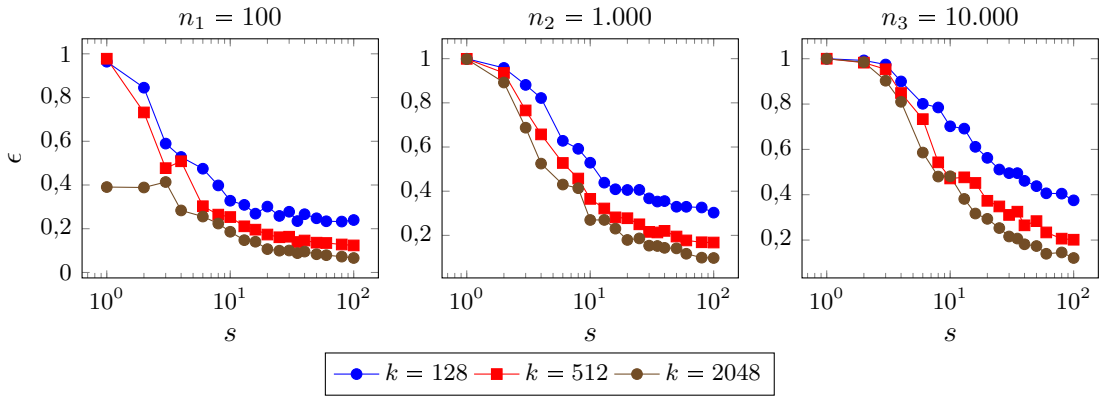

Abbildung 3.6: Die Graphen zeigen die Güte der CW-Einbettung bei Eingabedaten mit n Vektoren, die jeweils s Nicht-Null-Einträge hatten. Die Dimension der Vektoren war jeweils $d = 20.000$.

wurden mit den Originaldaten vergleichen und anschließend nach ihrer Güte sortiert. Die schlechtesten drei Einbettungen wurden ignoriert (d.h. $\delta = 1/3$). Der höchste ϵ-Wert der restlichen sechs Einbettungen bildete das Ergebnis der jeweiligen (s, n, k)-Kombination.

Auswertung Die Diagramme in Abbildung 3.6 zeigen die Ergebnisse dieses Tests. Alle Kurven fallen, das heißt, je dichter die Eingabedaten besetzt sind, desto genauer wird die Einbettung. Ob die benötigte Zieldimension k logarithmisch von n abhängt, kann überprüft werden, indem der Quotient zweier ϵ-Werte bei $n_1 = 100$ und $n_2 = 1000$ und jeweils konstanten k und s betrachtet wird: Bei logarithmischer Abhängigkeit müsste dieser gleich $\sqrt{\frac{\ln 1000}{\ln 100}} \approx 1{,}22$ sein, bei quadratischer Abhängigkeit dagegen $\sqrt{\frac{1000^2}{100^2}} = 10$. Der linke Graph von Abbildung 3.7 zeigt den Quotienten für alle $\epsilon < 0.5$, also für Einbettungen, die als „erfolgreich“ bezeichnet werden können. Einen Wert von 1,22 erreicht kein einziger Quotient, allerdings sind alle Quotienten deutlich kleiner als 100. Daraus folgt, dass k nicht in $\mathcal{O}(\log n)$ liegt, aber auch nicht in $\Omega(n^2)$. Insgesamt sinken die Quotienten mit zunehmendem s weiter ab, d.h. bei hinreichend großem s hängt k logarithmisch von n ab. Bei einer Anzahl von $s \geqslant s_{n_1 n_2} = 30$ Nicht-Null-Einträgen liegt der Quotient der meisten Datenpunkte ungefähr bei höchstens 1,5. Somit könnte es sein, dass k in $\mathcal{O}(\log^2 n)$ liegt.

Zur Überprüfung der These wurde eine Datenreihe für $n_3 = 10.000$ erstellt (siehe rechter Graph von Abbildung 3.6). Der Quotient der ϵ-Werte für $n_2 = 1.000$ und $n_3 = 10.000$ müsste demnach kleiner als $\frac{\ln 10.000}{\ln 1.000} \approx 1{,}33$ sein. Dies ist etwa bei $s \geqslant s_{n_2 n_3} = 40$ der Fall, also erst für Daten mit mehr Nicht-Null-Einträgen als beim Vergleich zwischen $n_1 = 100$ und $n_2 = 1.000$ (hier galt die These bereits bei $s_{n_1 n_2} = 30$). Betrachtet man den Quotienten $\frac{\ln n_i}{s_{n_{i-1} n_i}}$, so stellt man fest, dass dieser für beide betrachteten Testreihen gleich ist, nämlich $\frac{\ln 10.000}{40} = \frac{\ln 1.000}{30} \approx 0{,}23$.

