universität koblenz
Informatik

Nächste-Nachbarn-Suche in hochdimensionalen und dünnbesetzten Vektorräumen

Masterarbeit

zur Erlangung des Grades Master of Science (M.Sc.)
im Studiengang Computervisualistik

vorgelegt von

Benjamin Zuravlev

Erstgutachter:	Prof. Dr.-Ing. Peer Neubert (Institut für Computervisualistik)
Zweitgutachter:	Kevin Weirauch, M.Sc. (Institut für Computervisualistik)

Koblenz, im September 2024

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Visuelle SLAM-Algorithmen nutzen die *Visual Place Recognition*, um fehlerhafte Daten, die aus Bewegungsinformationen gewonnen werden, zu korrigieren. In diesem Kontext nimmt die Suche des nächsten Nachbarn eine wichtige Rolle ein. Die Ausführung dieser Suche erfordert, dass die Bilder als Vektoren vorliegen. Eine Form dieser Vektoren stellen die sogenannten SEER-Deskriptoren dar, deren Berechnung sowie einige daraus resultierende Eigenschaften im Rahmen dieser Arbeit erörtert werden. Die hohe Dimensionalität und Dünnbesetztheit sind zwei wesentliche Eigenschaften dieser Deskriptoren, welche den Rahmen für die vorliegende Arbeit bilden. Um eine gute Methode zu identifizieren, die für einen solchen Vektor den ihm ähnlichsten Vektor aus einem Datensatz liefert, werden einige Metriken und Algorithmen für dieses Problem aus verschiedenen Publikationen vorgestellt. Die Implementierung der genannten Verfahren in der Programmiersprache Python sowie eine Bewertung ihrer Performance sind ebenfalls Gegenstand dieser Arbeit. In einigen Fällen werden Modifikationen vorgenommen, um der spezifischen Beschaffenheit der in dieser Arbeit vorliegenden Vektoren Rechnung zu tragen. Des Weiteren wird eine neue Ähnlichkeitsfunktion vorgestellt, deren Implementierung und Evaluierung ebenfalls Gegenstand dieser Arbeit ist. Die durchgeführten Untersuchungen legen nahe, dass der Cosinusabstand das geeignetste Ähnlichkeitsmaß darstellt. Darüber hinaus wird aufgezeigt, dass die neu entwickelte Ähnlichkeitsfunktion hinsichtlich der Qualität der Ergebnisse sowie der Laufzeit eine geeignete Alternative darstellt. Im Weiteren wird dargelegt, dass die Übertragbarkeit der Ergebnisse im Kontext der SEER-Deskriptoren auf den allgemeinen hochdimensionalen dünnbesetzten Fall nur eingeschränkt möglich ist.

Abstract

Visual SLAM algorithms use visual place recognition to correct erroneous data obtained from movement information. In this context, the nearest neighbor search plays an important role. The execution of this search requires the images to be available as vectors. One form of these vectors are the so-called SEER descriptors, whose calculation and some resulting properties are discussed in this paper. The high dimensionality and sparseness are two essential properties of these descriptors, which form the framework for the present work. In order to identify a good method that provides the most similar vector from a data set for such a vector, some metrics and algorithms for this problem from various publications are presented. The implementation of these methods in the Python programming language and an evaluation of their performance are also the subject of this work. In some cases, modifications are made to take into account the specific nature of the vectors presented in this thesis. Furthermore, a new similarity function is presented, the implementation and evaluation of which is also the subject of this thesis. The investigations carried out suggest that the cosine distance is the most suitable similarity measure. Moreover, it is shown that the newly developed similarity function is a suitable alternative in terms of the quality of the results and the runtime. Furthermore, it is shown that the transferability of the results in the context of the SEER descriptors to the general high-dimensional sparse case is only possible to a limited extent.

Inhaltsverzeichnis

1 Einleitung

Die Suche nach nächsten Nachbarn im Allgemeinen ist ein Optimierungsproblem, welches in vielen verschiedenen Praxisfällen Anwendung findet, wie zum Beispiel beim Suchen von doppelt vorhandenen Dateien und Webseiten [And09] oder bei der Klassifikation bei *Machine-Learning*-Algorithmen. Formal kann das Problem folgendermaßen definiert werden [HAK00]: Für einen Punkt $q \in \mathbb{R}^d$ soll der Nächste Nachbar x_{NN} aus dem Datensatz $D \subset \mathbb{R}^d$ gefunden werden:

$$x_{NN} = \{x' \in D \mid \forall\, x \in D, x \neq x' : dist(x', q) \leq dist(x, q)\} \tag{1}$$

$dist(.,.)$ ist dabei die gewählte Distanzfunktion.

Den Kontext dieser Arbeit bildet das Problem der *Visual Place Recognition*, die z. B. in SLAM-Algorithmen eingesetzt wird. Ziel ist es, Schleifenschlüsse visuell zu erkennen, damit fehlerbehaftete Informationen von den Bewegungssensoren durch die Kameradaten korrigiert werden können. Hierbei wird anhand des Vergleichs von Kamerabildern festgestellt, ob das mobile System (meistens ein Roboter) vorher bereits an dem Ort war, an dem es sich an einem gegebenen Zeitpunkt befindet. Da in den seltensten Fällen ein Pixel-für-Pixel-Vergleich sinnvoll ist, werden Bilder üblicherweise mithilfe von Merkmalsvektoren codiert, die sich wiederum für mathematische Operationen eignen.
Diese Merkmalsvektoren (im Folgenden auch Deskriptoren genannt) können mithilfe verschiedener Methoden berechnet werden, was in dieser Arbeit jedoch, bis auf die SEER-Deskriptoren (in Kapitel 3), nicht thematisiert wird.
Eine Möglichkeit, Bilder mathematisch zu repräsentieren, sind holistische Deskriptoren. Diese beschreiben das Bild in einem einzigen Vektor, im Gegensatz zu lokalen Deskriptoren, bei denen für jede Landmarke ein eigener Vektor genutzt wird. Dabei ist es naheliegend, dass zum Einen sehr viele einzelne Merkmale festgehalten werden müssen, zum Anderen wiederum viele Merkmale eines Bildes sich signifikant von denen eines anderen Bildes unterscheiden. Daraus entsteht ein wesentliches Merkmal der in dieser Arbeit untersuchten Vektoren, nämlich die hohe Dimensionalität. Die Dünnbesetztheit ist im Kontrast dazu keine Eigenschaft, die alle holistischen Deskriptoren ausmacht, sondern ein wesentliches Merkmal der SEER-Deskriptoren.

Die vorliegende Arbeit verfolgt das Ziel, einen Algorithmus zu identifizieren, der unter den gegebenen Rahmenbedingungen eine hinreichende Performance aufweist. Dies bezieht sich erstens darauf, dass der vom Algorithmus bestimmte nächste Nachbar die codierte Darstellung des Ortes darstellt, der auch vom Anfragebild codiert dargestellt wird. Zweitens soll der Algorithmus eine geringe Laufzeit benötigen, da dieses Problem im Kontext der VPR in Echtzeit gelöst werden muss. Gerade bezüglich der ersten Anforderung ist es wichtig zu beachten, dass kein Algorithmus unter allen Bedingungen perfekte Ergebnisse liefern kann, somit wird der Anteil der guten Ergebnisse betrachtet.

2 Verwandte Arbeiten / Stand der Forschung

Die Nächste-Nachbarn-Suche ist schon länger ein Forschungsfeld auf dem verschiedene Optimierungsansätze entwickelt werden, so zum Beispiel der *Orchards Algorithm* [Orc91], der die Dreiecksungleichungen [Cla06] nutzt, um Informationen über Entfernungen zu Punkten zu gewinnen, ohne sie explizit zu berechnen. Dieser Algorithmus wurde noch durch die Annulus Methode optimiert [Cla06]. Der *Approximating and Eliminating Search Algorithm* (kurz AESA) [Rui86] [Vid94] geht von den gleichen mathematischen Grundlagen aus, nutzt sie aber effizienter.

Taunk et al. [TDVS19] führen aus, dass kNN (*k Nearest Neighbors* häufig für Klassifikation gewählt wird. Dieser Algorithmus hat einige Schwächen, die durch verschiedene Modifikationen, wie unter anderem *locally adaptive kNN* [WD93], *weight adjusted kNN* [HKK01] oder *adaptive kNN* [HKK01] , beseitigt werden können.

Georgescu et al.[GSM03] zeigen, dass bei dem weit verbreiteten k-Means Algorithmus die Berechnungskomplexität für die nächste-Nachbarn-Suche durch *Locality Sensitive Hashing* reduziert werden kann während die Qualität der Klassifikation erhalten bleibt.

Hinneburg et al. [HAK00] gehen das Problem der geringeren Aussagekraft von klassischen Distanzmetriken im hochdimensionalen Fall dadurch an, dass die optimale Projektion der Daten in einen niedrigerdimensionalen Raum gefunden werden soll, um dann dort nächste Nachbarn zu finden.
Auch Vijayukamar et al. [VDS06] verfolgen den Ansatz der Dimensionalitätsreduktion, wenden ihn aber nur lokal an. Somit identifizieren sie verschiedene lokale Statistiken, die für die Klassifikation wichtig sind und vermeiden dadurch die Suche im gesamten Merkmalsraum. Dabei werden mehrere lokale lineare Modelle des Datensatzes erstellt.

Auch im Feld der Datenstrukturen gibt es verschiedene Ansätze, um die Laufzeit und / oder den Suchraum der Algorithmen zu reduzieren. Ein Beispiel ist das *Locality Sensitive Hashing* von Indyk und Motwani [IM98], das von Frome und Malik erfolgreich auf Bilddeskriptoren getestet wurde [FM06][GIM+99]. Ein anderer Ansatz mit dem gleichen Ziel sind die *Balltree*-basierten Datenstrukturen von Liu, Moore und Gray [LMGC06], die Baumstrukturen mit Knoten und Blättern nutzen.

Tao et al. [TYSK09] kombinieren beide Herangehensweisen und stellen den *Locality Sensitive B-Tree* als eine Datenstruktur vor, die sowohl eine schnelle Suche als auch gute Ergebnisse kombiniert und durch die Kombination mehrerer solcher LSB-*Trees* als LSB-*Forest* weiter optimiert werden kann.

Gong et al. [GSW+22] nutzen einen dünn besetzten kNN-*classifier* um besser verschiedenartige Muster zu klassifizieren, wobei die Korrelationen verschiedener Muster vom euklidischen Raum auf einen dünn besetzten Raum abgebildet werden. Mithilfe dieser Klassifikationsmethode wurden in Experimenten gute Klassifikationsergebnisse auf hochdimensionalen Datensätzen erzielt Auch Ma et al. [MGW+17] nutzen ebenfalls die Vorteile von dünn besetzten Koeffizienten um die Nachbarschaft von Datenpunkten präzise wiedergeben zu können. Damit erreichen sie gute Ergebnisse bei der nächste-Nachbarn-Klassifikation mit den Klassifikatoren *oefficient-weighted kNN Classifier* und *residual-weighted kNN Classifier*.

Die genannten Arbeiten adressieren oftmals zwei der drei wesentlichen Punkte dieser Arbeit in verschiedenen Kombinationen (nächste-Nachbarn-Suche, hohe Dimensionalität, Dünnbesetztheit), jedoch wird in keiner der Arbeiten der hier untersuchte Anwendungsfall hinreichend thematisiert. Zudem ist der Kontext der meisten Arbeiten auf dem Gebiet der nächste-Nachbarn-Suche oft eine Zuordnung eine Datenpunktes zu einer von mehreren definierten Klassen oder Clustern, was den Kern der hier untersuchten Fragestellung nicht trifft. Daher bieten die gewonnen Erkenntnisse einen Mehrwert für die Forschung auf diesem Gebiet.

3 Anwendungsfall: SEER-Deskriptoren

2022 wurden *Sparse Exemplar Ensemble Representations* (SEER) von Neubert und Schubert vorgestellt [NS22]. Diese Deskriptoren eignen sich dazu, ohne näheres Wissen über einen Datensatz, Deskriptoren zu berechnen, die die Spezifischen Merkmale desselben abbilden.

3.1 Algorithmus

Der Algorithmus bekommt als Eingabe zum einen L2-normalisierte Deskriptoren $x \in \mathbb{R}^{d_x}$ und zum Anderen die Matrix M, die als Spalten die dünn besetzten Exemplare mit je d_M Einträgen $\neq 0$ beinhaltet. Diese Matrix kann entweder schon als fertige Matrix übergeben, oder neu aufgebaut werden, falls sie für den Datensatz noch nicht berechnet wurde.

d_M ist einer der Parameter, die benötigt werden, um die Matrix neu aufzubauen und bezeichnet die Anzahl der Eintäge $\neq 0$ in jedem Exemplar. Außerdem werden die Parameter λ und k benötigt, wobei k die Anzahl der benötigten ähnlichen Exemplare in der Matrix M pro Eingabedeskriptor festlegt und λ als Reaktivierungsfaktor die Anzahl der Nicht-Null-Einträge ($\lambda \cdot k$) in den Ausgabedeskriptoren bestimmt. Diese Werte sind standardmäßig mit $k = 50$ und $\lambda = 2$ belegt.
Die Ausgabe des Algorithmus beinhaltet immer die dünn besetzten Deskriptoren y. Falls die Matrix M nicht als (bereits berechneter) Parameter übergeben wurde, gehört sie ebenfalls zu den Ausgabewerten.

Die Berechnung der Matrix und der Ausgabedeskriptoren läuft folgendermaßen ab: Für den Eingabedeskriptor x_i wird das Skalarprodukt mit der Matrix M gebildet, und anschließend die Anzahl der Werte im Ergebnisvektor S ermittelt, die größer sind, als der Quotient d_M/d_X. Dabei bezeichnet d_X die Dimensionalität der Eingabevektoren. Ist diese Anzahl mindestens k, wird der Ausgabedeskriptor y_i aus S bestimmt. Dies geschieht dadurch, dass alle Einträge, bis auf die $\lambda \cdot k$ höchsten, auf 0 gesetzt werden.
Falls jedoch nicht genügend Einträge groß genug sind, werden zusätzliche Exemplare für M aus dem Eingabedeskriptor gesampelt. Dabei entspricht die Wahrscheinlichkeit dafür, dass eine Dimension gesampelt wird, dem Betrag ihres Wertes im Eingabevektor. Wenn genügend Exemplare hinzugefügt wurden, wird zum einen S um die neuen Einträ-

ge erweitert um dann y_i zu bestimmen, und außerdem werden an die vorher berechneten Deskriptoren Nullen angehängt, damit die Dimensionalität aller Ausgabedeskriptoren gleich ist.
Eine schmatische Übersicht über die Erstellung von SEER-Deskriptoren ist in Abbildung 1 zu sehen.

