T2000

Jan Herrmann

10. Dezember 2025

Inhaltsverzeichnis

Abkürzungsverzeichnis

ANNS Approximierende Nearest-Neighbor-Suche. 9

KI Künstliche Intelligenz. 5

ML Maschinelles Lernen. 5

Vektor-Store Leichtgewichtiges System zur Speicherung von Embeddings und Ausführung semantischer Ähnlichkeitssuchen. 8

Vektordatenbank Datenbanksystem für Embeddings mit Mechanismen wie Sharding, Partitionierung, Caching und Replikation. 8

1 Einleitung

1.1 Motivation

1.2 Problemstellung

1.3 Zielsetzung

2 Theoretischer Hintergrund

2.1 Künstliche Intelligenz

Bevor eine präzise Definition von künstlicher Intelligenz möglich ist, muss geklärt werden, welches Ziel ein intelligentes System verfolgen soll. Russell und Norvig zeigen, dass gängige KI-Definitionen in der wissenschaftlichen Literatur entlang zweier zentraler Dimensionen variieren (vgl. [1], Kap. 1.1):

- **Mensch vs. Rationalität** Soll ein System wie ein Mensch denken oder handeln, oder soll es unabhängig vom Menschen ideal rational agieren?
- **Denken vs. Handeln** Soll Intelligenz anhand interner Denkprozesse oder anhand des beobachtbaren Verhaltens beurteilt werden?

Aus diesen beiden Dimensionen ergeben sich vier grundlegende Perspektiven auf KI, die unterschiedliche historische Forschungsrichtungen geprägt haben. Eine Übersicht dieser Einordnung zeigt Tabelle 1

Kategorie	**Beschreibung**
Systeme, die wie Menschen denken	Fokus auf Nachbildung menschlicher Denkprozesse, z.B. durch kognitive Modelle oder psychologische Theorien.
Systeme, die wie Menschen handeln	Intelligenz wird anhand menschlich ähnlichen Verhaltens beurteilt, unabhängig vom zugrunde liegenden Denkprozess.
Systeme, die rational denken	Fokus auf logische Schlussfolgerungen und formale Wissensrepräsentation.
Systeme, die rational handeln	Intelligente Agenten handeln zielgerichtet und optimal in ihrer Umgebung.

Tabelle 1: Eigene Darstellung in Anlehnung an [1], Kap. 1.1

Abgrenzung von KI und ML

Künstliche Intelligenz (KI) umfasst alle Verfahren, die darauf abzielen, intelligentes Verhalten technisch zu realisieren. Dazu gehören sowohl symbolische Ansätze wie Wissensrepräsentation und logisches Schließen als auch datengetriebene Methoden zur Wahrnehmung oder Sprachverarbeitung (vgl. [1], Kap. 1.1).

Maschinelles Lernen (ML) stellt ein klar abgegrenztes Teilgebiet der KI dar. Russell und Norvig beschreiben es als das Teilfeld der künstlichen Intelligenz, das sich mit Programmen befasst, die aus Erfahrung lernen (vgl. [1], Einleitung zu Teil VI).

Während KI somit als Oberbegriff sämtliche Methoden intelligenter Problemlösung einschließt, konzentriert sich ML ausschließlich auf Verfahren, die Wissen nicht explizit vorgegeben bekommen, sondern selbstständig aus Daten oder Erfahrungen

erschließen. ML bildet damit die Grundlage für viele moderne KI-Anwendungen, insbesondere für datengetriebene Systeme wie neuronale Netze oder Large Language Models.

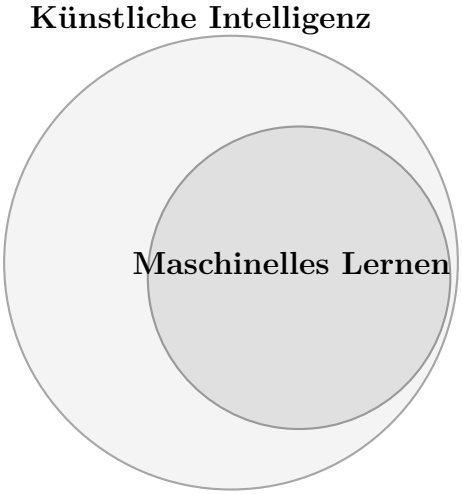

Abbildung 1: Einordnung von Maschinellem Lernen als Teilgebiete der Künstlichen Intelligenz