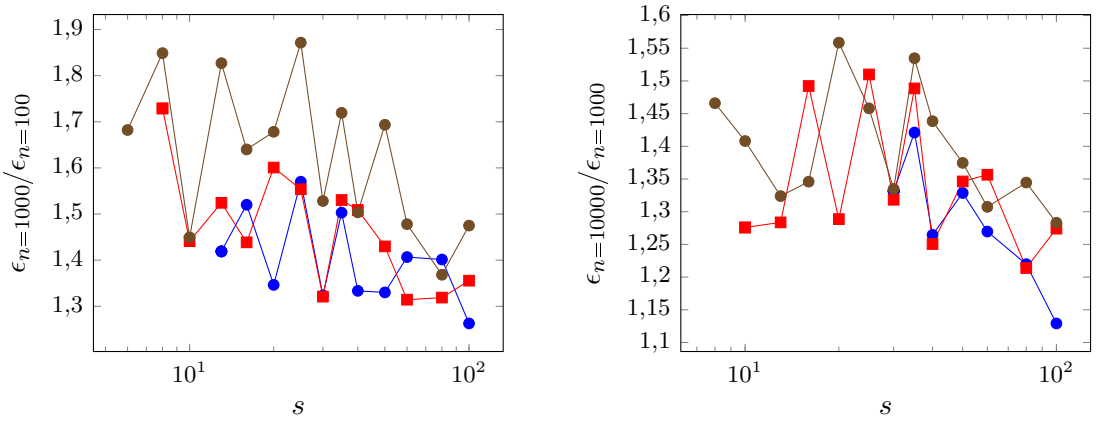

Abbildung 3.7: Der Graph zeigt die Quotienten der ϵ-Werte bei $n = 100$ und $n = 1.000$ bei verschieden Werten von s.

Zusammenfassend ergibt sich folgendes empirisches Ergebnis: Enthält jeder Eingabevektor mindestens $s \geqslant \frac{\ln n}{0{,}23}$ Nicht-Null-Einträge, so liegt die benötigte Zieldimension k in $\mathcal{O}(\log^2 n)$.

3.6.2 Konstanter Faktor der Zieldimension

Fragestellung In diesem Abschnitt soll nun auch für das CW-Verfahren der konstante Faktor für die Zieldimension bestimmt werden. Dabei wird zwischen voll und dünn besetzten Daten unterschieden: Im ersten Fall liegt k wie auch bei den anderen Verfahren in $\mathcal{O}\left(\frac{\log(1/\delta)}{\epsilon^2} \cdot \log n\right)$. Für den zweiten Fall gilt $k \in \mathcal{O}\left(\frac{\log(1/\delta)}{\epsilon^2} \cdot \log^2 n\right)$, wobei im Folgenden immer vorausgesetzt wird, dass $s \geqslant 10 \cdot \log_{10} n$ ist[7].

Durchführung Insgesamt wurden die untersuchten Datensätzen auf acht verschiedene Weisen aufgebaut, wobei die Größe der Datensätze, die Art und Weise der Eintragsgenerierung und die Spärlichkeit der Besetzung variiert wurden. Die $(d \times n)$-Matrizen hatten eine Größe von $(2 \cdot 10^4 \times 100)$ oder von (5.000×1.000). Die Anzahl der Nicht-Null-Einträge betrug bei der Hälfte der Konfigurationen $s = 10 \cdot \log_{10} n$ (dünn besetzt) und bei der anderen Hälfte $s = n$ (voll besetzt). Als Wertebereich wurden entweder ausschließlich -1 und $+1$ Einträge verwendet oder die Elemente wurden durch eine Gleichverteilung erstellt. Insgesamt ergeben sich so acht mögliche Kombinationen für den Aufbau der Datensätze. Von jeder dieser Konfigurationen wurden jeweils drei Datensätze generiert. Jeder Datensatz wurde 15 mal mit $\epsilon_{\text{int}} = 0{,}1$ und $\delta = 1/3$ eingebettet. Von den 10 besten Einbettungen wurde aus der gemessenen Güte ϵ_{out} die Konstante für die Zieldimension bestimmt. Zum Schluss wurde das Maximum für jede der acht Konfigurationen berechnet.

[7] Dies ist äquivalent zu der obigen Formel, denn es gilt: $\frac{\ln n}{\ln 1.000/30} = 30 \cdot \frac{\ln n}{\ln 1.000} = 30 \cdot \frac{\log_{10} n}{\log_{10} 1.000} = 30 \cdot \frac{\log_{10} n}{3} = 10 \cdot \log_{10} n$

Auswertung Tabelle 3.14 zeigt die berechneten Konstanten für die erwähnten Datensätze: Für voll besetzte Daten ergibt sich somit ein Faktor von etwa 1,6, für dünn besetzte Vektoren ein Faktor von 1,2. Damit reduziert CW bei voll besetzten Eingabedaten die Dimension des Datensatz bei gegebenem ϵ und δ genauso stark wie SRHT, BCH-Sketch und BCH-Null.

Datensatz	1	2	3	4	Max
Typ	zufällig	worst	zufällig	worst	
$n =$	100	100	1000	1000	
$d =$	20000	20000	5000	5000	
voll besetzt	1,608	1,636	1,036	1,061	1,636
dünn besetzt	1,181	0,550	1,076	0,759	1,181

Tabelle 3.14: Koeffizienten für die Zieldimension für voll und dünn besetzte Daten. Die Vektoren der dünn besetzten Daten enthielten bei $n = 100$ $s = 20$ und bei $n = 1.000$ $s = 30$ Nicht-Null-Einträge. Für einen einzelnen Tabelleneintrag wurden drei Datensätze mit den entsprechenden Parametern erstellt. Jeder Datensatz wurde 15 mal eingebettet, wobei die schlechtesten 5 Einbettungen ignoriert wurden ($\delta = 1/3$). Die letzte Spalte zeigt den größten Koeffizienten der jeweiligen Zeile.