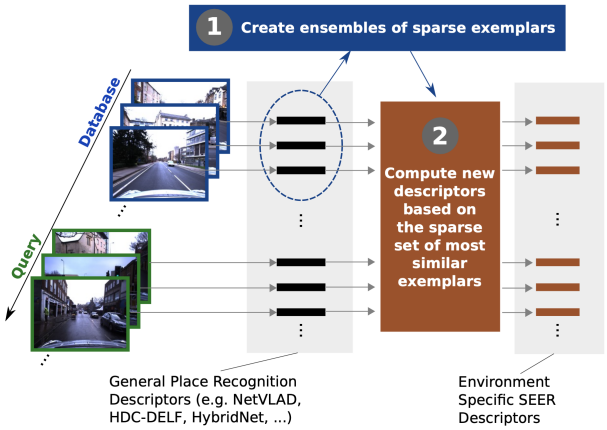


Abbildung 1: Schematische Darstellung der Erstellung der SEER-Deskriptoren. Entnommen aus [NS22]

Bei den SEER-Deskriptoren kommen zwei Schritte für die nächste-Nachbarn-Suche in Frage: Zum Einen die Prüfung auf ähnliche Exemplare in der Matrix M, zum Anderen der Vergleich unter den kodierten Deskriptoren, um einen Schleifenschluss für SLAM zu detektieren. Die erste Ähnlichkeitsprüfung wird mithilfe des Skalarproduktes durchgeführt. Diese Prüfung wird in dieser Arbeit nicht weiter untersucht. Gegenstand dieser Arbeit ist das Finden des ähnlichsten Deskriptors zu einem gegebenen Deskriptor, im Idealfall unter besonderer Berücksichtigung der Dünnbesetztheit.

3.2 Implementierung

Einer der Datensätze, die in dieser Arbeit Gegenstand der Untersuchung sind, wurde mithilfe des Python-Codes[1] in Codeabschnitt 1 generiert. (Die anderen beiden Datensätze wurden in MATLAB erzeugt [NS22]).

```
import numpy as np
from scipy.sparse import lil_matrix
import scipy.sparse as sp
from sklearn.preprocessing import normalize

def compute_M_and_Y(self, M, X, d_M=200, k=50, lambda_val=2):
    #Allocate memory for output descriptor
    DS = lil_matrix((0, 0), dtype=np.float32)
```

[1]Das zu dieser Arbeit gehörige Repository ist unter `https://github.com/bzuravlev/nnsearch_sparse_hd_data` zu erreichen

```
9       #L2 normalize each input descriptor
10      X = normalize(X, axis=1, norm='l2')
11      mat_empty = True
12      expectedSim = d_M / X.shape[1]
13      print("expectedSim: ", expectedSim)
14      for i in range(X.shape[0]):
15          if i % 10 == 0:
16              n_ex = 0
17              if M is not None:
18                  n_ex = M.shape[1]
19              print(f'  running image {i+1} of {X.shape[0]} ({n_ex}
    exemplars) ')
20          c = 0
21          if M is not None:
22              S = X[i] @ M
23              mat_empty = False
24              c = np.count_nonzero(S > expectedSim)
25          else:
26              S = np.array([])
27              mat_empty = True
28          #sample k-c new exemplars
29          if k > c:
30              PI,PV = self.create_exemplars(X[i, :], d_M, k-c)
31              if mat_empty:
32                  M = np.empty((X.shape[1],0))
33
34              #append empty columns to M
35              new_cols = np.zeros((X.shape[1],k-c), dtype=np.float32)
36              M = np.hstack([M, new_cols])
37              for p_idx in range(k-c, 0, -1):
38                  for p_idy in range(PI.shape[0]):
39                      #fill new columns with values
40                      #use deepcopy
41                      M[ PI[p_idy, -p_idx], -p_idx] = np.copy(PV[p_idy, -
    p_idx])
42
43              mat_empty = False
44              #append to S
45              for index in range(k-c, 0, -1):
46                  #compute and append missing values
47                  new_val = X[i] @ M[:,-index]
48                  S = np.append(S, new_val)
49          #sparsify
50          S= self.keep_n_largest(S, lambda_val*k)
51          DS= self.pad_and_append_SEER(DS,S)
52      return M, DS
```

Codeausschnitt 1: Code SEER-Deskriptoren

Im Folgenden werden nur die wichtigsten Schritte des Codeausschnitts aufgegriffen und erläutert.
In Zeile 22 wird das Skalarprodukt aus der Matrix `M` (außer im ersten Schleifendurchlauf, da die Matrix `M` dann noch leer ist) und dem aktuellen Deskriptor gebildet. In dem

Ergebnis dieser Multplikation werden dann in Zeile 24 die Werte gezählt, die den Wert `expectedSim` übersteigen. Diese Anzahl entspricht der Anzahl der Exemplare in der Matrix `M`, die den Vektor repräsentieren.

Falls es nicht genügend Exemplare gibt, die den Vektor repräsentieren (Prüfung `k > c`) wird in Zeile 30 eine entsprechende Anzahl von Exemplaren erzeugt. Im Anschluss an die Erweiterung der Matrix auf die neuen Dimensionen, wie in Zeile 36 dargestellt, erfolgt eine Füllung mit den generierten Werten an den entsprechenden Stellen in Zeile 41.
Nachdem die Matrix genügend Exemplare aufweist, wird das vorher berechnete Skalarprodukt um die Werte erweitert, die aus dem Skalarprodukt des Eingabevektors mit den neu erzeugten Spalten der Matrix entstehen. (Zeilen 47 und 48). Der letzte Schritt in jedem Schleifendurchlauf besteht darin, dass von dem endgültigen Skalarprodukt `S` alle Werte, bis auf die `lambda_val*k` größten, auf 0 gesetzt werden (siehe Zeile 50). Dieser Schritt sorgt für die Dünnbesetztheit der Ausgabedeskriptoren.
Die Berechnung des SEER-Deskriptors ist damit abgeschlossen und er wird an die Liste der SEER-Deskriptoren (im Code in Form einer Matrix) angehängt, was in Zeile 51 zu sehen ist. Dabei wird geprüft, ob die Dimensionalität der bisher erzeugten Deskriptoren derjenigen des neuen Deskriptors entspricht. Falls das nicht der Fall ist, werden die vorhandenen Deskriptoren mit Nullen aufgefüllt.

Die Implementation der Hilfsfunktionen, die genutzt wurden, ist aus Codeabschnitt 2 ersichtlich.

```
1  def keep_n_largest(self, arr, n):
2      #flat array for simple manipulation
3      flat_arr = arr.flatten()
4      #find threshold for n greatest values
5      if n >= len(flat_arr):
6          threshold = np.min(flat_arr)
7      else:
8          threshold = np.partition(flat_arr, -n)[-n]
9      #create mask which is True for n greatest values
10     mask = arr >= threshold
11     #create new array using this mask to set lower values to zero
12     result = np.where(mask, arr, 0)
13     return result
14
15
16 def pad_and_append_SEER(self, D, new_row):
17     if D.shape[1] > 0:
18         #Get the number of columns in the existing matrix and the new row
19         num_cols_M = D.shape[1]
20         num_cols_new = new_row.size
21         #Pad the existing matrix with zeros to match the number of
       columns in the new row
22         if num_cols_M < num_cols_new:
23             padding_M = sp.lil_matrix((D.shape[0], num_cols_new -
       num_cols_M), dtype=D.dtype)
24             D = sp.hstack([D, padding_M])
25         #Convert the new row to a sparse matrix
```

```
        new_row_sparse = sp.lil_matrix(new_row)
        #Append the new row to the matrix D in place
        D = sp.vstack([D, new_row_sparse])
    else:
        #Convert the new row to a sparse matrix and assign it to D
        D = sp.lil_matrix(new_row)
    return D


def create_exemplars(self, input_Y, n_dim_samples, n_patterns):
    PI = np.zeros((n_dim_samples, n_patterns), dtype=int)
    PV = np.zeros((n_dim_samples, n_patterns), dtype=int)
    for i in range(n_patterns):
        if n_dim_samples == input_Y.shape[0]:
            PI[:, i] = np.arange(input_Y.shape[0])
        else:
            PI[:, i] = np.random.choice(input_Y.shape[0], n_dim_samples,
    replace=False, p=np.abs(input_Y)/np.sum(np.abs(input_Y)))

    PV = input_Y[PI]

    #sparsify
    for col in range(PV.shape[1]):
        PV[:,col] = self.keep_n_largest(PV[:,col], 200)
    return PI, PV
```

Codeausschnitt 2: Hilfsfunktionen SEER-Deskriptoren

Da die Erstellung der SEER-Deskriptoren nicht der zentrale Inhalt dieser Arbeit ist, wird hier auf eine detaillierte Dokumentation verzichtet. Der Zweck dieser Funktionen ist aus ihrer Nutzung innerhalb des SEER-Algorithmus ersichtlich.

Auf Grundlage dieser Ausführungen sollen zunächst verschiedene Distanz- und Ähnlichkeitsmaße eingeführt werden, deren Eignung zur Identifikation des nächsten Nachbarn auf SEER-Daten untersucht werden soll.

4 Funktionen für Distanz und Ähnlichkeit

In der vorliegenden Arbeit wird eine Differenzierung zwischen Distanz und Ähnlichkeit vorgenommen. Dabei wird der Begriff der Ähnlichkeit so definiert, dass zwei Vektoren dann ähnlich sind, wenn entweder die Distanz minimal oder das Ähnlichkeitsmaß maximal ist. Falls beide Werte auf den Bereich zwischen 0 und 1 normiert sind, gilt die Formel 2 und die Werte können umgerechnet werden

$$similarity = 1 - distance \tag{2}$$

Da diese Voraussetzung nicht bei allen vorgestellen Metriken erfüllt ist, ist diese Differenzierung sinnvoll.

4.1 Distanzfunktionen

Nachfolgend werden zunächst die Funktionen betrachtet, die für ähnliche Vektoren ein möglichst geringes Ergebnis (im Idealfall 0) liefern.

4.1.1 Grundlagen

Üblicherweise wird in zwei- und dreidimensionalen Räumen die L_k-Norm als Distanzfunktion genutzt [HAK00], die für zwei Vektoren $x, y \in \mathbb{R}^d$ folgendermaßen definiert ist:

$$dist_{L_k}(x, y) = \sqrt[k]{\sum_{i=1}^{d}(x_i - y_i)^k} \tag{3}$$

gleichbedeutend mit

$$dist_{L_k}(x, y) = \left(\sum_{i=1}^{d}(x_i - y_i)^k\right)^{\frac{1}{k}} \tag{4}$$

In den meisten Anwendungsfällen wird $k = 1$ (Manhattan Distanz) oder $k = 2$ (Euklidische Norm) genutzt.

Für den hochdimensionalen Fall haben Beyer et al. [BGRS99] gezeigt, dass je nach Verteilung alle Punkte nahezu äquidistant zueinander sind, was wiederum dazu führt, dass die Lösung des nächste-Nachbarn-Problems instabil ist. Instabilität bedeutet in diesem Kontext, dass der nächste Nachbar schon durch eine minimale Verschiebung zum weitesten Nachbarn werden kann. Das wiederum hat zur Folge, dass bereits geringes Rauschen zu vollständig veränderten Ergebnissen führen würde. Dadurch, dass Rauschen gerade bei Kameraaufnahmen ein durchaus realistisches Szenario ist, ist diese Erkenntnis von hoher Relevanz für das untersuchte Problem.

4.1.2 Fraktionale Distanzfunktionen

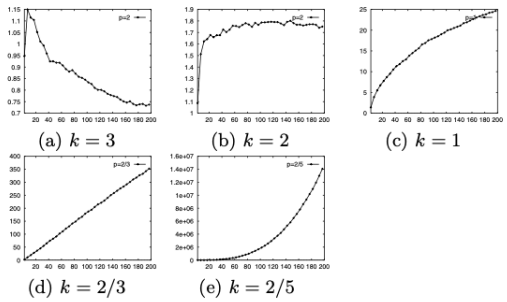
Im Gegensatz zu einer Berechung von z. B. der Distanz eines Objektes zu einem Sensor in 2D oder 3D in der realen Welt, geht es in hochdimensionalen Vektorräumen nicht um das konkrete Ergebnis einer Distanzberechnung, sondern um das Verhältnis zwischen verschiedenen Distanzberechnungen. Aus diesem Grund ist die Wahl der Distanzfunktion nicht trivial. Aggarwal et al. [AHK01] haben gezeigt, dass mit zunehmender Anzahl von Dimensionen die Aussagekraft der L_2-Norm gegenüber der L_1-Norm abnimmt. Der Begriff der Aussagekraft bezeichnet hier das Maß der Diversität einer Metrik. Je weiter verteilt die einzelnen Werte, die von dieser Metrik geliefert werden, sind, desto Aussagekräftiger ist sie.

Als Maß für besagte Aussagekraft haben Aggarwal et al. folgendes Verhältnis gewählt:

$$\frac{dist_{max} - dist_{min}}{dist_{min}} \tag{5}$$

$dist_{max}$ und $dist_{min}$ bezeichnen die jeweils größte und kleinste Distanz im Datensatz. In dem Konferenzbeitrag von Beyer et al. [BGRS99] wurde gezeigt, dass, wenn die Anzahl

der Dimensionen *dim* gegen Unendlich läuft, dieses Verhältnis für ein festes $k > 2$ gegen 0 geht (vgl. Abb. 2).



(a) $k = 3$ (b) $k = 2$ (c) $k = 1$

(d) $k = 2/3$ (e) $k = 2/5$

Abbildung 2: | $dist_{max} - dist_{min}$ | in Abhängigkeit von *dim* für verschiedene k. Entnommen aus [AHK01]

Weiterhin wurde gezeigt, dass die Differenz $dist_{max} - dist_{min}$ mit der Rate $dim^{\frac{1}{k}-\frac{1}{2}}$ wächst. Das bedeutet zusätzlich folgendes:

- für $k = 1$ (Manhattan Distanz) geht die Rate gegen ∞,
- für $k = 2$ (Euklidische Norm) bleibt die Rate konstant,

Da das Distanzverhältnis mit kleinerem k aussagekräftiger ist, wurden einige Fälle mit $0 < k < 1$ von Aggarwal et al. untersucht, mit dem Ergebnis, dass Entfernungsverhältnisse und Verteilungen deutlich besser beschrieben wurden als mit ganzzahligen k.

Die Wahl des genauen Wertes für k wurde nicht vertieft, daher werden in dieser Arbeit die experimentellen Ergebnisse der Autoren genutzt.

Die Nutzung der fraktionalen L_k-Metriken bildet somit eine mögliche Lösung für das Problem der Instabilität bei den ganzzahligen Werten für k im hochdimensionalen Fall.

