BHT Berliner Hochschule für Technik

Bachelorarbeit im Studiengang Medieninformatik

Aufbau einer semantischen Produktsuche mit Vector-Search-Technologie

Zur Erlangung des akademischen Grades „Bachelor of Science (B.Sc.)“ im Studiengang Medieninformatik der Berliner Hochschule für Technik

im Fachbereich IV

vorgelegt von:	Max Noah Heinrich
Studiengang:	Medieninformatik
Matrikelnummer:	941795
Betreuer:	Prof. Dr. Siamak Haschemi
Gutachter:	Prof. Dr. Zbigniew Jerzak
Bearbeitungszeitraum:	01.04.2025 - 01.07.2025

1. Juli 2025

Zusammenfassung

Im Rahmen dieser Arbeit wird die Konzeption und Implementierung einer semantischen Produktsuche mittels Vector-Search-Technologie für E-Commerce-Anwendungen untersucht. Die zentrale Herausforderung ist hierbei die semantische Lücke zwischen natürlichsprachigen Suchanfragen von Nutzern und den strukturierten Daten in relationalen Datenbanksystemen. Konventionelle mustervergleichende Suchverfahren scheitern systematisch an der semantischen Intention von Nutzeranfragen, da sie weder Synonyme noch kontextuelle Bedeutungsebenen erfassen können.

Diese Arbeit entwickelt eine Lösung auf Vektordatenbanken und Modellen zur natürlichen Sprachverarbeitung, die Suchanfragen semantisch verarbeitet und relevante Produktempfehlungen für mehrere Millionen Produkte generiert. Die Implementierung erfolgte als eigenständiger Python-Microservice unter Verwendung von PostgreSQL mit pgvector. Das System integriert kontinuierliche Datensynchronisation mit einem extern verwalteten Produktkatalog, automatisierte Embedding-Generierung und skalierbare API-Schnittstellen. Drei Embedding-Modelle wurden vergleichend evaluiert:

- `sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2`
- `Alibaba-NLP/gte-multilingual-base`
- `intfloat/multilingual-e5-base`

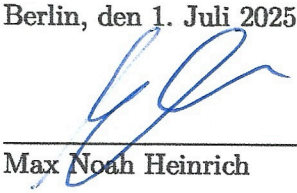
Die Arbeit demonstriert die technische Machbarkeit semantischer Produktsuche in bestehenden PostgreSQL-Infrastrukturen. Der entwickelte Microservice ermöglicht eine nahtlose Integration ohne zusätzliche externe proprietäre Anbieter. Die Evaluation zeigt deutliche Zielkonflikte zwischen Performance und Modellqualität auf, wobei die Wahl des Embedding-Modells domänenspezifisch optimiert werden sollte. Die Implementierung stellt eine solide Grundlage für produktive E-Commerce-Anwendungen dar, identifiziert jedoch auch Optimierungspotential und Raum für weitere Entwicklungen in Bereichen wie multimodale Ansätze, domänenspezifisches Fine-Tuning und Evaluationsmethoden für Embedding-Modelle.

Eidesstattliche Erklärung

Ich, Max Noah Heinrich, versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Zusätzlich wurden vereinzelt KI-Tools verwendet, beispielsweise zur grammatikalischen Korrektur. Etwaige Änderungen wurden sorgfältig überprüft, um die wissenschaftliche Integrität und die Originalität der Arbeit sicherzustellen.

Diese Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Berlin, den 1. Juli 2025



Max Noah Heinrich

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface – standardisierte Programmierschnittstelle zur Kommunikation zwischen Softwarekomponenten
Audit-Trails	Protokollierung und Nachverfolgung von Systemaktivitäten für Compliance und Sicherheitsanalysen
Brute-Force	Algorithmischer Ansatz, der alle möglichen Lösungen systematisch durchprobiert, ohne Optimierungen zu verwenden
DAU	Daily active users – die Anzahl der einzelnen Nutzer, die innerhalb eines 24-Stunden-Fensters mit einer Anwendung interagieren
Data-Residency	Rechtliche und technische Anforderungen bezüglich des geografischen Speicherorts von Daten
DX	Developer Experience bezeichnet die Gesamtheit der Erfahrungen und Interaktionen, die Entwickler beim Arbeiten mit Software-Tools, APIs, Frameworks, Bibliotheken oder Entwicklungsumgebungen machen
GIL	Global Interpreter Lock – Mechanismus in Python, der verhindert, dass mehrere Threads gleichzeitig Python-Code ausführen
Idempotenz	Eigenschaft einer Operation, dass mehrfache Ausführung mit denselben Parametern zum gleichen Ergebnis führt
NLP	Natural Language Processing – Teilbereich der künstlichen Intelligenz zur Verarbeitung natürlicher Sprache
Normalisierung	Datenbankdesign-Technik zur Reduzierung von Datenredundanz und Vermeidung von Anomalien
ONNX	Open Neural Network Exchange – offener Standard für die Darstellung von Machine Learning-Modellen