3.6.3 Laufzeit von CW

Vorüberlegungen Bei dem CW-Verfahren wird jedes Element des Eingabevektors auf genau ein Element der eingebetteten Matrix abgebildet. Da die verwendeten Unabhängigkeiten der Zufallsfunktionen konstant sind, ergibt sich für die Einbettung eines Eintrags eine konstante Laufzeit. Für die Initialisierung muss einmalig eine $(n \times k)$-Nullmatrix erstellt werden. Insgesamt wird somit eine Laufzeit von $\mathcal{O}(n \cdot (s + k))$ benötigt. In diesem Abschnitt geht es nun darum, die Konstante, die hinter der asymptotischen Laufzeit steckt, zu bestimmen.

	Laufzeit	Laufzeit pro Element
Initialisierung	$1,60 \pm 0,10$ s	$(2{,}66 \pm 0{,}17) \cdot 10^{-8}$ ms
Einbettung	$9,00 \pm 1,56$ s	$(7{,}50 \pm 1{,}30) \cdot 10^{-8}$ ms

Tabelle 3.15: Laufzeit für die Initialisierung und Einbettung bei dem CW-Verfahren. Die eingebetteten Datensätze hatten $n = 12.000$ Vektoren im $d = 200.000$ dimensionalen Raum, wobei jeder Vektor aus $s = 10.000$ Nicht-Null-Einträgen bestand. Die Dimension der Daten wurde auf $k = 5.000$ reduziert. Insgesamt wurden 50 Datensätze eingebettet und die Laufzeiten gemittelt sowie die Standardabweichung berechnet. Die letzte Spalte zeigt die Laufzeit pro Element, das heißt bei der „Initialisierung“ wurde die Laufzeit durch $n \cdot k$ geteilt und bei der „Einbettung“ durch $n \cdot s$.

Durchführung und Auswertung Die Laufzeitanteile für die Initialisierung und Einbettung werden dabei getrennt analysiert. Insgesamt wurden 50 gleich große Datensätze eingebettet und die Laufzeiten gemittelt. Das Ergebnis ist in Tabelle 3.15 zu sehen. Für CW ergibt sich somit eine Laufzeit von

$$T_{\mathrm{CW}} = 7{,}50 \cdot 10^{-8} \cdot n \cdot s + 2{,}66 \cdot 10^{-8} \cdot n \cdot k$$

3.7 Verkettung verschiedener Einbettungsverfahren

Nach den bisherigen Untersuchungen stellt sich nun die Frage: Kann eine Verkettung verschiedener Einbettungsverfahren in bestimmten Anwendungsszenarien sinnvoll sein? Dabei ist bei einer gegebenen anstrebten Güte sowohl eine hohe Dimensionsreduktion von Interesse, als auch eine geringe Laufzeit. Letztere kann mit den in Abschnitt 3.3.2 und 3.6.3 ermittelten Laufzeitformeln approximiert werden.

Zunächst wird untersucht, wie sich die Güte ϵ und die Fehlerwahrscheinlichkeit δ der Gesamteinbettung berechnen lässt.

3.7.1 Theorem. *Sei D ein beliebiger Datensatz und π_1 sowie π_2 zwei Einbettungsverfahren mit der Güte ϵ_1 bzw. ϵ_2 und der Versagenswahrscheinlichkeit δ_1 bzw. δ_2. Wird D zunächst von π_1 und das Ergebnis anschließend von π_2 einbettet, dann gilt für die Gesamteinbettung:*

a) $\epsilon = \epsilon_1 + \epsilon_2 + \epsilon_1\epsilon_2$

b) $\delta = \delta_1 + \delta_2 - \delta_1\delta_2$

Beweis.

a) Bei der Gesamteinbettung $\pi_2 \circ \pi_1$ gilt für die Approximation des Abstandes zwischen den Vektoren $p, q \in N$:

$$(1-\epsilon_1)(1-\epsilon_2)\|p-q\|_2 \leqslant \|\pi_2(\pi_1(p)) - \pi_2(\pi_1((q))\|_2 \leqslant (1+\epsilon_1)(1+\epsilon_2)\|p-q\|_2$$

Der Faktor für die Abschätzung nach oben ist damit gleich $(1+\epsilon_1)(1+\epsilon_2) = 1+\epsilon_1+\epsilon_2+\epsilon_1\epsilon_2$. Für die Abschätzung nach unten gilt: $(1-\epsilon_1)(1-\epsilon_2) = 1-\epsilon_1-\epsilon_2+\epsilon_1\epsilon_2 > 1-\epsilon_1-\epsilon_2-\epsilon_1\epsilon_2$. Somit gilt für die Gesamtgüte: $\epsilon = \epsilon_1 + \epsilon_2 + \epsilon_1\epsilon_2$. □

b) Die Wahrscheinlichkeit, dass beide Einbettungen erfolgreich waren, liegt bei $(1-\delta_1)(1-\delta_2)$. Somit ist die Versagenswahrscheinlichkeit für die Gesamteinbettung gleich $\delta = 1-(1-\delta_1)(1-\delta_2) = \delta_1+\delta_2-\delta_1\delta_2$. □

Aus Theorem 3.7.1 lässt sich ableiten, dass eine minimale Zieldimension bei gegebenen $\epsilon_{\mathrm{gegeben}}$ nur mit einer einfachen Einbettung erreicht werden kann. Denn bei Verwendung

von zwei Einbettungen bestimmt ϵ_2 die spätere Zieldimension. Da $\epsilon_2 < \epsilon_{\text{gegeben}}$ ist und k umgekehrt quadratisch von ϵ abhängt, ergibt sich für die zweite Einbettung eine größere Zieldimension als bei der einfachen Einbettung mit $\epsilon = \epsilon_{\text{gegeben}}$.