4.1.3 Hamming-Distanz

Nachfolgend wird die Hamming-Distanz eingeführt, die im Verfahren des *Sign-Random-Projection Locality-Sensitive Hashing* in Kapitel 4.1.4 zur Anwendung kommt. Vorgestellt wurde diese Distanzmetrik im Jahr 1950 von Richard W. Hamming [Ham50]. Üblicherweise wird das Verfahren genutzt, um Fehler in Zeichenketten zu korrigieren. Im Kontext dieser Arbeit geht es aber nur um das Abstandsmaß, das definiert ist als

$$dist_{ham}(x, y) := | \{j \in \{1, ..., n\} \mid x_j \neq y_j\} | \tag{6}$$

Die Hamming-Distanz ist also die Anzahl der Stellen an denen zwei miteinander verglichene Codes unterschiedliche Werte aufweisen.

4.1.4 Sign-Random-Projection Locality-Sensitive Hashing

Ji et. Al. stellen in ihrem Paper *Super-Bit Locality-Sensitive Hashing* [JLY+12] die Algorithmengruppe des *Locality-Sensitive Hashing* vor, die Annäherungen an die Winkelähnlickeit darstellt. Konkret wird der *Sign-Random-Projection LSH*-Algorithmus vorgestellt, der folgendermaßen funktioniert:
Sei v ein dim-dimensionaler Vektor aus der Normalverteilung $\mathcal{N}(0, I_{dim})$. Sei außerdem x ein Vektor aus dem zu untersuchenden Datensatz (ebenfalls dim-dimensional). Dann ist die SRP-LSH Funktion $h_v(x)$ definiert als $h_v(x) = sgn(v^T x)$, wobei gilt:

$$sgn(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases} \tag{7}$$

Goemans und Williamson [GW95] haben gezeigt dass folgende Gleichung gilt:

$$Pr(h_v(a) \neq h_v(b)) = \frac{\theta_{a,b}}{\pi} \tag{8}$$

Diese Gleichung zeigt, dass die Wahrscheinlichkeit für die Unähnlichkeit der Hashwerte der beiden Vektoren a und b dem eingeschlossenen Winkel $\theta_{a,b}$ entspricht. Dieser wird durch den Faktor $\frac{1}{\pi}$ auf den Wertebereich zwischen 0 und 1 umgerechnet, sodass ein Winkel von 0° gleiche Hashwerte zur Folge hat und analog dazu ein Winkel von 180° in einer Wahrscheinlichkeit von 0 für gleiche Hashwerte resultiert.
Wenn man K dim-dimensionale Vektoren $v_1, ..., v_K$ aus der Normalverteilung $\mathcal{N}(0, I_{dim})$ sampelt, kann man die Funktion $h(x) = (h_{v_1}(x), h_{v_2}(x), ..., h_{v_K}(x))$ definieren. Diese Funktion enthält K SRP-LSH Funktionen und erzeugt somit K-stellige Bitcodes. Daraus folgt

$$\mathbb{E}\left[dist_{ham}(h(a), h(b))\right] = \frac{K\theta_{a,b}}{\pi} = C\theta_{a,b} \tag{9}$$

Dies zeigt, dass der Erwartungswert der Hamming-Distanz zwischen den Bitcodes der Vektoren a und b eine Annäherung an den Winkel zwischen den beiden Vektoren, multipliziert mit der Konstante $C = K/\pi$, darstellt. Somit liefert SRP-LSH eine Annäherung an die Winkelähnlichkeit.

In dem gleichen Paper führen Ji et. Al. weiter aus, dass die große Varianz von $dist_{ham}(h(a), h(b))$ zu einem großen Annäherungsfehler führen kann. Um eine gute Annäherung zu gewährleisten, muss der erzeugte Bitcode lang genug sein. Das bedeutet, dass eine entsprechende Anzahl von Zufallsvektoren vorliegen muss. Je mehr Vektoren aus der Verteilung gezogen werden, desto wahrscheinlicher ist eine lineare Abhängigkeit innerhalb dieser Vektoren. Dies wiederum führt dazu, dass der Informationsgehalt der Bitcodes sinkt. Daher wird eine Orthogonalisierung der Vektoren vorgeschlagen, die die Varianz senkt und somit den Informationsgehalt erhöht.

4.2 Ähnlichkeitsmetriken

4.2.1 Skalarprodukt und Cosinusähnlichkeit

In Kapitel 3 auf Seite 3 werden SEER-Desriptoren erklärt, die das Skalarprodukt als Ähnlichkeitsmetrik nutzen. Der zugrundeliegende Gedanke ist der folgende: Das Ska-

larprodukt der Vektoren a und b beinhaltet die Information über den Winkel, den sie einschließen [LZX+18]:

$$a \cdot b = |a| \cdot |b| \cdot \cos\theta \tag{10}$$

Da die Norm der Vektoren positiv ist, lassen sich aus dem Skalarprodukt generell folgende Erkenntnisse festhalten:

- $a \cdot b < 0$: θ ist größer als 90°
- $a \cdot b = 0$: θ ist gleich 90°
- $a \cdot b > 0$: θ ist kleiner als 90°

Dabei gilt die Einschränkung, dass wegen der Kommutativität des Skalarproduktes ($a \cdot b = b \cdot a$), der Winkel θ im Wertebereich zwischen 0° und 180° liegt. Dadurch gilt ebenfalls, dass ein hohes Skalarprodukt zweier Vektoren auf eine hohe Ähnlichkeit hinweist. Dieser Effekt wird bei dem Test auf genügend Exemplare in der Matrix M bei der Erstellung der SEER-Deskriptoren genutzt.
Das Skalarprodukt findet häufig Anwendung bei der Ähnlichkeitsbestimmung in hochdimensionalen Vektorräumen, zum Beispiel bei der Ermittlung der Ähnlichkeit zwischen zwei Textdokumenten [LZX+18].

Aus dem Skalarprodukt geht durch Umstellen unmittelbar die Formel für die Cosinusähnlichkeit hervor:

$$\cos\theta = \frac{a \cdot b}{|a| \cdot |b|} \tag{11}$$

Da gilt, dass $\cos(0°) = 1$ und $\cos(180°) = -1$, und dies auch den gesamten Wertebereich des Cosinus abdeckt, gilt ebenso: Je größer das Ergebnis der Cosinusähnlichkeit ausfällt, desto ähnlicher sind die zwei Vektoren zueinander, in dem Sinne, dass sie in die gleiche Richtung zeigen. Da jedoch auch die Längen der Vektoren in dem Ergebnis berücksichtigt werden, ist es ratsam, mit normalisierten Vektoren zu rechnen. Die Cosinusähnlichkeit ist eine in der Forschung etablierte Ähnlichkeitsmetrik für hochdimensionale Datensätze, wie aus [Ass12], [SEK04] und [LZX+18] hervorgeht.

4.2.2 IGRID Index

Aggarwal und Yu haben im Jahr 2000 den IGRID Index als einen vielversprechenden Ansatz für Nächste-Nachbarn-Suche in hochdimensionalen Räumen vorgestellt [AY00]. Der zugrundeliegende Gedanke ist, dass nicht alle Dimensionen bei der Entfernungsmessung bzw. Ähnlichkeitsprüfung berücksichtigt werden, sondern nur die, auf denen die Punkte nah genug aneinander liegen. Somit wird erreicht, dass Punkte, die in vielen Dimensionen nah aneinander liegen, eine höhere Ähnlichkeitsbewertung erreichen, als solche, die überall weit voneinander entfernt sind.

4.2.2.1 Ähnlichkeitsfunktion

Für den IGRID Index wurde die folgende Ähnlichkeitsfunktion definiert:

$$IDist(x,y) = \left[\sum_{i=1}^{d}\left(1 - \frac{\mid x_i - y_i \mid}{u_i - l_i}\right)^p\right]^{1/p} \tag{12}$$

x ist einer der Vekoren, die auf Ähnlichkeit untersucht werden sollen mit den Einträgen x_i für die Dimension i (analog für y). *dim* ist die Anzahl der Dimensionen, u_i und l_i (vom Englischen *upper/lower*) sind die Grenzen des Wertebereichs für die Dimension i. Die Berücksichtigung des Wertebereichs stellt eine Normalisierung innerhalb des Datensatzes sicher. Um den Begriff „Nähe“ zu definieren, werden die Wertebereiche aller Dimensionen in k_d Bereiche eingeteilt. Die Einträge, die gemeinsam in einem Bereich liegen, werden als nah zueinander betrachtet. Bei dieser Einteilung ist zu beachten, dass es nicht um gleich „breite“ Bereiche handelt, sondern um Bereiche mit gleicher „Tiefe“. Dies bedeutet, dass die Bereiche so aufgeteilt werden, dass in allen Bereichen die gleiche Anzahl von Einträgen liegt. Dies hat dann zur Folge, dass eng besetzte Dimensionen deutlich engere Bereiche aufweisen, als Dimensionen, auf denen die Datenpunkte weiter verteilt sind. In jedem dieser Bereiche befinden sich folglich $1/k_d$ Punkte. Diese Tatsache gilt, wie auch einige andere Grundannahmen in diesem Algorithmus, jedoch nur für voll besetzte Datensätze. Der Bereich an der Stelle j in Dimension i heißt dann $\mathcal{R}[i,j]$. Für die Berechnung der Ähnlichkeit bedeutet dies Folgendes: Seien x und y zwei Vektoren. Die Menge der Dimensionen, auf denen beiden Vektoren ähnlich sind besteht aus den Dimensionen, auf denen die Einträge im gleichen Bereich liegen. Somit ist $\mathcal{S}[x,y,k_d]$ die Menge der ähnlichen Dimensionen der Vektoren x und y für den Diskretisierungsgrad k_d. Außerdem seinen m_i und n_i die oberen und unteren Grenzen des jeweiligen Bereiches. Dann ist die Formel für die Ähnlichkeit zweier Vektoren bei der gegebenen Diskretisierungsstufe:

$$PIDist(x,y.k_d) = \left[\sum_{i \in \mathcal{S}[x,y,k_d]}\left(1 - \frac{\mid x_i - y_i \mid}{m_i - n_i}\right)^p\right]^{1/p} \tag{13}$$

Da jeder einzelne Summand zwischen 0 und 1 liegen wird, liegt der Wertebereich der Gesamtfunktion zwischen 0 und $\mid \mathcal{S}[x,y,k_d] \mid$. Je ähnlicher sich die Vektoren sind, desto mehr Komponenten enthält die Summe mit gleichzeitig höheren Werten. Somit wird bei diesem Ansatz nicht die Distanz (also das Maß der Unähnlichkeit) gemessen, mit dem Ziel das Minimum zu finden, sondern der Fokus liegt auf dem Maß der Ähnlichkeit. Das Ignorieren der Dimensionen, die nicht zur Menge $\mathcal{S}$ gehören, führt grundsätzlich zu einem Informationsverlust. Die empirisches Tests von Aggarwal und Yu haben jedoch gezeigt, dass die dadurch ebenfalls herbeigeführte Rauschreduktion überwiegt.

4.2.2.2 Indexstruktur

Um die Zugriffszeit bei der Suche nach dem nächsten Nachbarn zu reduzieren, haben Aggarwal und Yu zusätzlich die folgende Indexstruktur entwickelt:

Für jede Dimension und jeden Bereich innerhalb dieser Dimension werden die Ober- und Untergrenze gespeichert. Außerdem wird für jeden dieser Einträge eine Liste der Indizes gespeichert, die in diesem Bereich liegen, mit dem jeweiligen Eintrag auf der betreffenden Dimension. Dies vereinfacht die Suche nach den Nächsten Nachbarn, da für jeden Punkt p sämtliche Kandidaten in der Indexstruktur schnell abrufbar sind und dadurch nur die relevante Teilmenge des Datensatzes durchsucht wird.

Potenziell kann das Problem auftreten, dass zwei Punkte aus benachbarten Bereichen jeweils im Grenzbereich liegen und eine geringere Distanz zueinander aufweisen, als zu Punkten aus ihren Bereichen. Durch die Einteilung in feste Bereiche könnte diese Nachbarschaft unentdeckt bleiben und zu falschen Ergebnissen führen. Um einem solchen Fehler entgegenzuwirken werden die Bereiche jeweils noch in l Unterbereiche eingeteilt. So können bei der Kandidatensuche jeweils noch n (üblicherweise 1) benachbarte Unterbereiche einbezogen werden.

4.2.3 Anzahl der gemeinsam belegten Dimensionen

4.2.3.1 Ähnlichkeitsfunktion

Der in dieser Arbeit neu vorgestellte Ansatz der Anzahl von gemeinsam belegten Dimensionen folgt der simplen formalen Definition die in der Gleichung 14 zu sehen ist.

$$sim(x, y) =\mid S(x, y) \mid \tag{14}$$

Wobei gilt, dass

$$S = D_x \cap D_y \tag{15}$$

und

$$\begin{aligned} D_x &= \{i \in \{1, 2, .., n\} \mid x_i \neq 0\} \\ D_y &= \{i \in \{1, 2, .., n\} \mid y_i \neq 0\} \end{aligned} \tag{16}$$

Somit werden hier explizit nur die Dimensionen betrachtet, die besetzt sind, und auch nur die, die von beiden untersuchten Vektoren x und y gemeinsam besetzt sind. Implizit geschieht das auch beim Skalarprodukt bzw. bei der Cosinusähnlichkeit, da nur die Summanden in die Summe eingehen, bei denen beide Faktoren ungleich Null sind. Da die Berechnung trotzdem ausgeführt wird, ist das nur eine implizite Berücksichtigung der Dünnbesetztheit.
Hier wird nur die Anzahl der besagten Dimensionen als Ähnlichkeitsmetrik verwendet, ohne auf die konkreten Zahlenwerte einzugehen.

4.2.3.2 Indexstruktur

Für die Laufzeitreduktion dieses Algorithmus eignet sich die folgende Datenstruktur, die der Indexstruktur für den IGRID-Index ähnelt: Zu Beginn wird eine Indextruktur, zum Beispiel eine Liste von Listen so initialisiert, dass für jede Dimension des vorliegenden Datensatzes eine „Unterliste“ vorhanden ist. Dann werden für alle Datenbankdeskriptoren die folgenden Schritte ausgeführt:

1. Bestimmung aller Nicht-Null-Dimensionen

2. Einfügen des Index des aktuellen Deskriptors in jede ensprechende Unterliste

Somit liegt am Ende eine Struktur vor, die für jeden Deskriptor x, für den der nächste Nachbar gesucht werden soll, alle möglichen Kandidaten liefert. Dafür müssen nur die Nicht-Null-Dimensionen des Deskriptors identifiziert werden und dann eine Gesamtliste aus den entsprechenden Listen aus der Indexstruktur gebildet werden. Für alle Deskriptoren y außerhalb dieser Liste gilt $sim(x, y) = 0$. Für alle Indizes innerhalb der Liste wird die Anzahl der Vorkommnisse ermittelt und ergibt somit die Anzahl der Dimensionen, die beide Vektoren gemeinsam besetzen. Diese Vorgehensweise verhindert viele unnötige Gleichheitsprüfungen und wirkt sich positiv auf die Anzahl der Rechenopeationen und Speicherzugriffe und somit auch auf die Laufzeit aus.