2.2 Machine Learning

Für die in dieser Arbeit betrachtete Anwendung ist Machine Learning insofern von zentraler Bedeutung, als dass es die Grundlage für die automatisierte Verarbeitung und semantische Strukturierung von Text- und Produktdaten bildet. Im Folgenden werden zunächst konventionelle Verfahren des Machine Learning vorgestellt, bevor anschließend das Representation Learning und das Deep Learning als zunehmend automatisierte und leistungsfähigere Formen der Merkmalsextraktion betrachtet werden. Abbildung 2 veranschaulicht diese Einordnung innerhalb des Gesamtfeldes.

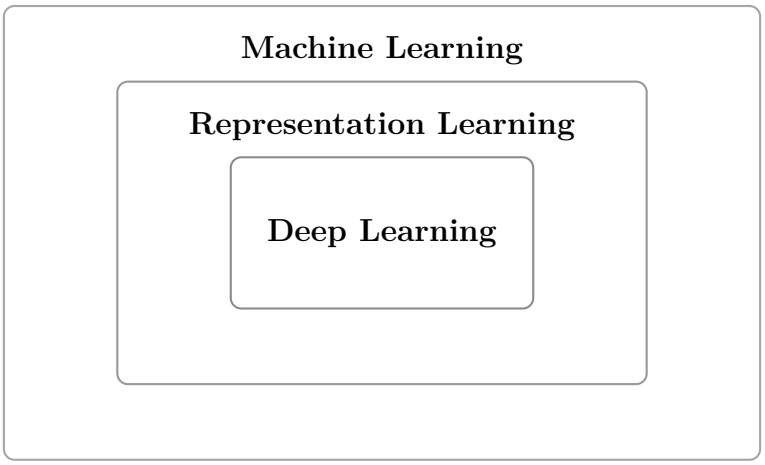

Abbildung 2: Einordnung von Representation Learning und Deep Learning als Teilgebiete des Machine Learning

2.2.1 Konventionelles Machine Learning (ML)

Nach der klassischen Definition von Mitchell gilt ein Programm als lernfähig, wenn *"a computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E"* [2, S. 2]. Konventionelles maschinelles Lernen umfasst somit Verfahren, die statistische Muster in Daten erkennen und auf dieser Basis Vorhersagen oder Entscheidungen treffen können.

LeCun, Bengio und Hinton ordnen maschinelles Lernen als zentralen Bestandteil moderner digitaler Systeme ein. ML-Methoden werden unter anderem zur Objekterkennung in Bildern, zur Umwandlung von Sprache in Text oder zur Relevanzsortierung von Inhalten eingesetzt [3, S. 436f.]. Diese Anwendungen verdeutlichen den praktischen Nutzen konventioneller ML-Verfahren in einer datengetriebenen Gesellschaft.

Ein wesentlicher Engpass konventioneller ML-Verfahren besteht darin, dass Modelle Rohdaten wie Pixelwerte, Audiosignale oder Text nicht direkt verarbeiten können. Stattdessen müssen die relevanten Merkmale zunächst manuell aus den Daten extrahiert werden. Dieser arbeitsintensive Prozess, bei dem Expertinnen und Experten geeignete Eingaberepräsentationen gestalten, wird als *Feature Engineering* bezeichnet.

Da die Modellleistung maßgeblich von der Qualität dieser handentwickelten Merkmale abhängt, stellte dieser Ansatz über Jahrzehnte hinweg eine zentrale Begrenzung klassischer ML-Systeme dar [3, S. 436f.].

Diese Einschränkung bildet den Ausgangspunkt für das Representation Learning, das darauf abzielt, Merkmale automatisch aus Rohdaten zu lernen und die Abhängigkeit vom Feature Engineering zu reduzieren.