Im Folgenden werden nun mehrere Datensätze betrachtet und die Laufzeiten und Zieldimension für verschiedene einfache und verkettete Einbettungen berechnet.

Datensatz $\mathcal{A}$	$n = 20.000$, $d = 10^9$, $s = 10^7$			
	einfache Einbettung $\epsilon = 0{,}1$, $\delta = 0{,}333$		Verkettung $\epsilon_1 = 0{,}0488$, $\delta_1 = 0{,}183$	
	T in s	k	T in s	k
SRHT	8.806.218	1.741	11.013.571	11.300
KN	2.611.741	1.088	9.907.586	7.062
CW	15.007	12.930	15.045	83.931
			$d_{\text{neu}} = s_{\text{neu}} = k_{\text{CW}}$ $\epsilon_2 = 0{,}0488$, $\delta_2 = 0{,}183$	
			T in s	k
SRHT			924	11.300
KN			83.156	7.062
Optima	KN	CW	CW+SRHT	CW+KN
T in h	725	4:10	4:26	27:17
k	1.088	12.930	11.300	7.062

Tabelle 3.16: Laufzeit und Zieldimension für einfache und verkettete Einbettung. Die zweite und dritte Spalte zeigen die einfache Einbettung mit SRHT, KN und CW. In der vierten und fünften Spalte wurde der Datensatz verkettet eingebettet, wobei die zweite Einbettung auf dem Ergebnis von CW arbeitet.

In Tabelle 3.16 wurde ein hypothetischer Datensatz auf verschiedene Weisen eingebettet: Die zweite und dritte Spalte zeigen die Laufzeit und Zieldimension für die einfache Einbettung mit SRHT, KN bzw. CW. Für die verkettete Einbettung (vierte und fünfte Spalte) wurde die Güte und Fehlerwahrscheinlichkeit so gewählt, dass $\epsilon_1 = \epsilon_2$ und $\delta_1 = \delta_2$ ist und dass sich eine Gesamtgüte und -fehlerwahrscheinlichkeit wie bei der einfachen Einbettung ergibt. Da CW bei der ersten Einbettung die geringste Laufzeit hat, wurde die zweite Einbettung auf der CW-Einbettung berechnet. Die letzten drei Zeilen der Tabelle fassen die relevanten Ergebnisse des Tests zusammen: Wie bereits klar war, wird die geringste Zieldimension mit der einfachen Anwendung des KN-Verfahrens erzielt. Allerdings ist eine Rechenzeit von ungefähr einem Monat (725 Stunden) nicht tolerierbar. CW ist hier mit gerade einmal vier Stunden erheblich schneller und reduziert die Dimension auf $k = 12.930$.

Das interessante ist nun, dass die Dimension weiter reduziert werden kann, wenn nach der Anwendung von CW mit SRHT eingebettet wird: Die Dimension sinkt dabei jedoch nur 12% auf $k = 11.300$. Angesichts der Tatsache, dass sich die Laufzeit jedoch gerade einmal 16 Minuten erhöht, könnte sich die Verkettung durchaus lohnen, vor allem wenn die Laufzeit des nachfolgenden Algorithmus exponentiell von der Dimension der Daten abhängt. Man muss bei der Verkettung von CW mit SRHT aber erwähnen, dass die zwischenzeitliche CW-Einbettung mit $k \cdot n = 1,68 \cdot 10^9$ Einträgen etwa 13 GB Arbeitsspeicher benötigt[8]. Verwendet man als zweites Einbettungsverfahren KN statt SRHT, so ergibt sich eine nochmals kleinere Zieldimension von $k = 7.062$. Die Laufzeit erhöht sich jedoch auf ungefähr einen Tag.

Letztendlich haben alle vier möglichen Methoden ihre Berechtigung: Eine kleinere Zieldimension führt bei Datensatz $\mathcal{A}$ zu einer längeren Rechenzeit und umgekehrt. Es ergibt sich folgendes Ranking mit steigenden Laufzeiten bzw. sinkenden Zieldimensionen: CW, CW+SRHT, CW+KN, KN. Welche Methode nun die Beste ist, hängt vom jeweiligen Anwendungsfall ab: Hängt die Laufzeit des nachfolgenden Algorithmus linear von der Dimension ab, ist wahrscheinlich CW oder CW+SRHT die richtige Wahl. Wenn die Laufzeit dagegen exponentiell mit der Dimension steigt und generell sehr groß ist, ist voraussichtlich CW+KN oder vielleicht sogar KN die bessere Wahl.