5 Analyse der verschiedenen Verfahren

Dadurch, dass die Daten dünn besetzt sind, finden die meisten vorher ausgeführten Lösungsansätze für das NN-Problem nur eingeschränkt Anwendung. Die eventuellen Änderungen werden an der jeweiligen Stelle kurz erläutert. Grundsätzlich wird bei allen Verfahren zuerst die Genauigkeit betrachtet, weil eine Optimierung hinsichtlich der Geschwindigkeit nur sinnvoll ist, wenn eine gewisse Klassifikationsrate erreicht wird. Diesbezüglich werden drei Zahlen erhoben:

- Die Distanz (in Bildern / Frames) des erkannten nächsten Nachbarn von dem wirklichen nächsten Nachbarn (idealerweise 0) (In den Tabellen unter d aufgeführt: Median: d_{med}, Mittelwert: d_{mean})

- Die Position des wirklichen Nächsten in den sortierten Ergebnissen des Algorithmus (idealerweise Index 0 in den meisten Programmiersprachen)(In den Tabellen unter p aufgeführt: Median: p_{med}, Mittelwert: p_{mean})

- Die Proportion der Ergebnisse, bei denen der wirkliche nächste Nachbar in den besten 5 zurückgegebenen Ergebnissen liegt, zur Anzahl der Anfragedeskriptoren. (In den Tabellen unter *prop* aufgeführt)

Grundsätzlich werden in den Codeauszügen der Übersichtlichkeit halber die Imports weggelassen, die einmal im Codeabschnitt 3 aufgeführt sind:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import scipy.sparse as sp
4 from collections import Counter
5 from scipy.spatial import distance
6 from time import perf_counter
```

Codeausschnitt 3: Python Imports

5.1 Verwendete Daten

Wie in Kap. 3 ausgeführt, wird in dieser Arbeit die Nächste-Nachbarn-Suche für SEER-Deskriptoren untersucht. Die dazu verwendeten Daten sind folgende:

- Zwei Versionen von SEER-Deskriptoren des *Garden Points Walking* Datensatzes[2] [NS22] (je 200 Bilder für Basisdaten und Querydaten)
  - 6716 Dimensionen mit max. 100 nicht-Null Einträgen pro Deskriptor (Erzeugt mit dem Code aus Kapitel 3)
  - 7696 Dimensionen mit max. 100 nicht-Null Einträgen pro Deskriptor
- Je 1000 Deskriptoren aus dem Nordland-Datensatz[3] (Frühling und Winter) mit 49389 Dimensionen und max. 100 nicht-Null Einträgen pro Deskriptor [SNP13]

5.2 Python-Code für die Analyse

Für die Auswertung aller Algorithmen wurde der Code in den Codeabschnitten 4, 5 und 6 genutzt.

```
def run_and_evaluate_func_on_data(function, dataX, dataY, datastring, functionstring, similarity_mode, *args):
    start = perf_counter()
    #call function with given data and optional arguments
    res_matrix = function(dataX, dataY, *args)
    end = perf_counter()
    duration = end - start
    #create a matrix where only the best match is set to 1 and all others to 0
    res_matrix_dist_hard = dist_principal_diagonal(calc_hard_matrix(res_matrix, similarity_mode))
    #create a matrix where the entries are the indices of the sorted NNs
    res_rank = get_position_of_correct_frame(res_matrix, similarity_mode)
    print_matrices(res_matrix, res_matrix_dist_hard, res_rank, datastring, functionstring, *args)

    #create a mask for the top 5 findings for each descriptor
    mask = res_rank < 5
    #Sum the boolean mask to get the count of elements less than 5
    res_matrix_count_tolerated = np.sum(mask)
    print(f"Duration {functionstring}: {duration} seconds")
    print(f"Median Distance to GT-NN: {np.median(res_matrix_dist_hard)}")
    print(f"Average Distance to GT-NN: {np.mean(res_matrix_dist_hard)}")
    print(f"Median Rank of GT-NN: {np.median(res_rank)}")
    print(f"Average Rank of GT-NN: {np.mean(res_rank)}")
    print(f"{res_matrix_count_tolerated} out of {res_matrix.shape[0]} within top 5 NNs")
```

[2] Download unter `https://www.tu-chemnitz.de/etit/proaut/en/research/rsrc/prstructure/SEER/data.zip`

[3] Download unter `https://nrkbeta.no/2013/01/15/nordlandsbanen-minute-by-minute-season-by-season/`

```
    return res_matrix, res_matrix_dist_hard, res_rank
```

Codeausschnitt 4: Python Code für die Auswertung der Algorithmen

```
def print_matrices(gs_mat, dist_hard, rank, datastring, functionstring, *args):
    fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(15, 5))
    s = args
    fig.suptitle(f"Dataset: {datastring}, Function: {functionstring}, Parameters: {s}")
    x = np.arange(0, gs_mat.shape[0])
    ax1.imshow(gs_mat, cmap='gray')
    ax1.set_title("Grayscale Results")

    values1, bins1, bars1 = ax2.hist(dist_hard, bins=[0, 1, 2, 3, 4, 5, 10, 20, 50, 200], edgecolor='white')
    ax2.bar_label(bars1, fontsize=10, color='navy')
    ax2.set_xscale('log')
    ax2.set_title("Distance to correct NN (frames)")

    values2, bins2, bars2 = ax3.hist(rank, bins=[0, 1, 2, 3, 4, 5, 10, 20, 50, 200], edgecolor='white')
    ax3.bar_label(bars2, fontsize=10, color='navy')
    ax3.set_xscale('log')
    ax3.set_title("Index of correct frame in sorted NNs")


    s1 = (f"_{s}_").replace('(','').replace(')','').replace(", ",'_').replace('.','-')
    dir = "./exports/"

    # Save the full figure
    file_full = (f"{dir}{functionstring}_{s1}_{datastring}_full").replace(' ', '')
    fig.savefig(file_full)
```

Codeausschnitt 5: Python Code für die Ausgabe der Auswertungsmatrizen

```
def get_position_of_correct_frame(matrix, similarity_mode):
    length = matrix.shape[0]
    positions = np.zeros(length)
    #input: similarity matrix
    for i in range(length):
        sorted_indices = np.argsort(matrix[i])
        #similarity or distance mode
        if similarity_mode:
            pos = length - np.where(sorted_indices==i)[0][0] - 1
        else:
            pos = np.where(sorted_indices==i)[0][0]
        positions[i] = pos
    return positions

def calc_hard_matrix(matrix, similarity_mode):
    #Create a new matrix with the same shape, filled with zeros
```

```
    result = np.zeros_like(matrix)
    if similarity_mode:
        indices = np.argmax(matrix, axis=1)
    else:
        indices = np.argmin(matrix, axis=1)
    result[np.arange(matrix.shape[0]), indices] = 1
    #return a matrix where the best candidate per row is set to one
    return result

def dist_principal_diagonal(matrix):
    length = matrix.shape[0]
    differences = np.zeros(length)
    #use the fact, that we need a principal diagonal here
    for i in range(length):
        difference = np.abs(np.argmax(matrix[i]) - i)
        differences[i] = difference
    return differences
```

Codeausschnitt 6: Python Code für die Berechnung der Auswertungsmatrizen

Durch die einheitliche Auswertung aller Algorithmen wird eine hohe Vergleichbarkeit erreicht. Zusätzlich vereinfacht diese Herangehensweise das Ausführen der einzelnen Funktionen mit variierenden Parametern durch eine einheitliche Implementierung, bei der nur die Parameter angepasst werden müssen. Einige mit diesem Code erstellte Diagramme sind im Text zu sehen, in dem GitHub-Repository dieser Arbeit[4] können alle Anderen ebenfalls eingesehen werden.

5.3 Fraktionale Distanzfunktionen

Die fraktionalen Distanzfunktionen aus Kapitel 4.1.2 wurden auf alle Paarungen zwischen Basis- und Querydaten angewendet. Ausgeführt wurde dies mit dem Python-Code in Codeabschnitt 7. Dabei ist wichtig zu beachten, dass die Dünnbesetztheit der Vektoren hier keine besondere Berücksichtigung findet.

```
def lk_distance(k, x1, x2):
    #test if x1 and x2 have equal length
    if (len(x1) != len(x2)):
        print('Input vectors have different lengths!')
        return
    else:
        #compute and return lk Length
        result = np.sum(np.abs(x1 - x2) ** k) ** (1/k)
        return result

def sequential_lk_dist(Xin, Yin, k):
    result_Matrix = np.empty((Xin.shape[0], Yin.shape[0]), dtype=np.
    float32)
    for i in range(Xin.shape[0]):
        for j in range(Yin.shape[0]):
```

[4] https://github.com/bzuravlev/nnsearch_sparse_hd_data

```
15              dist = lk_distance(k, Xin[i].toarray()[0], Yin[j].toarray()
    [0])
16              result_Matrix[j][i] = dist
17      return result_Matrix
```

Codeausschnitt 7: Code Fraktionale Distanzen

Soweit möglich wurden bei allen Algorithmen für die Implementierung Funktionen aus der Numpy-Bibliothek genutzt, da diese compilierten C-Code nutzen und somit hinsichtlich der Ausführungsgeschwindigkeit eine sehr gute Wahl in Python-Code darstellen. Zusätzlich können einige Operationen auf gesamten Arrays ausgeführt werden, sodass nicht jedes Element in einer Schleife einzeln adressiert werden muss (siehe Zeile 8 in Codeabschnitt 7).

Die Ergebnisse für die Fraktionalen Distanzfunktionen sind in den Tabellen 1 und 2 zu sehen.

Fraktionale Distanzen, $k = 0.1$					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	81.5	83.88	6.0	15.06	76/200
Garden Points 2	87.5	88.46	5.0	17.43	97/200
Nordland	0.0	26.62	0.0	7.35	947/1000

Tabelle 1: Ergebnisse für Fraktionale Distanzen, $k = 0.1$

Fraktionale Distanzen, $k = 0.2$					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	81.5	83.66	6.0	17.48	75/200
Garden Points 2	88.5	89.55	5.0	21.43	94/200
Nordland	0.0	26.53	0.0	8.86	950/1000

Tabelle 2: Ergebnisse für Fraktionale Distanzen, $k = 0.2$

Als Beobachtungen lassen sich einige wesentliche Punkte festhalten. Zunächst fällt auf, dass der Algorithmus auf den kleineren Datensätzen deutlich schlechtere Ergebnisse erzielt, als auf dem größeren Nordland-Datensatz. Dies liegt unter Anderem daran, dass in einem kleineren Datensatz einzelne Fehler stärker gewichtet sind, als in einem großen Datensatz. Dieser Effekt tritt auch bei den meisten anderen untersuchten Ansätzen auf. Außerdem sieht man, dass die Unterschiede zwischen den Ergebnissen für $k = 0.1$ und $k = 0.2$ marginal sind, mit leicht besseren Ergebnissen für $k = 0.1$.

Eine Diskrepanz, die stark auffällt, ist, dass die Distanz zum korrekten nächsten Nachbarn sowohl im Mittelwert als auch im Median vergleichsweise hoch ausfällt, wobei die Werte für die Position des korrekten nächsten Nachbarn in den sortierten Ergebnissen deutlich besser sind. Außerdem tritt dies nur bei den Garden-Points-Datensätzen auf. Bei der näheren Analyse der Daten hinsichtlich des Zustandekommens dieses Ergebnisses ist aufgefallen, dass die ersten Deskriptoren der Garden-Points-Datensätze weniger als 100

nicht-Null Einträge aufweisen. Diese werden daher als Nächste Nachbarn für fast alle folgende Deskriptoren gefunden und alle folgenden Kandidaten werden als weiter entfernt eingestuft.
Ein Testdurchlauf, bei dem die ersten 5 Deskriptoren weggelassen wurden, bestätigt diese Annahme und zeigt deutliche Verbesserungen, wie in Tabelle 3 zu sehen ist.
Diese Verbesserung ist auch in den grafischen Ausgaben der Auswertungsfunktion in Abb. 3 zu sehen, hier am Beispiel des zweiten Garden Points Datensatzes.

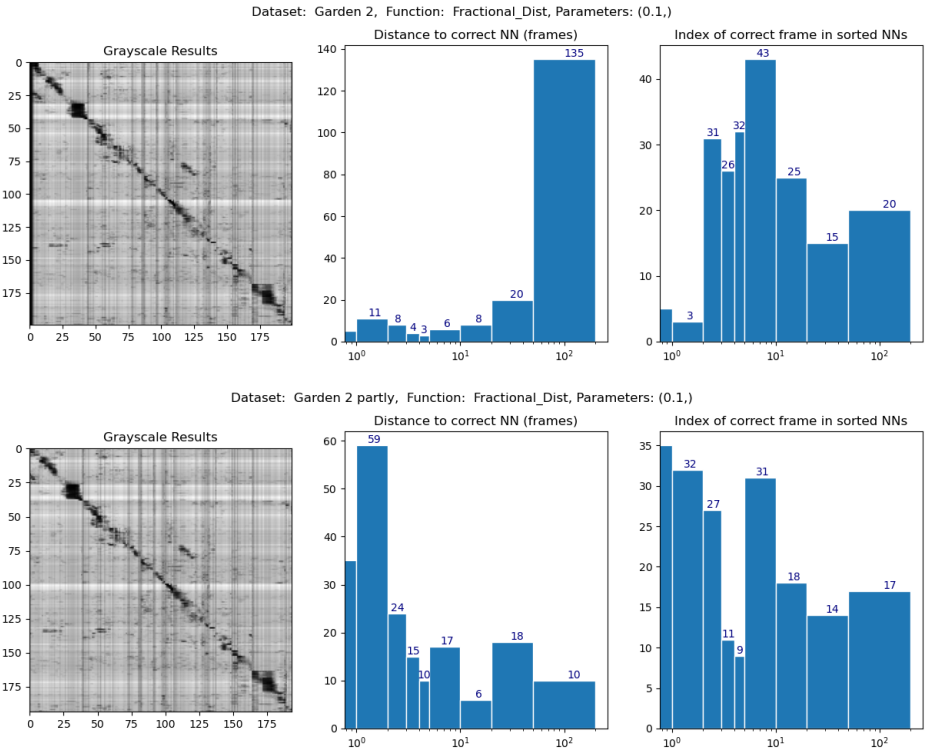


Abbildung 3: Visuelle Darstellung der Ergebnisse der Fraktionalen Distanzen für $k = 0.1$ und der Verbesserung durch das Weglassen der ersten 5 Deskriptoren. Auffällig ist der schwarze Balken am linken Rand der oberen Graustufendarstellung, der eine geringe Distanz darstellt. Dieser Balken fehlt unten, weil die entsprechenden Deskriptoren nicht berücksichtigt werden. Darüber hinaus fällt die positive Verschiebung beider Auswertungsmetriken nach links auf.

5.4 L_k-Distanzfunktionen auf gemeinsamen Dimensionen

Um das Problem zu adressieren, welches durch die niedrige Zahl der Dimensionen im ersten Deskriptor entsteht, sollen nur die Dimensionen in die Berechnung einbezogen werden, bei denen in beiden zu vergleichenden Deskriptoren ein Wert ungleich Null eingetragen ist. Dafür wird der Code aus dem Codeabschnitt 8 verwendet:

Fraktionale Distanzen, $k = 0.1$					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	2.0	9.35	3.0	12.54	115/195
Garden Points 2	2.0	9.1	3.0	15.20	114/195

Tabelle 3: Ergebnisse für Fraktionale Distanzen, $k = 0.1$, erste 5 Bilder weggelassen