2.2.2 Representation Learning (RL)

Representation Learning umfasst Methoden, die es ermöglichen, aus Rohdaten automatisch diejenigen Merkmale (Features) zu extrahieren, die für eine bestimmte Aufgabe relevant sind. Im Gegensatz zum konventionellen Machine Learning entfällt dabei die Notwendigkeit, Merkmalsextraktoren manuell zu definieren. Stattdessen lernt das Modell eigenständig geeignete interne Repräsentationen, die zentrale Informationen betonen und irrelevante Variationen unterdrücken [3, S. 436].

Ein Teil der Leistungsfähigkeit moderner KI-Systeme beruht darauf, dass diese Repräsentationen nicht nur auf einer Ebene entstehen, sondern schrittweise über mehrere Transformationen hinweg verfeinert werden können. Verfahren, die solche hierarchischen Merkmalsräume durch mehrere nichtlineare Schichten lernen, werden unter dem Begriff Deep Learning zusammengefasst.

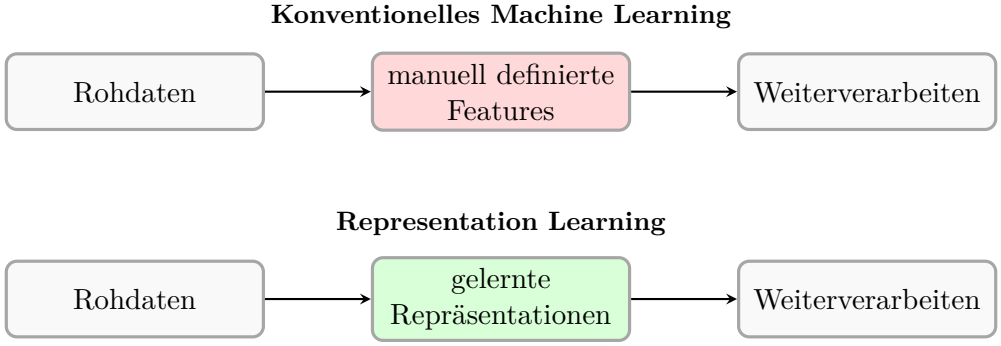

Abbildung 3: Gegenüberstellung von konventionellem Machine Learning und Representation Learning. In klassischen Verfahren (rot) müssen relevante Merkmale aus den Rohdaten manuell spezifiziert werden, bevor ein Modell sie weiterverarbeiten kann. Representation Learning (grün) ersetzt diesen manuellen Schritt durch ein lernfähiges Abbildungsmodell, das aus den Rohdaten eigenständig geeignete interne Repräsentationen ableitet. Dadurch entstehen schrittweise verfeinerte, hierarchische Merkmalsräume, wie sie im vorangegangenen Text beschrieben wurden und die die Grundlage moderner Deep-Learning-Methoden bilden.

2.2.3 Deep Learning (DL)

Deep Learning stellt eine mehrschichtige Ausprägung des Representation Learning dar, bei der die Repräsentationen der Features nicht nur auf einer Ebene entstehen,

sondern über zahlreiche hintereinandergeschaltete nichtlineare Transformationen hinweg sukzessive abstrahiert und verfeinert werden. Jede dieser Transformationen bildet die Ausgabe einer Schicht in eine neue Repräsentation um, die etwas abstrakter ist als die vorherige. Werden genügend solcher Schichten kombiniert, können tiefe neuronale Netze auch hochkomplexe Funktionen modellieren und Strukturen in den Daten erfassen, die mit flachen Modellen nicht zugänglich wären (vgl.[3, S. 436 f.]).

Ein zentraler Vorteil tiefer Architekturen besteht darin, dass höherliegende Repräsentationsebenen diejenigen Aspekte der Eingangsdaten verstärken, die für die jeweilige Aufgabe bedeutsam sind, während störende oder irrelevante Variationen unterdrückt werden. Besonders anschaulich zeigt sich dieses Prinzip in der Bildverarbeitung: Obwohl ein Bild lediglich als Anordnung von Pixelwerten vorliegt,lernen die ersten Schichten eines tiefen Netzes meist einfache Merkmale wie Kanten in unterschiedlichen Orientierungen und Positionen. Die folgenden Schichten erkennen Kombinationen dieser Kanten, etwa Texturen oder kleine Motive. Noch höhere Ebenen fassen diese Motive zu komplexeren Strukturen wie Objektteilen zusammen, aus denen schließlich vollständige Objekte oder abstrakte Konzepte abgeleitet werden können. Der entscheidende Punkt besteht darin, dass keine dieser Merkmalsstufen manuell spezifiziert wird: Die Merkmale und ihre Hierarchie entstehen vollständig durch den Lernprozess selbst, basierend auf einem allgemeinen Optimierungsverfahren (vgl.[3, S. 436 f.]).