Datensatz $\mathcal{B}$	$n = 10^7$, $d = 10^9$, $s = 1.000$			
	einfache Einbettung $\epsilon = 0{,}1$, $\delta = 0{,}333$		Verkettung $\epsilon_1 = 0{,}0488$, $\delta_1 = 0{,}183$	
	T in s	k	T in s	k
SRHT	4.690.501.908	2.833	5.794.178.863	18.391
KN	223.170	1.771	822.426	11.494
CW	9.860	34.249	59.887	222.318
			$d_{\text{neu}} = k_{\text{CW}}$, $s_{\text{neu}} = s$ $\epsilon_2 = 0{,}0488$, $\delta_2 = 0{,}183$	
			T in s	k
SRHT			1.288.153	18.391
KN			820.797	11.494
Optima	KN	CW	CW+SRHT	CW+KN
T in h	62	2:44	374	245
k	1.771	34.249	18.391	11.494

Tabelle 3.17: Laufzeit und Zieldimension für einen sehr dünn besetzten Datensatz. Der Aufbau der Tabelle ist analog zu Tabelle 3.16.

[8] Dies gilt bei der Verwendung von Fließkommazahlen mit 8 Byte (Typ *double* bei C++)

Der zweite untersuchte Datensatz enthielt deutlich mehr Vektoren mit sehr viel dünnerer Besetzung. Ein Blick auf die letzten Zeilen von Tabelle 3.17 macht deutlich, dass hier eine Verkettung keinen Sinn ergibt: Sowohl die Laufzeit als auch die Zieldimension sind bei KN geringer als bei Verkettung zweier Einbettungsverfahren.

Die dünne Besetzung des Datensatzes $\mathcal{B}$ verhindert, dass eine Verkettung der Einbettungsverfahren nützlich sein kann: Nach Anwenden des CW-Verfahrens liegt die Anzahl der Nicht-Null-Einträge pro Vektor immer noch bei annähernd $s_{\text{neu}} \approx 1.000$.[9] Somit ergeben sich für das KN-Verfahren durch die verkettete Einbettung keine Vorteile. Auch die Methode CW+SRHT ist uninteressant, da die Dimension der Vektoren nach der CW-Einbettung mit $d_{\text{neu}} = 222.318$ noch deutlich größer ist als die Anzahl der Nicht-Null-Einträge mit $s_{\text{neu}} \leqslant 1.000$. Da SRHT die dünne Besetzung nicht ausnutzt, ergibt sich eine recht hohe Laufzeit.

Im Gegenschluss bedeutet dies, dass sich bei einer Matrix mit noch höherer Dimensionszahl als bei Datensatz $\mathcal{A}$ und entsprechend mehr Nicht-Null-Einträgen die Verkettung besonders lohnen müsste. Zur besseren Vergleichbarkeit sollte der Speicherverbrauch der Eingabedaten gleich bleiben, das heißt $n \cdot s$ sollte konstant bleiben.

Datensatz $\mathcal{C}$	$n = 200$, $d = 10^{11}$, $s = 10^9$			
	einfache Einbettung $\epsilon = 0{,}1$, $\delta = 0{,}333$		Verkettung $\epsilon_1 = 0{,}0488$, $\delta_1 = 0{,}183$	
	T in s	k	T in s	k
SRHT	8.068.063	931	10.273.076	6.033
KN	1.338.953	582	5.340.781	3.771
CW	15.000	3.701	15.000	23.975
			$d_{\text{neu}} = s_{\text{neu}} = k_{\text{CW}}$ $\epsilon_2 = 0{,}0488$, $\delta_2 = 0{,}183$	
			T in s	k
SRHT			2	6.033
KN			128	3.771
Optima	KN	CW	CW+SRHT	CW+KN
T in h	371	4:10	4:10	4:12
k	582	3.701	6.033	3.771

Tabelle 3.18: Laufzeit und Zieldimension für einen hochdimensionalen Datensatz mit wenig Vektoren. Der Aufbau der Tabelle ist analog zu Tabelle 3.16.

Das Ergebnis für diesen Test zeigt Tabelle 3.18: Erstaunlicherweise lohnt sich eine Verkettung wiederum nicht, diesmal aber aus einem anderen Grund: Das einfache Anwenden

[9] Treten bei der CW-Einbettung keine Kollisionen auf, gilt $s_{\text{neu}} = 1.000$. Ansonsten ist $s_{\text{neu}} < 1.000$.

von CW liefert eine kleinere Zieldimension als die verketteten Methoden. Die Ursache dafür ist die geringe Anzahl an Vektoren: Grundsätzlich lohnte sich die Verkettung von CW mit anderen Verfahren, weil die Zieldimension bei CW quadratisch von $\ln n$ abhängt. Ist n jedoch wie Datensatz $\mathcal{C}$ klein, ist auch der zusätzliche Faktor $\ln n$ nicht besonders groß. Hinzu kommt, dass die zweite Einbettung eine generell größere Zieldimension liefert, als es bei einer einfachen Einbettung der Fall wäre, da bei der betrachteten Verkettung $\epsilon_2 < \frac{1}{2} \cdot \epsilon$ ist und $\delta_2 < \delta_1$ ist. Zusammen mit den konstanten Faktoren für die Berechnung der Zieldimension ergibt sich dann eine höhere Zieldimension bei Verkettung als wenn eine einfache Einbettung mit CW vorgenommen wird.