```
def get_common_indices(sparsem1, sparsem2):
    #Get and return the indices of the dimensions which are present in
    both sparse vectors
    return np.intersect1d(sparsem1.nonzero()[1],  sparsem2.nonzero()[1])

def get_values_on_common_dimension(sparsem1, sparsem2):
    #Get common indices
    common_indices = get_common_indices(sparsem1, sparsem2)

    #Get and return corresponding values
    common_values1 = sparsem1[0, common_indices].toarray()[0]
    common_values2 = sparsem2[0, common_indices].toarray()[0]
    return common_values1, common_values2

def distance_on_matching_dims(sparsem1, sparsem2, k):
    common_values1, common_values2 = get_values_on_common_dimension(
    sparsem1, sparsem2)
    if common_values1.size == 0:
        return np.inf
    else:
        #compute and return distance on common distances
        result = lk_distance(k, common_values1, common_values2)
        return result
```

Codeausschnitt 8: Distanzfunktionen auf gemeinsamen Dimensionen

Auch hier wird, soweit möglich, auf Funktionen aus der Numpy Bibliothek zurückgegriffen.

Die erzielten Ergebnisse sind in den Tabellen 4, 5 und 6 zu sehen.

Distanz auf gemeinsamen Dimensionen, $k = 2$					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	54.0	60.57	74.5	75.35	0/200
Garden Points 2	49.5	59.76	54.5	55.77	2/200
Nordland	247.0	308.37	81.0	87.89	0/1000

Tabelle 4: Distanz auf gemeinsamen Dimensionen, $k = 2$

Die Ergebnisse verschlechtern sich drastisch, was wiederum auf die Dünnbesetztheit zurückzuführen ist. Weisen zwei Vektoren eine geringe Anzahl vom gemeinsam besetzten Dimensionen auf, werden sie vom Algorithmus als näher aneinander eingestuft, als zwei Vektoren, die deutlich mehr gemeinsame Dimensionen aufweisen. Dies liegt daran,

Distanz auf gemeinsamen Dimensionen, $k = 1$					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	54.5	61.22	77.0	77.11	0/200
Garden Points 2	50.0	60.41	58.0	56.92	2/200
Nordland	245.0	307.45	81.0	87.91	0/1000

Tabelle 5: Distanz auf gemeinsamen Dimensionen, $k = 1$

Distanz auf gemeinsamen Dimensionen, $k = 0.1$					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	54.0	60.54	79.0	77.65	0/200
Garden Points 2	50.0	59.32	57.0	58.17	1/200
Nordland	240.5	305.52	81.0	87.70	0/1000

Tabelle 6: Distanz auf gemeinsamen Dimensionen, $k = 0.1$

dass mehr Summanden in die Distanzfunktionen eingehen, wenn mehr Dimensionen bei beiden Vektoren mit Werten belegt sind. Eine höhere Anzahl von gemeinsam belegten Dimensionen weist jedoch tendenziell auf eine höhere Ähnlichkeit der zwei Vektoren hin. Somit muss festgehalten werden, dass dieser Ansatz für das Lösen des gestellten Problems ungeeignet ist. Das beschriebene Verhalten ist auch auf Abb. 4 zu erkennen.

Eine mögliche Gegenmaßnahme wäre eine Berücksichtigung der Anzahl der Dimensionen in der Distanzmetrik zum Beispiel dadurch, dass die Distanz durch die Anzahl der Summanden geteilt wird. Dies führt zu leichten Verbesserungen, jedoch kann das Ergebnis auch dann nicht als zufriedenstellend bezeichnet werden.

5.5 Vereinfachte PIDist-Ähnlichkeit

Ein nächster sinnvoller Schritt um die schlechten Ergebnisse des vorangestellten Ansatzes zu verbessern, ist der Wechsel von der Distanzberechnung hin zur Berechnung der Ähnlichkeit. Die von der PIDist aus Kapitel 4.2.2 abgeleitete vereinfachte Ähnlichkeitsfunktion verfolgt genau diesen Ansatz. Die Ähnlichkeit wird wie folgt berechnet: Für jedes Paar betrachteter Vektoren wird ein Ähnlichkeitswert inkrementell über alle gemeinsamen Dimensionen summiert:

$$PIDist_{simple}(x, y) = \left[\sum_{i \in \mathcal{S}[X,Y]} \left(1 - \frac{\mid x_i - y_i \mid}{max_i - min_i} \right)^p \right]^{(1/p)} \tag{17}$$

x und y sind die beiden betrachteten Deskriptoren, S die Menge der Dimensionen, die gemeinsam mit Werten belegt sind. Die Vereinfachung im Vergleich zur $PIDist$-Funktion von Aggarwal und Yu besteht darin, dass keine Diskretisierung in Bereiche innerhalb der Dimensionen stattfindet. Dies hat den Hintergrund, dass bei den vorliegenden Daten die Dünnbesetztheit ein zentrales Kriterium ist, und somit, den gesamten Datensatz betreffend, keine generellen Annahmen über die Anzahl der Einträge in einer Dimension

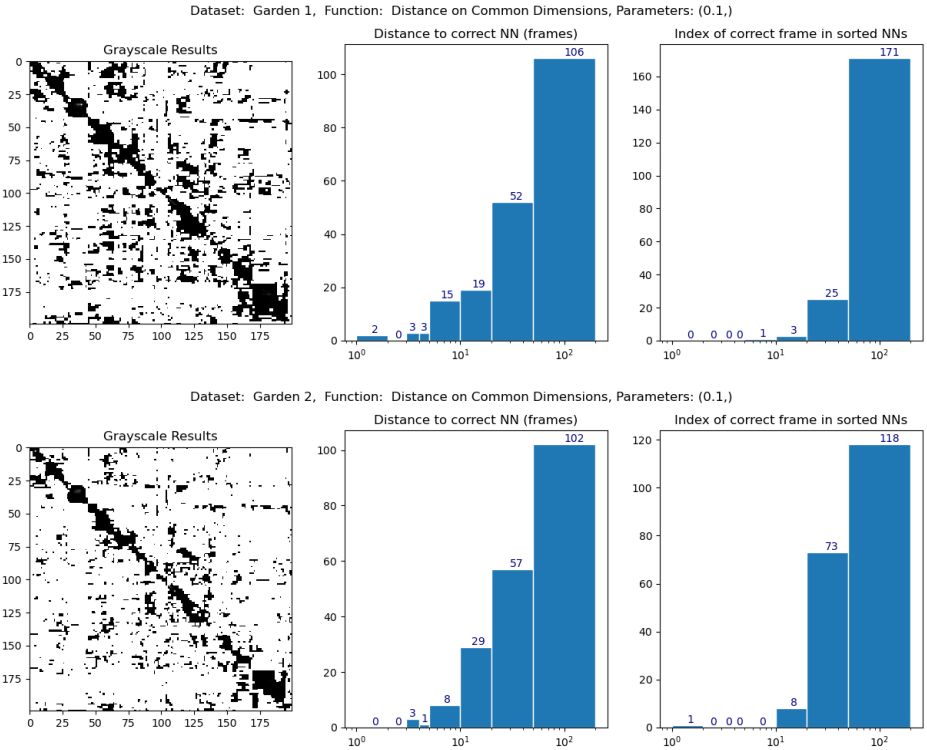


Abbildung 4: Visuelle Darstellung der Ergebnisse der Distanz auf gemeinsamen Dimensionen auf den Garden Points Datensätzen. Deutlich sichtbar ist der Effekt, dass die eigentlich Ähnlichen Deskriptoren als weit voneinander entfernt eingestuft wurden, die Hauptdiagonale beschreibt hier sehr große Distanzen.

getroffen werden können. Es gibt jedoch eine sehr hohe Wahrscheinlichkeit dafür, dass auf eine Dimension gesehen so wenige Einträge vorhanden sind, dass eine Diskretisierung wenig Sinn ergibt. Daher wird in dieser Arbeit der Wertebereich der Dimension ($max_i - min_i$) als Normalisierungsfaktor gewählt. Die Implementierung ist im Codeabschnitt 9 zu sehen.

```
def get_lowest_per_dimension(matrix):
    #convert all zeros to infinity
    arr_with_inf = np.where(matrix.toarray() == 0, np.inf, matrix.toarray
    ())
    #get lowest per dimension while ignoring zeros
    lowest_per_dim = np.amin(arr_with_inf, axis=0)
    return lowest_per_dim

def piDist_simple_sequential(Xin, Yin, p):
    #iterate over all points
    result_Matrix = np.zeros((Xin.shape[0], Yin.shape[0]), dtype=np.
    float32)
    highest_per_dim = np.amax(Xin, axis=0).toarray()[0]
    lowest_per_dim = get_lowest_per_dimension(Xin)
    range_per_dim = highest_per_dim - lowest_per_dim
    for i in range(Xin.shape[0]):
        matrix1 = Xin[i]
        for j in range(Yin.shape[0]):
            matrix2 = Yin[j]
            common_indices = get_common_indices(matrix1, matrix2)
            valuesX, valuesY = get_values_on_common_dimension(matrix1,
    matrix2)
            differences = np.abs(valuesX - valuesY)
            similarity = (np.sum( (1- differences / range_per_dim[
    common_indices])**p ))** (1/p)
            result_Matrix[j][i] = similarity
    return result_Matrix
```

Codeausschnitt 9: Angepasster vereinfachter PIDist

Auch hier werden die Eigenschaften von Numpy-Arrays ausgenutzt, die eine effiziente Berechnung der benötigten Werte möglich machen. Sämtliche Differenzen und Quotienten werden ohne zusätzliche Schleifen in einer Operation berechnet.

Die mit diesem Ansatz erzielten Ergebnisse sind in den Tabellen 7, 8 und 9 aufgeführt.

PIDist vereinfacht, $p = 2$					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	32.0	46.80	7.5	17.69	60/200
Garden Points 2	2.0	15.89	4.0	15.39	109/200
Nordland	250.0	334.53	4.0	12.02	503/1000

Tabelle 7: $PIDist$ vereinfacht, $p = 2$

Auffällig ist hierbei zunächst, dass die beiden Garden-Points-Datensätze teils stark unterschiedliche Ergebnisse liefern. Diese sind auch in Abb. 5 zu sehen. Eine mögliche

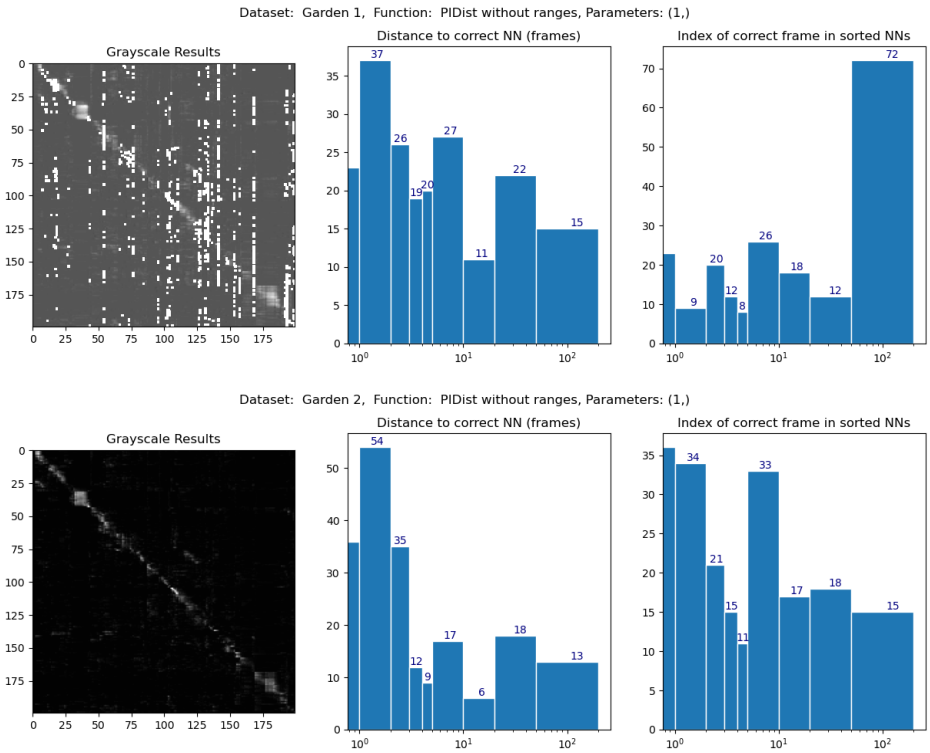


Abbildung 5: Visuelle Darstellung der Ergebnisse der vereinfachten $PIDist$ Metrik auf den Garden Points Datensätzen. Auffällig ist das stark unterschiedlich Verhalten trotz der Ähnlichkeit der Datensätze und des gleichen Wertes für p.

PIDist vereinfacht, $p = 1$					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	3.0	13.27	11.0	67.77	72/200
Garden Points 2	2.0	10.33	3.0	14.60	117/200
Nordland	82.0	183.23	994.0	960.71	12/1000

Tabelle 8: *PIDist* vereinfacht, $p = 1$

PIDist vereinfacht, $p = 0.1$					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	85.5	83.98	16.0	23.58	31/200
Garden Points 2	2.0	16.94	3.0	13.19	123/200
Nordland	289.0	368.63	9.0	15.82	252/1000

Tabelle 9: *PIDist* vereinfacht, $p = 0.1$

Ursache hierfür wird im folgenden Kapitel erläutert. Die Beobachtungen hinsichtlich der Parameterwahl für p stimmen mit den von Aggarwal und Yu [AY00] präsentierten Ergebnissen überein, auch hier liefert $p = 1$ die besten Ergebnisse.

5.6 Angepasste PIDist-Ähnlichkeit

Die angepasste *PIDist*-Funktion ist in ihrer Funktionsweise mit einer Diskretisierung der Dimensionen vergleichbar, da lediglich Distanzen berücksichtigt werden, die innerhalb eines Bruchteils des Wertebereichs der Dimensionen liegen. Damit wird der Effekt der Nähe aus der originalen Funktion angenähert. Es muss für die Berücksichtigung für die Ähnlichkeitsfunktion also gelten:

$$| x_i - y_i | \leq (max_i - min_i) * t \tag{18}$$

t ist der gewählte Schwellwert, es gilt $0 < t \leq 1$.

Dafür wird der Code minimal angepasst, wie in Codeabschnitt 10 zu sehen ist.