Diese Fähigkeit, ausgehend von Rohdaten eine gestufte, zunehmend abstrakte Repräsentation zu lernen, bildet den Kern des Deep-Learning-Paradigmas und erklärt, warum tiefe neuronale Netze in Bereichen wie Computer Vision, Sprachverarbeitung oder allgemeinen Mustererkennungsaufgaben so leistungsfähig sind.

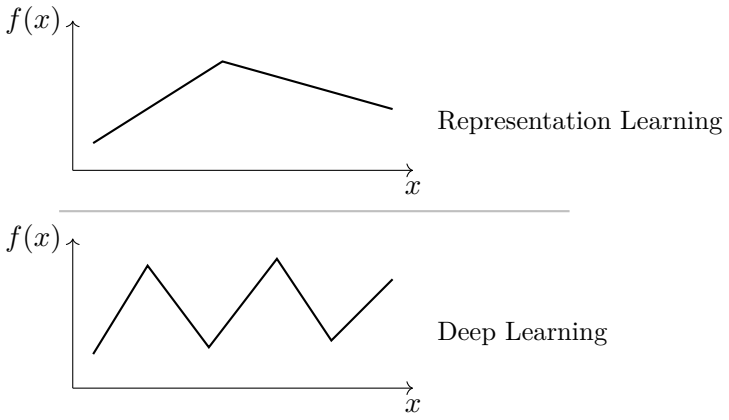

Abbildung 4: Schematische Gegenüberstellung der durch ein flaches Modell (oben) und ein tiefes neuronales Netz (unten) darstellbaren Funktionen. Während ein flaches Modell nur relativ einfache, schwach geknickte Abbildungen $f(x)$ realisieren kann, erlauben tiefe Netze durch die Hintereinanderschaltung mehrerer nichtlinearer Schichten stark nichtlineare, „um mehrere Ecken gehende“ Funktionen.

2.3 Large Language Models Transformer

1. 2.3 LLM vollständig schreiben

Du brauchst hier: Transformer-Grundlagen Tokenisierung Embedding Layer Aufmerksamkeit kurzer Hinweis: LLM erzeugt Embeddings → Grundlage für 2.4 Nur ca. 1 Seite.

Am Ende von 2.3 kurz Embeddings erklären Damit 2.4 nicht unmotiviert Embeddings voraussetzt

2.4 Vektor Store

Vektor-Stores bilden die Grundlage für moderne KI-Anwendungen, die auf semantischer Ähnlichkeitssuche basieren. Sie dienen der Speicherung von Embeddings, also numerischen Repräsentationen von Text- oder Produktdaten, und ermöglichen effiziente Nearest-Neighbor-Abfragen. Damit stellen sie einen zentralen Baustein für Retrieval-Augmented Generation (RAG) und für Systeme dar, die kontextbezogene Informationen an große Sprachmodelle übergeben (vgl. Abschnitt **??**).

Da der Begriff „Vektorstore“ in der Literatur häufig als Sammelbezeichnung für leichtgewichtige, embeddingbasierte Speichersysteme verwendet wird, ist eine Abgrenzung zu vollwertigen Vektordatenbanken notwendig. Letztere integrieren zusätzliche Mechanismen wie Sharding, interne Partitionierung, Caching oder Replikation und unterscheiden sich damit deutlich in ihrer Komplexität und Skalierbarkeit.

Im Folgenden werden die in [4] vorgestellten Speicher- und Suchmechanismen von Vektordatenbanken zusammengefasst und auf das Anwendungsszenario dieser Arbeit übertragen.