Datensatz $\mathcal{A}$ war somit für die verkettete Einbettung optimal geeignet: Weder die Anzahl der Vektoren noch die Anzahl der Nicht-Null-Einträge war zu klein. Hinsichtlich des Speicherverbrauchs war Datensatz $\mathcal{A}$ aber bereits grenzwertig: Die Zwischeneinbettung benötigte 13 GB Arbeitsspeicher; für den Linux-Rechner, auf dem ich meine Experimente durchführte, wäre dies bereits zu viel gewesen. Ist n wesentlich größer (z.B. um Faktor 10), muss über Alternativen nachgedacht werden, da der Speicherverbrauch der Zwischeneinbettung linear mit n steigt.

Liegen die Originaldaten spaltenweise gespeichert vor, könnte man jeden Vektor einzeln einbetten und anschließend auf einer Festplatte zwischenspeichern. Die zweite Einbettung verarbeitet dann die auf der Festplatte gespeicherte Zwischeneinbettung. Die Abhängigkeit von n fällt somit weg und damit auch das Speicherproblem.

Bisher wurden nur die Parameter n, d und s des Datensatzes variiert, jedoch nicht die Einbettungsparameter ϵ und δ. In Tabelle 3.19 wurde der Datensatz $\mathcal{A}$ mit $\epsilon = 0,01$ und $\delta = 1/10$ eingebettet. Die Laufzeit von KN erhöht sich dadurch auf mehrere Monate, sodass das Verfahren in diesem Fall unbrauchbar wird. Bei der Verkettung von CW mit KN ergibt sich das gleiche Problem. Somit sind nur noch die Varianten CW und CW+SRHT von Interesse. Letztere hat eine um etwa 30% geringere Zieldimension als CW, aber auch eine deutlich höhere Laufzeit von ungefähr 11 Stunden gegenüber 4 Stunden bei CW. Im Vergleich zu Tabelle 3.16 kann man also festhalten, dass kleinere ϵ- und δ-Werte zwar die Laufzeit der CW+SRHT-Variante erhöhen, aber auch die relative Reduktion der Zieldimension gegenüber der einfachen CW-Einbettung verbessern.

Bei den sehr kleinen ϵ- und δ-Werten, wie sie hier in Tabelle 3.19 verwendet wurden, ergibt sich jedoch ein anderes Problem, das bereits in schwächerer Form in Tabelle 3.16 auftrat: Die Zwischeneinbettung hat eine Dimension von fast 1,6 Millionen und benötigt somit 235 GB Speicher. Es gibt durchaus Rechner, die entsprechend viel Hauptspeicher haben[10], aber in der Regel dürfte es notwendig sein, die erste Einbettung spaltenweise vorzunehmen. Im Gegensatz zu Tabelle 3.16 verbraucht aber auch die zweite Einbettung mit circa 32 GB noch sehr viel Speicher, das heißt die gewählten Einbettungsparameter würde man in der Praxis nur dann wählen, wenn der benutzte Rechner ebenso viel Speicher

[10] Der *SPARC T4-1* von Oracle unterstützt beispielsweise bis zu 512 GB Arbeitsspeicher

Datensatz $\mathcal{A}$	$n = 20.000$, $d = 10^9$, $s = 10^7$			
	einfache Einbettung $\epsilon = 0{,}03$, $\delta = 0{,}1$		Verkettung $\epsilon_1 = 0{,}0149$, $\delta_1 = 0{,}0{,}0513$	
	T in s	k	T in s	k
SRHT	12.521.167	40.540	14.474.946	212.269
KN	24.851.711	25.337	75.461.057	132.668
CW	15.160	301.114	15.839	1.576.651
			$d_{\text{neu}} = s_{\text{neu}} = k_{\text{CW}}$ $\epsilon_2 = 0{,}0149$, $\delta_2 = 0{,}0{,}0513$	
			T in s	k
SRHT			22.822	212.269
KN			11.897.577	132.668
Optima	KN	CW	CW+SRHT	CW+KN
T in h	6903	4:13	10:44	3309
k	25.337	301.114	212.269	132.668

Tabelle 3.19: Laufzeit und Zieldimension für die Einbettung von Datensatz $\mathcal{A}$ bei Verwendung besserer Gütewerte und einer geringen Versagenswahrscheinlichkeit. Der Aufbau der Tabelle ist analog zu Tabelle 3.16.

zur Verfügung hat. Denn schließlich ist das primäre Ziel von Einbettungen die Datengröße soweit zu verringern, dass sie von nachfolgenden Algorithmen verarbeitet werden können ohne dafür auf ein langsamen Massenspeicher wie eine Festplatte zugreifen zu müssen.

Möchte man das gegebene ϵ nicht erhöhen, aber trotzdem die Dimension der zweiten Einbettung verringern, könnte man in dem gegebenen Szenario ϵ_2 auf 0,025 erhöhen. Im Gegenzug wird ϵ_1 auf 0,00488 verkleinert, sodass die Gesamteinbettung eine Güte von 0,03 beibehält. Als Folge verringert sich die Zieldimension der CW+SRHT-Variante von 212.269 auf 75.292 und damit der Speicherverbrauch auf etwa 11 GB — ein sehr positives Ergebnis. Kostenlos ist dieser Gewinn jedoch nicht: Durch das sehr kleine ϵ_1 vergrößert sich die Dimension der Zwischeneinbettung auf 14,7 Millionen und damit der Speicherverbrauch auf 2,1 TB. Als Folge steigt auch die Rechenzeit der nachfolgenden SRHT-Einbettung: Insgesamt benötigt die verkettete Einbettung nun 60 Stunden.