```
def piDist_threshold_sequential(Xin, Yin, p, t):
    #initial similarity value is 0
    result_Matrix = np.zeros((Xin.shape[0], Yin.shape[0]), dtype=np.
    float32)
    #get highest value in each dimension
    highest_per_dim = np.amax(Xin, axis=0).toarray()[0]
    #get lowest value in each dimension except for 0
    lowest_per_dim = get_lowest_per_dimension(Xin)
    #compute ranges in all dimensions
    range_per_dim = highest_per_dim - lowest_per_dim
    #iterate over all points
    for i in range(Xin.shape[0]):
        matrix1 = Xin[i]
        for j in range(Yin.shape[0]):
            matrix2 = Yin[j]
```

```
            common_indices = get_common_indices(matrix1, matrix2)
            valuesX, valuesY = get_values_on_common_dimension(matrix1,
    matrix2)
            #compute all differences on common dimensions
            differences = np.abs(valuesX - valuesY)

            #filter the values so that only close pairings are considered
            close_indices = np.where(differences < (range_per_dim[
    common_indices]*t))

            #compute the similarity value
            c_i_close = common_indices[close_indices]
            d_close = differences[close_indices]
            similarity = (np.sum( (1- d_close / range_per_dim[c_i_close])
    **p ))** (1/p)
            result_Matrix[j][i] = similarity
    return result_Matrix
```

Codeausschnitt 10: Angepasster PIDist mit Schwellwert

Die Ergebnisse für ausgewählte Parameter sind in den Tabellen 10 und 11 zu sehen. Auch andere Parameterbelegungen wurden evaluiert, jedoch ohne eine signifikante Verbesserung des Ergebnisses zu erzielen.

PIDist angepasst, $p = 2$**,** $t = 0.5$					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	3.0	13.04	7.0	26.51	82/200
Garden Points 2	97.5	98.20	99.5	98.80	7/200
Nordland	37.5	149.81	6.0	72.88	450/1000

Tabelle 10: $PIDist$ angepasst, $p = 2$, $t = 0.5$

PIDist angepasst, $p = 1$**,** $t = 0.5$					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	3.0	13.04	6.0	25.96	84/200
Garden Points 2	96.0	97.54	99.5	98.80	7/200
Nordland	41.0	151.74	6.0	72.93	449/1000

Tabelle 11: $PIDist$ angepasst, $p = 1$, $t = 0.5$

Zur besseren Vergleichbarkeit werden in der Tabelle 12 die Ergebnisse der zwei verschiedenen Implementationen der PIDist-Metrik gegenübergestellt:

Im Vergleich fällt vor allem die Abhängigkeit der Ergebnisse von dem Datensatz auf. In beiden Fällen fällt jeder Wert für den zweiten Garden-Points Datensatz signifikant ab. Die Verbesserungen auf den beiden anderen Datensätzen sind teilweise sehr stark, jedoch sind die Ergebnisse auch dann trotzdem noch nicht zufriedenstellend.

Eine Mögliche Ursache für das diffuse Verhalten, das auch in der Abbildung 5 dieser beiden Varianten des IGRID-Index liegt in der Normalisierung der Distanzen auf den

PIDist vereinfacht, $p = 2$					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	32.0	46.80	7.5	17.69	60/200
Garden Points 2	2.0	15.89	4.0	15.39	109/200
Nordland	250.0	334.53	4.0	12.02	503/1000
PIDist angepasst, $p = 2$, $t = 0.5$					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	↑ 3.0	↑ 13.04	↗ 7.0	↘ 26.51	↑ 82/200
Garden Points 2	↓ 97.5	↓ 98.20	↓ 99.5	↓ 98.80	↓ 7/200
Nordland	↑ 37.5	↑ 149.81	↘ 6.0	↓ 72.88	↓ 450/1000

Tabelle 12: Vergleich der *PIDist* Varianten, $p = 2$

PIDist vereinfacht, $p = 1$					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	3.0	13.27	11.0	67.77	72/200
Garden Points 2	2.0	10.33	3.0	14.60	117/200
Nordland	82.0	183.23	994.0	960.71	12/1000
PIDist angepasst, $p = 1$, $t = 0.5$					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	→ 3.0	↗ 13.04	↑ 6.0	↑ 25.96	↗ 84/200
Garden Points 2	↓ 96.0	↓ 97.54	↓ 99.5	↓ 98.80	↓ 7/200
Nordland	↑ 41.0	↗ 151.74	↑ 6.0	↑ 72.93	↑ 449/1000

Tabelle 13: Vergleich der *PIDist* Varianten, $p = 1$

jeweiligen Dimensionen. Bei diesem Schritt wird die Distanz zwischen Ober- und Untergrenze gebildet um diese dann für die Normalisierung zu verwenden. In den Experimenten wird der Anwendungsfall so simuliert, dass die Anfragebilder erst dann vorliegen, wenn die entsprechende Ähnlichkeitsprüfung vorgenommen werden soll. Dadurch werden sie bei der Ermittlung der Extremwerte auf den Dimensionen nicht berücksichtigt. Liegt nun ein Eintrag außerhalb dieser Grenzen, hat die Normalisierung keinen sinnvollen Einfluss mehr auf das Ergebnis und das Verhalten der Werte wird unkalkulierbar. Dadurch hat auch die Subtraktion des Distanzwertes von 1 keinen gleichbleibenden Einfluss auf alle Summanden der Ähnlichkeitsfunktion. Dieses Problem tritt mit einer signifikant höheren Wahrscheinlichkeit in dünnbesetzten Vektorräumen auf, daher ist dieser Algorithmus, ob vereinfacht oder angepasst, nicht dazu geeignet, nächste Nachbarn in hochdimensionalen aber dünn besetzten Vektorräumen zu bestimmen.

5.7 Sign-Random-Projection Locality-Sensitive Hashing

Eine gänzlich andere Sichtweise auf dieses Problem liegt der Winkeldistanz zugrunde, die durch das *Sign-Random-Projection Locality-Sensitive Hashing* aus Kapitel 4.1.4 angenähert wird. Der Python-Code für diesen Algorithmus ist im Codeabschnitt 11 zu sehen.

```
from scipy.spatial import distance

def srp_lsh_sequential(Xin, Yin, num_rand_vectors):
  #initialize matrix with zeros
    result_Matrix = np.zeros((Xin.shape[0], Yin.shape[0]), dtype=np.
    float32)
    d = Xin.shape[1]
    mu = np.zeros(d)
    cov_matrix = np.eye(d)
    #create num_rand_vectors with mean 0 and an identity matrix as
    covariance matrix
    rand_vectors = np.random.multivariate_normal(mu, cov_matrix,
    num_rand_vectors)
    for i in range(Xin.shape[0]):
        vec1 = Xin[i].toarray()
        for j in range(Yin.shape[0]):
            #compute similarity
            vec2 = Yin[j].toarray()
            code1 = (np.dot(vec1, np.transpose(rand_vectors)) >= 0)[0]
            code2 = (np.dot(vec2, np.transpose(rand_vectors)) >= 0)[0]
            dist = distance.hamming(code1, code2)
            result_Matrix[j][i] = dist
    return result_Matrix
```

Codeausschnitt 11: SRP-LSH

Die Auswertung des SRP-LSH-Algorithmus mit $K = 100$ Zufallsvektoren ist in Tabelle 14 zu sehen.

Aufgrund der langen Laufzeit des Algorithmus für den Nordland-Datensatz (fast 16,5 Stunden für $K = 100$), wurden die anderen Belegungen für K nur für die Garden Points Datensätze getestet. Die Ergebnisse sind in der Tabelle 15 zu sehen. Diese zeigen klar, dass die Qualität mit steigender Menge der Zufallsvektoren steigt, dies stellt jedoch aufgrund der Laufzeit ein Problem dar.

Die hohe Laufzeit bei diesem Algorithmus ist vor allem auf die Erstellung der Zufallsvektoren zurückzuführen. Diese wächst sowohl mit der Anzahl K der gewählten Vektoren, solange $K < dim$, als auch mit der gewünschten Dimensionalität dim. Dabei ist gerade das Wachstum durch eine höhere Dimensionalität nichtlinear, wie aus den Tabellen 16 und 17 ersichtlich.

Die Dimensionalität ist durch den Datensatz vorgegeben, und da ein größerer Datensatz tendenziell auch eine höhere Dimensionalität bei den SEER-Deskriptoren erzeugt,

SRP-LSH, $K = 100$					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	9.0	32.42	18.0	43.53	55/200
Garden Points 2	18.5	39.1	31.5	48.78	51/200
Nordland	0.0	89.46	0.0	41.91	767/1000

Tabelle 14: Ergebnisse für SRP-LSH, $K = 100$

SRP-LSH, ausgewählte K					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
	SRP-LSH, $K = 360$				
Garden Points 1	3.0	19.44	6.0	26.63	91/200
Garden Points 2	2.0	14.72	5.0	32.87	92/200
	SRP-LSH, $K = 1080$				
Garden Points 1	2.0	12.56	4.0	21.62	103/200
Garden Points 2	2.0	11.11	4.0	24.31	111/200
	SRP-LSH, $K = 1260$				
Garden Points 1	2.0	12.40	3.5	18.95	112/200
Garden Points 2	2.0	9.72	3.0	24.37	111/200
	SRP-LSH, $K = 3600$				
Garden Points 1	2.0	11.25	3.0	15.07	117/200
Garden Points 2	2.0	10.2	3.0	22.57	113/200
	SRP-LSH, $K = 5400$				
Garden Points 1	2.0	11.19	3.0	16.57	118/200
Garden Points 2	1.5	8.21	3.0	16.41	115/200

Tabelle 15: Ergebnisse für SRP-LSH, verschiedene Werte für K

bedingt durch die benötigte Anzahl entsprechender Exemplare, ist für alle Anwendungsfälle eine hohe Laufzeit wahrscheinlich.

Laufzeit `numpy.random.multivariate_normal`		
Anzahl Vektoren	**Dimensionalität**	**Erstellungsdauer in Sekunden**
3600	1625	1.30
3600	3250	21.42
3600	6500	68.26
3600	13000	385.41

Tabelle 16: Laufzeit für die Erstellung der Zufallsvektoren

Da die Erstellung der Zufallsvektoren bereits durch eine Numpy-Funktion bewerkstelligt wird, und damit bereits die Laufzeit betreffend optimiert ist, ist dieser Algorithmus für das effiziente Lösen des VPR-Problems ungeeignet. Inwiefern sich das auf Basis des Python-Codes gewonnene Fazit auf Implementierungen in anderen Programmiersprachen übertragen lässt, muss in künftigen Untersuchungen evaluiert werden.

5.8 Cosinusähnlichkeit

Eine konkrete Bestimmung der Winkelähnlichkeit stellt, vor allem wegen der schlechten Laufzeit der ANnäherung, eine sinnvolle Alternative zur Ähnlichkeitsprüfung dar. Hierfür

Laufzeit `numpy.random.multivariate_normal`		
Anzahl Vektoren	**Dimensionalität**	**Erstellungsdauer in Sekunden**
1800	6500	61.78
3600	6500	68.26
5400	6500	86.73
7200	6500	79.23
9000	6500	74.51

Tabelle 17: Laufzeit für die Erstellung der Zufallsvektoren

wurde die in der Bibliothek `scikit-learn`[5] verfügbare Funktion `cosine_similarity` in unveränderter Fassung verwendet. Die damit erzielten Ergebnisse sind in der Tabelle 18 einzusehen.

Cosinusähnlichkeit					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	1.0	5.61	2.0	10.03	129/200
Garden Points 2	1.0	10.38	2.0	10.91	129/200
Nordland	0.0	9.18	0.0	4.73	978/1000

Tabelle 18: Cosinusähnlichkeit

Die Ergebnisse zeigen eine signifikante Verbesserung der Klassifikationsergebnisse im Vergleich zu den vorher präsentierten Algorithmen, die vor allem bei dem größeren Datensatz sichtbar wird. Bei den kleineren Datensätzen fallen die einzelnen Fehler, wie bereits bei anderen Methoden festgestellt, stärker ins Gewicht.

5.9 Anzahl der gemeinsam belegten Dimensionen

Der letzte hier getestete Algorithmus ist der in dieser Arbeit neu vorgestellte Ansatz aus Kapitel 4.2.3. Hier wird, wie bereits ausgeführt, die konkrete Belegung der Dimensionen vernachlässigt und nur die Anzahl der gemeinsam belegten Dimensionen als Ähnlichkeitsmaß genutzt.

Diese Methode nutzt zudem explizit die Dünnbesetztheit der Deskriptoren aus, da nur belegte Dimensionen in die Berechnung miteinbezogen werden.
Dies wird mit dem Code in Quellcode 12 erreicht:

```
1 def compute_matching_dimensions_sequential(Xin, Yin):
2         result_Matrix = np.zeros((Xin.shape[0], Yin.shape[0]), dtype=np.
    float32)
3         for i in range(Xin.shape[0]):
4             for j in range(Yin.shape[0]):
5                 #get number of matching dimensions
6                 matches = np.size(get_common_indices(Xin[i], Yin[j]))
```

[5] `https://scikit-learn.org/stable/`