Tabelle 2: Abgrenzung von Vektor-Store und Vektordatenbank

Merkmal	**Vektor-Store**	**Vektordatenbank**
Speicherung von Embeddings	✓	✓
k-nächste-Nachbarn-Suche (k-NN)	✓	✓
Sharding über mehrere Knoten	✗	✓
Interne Partitionierung innerhalb eines Knotens	✗	✓
Caching-Mechanismen	✗	✓
Replikation	✗	✓
Approximate Nearest Neighbor (ANNS)-Indexe	✗	✓
Metadatenverwaltung und Konsistenzmodelle	✗	✓

2.4.1 Abgrenzung Vektor-Store und Vektor-Datenbank

Der Begriff Vektor-Store wird in der Literatur nicht einheitlich verwendet und dient häufig als Sammelbezeichnung für leichtgewichtige Systeme, die Embeddings speichern und eine grundlegende semantische Ähnlichkeitssuche bereitstellen. Solche Systeme konzentrieren sich in der Regel auf das Einfügen von Embeddings und die Ausführung von k-nächste-Nachbarn-Abfragen, ohne jedoch erweiterte Datenbankfunktionen bereitzustellen.

Unter einer Vektordatenbank werden hingegen vollwertige Datenbanksysteme verstanden, die neben der Speicherung und Suche von Embeddings zusätzliche Verwaltungs- und Infrastrukturmechanismen integrieren. Dazu zählen insbesondere Sharding, interne Partitionierung, Caching, Replikation, sowie Indexstrukturen für Approximate-

Nearest-Neighbor-Search (ANNS). Diese Systeme sind auf hohe Skalierbarkeit, Robustheit und Performanz im produktiven Einsatz ausgelegt.

Für die vorliegende Arbeit wird daher folgende Unterscheidung getroffen:

- **Vektor-Store:** leichtgewichtiges System zur Speicherung von Embeddings und zur Durchführung semantischer Ähnlichkeitssuchen.
- **Vektordatenbank:** vollständiges Datenbankmanagementsystem mit skalierbaren Speicher- und Verwaltungsmechanismen.

Diese definitorische Abgrenzung dient der Klarheit und legt die einheitliche Verwendung der Begriffe im weiteren Verlauf der Arbeit fest.

2.4.2 Speicherverfahren für Vektordatenbanken

Die im Folgenden beschriebenen Mechanismen stellen zentrale Bestandteile moderner, skalierbarer Vektordatenbanksysteme dar. Sie werden im produktiven Umfeld zur Optimierung von Leistung, Verfügbarkeit und Robustheit eingesetzt. Für den im Rahmen dieser Arbeit entwickelten Prototypen spielen diese Konzepte jedoch keine operative Rolle, da ein leichtgewichtiges Single-Node-System ohne verteilte Architektur eingesetzt wird. Die Ausführungen dienen daher der theoretischen Einordnung und sollen den technologischen Kontext verdeutlichen, in dem sich Vektordatenbanken typischerweise bewegen.

horizontale Datenpartitionierung über mehrere Maschinen (auch Sharding genannt) bezeichnet ein Verfahren, bei dem eine Datenbank in mehrere logisch getrennte und auf verschiedene physische Knoten verteilte Teilmengen („Shards“) aufgeteilt wird. Durch diese Aufteilung wird das Gesamtdatenset in kleinere, handhabbarere Einheiten zerlegt, was Skalierbarkeit, Lastverteilung und die Verwaltung großer Datenmengen erleichtert (vgl. [4, S. 3f]).

horizontale Datenpartitionierung innerhalb einer Maschine die horizontale Datenpartitionierung innerhalb einer Maschine bezeichnet die Aufteilung der in einer einzelnen Datenbankinstanz gespeicherten Vektordaten in mehrere logisch getrennte Teilmengen („Partitionen“). Im Unterschied zum Sharding, das Daten über mehrere physische Knoten verteilt, erfolgt diese Form der Partitionierung ausschließlich innerhalb eines Systems. Ziel ist es, lokale Abfragen effizienter auszuführen, Speicherressourcen besser auszunutzen und parallele Verarbeitung zu ermöglichen. Partitionen können beispielsweise über Wertebereiche (Range-Partitioning), vordefinierte Kategorien (List-Partitioning) oder Hash-Verfahren gebildet werden. Durch diese interne Strukturierung müssen Suchanfragen nur gegen relevante Partitionen ausgeführt werden, was insbesondere bei großen Einbettungsräumen die Latenz der semantischen Ähnlichkeitssuche reduziert (vgl. [4, S. 3f.]).