Dieses Gedankenexperiment zeigt, dass es keinesfalls immer sinnvoll ist ϵ_1 und ϵ_2 gleich zu wählen. Durch Erhöhung von ϵ_2 und entsprechender Verringerung von ϵ_1 kann bei gleich bleibender Gesamtgüte die Zieldimension der verketteten Einbettung auf Kosten der Rechenzeit stark gesenkt werden. Dies ist auch auf die Versagenswahrscheinlichkeit δ_1 und δ_2 übertragbar. Allerdings ist hier der Effekt geringer, da die Zieldimension logarithmisch statt quadratisch von dem Parameter abhängt.

Zusammenfassend lässt sich qualitativ feststellen, dass sich die Verkettung von CW mit SRHT oder KN bei sehr großen Datensätzen durchaus lohnt. „Groß“ bezieht sich dabei sowohl auf die Vektorenanzahl n als auch auf die Anzahl der Nicht-Null-Einträge s. Ist n besonders groß und/oder ϵ bzw. δ besonders klein, muss berücksichtigt werden, dass die Zwischeneinbettung viel Speicher benötigt. Die erste Einbettung muss somit ggf. spaltenweise vorgenommen werden, sodass die eingebetteten Vektoren in einem Massenspeicher zwischengespeichert werden können.

Kapitel 4

Fazit

In dieser Arbeit wurden lineare l_2-Einbettungen von dünn besetzten Eingabedaten hinsichtlich ihrer Laufzeit, ihrer Dimensionreduktion und der benötigten Unabhängigkeit experimentell untersucht. Dabei ergaben sich die folgenden Ergebnisse:

Zunächst (Abschnitt 3.2.1) wurden die konstanten Faktoren der Zieldimension bestimmt. Dabei fiel der geringe Koeffizient von 1,0 beim Verfahren von Kane und Nelson auf — alle anderen Einbettungsmethoden kamen auf einen Koeffizienten von 1,6. Bei der Einbettung von Clarkson und Woodruff konnte bewiesen werden, dass es Datensätze gibt, für die die Zieldimension k in $\Omega(n^2)$ liegt.

Bei der Untersuchung der Laufzeit (Abschnitt 3.3) stellten sich die auf zufälligen Rademacher-Matrizen basierenden Verfahren BCH-Sketch und BCH-Null als uninteressant heraus: In jedem Fall war die Einbettung von Kane und Nelson schneller. In den darauf folgenden Experimenten konnten für SRHT und KN mehr oder weniger exakte Formeln zur Vorhersage der Laufzeit bestimmt werden. Qualitativ kann man sagen, dass bei hinreichend dünn besetzten Eingabedaten das KN-Verfahren in kürzerer Zeit einbettet als die schnelle Johnson-Lindenstrauss-Transformation (SRHT).

Die Untersuchung der Dimensionsreduktion (Abschnitt 3.4) brachte keine wirklich neuen Erkenntnisse: Letztendlich konnte die Aussage aus Abschnitt 3.2.1 bestätigt werden, dass KN die Dimension am stärksten reduziert.

Bei der Überprüfung der Unabhängigkeit (Abschnitt 3.5) stellte sich heraus, dass bei allen Zufallsvariablen der Einbettungsverfahren vierfache Unabhängigkeit ausreicht. Höhere Unabhängigkeit bringt keine Vorteile, geringere Unabhängigkeit führt bei entsprechenden Eingabedaten zu unbrauchbaren Einbettungen.

Sehr interessante Ergebnisse lieferte die genauere Untersuchung der Einbettung von Clarkson und Woodruff (Abschnitt 3.6): Wenn man weiß, dass die Eingabevektoren hinreichend dicht besetzt sind, kann man das CW-Verfahren anwenden, ohne eine quadratisch von n abhängende Zieldimension zu wählen. Die ist ein nützliches Resultat, denn CW ist nochmals deutlicher schneller als die Einbettungsverfahren SRHT und KN.

Aufschlussreich war auch der letzte Abschnitt (3.7) des experimentellen Teils: Hier ging es um die Frage, ob eine Verkettung verschiedener Einbettungsverfahren sinnvoll sein kann. Diese Frage konnte mit „ja“ beantwortet werden: Bei sehr großen Datensätzen lohnt es sich zunächst das CW-Verfahren anzuwenden und anschließend das Ergebnis mit SRHT oder KN einzubetten. Die Zieldimension ist zwar stets größer als bei einfacher Einbettung mit KN, jedoch ist auch die Laufzeit bei verketteter Einbettung um ein Vielfaches geringer.

Es gibt zahlreiche Möglichkeiten im Bereich der linearen Einbettungen weiter zu forschen. Beim Einbettungsverfahren von Clarkson und Woodruff wäre es wünschenswert die empirischen Ergebnisse theoretisch zu untermauern. Interessant wäre auch ein praxisorientierter Laufzeitvergleich bei einem konkreten Anwendungsfall. Bei der Clusteranalyse könnte man untersuchen, ab welche Datengröße sich das Einbetten trotz des entstehenden Overheads lohnt. Die Forschung im Bereich der linearen Einbettungen ist also bei Weitem noch nicht abgeschlossen.