```
7                   result_Matrix[j][i] = matches
8           return result_Matrix
```

Codeausschnitt 12: Anzahl der gemeinsamen Dimensionen

Die Ergebnisse sind in Tabelle 19 zu sehen.

Anzahl der Gemeinsamen Dimensionen					
Datensatz	d_{med}	d_{mean}	p_{med}	p_{mean}	$prop$
Garden Points 1	2.0	10.405	3.0	11.915	122/200
Garden Points 2	1.0	8.21	3.0	13.38	123/200
Nordland	0.0	28.608	0.0	6.556	949/1000

Tabelle 19: Anzahl der Gemeinsamen Dimensionen

Der beschriebene Ansatz führt zu signifikant besseren Ergebnissen als die bisher aufgeführten Algorithmen, ausgenommen die Cosinusähnlichkeit. An den Medianen d_{med} und p_{med} ist erkennbar, dass die meisten Indizes auf eine vergleichsweise gute Klassifikation schließen lassen. Gleichzeitig zeigen die verhältnismäßig hohen Mittelwerte d_{mean} und p_{mean}, dass auch einige Ausreißer vorliegen.

Diese Ergebnisse sind auch in der Ausgabe der Auswertungsfunktion in Abbildung 6 zu sehen.

Die Indexstruktur aus Kapitel 4.2.3.2 wurde mit dem Code aus Quellcode 13 implementiert. Zu sehen sind sowohl die Erstellung der Indexstruktur als auch die Benutzung derselben.

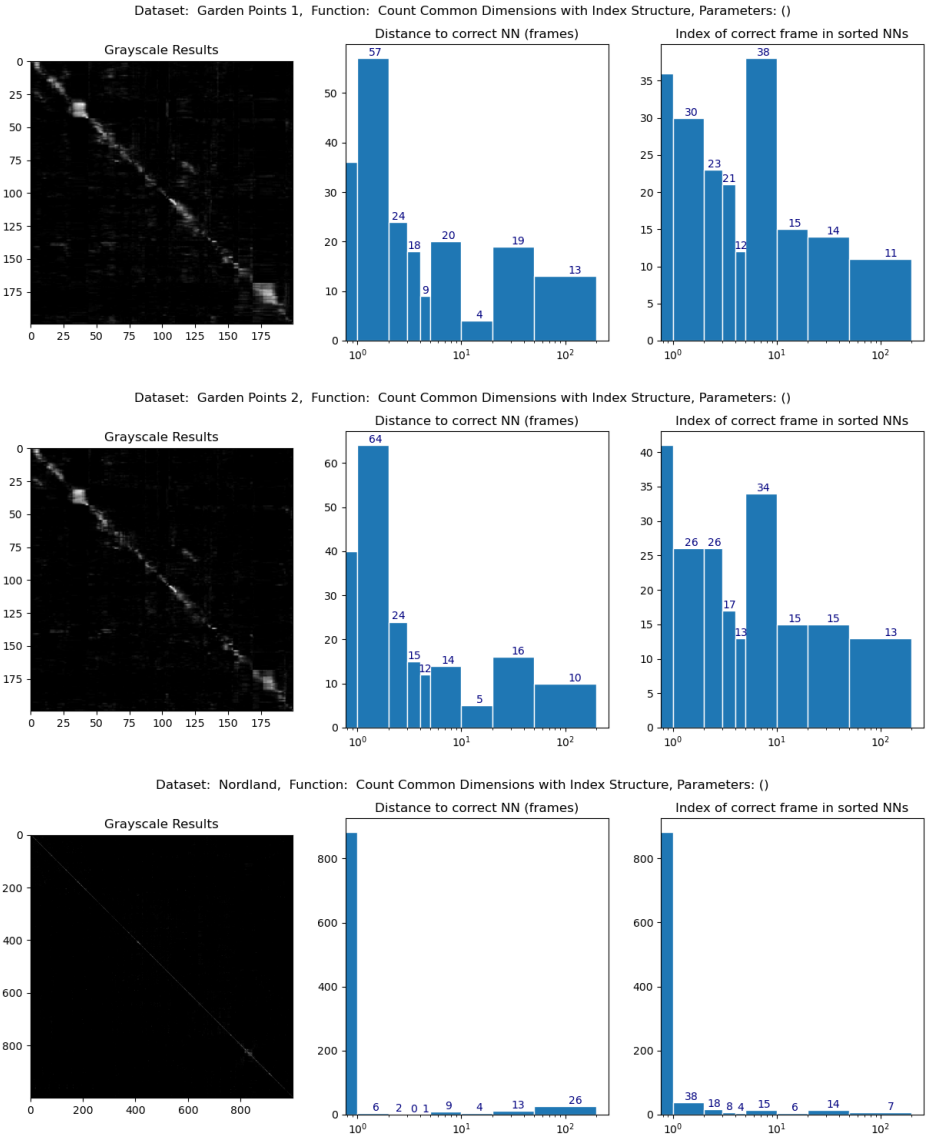


Abbildung 6: Visuelle Darstellung der Ergebnisse der Anzahl der gemeinsam belegten Dimensionen. Trotz der verschiedenen Skalen ist ersichtlich, dass die Performance auf dem Nordlanddatensatz die auf den Garden Points Datensätzen übertrifft.

```
def build_index_structure_matching_dims(Q_Data):
    indices_lists = [[] for _ in range(Q_Data.shape[1])]
    for i in range(Q_Data.shape[0]):
        vec = Q_Data[i]
        #get populated dimensions
        used_dims = vec.nonzero()[1]
        for dim in used_dims:
            indices_lists[dim].append(i)
    return indices_lists