In modernen Vektordatenbanken wird Partitionierung häufig mit Sharding kombiniert (vgl. Abbildung 5).

Während Sharding die Skalierung über mehrere Knoten ermöglicht, optimiert die interne Partitionierung die Datenorganisation innerhalb jedes Shards. Beide Verfahren bilden damit die Grundlage für performante Retrieval-Systeme in Vektor Stores und Vektordatenbanken.

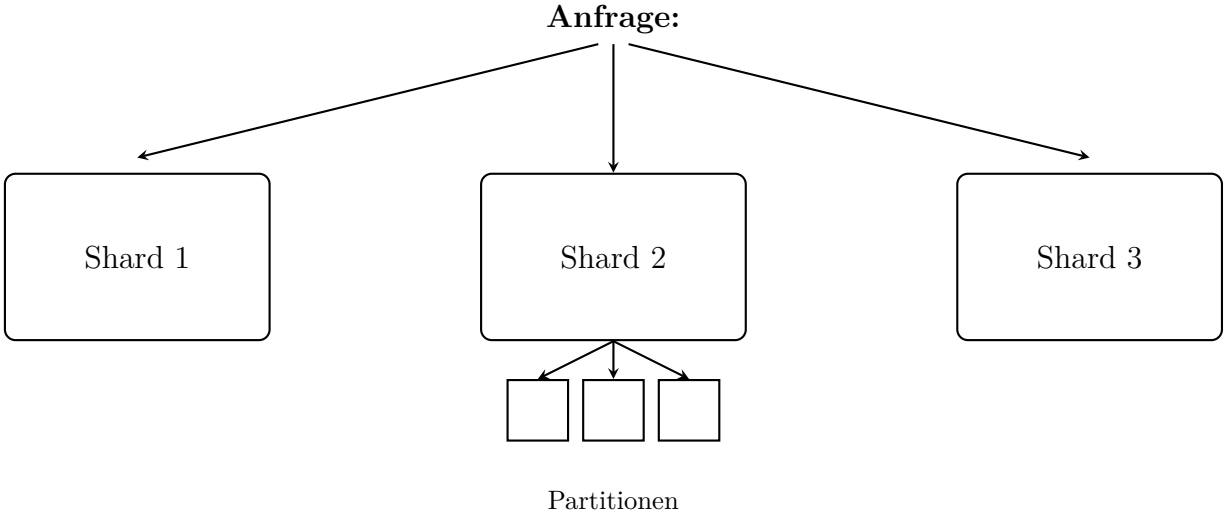

Abbildung 5: Sharding über mehrere Maschinen und interne Partitionierung innerhalb eines Shards

Caching-Mechanismen in Vektordatenbanken beschreiben das Zwischenspeichern häufig oder kürzlich genutzter Daten in besonders schnellen Speichermedien (z.B. RAM), um Zugriffszeiten zu reduzieren und die Last auf der eigentlichen Datenbank zu verringern. Für Vektordatenbanken ist Caching ein zentraler Mechanismus, da semantische Ähnlichkeitssuchen typischerweise rechenintensiv sind und von wiederholten Abfragen profitieren (vgl. [4, S. 4f.]).

Im Gegensatz zu klassischen Schlüssel-Wert-Caches (etwa Redis) ist das Caching in VDBs herausfordernder, da Embeddings hochdimensionale Vektoren darstellen und identische Anfragen selten auftreten. Daher kommen allgemeine Cache-Strategien zum Einsatz, die unabhängig vom genauen Vektorinhalt arbeiten, da der Vergleich hochdimensionaler Vektoren zur Bestimmung von Cache-Schlüsseln ungeeignet ist.