Anhang A

Notation

A.0.1 Symbole

$[m]$	Menge der natürlichen Zahlen von 1 bis m
$\|x\|_2$	Euklidische Länge des Vektors x
$\mathbb{R}$	Menge der reellen Zahlen
$\mathbb{E}[X]$	Erwartungswert der Zufallsvariablen X

A.0.2 Variablen

n	Anzahl der Vektoren im Datensatz
d	Dimension der Vektoren
s	Anzahl der Nicht-Null-Einträge pro Vektor
k	Zieldimension (Dimension der eingebetteten Vektoren)
ϵ	Genauigkeit der Einbettung: $(1 \pm \epsilon)$ Approximation der paarweisen Abstände
ϵ_{in}	Angestrebte Genauigkeit
ϵ_{out}	Tatsächliche Genauigkeit
δ	Versagenswahrscheinlichkeit
N	Menge aller Vektoren im Datensatz
p, q, x	Ein Vektor im Datensatz
π	Einbettungsfunktion ($\pi : \mathbb{R}^d \to \mathbb{R}^k$)

A.0.3 Abkürzugnen für Einbettungsverfahren

BCH-Sketch	Voll besetzte Rademachermatrizen	(siehe Seite 6)
BCH-Null	Zu einem Drittel besetzte Rademachermatrizen	(siehe Seite 6)
SRHT	Schnelle Johnson-Lindenstrauss Transformation	(siehe Seite 7)
CW	Dünn besetzte Einbettungsmatrix von Clarkson und Woodruff	(siehe Seite 9)
KN	Einbettungsverfahren von Kane und Nelson	(siehe Seite 9)

A.0.4 Spezielle Variablen bei bestimmten Einbettungsverfahren

R	Rademachermatrix	BCH-Sketch, BCH-Null
P	Auswahl-Matrix	SRHT
H	Hadamard-Matrix	SRHT
D	Diagonal-Matrix mit zufälligen Einträgen	SRHT
b	Anzahl der Bereiche	KN
h	Hashfunktion ($h : [d] \times [b] \to [k/b]$) für die Abbildung auf einen Index innerhalb eines Bereichs	KN
σ	Hashfunktion ($\sigma : [d] \times [b] \to \{-1, +1\}$), die festlegt, ob addiert oder subtrahiert wird	KN

Literaturverzeichnis

[1] Achlioptas, Dimitris: *Database-friendly random projections: Johnson-Lindenstrauss with binary coins.* J. Comput. Syst. Sci., 66(4):671–687, 2003.

[2] Ailon, Nir und Bernard Chazelle: *Approximate nearest neighbors and the fast Johnson-Lindenstrauss transform.* In: *STOC*, Seiten 557–563, 2006.

[3] Alon, Noga, László Babai und Alon Itai: *A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem.* J. Algorithms, 7(4):567–583, 1986.

[4] Alon, Noga, Yossi Matias und Mario Szegedy: *The Space Complexity of Approximating the Frequency Moments.* J. Comput. Syst. Sci., 58(1):137–147, 1999.

[5] Clarkson, Kenneth L. und David P. Woodruff: *Low Rank Approximation and Regression in Input Sparsity Time.* CoRR, abs/1207.6365, 2012.

[6] Coppersmith, Don und Shmuel Winograd: *Matrix Multiplication via Arithmetic Progressions.* J. Symb. Comput., 9(3):251–280, 1990.

[7] Fino, Bernard J. und V. Ralph Algazi: *Unified Matrix Treatment of the Fast Walsh-Hadamard Transform.* IEEE Trans. Computers, 25(11):1142–1146, 1976.

[8] Ingo Wegener: *Datenstrukturen, Algorithmen und Programmierung 2.* Seiten: 139-141, 2002. URL: http://ls2-www.cs.uni-dortmund.de/lehre/sommer2002/dap2/skript.pdf.

[9] Johnson, W. B. und J. Lindenstrauss: *Extensions of Lipschitz mapping into Hilbert space.* In: *Conf. in modern analysis and probability*, Band 26 der Reihe *Contemporary Mathematics*, Seiten 189–206. American Mathematical Society, 1984.

[10] Kane, Daniel M. und Jelani Nelson: *Sparser Johnson-Lindenstrauss transforms.* In: *SODA*, Seiten 1195–1206, 2012.

[11] Nelson, Jelani und Huy L. Nguyen: *OSNAP: Faster numerical linear algebra algorithms via sparser subspace embeddings.* CoRR, abs/1211.1002, 2012.

[12] Nelson, Jelani und Huy L. Nguyen: *Sparsity Lower Bounds for Dimensionality Reducing Maps.* CoRR, abs/1211.0995, 2012.

[13] Rusu, Florin und Alin Dobra: *Pseudo-random number generation for sketch-based estimations.* ACM Trans. Database Syst., 32(2):11, 2007.

[14] Tropp, Joel A.: *Improved Analysis of the subsampled Randomized Hadamard Transform.* Advances in Adaptive Data Analysis, 3(1-2):115–126, 2011.

[15] Venkatasubramanian, Suresh und Qiushi Wang: *The Johnson-Lindenstrauss Transform: An Empirical Study.* In: *ALENEX*, Seiten 164–173, 2011.

[16] Williams, Virginia Vassilevska: *Multiplying matrices faster than coppersmith-winograd.* In: *STOC*, Seiten 887–898, 2012.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 14. Juli 2013

Jens Quedenfeld