def compute_matching_dimensions_w_index(Xin, Yin):
        result_Matrix = np.zeros((Xin.shape[0], Yin.shape[0]), dtype=np.
    float32)
        indices_lists = build_index_structure_matching_dims(Yin)
        #dtype=object for lists
        indices_array = np.array(indices_lists, dtype=object)
        for i in range(Xin.shape[0]):
            #get nonzero Elements of descriptor
            nonzero_dims = Xin[i].nonzero()[1]
            #get all candidate indices
            candidates_ind = indices_array[nonzero_dims]
            #combine lists and count occurences of every index
            combined_list = [item for sublist in candidates_ind for item
    in sublist]
            counter = Counter(combined_list)
            #write occurences to correct place
            for index, count in counter.items():
                result_Matrix[index][i] = count
        return result_Matrix
```

Codeausschnitt 13: Anzahl der gemeinsamen Dimensionen mit Indexstruktur

Die Verbesserung der Laufzeit ist aus Tabelle 21 zu entnehmen. Die Tatsache, dass damit die Laufzeit von ungefähr 1200 Sekunden auf 1,2 Sekunden bei dem Nordland-Datensatz reduziert werden konnte zeigt, dass diese Metrik durch die Indexstruktur in Echtzeit angewendet werden kann.

5.10 Laufzeitvergleich

Bei dem Laufzeitvergleich ist wichtig, dass beachtet wird, dass die in Python in diversen Bibliotheken verfügbaren Funktionen eine deutlich geringere Laufzeit benötigen als Funktionen die im Code programmiert werden müssen, wodurch die theoretische Komplexität nicht abgebildet werden kann.
Zudem können Laufzeiten durch verschiedene systembedingte Randbedingungen, wie z.B. Hardware- oder Softwarekonfigurationen, deutlich variieren. Dennoch bleiben die Größenordnungen der ermittelten Zahlen repräsentativ, und die Vergleichbarkeit der Ergebnisse ist gegeben.

Zunächst sollen der Vollständigkeit halber die Methoden ausgewertet werden, die nicht in der engeren Auswahl von geeigneten Algorithmen zur Lösung des NN-Problems

in hochdimensionalen, dünnbesetzten Vektorräumen verbleiben. Diese Auswertung ist in der Tabelle 20 zu sehen.

Laufzeit in Sekunden				
Metrik	**Parameter**	**GP1**	**GP2**	**NL**
Dist. gem. Dim	$k = 2$	5.26	5.25	705.74
	$k = 1$	5.24	5.25	705.52
	$k = 0.1$	5.25	5.18	708.40
PIDist vereinfacht	$p = 2$	7.13	8.49	762.50
	$p = 1$	7.21	8.51	760.45
	$p = 0.1$	7.21	8.52	761.72
PIDist, $t = 0.5$	$p = 2$	7.26	8.64	768.31
	$p = 1$	7.31	8.71	765.49
SRP-LSH	$k = 100$	101.98	127.37	59158.21
	$k = 360$	113.75	149.21	
	$k = 1080$	245.67	257.96	
	$k = 1260$	234.44	261.24	
	$k = 3600$	390.22	425.89	
	$k = 5400$	447.15	522.02	

Tabelle 20: Laufzeiten der ungeeigneten Algorithmen

Die Auswertung der Laufzeit derjenigen Algorithmen, welche die angestrebten Ergebnisse erzielten, ist in Tabelle 21 zu sehen.

Laufzeit in Sekunden			
Metrik	**GP1**	**GP2**	**NL**
Fraktionale Distanzen, $k = 0.1$	2.65	3.06	1456.70
Anzahl gemeinsamer Dimensionen	3.15	3.79	1285.22
Anz. gem. Dim. mit Index	0.03	0.02	1.21
Cosinusähnlichkeit	0.002	0.002	0.006

Tabelle 21: Laufzeiten der besseren Algorithmen

Es ist deutlich sichtbar, dass die Cosinusähnlichkeit in diesem Kontext die beste Wahl ist, wobei auch der neue Algorithmus durch die Indexstruktur eine zufriedenstellende Laufzeit aufweist.

5.11 Diskussion der Ergebnisse

Als Algorithmen mit guten Ergebnissen haben sich die Fraktionalen Distanzen mit $k = 0.1$, die Anzahl der gemeinsam belegten Dimensionen und die Cosinusähnlichkeit erwiesen.
Die vorher zum Teil geäußerten Annahmen für die Ursachen der Fehlschläge einiger Algorithmen sollen hier noch einmal aufgegriffen werden.

Als Schwäche der Distanz auf gemeinsam belegten Dimensionen wurde die falsche Handhabung der gemeinsam belegten Dimensionen identifiziert. Es ist dabei grundsätzlich erkennbar, dass die Algorithmen, die mit den konkreten Werten der Dimensionen in Kombination mit dem Nutzen der Dünnbesetztheit arbeiten, zum Beispiel durch das Ignorieren der unbesetzten Dimensionen, tendenziell schlechte Ergebnisse erzielen. Eine naheliegende Erklärung dafür ist die Beschaffenheit der SEER-Deskriptoren (vgl. Kapitel 3), wie im Folgenden erläutert wird:
Im Rahmen der Erstellung der SEER-Deskriptoren spielt der Zufallsfaktor eine wesentliche Rolle. Immer, wenn ein Eingabevektor in der Matrix M nicht ausreichend repräsentiert ist, werden neue Exemplare gesampelt. Dabei ist die Wahrscheinlichkeit dafür, dass ein Wert in das Exemplar gesampelt wird, für betragsmäßig große Einträge entsprechend größer, als für betragsmäßig kleine Werte. Am Ende werden außerdem viele Werte aus dem Skalarprodukt des Eingabevektors mit der Matrix M auf Null gesetzt bis auf die $\lambda * k$ größten. Somit kann nicht gesagt werden, dass der konkrete berechnete bzw. festgelete Wert eines Deskriptors einen konkreten Informationsgehalt aufweist. Vielmehr geht es um Größenordnungen, die eher lokal in einem Kontext stehen und nicht global. Daher spekuliert ein Ansatz, der sich sehr auf die konkreten Werte in den einzelnen Dimensionen fokussiert, auf ein Maß von Informationsgehalt, das diese Deskriptoren nicht liefern. Diese Herangehensweise nutzen auch die beiden PIDist-Varianten, die entsprechend schwache Ergebnisse liefern. Darüber hinaus wurde bezüglich der PIDist-Funktionen bereits ausgeführt, dass die Normalisierung, die von essentieller Bedeutung für das Ähnlichkeitsmaß ist, nicht auf die Weise funktioniert, wie für sinnvolle Ergebnisse nötig.

Einen Sonderfall in der Einordnung der Qualität der Algorithmen stellt der SRP-LSH-Algorithmus dar. Grundsätzlich werden die Ergebnisse mit zunehmender Anzahl von Zufallsvektoren immer besser und es steht durchaus zu vermuten, dass bei einer Anzahl, die der der Dimensionalität der Vektoren entspricht, gute Klassifikationsergebnisse zu erwarten sind. Jedoch erfüllt der Algorithmus aufgrund der langen Laufzeit nicht die Anforderungen an die Effizienz und ist somit im Kontext dieser Arbeit unbrauchbar.

Die Ähnlichkeitsmetriken bzw. Distanzfunktionen, die sich durch gute Ergebnisse hervorgetan haben, sind die Fraktionalen Distanzen mit $k = 0.1$ (sofern die Anzahl der besetzten Dimensionen stimmt), die Cosinusähnlichkeit und die Anzahl der gemeinsam besetzten Dimensionen.
Bei den Fraktionalen Distanzen werden die Ergebnisse der vorher schon zitierten Arbeit von Aggarwal et Al. [AHK01] genutzt, jedoch wird hier nur die hohe Dimensionalität berücksichtigt. Diese Tatsache ist wahrscheinlich die Ursache dafür, dass diese Metrik gute Ergebnisse erzielt, obwohl die sämtliche Werte in die Berechnung einfließen. Die große Anzahl der Null-Einträge ist in diesem Fall der Faktor mit dem größeren Informationsgehalt. Daher liefert diese Metrik zwar gute Ergebnisse, beantwortet aber nicht im engeren Sinne die gestellte Frage, da die Dünnbesetztheit der Daten nicht berücksichtigt, geschweige denn genutzt wird.
Wie schon in Kapitel 4.2.1 erwähnt, findet der Cosinusabstand bzw. die Cosinusähnlich-

keit bereits Anwendung in hochdimensionalen Vektorräumen, was auch durch die hier erzielten Ergebnisse gestützt wird. Ein großer Vorteil in der Python-Umgebung ist die Nutzung der sehr effizienten vorkompilierten Funktion. Diese benötigt sogar auf dem Nordland Datensatz nur 0.006 Sekunden für die Berechnung von $1000 \cdot 1000 = 1000000$ Vergleichen und liefert gleichzeitig sehr gute Ergebnisse. Auch SRP-LSH, eine Annäherung an den Winkelabstand, liefert mit aufsteigender Anzahl von gesampelten Zufallsvektoren zunehmend bessere Ergebnisse, wodurch ebenfalls gezeigt wird, dass der Winkelabstand im allgemeinen hochdimensionalen Fall eine verlässliche Metrik darstellt.

Die einzige Metrik, die die Dünnbesetztheit wirklich ausnutzt, ist die Anzahl der gemeinsam besetzten Dimensionen. Auch hier lässt sich das gute Ergebnis mit der Natur der SEER-Deskriptoren erklären. Da SEER-Deskriptoren die spezifischen Merkmale eines Datensatzes abbilden und, wie bereits vorher argumentiert, die konkreten Zahlenwerte keine Information im engeren semantischen Sinn beinhalten, reicht die Tatsache, dass die Dimension besetzt ist (das bedeutet, dass der Wert nach der Berechnung des Skalarproduktes groß genug war), bereits als Information aus. Dieser Algorithmus ist zwar nicht so schnell wie die Cosinusfunktion, jedoch ist er mathematisch gesehen weniger komplex. Daher müsste überprüft werden, ob dieser Ansatz in einer schnelleren Programmiersprache wie C oder C++ eine ähnliche oder sogar geringere Laufzeit benötigt, als die Cosinusfunktion. Ein Argument für diese Annahme stellt die immense Reduktion der Laufzeit bei Benutzung der Indexstruktur aus, bei der die Laufzeit, verglichen mit den anderen geprüften Algorithmen, sehr nah an der Laufzeit der Cosinusähnlichkeit liegt.

6 Fazit

6.1 Zusammenfassung

Das Ziel dieser Arbeit war es, einen Algorithmus zu finden, der in hochdimensionalen, dünn besetzten Vektorräumen ein gutes Ergebnis liefert und dabei ressourcensparend arbeitet. Dieses Ziel wurde insofern erreicht, als dass der Cosinusabstand als die Metrik identifiziert wurde, die sowohl gute Ergebnisse liefert, als auch eine sehr geringe Laufzeit benötigt. Darüber hinaus wurde die neue Metrik der Anzahl der gemeinsam besetzten Dimensionen als geeignete Möglichkeit für dieses Problem vorgestellt, bei der jedoch offen ist, ob sie die geringe Laufzeit der optimierten Cosinusähnlichkeit erreichen kann.

6.2 Übertragbarkeit der Ergebnisse

In dieser Arbeit wurde die nächste-Nachbarn-Suche nur auf SEER-Deskriptoren durchgeführt. Die spezielle Natur dieser Deskriptoren, die auf ihre Erstellung zurückzuführen ist, wurde ausgeführt. Die auf diesen Daten erzielten Ergebnisse sind folgendermaßen auf andere dünnbesetzte hochdimensionale Datensätze übertragbar:
Beinhaltet die belegte Dimension an sich schon mehr Information, als der Wert, mit dem

sie belegt wurde, lassen sich die Ergebnisse höchstwahrscheinlich uneingeschränkt übertragen.
Ist der konkrete Zahlenwert ebenfalls wichtig, ist es sehr wahrscheinlich, dass die Cosinusähnlichkeit gleich gute Ergebnisse erzielen wird, wie auf den SEER-Deskriptoren. Darüber hinaus werden wahrscheinlich auch mit den Fraktionalen Distanzen gute Ergebnisse hinsichtlich der Genauigkeit erzielt werden können.
Es muss festgehalten werden, dass die in dieser Arbeit neu vorgestellte Methode sehr wahrscheinlich nicht in jedem Fall auf hochdimensionalen, dünn besetzten Vektoren so gute Ergebnisse erzielen wird. Vielmehr ist die Qualität abhängig von der Grundcharakteristik des untersuchten Datensatzes. Ähnelt besagte Charakteristik der der SEER-Deskriptoren, sind ebenfalls gute Ergenisse erwartbar.

6.3 Ausblick

Diese Arbeit zeigt einige Punkte auf, die für weitere Forschungen von Relevanz sind:
Wie kann der SRP-LSH-Algorithmus beschleunigt werden und welche Qualität können die Ergebnisse dann erreichen?
Wie schnell kann die Metrik der gemeinsam besetzten Dimensionen arbeiten, wenn sie effizient programmiert und kompiliert wird?
Gibt es einen Normalisierungsfaktor für die Distanz auf den gemeinsam besetzten Dimensionen, der das Ergebnis so verbessert, dass der Aufwand einer effizienteren Programmierung lohnenswert wäre?
Außerdem wurden die *Outlier* in den als gut bewerteten Verfahren nicht weiter untersucht. Hier wäre eine eine Analyse lohnenswert, um herauszufinden, ob es Merkmale gibt, die diese Deskriptoren von den anderen unterscheiden. Eine kurze Untersuchung in diesem Bereich war ergebnislos, möglicherweise weil die Originalbilder nicht vorlagen, die Erklärungsansätze liefern könnten.

Garden Points 1																			
	Fraktionale Distanzen verschiedene k			Dist. auf gem. Dim. verschiedene k			PIDist vereinfacht verschiedene p			PIDist mit $t = 0.5$ versch. p		SRP-LSH verschiedene K						Anz. gem. Dim.	Cosinus-ähnlichkeit
Parameter	0.2	0.1	0.1 ohne erste 5 Bilder	2	1	0.1	2	1	0.1	2	1	100	360	1080	1260	3600	5400		
d_{med}	81.5	81.5	2.0	54.0	54.5	54.0	32.0	3.0	85.5	3.0	3.0	9.0	3.0	2.0	2.0	2.0	2.0	2.0	1.0
d_{mean}	83.66	83.88	9.35	60.57	61.22	60.54	46.80	13.27	83.98	13.04	13.04	32.42	19.44	12.56	12.40	11.25	11.19	10.41	5.61
p_{med}	6.0	6.0	3.0	74.5	77.0	79.0	7.5	11.0	16.0	7.0	6.0	18.0	6.0	4.0	3.5	3.0	3.0	3.0	2.0
p_{mean}	17.48	15.06	12.54	73.35	77.11	77.65	17.69	67.77	23.58	26.51	25.96	43.53	26.63	21.62	18.95	15.07	16.57	11.92	10.03
prop $n/200$	75	76	115/**195**	0	0	0	60	72	31	82	84	55	91	103	112	117	118	122	129

Tabelle 22: Alle Ergebnisse auf dem Garden Points 1 Datensatz

Garden Points 2																			
	Fraktionale Distanzen verschiedene k			Dist. auf gem. Dim. verschiedene k			PIDist vereinfacht verschiedene p			PIDist mit $t = 0.5$ versch. p		SRP-LSH verschiedene K						Anz. gem. Dim.	Cosinus-ähnlichkeit
Parameter	0.2	0.1	0.1 ohne erste 5 Bilder	2	1	0.1	2	1	0.1	2	1	100	360	1080	1260	3600	5400		
d_{med}	88.5	87.5	2.0	49.5	50.0	50.0	2.0	2.0	2.0	97.5	96.0	18.5	2.0	2.0	2.0	2.0	1.5	1.0	1.0
d_{mean}	89.55	88.46	9.1	59.76	60.41	59.32	15.89	10.33	16.94	98.20	97.54	39.2	14.72	11.11	9.72	10.2	8.21	8.21	10.38
p_{med}	5.0	5.0	3.0	54.5	58.0	57.0	4.0	3.0	3.0	99.5	99.5	31.5	5.0	4.0	3.0	3.0	3.0	3.0	2.0
p_{mean}	21.43	17.43	15.20	55.77	56.92	58.17	15.39	14.6	13.19	98.80	98.80	48.78	32.87	24.31	24.37	22.57	16.41	13.38	10.91
prop $n/200$	94	97	114/**195**	2	2	1	109	117	123	7	7	51	92	111	111	113	115	123	129

Tabelle 23: Alle Ergebnisse auf dem Garden Points 2 Datensatz

Nordland													
	Fraktionale Distanzen verschiedene k		Dist. auf gem. Dim. verschiedene k			PIDist vereinfacht verschiedene p			PIDist mit $t = 0.5$ versch. p		SRP-LSH verschiedene K	Anz. gem. Dim.	Cosinus-ähnlichkeit
Parameter	0.2	0.1	2	1	0.1	2	1	0.1	2	1	100		
d_{med}	0.0	0.0	247.0	245.0	240.5	250.0	82.0	289.0	37.5	41.0	0.0	0.0	0.0
d_{mean}	26.53	26.62	308.37	307.45	305.52	334.53	183.23	368.63	149.81	151.74	89.46	28.61	9.18
p_{med}	0.0	0.0	81.0	81.0	81.0	4.0	994.0	9.0	6.0	6.0	0.0	0.0	0.0
p_{mean}	8.86	7.35	87.89	87.91	87.70	12.02	960.71	15.82	72.88	72.93	41.91	6.57	4.73
prop $n/1000$	950	947	0	0	0	503	12	252	450	449	767	949	978

Tabelle 24: Alle Ergebnisse auf dem Nordland Datensatz

Abbildungsverzeichnis

Tabellenverzeichnis

Codeausschnitte

Literatur

[AHK01] Aggarwal, Charu C. ; Hinneburg, Alexander ; Keim, Daniel A.: On the surprising behavior of distance metrics in high dimensional space. In: *Database Theory—ICDT 2001: 8th International Conference London, UK, January 4–6, 2001 Proceedings 8* Springer, 2001, S. 420–434

[And09] Andoni, Alexandr: *Nearest neighbor search: the old, the new, and the impossible*, Massachusetts Institute of Technology, Diss., 2009

[Ass12] Assent, Ira: Clustering high dimensional data. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 2 (2012), Nr. 4, S. 340–350

[AY00] Aggarwal, Charu C. ; Yu, Philip S.: The IGrid index: reversing the dimensionality curse for similarity indexing in high dimensional space. In: *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2000, S. 119–129

[BGRS99] Beyer, Kevin ; Goldstein, Jonathan ; Ramakrishnan, Raghu ; Shaft, Uri: When is "nearest neighbor" meaningful? In: *Database Theory—ICDT'99: 7th International Conference Jerusalem, Israel, January 10–12, 1999 Proceedings 7* Springer, 1999, S. 217–235

[Cla06] Clarkson, Kenneth L.: Nearest-neighbor searching and metric space dimensions. (2006)

[FM06] Frome, Andrea ; Malik, Jitendra: Object recognition using locality sensitive hashing of shape contexts. (2006)

[GIM+99] Gionis, Aristides ; Indyk, Piotr ; Motwani, Rajeev u. a.: Similarity search in high dimensions via hashing. In: *Vldb* Bd. 99, 1999, S. 518–529

[GSM03] Georgescu ; Shimshoni ; Meer: Mean shift based clustering in high dimensions: A texture classification example. In: *Proceedings Ninth IEEE International Conference on Computer Vision* IEEE, 2003, S. 456–463

[GSW+22] Gong, Chaoyu ; Su, Zhi-Gang ; Wang, Pei-Hong ; Wang, Qian ; You, Yang: A sparse reconstructive evidential K-nearest neighbor classifier for high-dimensional data. In: *IEEE Transactions on Knowledge and Data Engineering* 35 (2022), Nr. 6, S. 5563–5576

[GW95] Goemans, Michel X. ; Williamson, David P.: Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. In: *Journal of the ACM (JACM)* 42 (1995), Nr. 6, S. 1115–1145

[HAK00] Hinneburg, Alexander ; Aggarwal, Charu C. ; Keim, Daniel A.: What is the nearest neighbor in high dimensional spaces? (2000)

[Ham50] Hamming, Richard W.: Error detecting and error correcting codes. In: *The Bell system technical journal* 29 (1950), Nr. 2, S. 147–160

[HKK01] Han, Eui-Hong ; Karypis, George ; Kumar, Vipin: Text categorization using weight adjusted k-nearest neighbor classification. In: *Advances in Knowledge Discovery and Data Mining: 5th Pacific-Asia Conference, PAKDD 2001 Hong Kong, China, April 16–18, 2001 Proceedings 5* Springer, 2001, S. 53–65

[IM98] Indyk, Piotr ; Motwani, Rajeev: Approximate nearest neighbors: towards removing the curse of dimensionality. In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998, S. 604–613

[JLY+12] Ji, Jianqiu ; Li, Jianmin ; Yan, Shuicheng ; Zhang, Bo ; Tian, Qi: Super-bit locality-sensitive hashing. In: *Advances in neural information processing systems* 25 (2012)

[LMGC06] Liu, Ting ; Moore, Andrew W. ; Gray, Alexander ; Cardie, Claire: New algorithms for efficient high-dimensional nonparametric classification. In: *Journal of Machine Learning Research* 7 (2006), Nr. 6

[LZX+18] Luo, Chunjie ; Zhan, Jianfeng ; Xue, Xiaohe ; Wang, Lei ; Ren, Rui ; Yang, Qiang: Cosine normalization: Using cosine similarity instead of dot product in neural networks. In: *Artificial Neural Networks and Machine Learning–ICANN 2018: 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4-7, 2018, Proceedings, Part I 27* Springer, 2018, S. 382–391

[MGW+17] Ma, Hongxing ; Gou, Jianping ; Wang, Xili ; Ke, Jia ; Zeng, Shaoning: Sparse coefficient-based k-nearest neighbor classification. In: *IEEE access* 5 (2017), S. 16618–16634

[NS22] Neubert, Peer ; Schubert, Stefan: SEER: Unsupervised and sample-efficient environment specialization of image descriptors. In: *Robotics: Science and Systems (R: SS)* (2022)

[Orc91] Orchard, Michael T.: A fast nearest-neighbor search algorithm. In: *Acoustics, Speech, and Signal Processing, IEEE International Conference on* IEEE Computer Society, 1991, S. 2297–2298

[Rui86] Ruiz, Enrique V.: An algorithm for finding nearest neighbours in (approximately) constant average time. In: *Pattern Recognition Letters* 4 (1986), Nr. 3, S. 145–157

[SEK04] Steinbach, Michael ; Ertöz, Levent ; Kumar, Vipin: The challenges of clustering high dimensional data. In: *New directions in statistical physics: econophysics, bioinformatics, and pattern recognition*. Springer, 2004, S. 273–309

[SNP13] Sünderhauf, Niko ; Neubert, Peer ; Protzel, Peter: Are we there yet? Challenging SeqSLAM on a 3000 km journey across all four seasons. In: *Proc. of workshop on long-term autonomy, IEEE international conference on robotics and automation (ICRA)* Citeseer, 2013, S. 2013

[TDVS19] Taunk, Kashvi ; De, Sanjukta ; Verma, Srishti ; Swetapadma, Aleena: A brief review of nearest neighbor algorithm for learning and classification. In: *2019 international conference on intelligent computing and control systems (ICCS)* IEEE, 2019, S. 1255–1260

[TYSK09] Tao, Yufei ; Yi, Ke ; Sheng, Cheng ; Kalnis, Panos: Quality and efficiency in high dimensional nearest neighbor search. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, 2009, S. 563–576

[VDS06] Vijayakumar, Sethu ; D'Souza, Aaron ; Schaal, Stefan: Approximate nearest neighbor regression in very high dimensions. (2006)

[Vid94] Vidal, Enrique: New formulation and improvements of the nearest-neighbour approximating and eliminating search algorithm (AESA). In: *Pattern Recognition Letters* 15 (1994), Nr. 1, S. 1–7

[WD93] Wettschereck, Dietrich ; Dietterich, Thomas: Locally adaptive nearest neighbor algorithms. In: *Advances in Neural Information Processing Systems* 6 (1993)