Zu den gängigen Verfahren zählen:

- **First-In First-Out (FIFO):** entfernt das älteste Element im Cache; einfach, aber ohne Berücksichtigung der Zugriffshäufigkeit.
- **Least Recently Used (LRU):** löscht den am längsten nicht genutzten Eintrag; gut geeignet für Arbeitslasten mit zeitlicher Lokalität.
- **Most Recently Used (MRU):** entfernt das zuletzt genutzte Element; sinnvoll bei einmaligen Zugriffsmustern.
- **Least Frequently Used (LFU):** bevorzugt das Entfernen seltener genutzter Einträge; vorteilhaft bei stabilen Zugriffshäufigkeiten.

Einige Systeme nutzen zudem **partitioniertes Caching**, bei dem Vektordaten in Gruppen (z.B. nach Kategorien oder Zugriffsmustern) getrennt gecacht werden, um Ressourcen gezielt zu optimieren. Insgesamt trägt Caching wesentlich zur Reduktion der Abfragelatenz und zur Stabilisierung der Systemlast bei, insbesondere bei wiederkehrenden Ähnlichkeitsanfragen.

Replikation bezeichnet das Anlegen und Verteilen mehrerer Kopien von Vektordaten auf unterschiedliche Knoten eines verteilten Systems, um Ausfallsicherheit, Verfügbarkeit und Leselastverteilung zu erhöhen. Während sie für die semantische Ähnlichkeitssuche nicht unmittelbar leistungsbestimmend ist, stellt sie einen zentralen Mechanismus für die betriebliche Robustheit moderner Vektordatenbanken dar (vgl. [4, S. 5f.]).

Insgesamt trägt Replikation wesentlich zur Robustheit und Verfügbarkeit verteilter Vektordatenbanksysteme bei, steht jedoch weniger im direkten Zentrum der Ähnlichkeitssuche als vielmehr ihrer infrastrukturellen Betriebsstabilität.

2.4.3 Suchverfahren

Diese Arbeit befasst sich mit zwei Kategorien von Suchverfahren, deren grundlegende Funktionsweise in Abbildung 6 schematisch gegenübergestellt ist. Die Darstellung der approximierenden Suche stellt hierbei ein vereinfachtes, heuristisches Beispiel dar: Es illustriert das Prinzip, dass ANN-Verfahren den Suchraum gezielt einschränken, ohne jeden Punkt im Vektorraum zu prüfen. In der Praxis existieren verschiedene ANN-Algorithmen (z.B. HNSW, IVF, LSH), die dieses Prinzip auf unterschiedliche Weise umsetzen. Die Abbildung dient somit der didaktischen Veranschaulichung des Grundkonzepts.

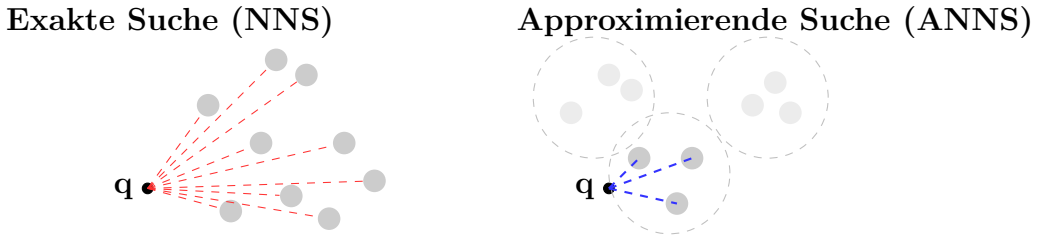

Abbildung 6: Vergleich zwischen exakter Nearest-Neighbor-Suche (NNS) und approximierender Suche (ANN). Bei exakter Suche berechnet das System die Distanz zwischen der Anfrage und *jedem* Punkt im Raum (rote Linien). ANN-Verfahren durchsuchen hingegen nur relevante Regionen des Suchraums (blauer Pfad).

Exakte Nearest-Neighbor-Suche (NNS) bezeichnet das Verfahren, für einen gegebenen Anfragevektor denjenigen Vektor in einer Menge zu bestimmen, der gemäß einem definierten Distanzmaß (z.B. euklidische Distanz) am nächsten liegt. In einfachster Form erfolgt dies durch eine lineare Suche, bei der für jedes gespeicherte Embedding die Distanz zum Anfragevektor berechnet und anschließend der Vektor mit dem geringsten Abstand ausgewählt wird (vgl. [4, S. 6f.]).

Da die Laufzeit dieses Ansatzes linear (O(n)) mit der Anzahl der gespeicherten

Embeddings wächst, stoßen Systeme bei großen Datenmengen schnell an ihre Leistungsgrenzen. In solchen Fällen können approximierende Verfahren eingesetzt werden. Die Grundidee besteht darin, eine Lösung zu liefern, die zwar nicht exakt optimal ist, dafür jedoch eine deutlich bessere Laufzeit und geringeren Speicherbedarf ermöglicht. Die leichte Abweichung vom exakten Ergebnis wird somit bewusst zugunsten einer höheren Effizienz in Kauf genommen.

Approximierende Nearest-Neighbor-Suche (ANNS) verzichtet auf eine vollständige Durchmusterung aller Embeddings und nutzt stattdessen probabilistische oder heuristische Verfahren, um den Suchraum gezielt einzugrenzen. Heuristische Verfahren nutzen vereinfachte Entscheidungsregeln, die eine schnelle, aber nicht zwingend optimale Abschätzung erlauben. Ziel ist es, möglichst schnell einen Vektor zu finden, der dem Anfragevektor sehr ähnlich ist, ohne zwingend den exakt nächsten Nachbarn bestimmen zu müssen. Dadurch lassen sich deutliche Laufzeitgewinne erzielen, insbesondere bei großen Datenmengen (vgl. [4, S. 6f.]).

Während exakte Verfahren insbesondere in kleinen Vektor-Stores mit geringem Datenvolumen eingesetzt werden können, sind approximierende Verfahren bei skalierbaren Vektordatenbanken mit einer großen Anzahl von Embeddings sinnvoll.

2.5 Schnittstellentechnologie

2.6 Retrieval-Augmented Generation

2.7 prompt Engineering

3 Anforderungsanalyse

3.1 Analyse

3.2 Benötigte Daten aus dem PIM System

3.3 Vergleich der LLM Modelle

4 Konzeption

4.1 Architektur

4.2 Datenfluss zwischen Vektor Store und Applikation

4.3 Schnittstellendesign

4.4 Promptdesign

5 Implementierung

5.1 Überblick über die Systemkomponenten

5.2 Umsetzung der Schnittstellen (Vector Store / Applikation / LLM)

5.3 Datenimport und -export

5.4 Integration des LLMs und Promptlogik

5.5 Fehlerbehandlung und Parallelität

6 Evaluation

6.1 Aufbau der Evaluationsbewertung

6.2 Bewertungsmetrik/ -kriterien

6.3 Durchführung

Wer bewertet Texte, Bewertungsbogen, Wie viele Bewertungen pro Text

6.4 Ergebnis

6.5 Diskussion

LLM ohne RAG halluziniert → schlecht für PIM-Daten (Faktentreue).

Prompt mit Rollenbeschreibung („Du bist ein Marketing-Experte...“) besser als generischer Prompt.

Techniktexte brauchen präzisere Struktur -> Template hilft.

Marketingtexte profitieren von höherer Kreativität → Temperatur-Einstellungen diskutieren.

Literatur

[1] S. J. Russell und P. Norvig, *Artificial intelligence: a modern approach* (Prentice Hall series in artificial intelligence). Upper Saddle River: Prentice Hall, 1995, 932 S., ISBN: 978-0-13-103805-9 978-0-13-360124-4.

[2] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.

[3] Y. LeCun, Y. Bengio und G. Hinton, „Deep learning,“ *Nature*, Jg. 521, Nr. 7553, S. 436–444, 28. Mai 2015, ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature14539. besucht am 7. Dez. 2025. Adresse: https://www.nature.com/articles/nature14539.

[4] L. Ma u. a., *A Comprehensive Survey on Vector Database: Storage and Retrieval Technique, Challenge*, 16. Juni 2025. DOI: 10.48550/arXiv.2310.11703. arXiv: 2310.11703[cs]. besucht am 3. Dez. 2025. Adresse: http://arxiv.org/abs/2310.11703.