ORM	Object-Relational Mapping – Programmiertechnik zur Abbildung relationaler Datenbankstrukturen auf objektorientierte Programmiersprachen
RAM	Random Access Memory – Arbeitsspeicher für temporäre Datenspeicherung
Separation of Concerns	Designprinzip zur Trennung von Programmfunktionalitäten in distinct sections, sodass jeder Bereich eine separate Verantwortlichkeit adressiert
SLA	Service Level Agreement – Vereinbarung über die zu erbringende Dienstleistungsqualität zwischen Anbieter und Kunde
Vendor Lock-in	Abhängigkeit von einem spezifischen Anbieter, die den Wechsel zu alternativen Lösungen erschwert oder verteuert

1 Einleitung

Die digitale Transformation des Handels und die damit einhergehende Verschiebung der Verbrauchergewohnheiten sind seit inzwischen mehr 30 Jahren integraler Bestandteil des Internets. Als 1994 über NetMarket in Form des Verkaufs einer Sting-CD das erste Mal eine Online-Transaktion über eine Kreditkarte abgewickelt wurde [1], war noch nicht abzusehen, welche Bedeutung E-Commerce-Plattformen haben werden und wie diese sich als die primären Anlaufstellen für Produktrecherche- und -erwerb entwickeln. Diese Entwicklung brachte jedoch auch neue Herausforderungen mit sich: digitale Plattformen müssen Nutzerbedürfnisse allein durch technische Lösungen erfassen und bedienen.

Eine entscheidende Rolle spielt hierbei die technische Umsetzung, mit der Verfahren für Such- und Empfehlungssysteme die Nutzerintention mit dem verfügbaren Produktangebot verbindet. Moderne Anwendungen müssen fähig sein, auch ungenau oder umgangssprachlich formulierte Suchanfragen zu verstehen und relevante Ergebnisse zu liefern. Die Integration von Machine Learning-Technologien in Such- und Empfehlungssysteme eröffnet hier neue Möglichkeiten, erfordert jedoch eine durchdachte Einbindung in bestehende technische Infrastrukturen.

1.1 Motivation und Problemstellung

Eine zentrale Herausforderung moderner E-Commerce-Plattformen liegt an der semantischen Lücke zwischen natürlichsprachigen Suchanfragen der Nutzer und den strukturierten Daten der zugrundeliegenden Datenbanksysteme.

Konventionelle Suchverfahren, die beispielsweise auf Mustererkennung und Textübereinstimmung basieren, scheitern systematisch an dieser semantischen Diskrepanz. Sie können weder Synonyme noch kontextuelle Bedeutungsebenen erfassen und führen damit zu Implikationen auf die Nutzererfahrung.

1.2 Zielsetzung und Forschungsfragen

Diese Arbeit verfolgt das Ziel, eine mögliche Lösung für die semantische Produktsuche zu entwickeln und deren Integration in bestehende E-Commerce-Infrastrukturen zu

untersuchen. Im Mittelpunkt steht die Konzeption und Implementierung einer Vector-Search-basierten Lösung, die natürlichsprachige Suchanfragen semantisch verarbeitet und relevante Produktempfehlungen generiert.

Die zentrale Forschungsfrage lautet:

> **Wie lässt sich eine semantische Produktsuche mit Vector-Search-Technologie konzipieren und in bestehende E-Commerce-Infrastrukturen integrieren?**

Hieraus ergeben sich folgende spezifische Forschungsfragen:

- Wie kann eine semantische Produktsuche für mehrere Millionen Produkte technisch effizient implementiert werden?
- Welche Embedding-Modelle liefern optimale Ergebnisse für deutschsprachige Produktdaten im E-Commerce-Kontext?
- Welche Herausforderungen in Hinblick auf die Architektur und Implementierung ergeben sich bei der Integration in bestehende PostgreSQL-basierte Infrastrukturen?

1.3 Aufbau der Arbeit

Diese Arbeit gliedert sich in sieben Hauptteile, die systematisch von den theoretischen Grundlagen zur praktischen Implementierung und Evaluation führen:

1. **Aufgabenstellung**: Definition des konkreten Anwendungskontexts und der spezifischen Anforderungen der donista-Infrastruktur
2. **Theoretische Grundlagen**: Einführung in Konzepte der Datenhaltung, Vektordatenbanken und Embedding-Technologien
3. **Stand der Technik**: Analyse aktueller Vector-Search-Technologien und deren Anwendungsbereiche
4. **Konzeption und Methodik**: Entwicklung der Systemarchitektur, Auswahl geeigneter Technologien und Definition der Evaluationsmethoden
5. **Implementierung**: Detaillierte Beschreibung der technischen Umsetzung als Microservice mit PostgreSQL und pgvector

6. **Evaluation und Ergebnisse**: Performance-Analyse verschiedener Embedding-Modelle und qualitative Bewertung der Suchergebnisse
7. **Diskussion, Fazit und Ausblick**: Interpretation der Ergebnisse, Diskussion von Limitationen und Aufzeigung zukünftiger Entwicklungsmöglichkeiten

2 Aufgabenstellung

donista ist ein deutsches Not-for-Profit Software-Startup aus Berlin, welches über eine Browser-Extension bei Online-Einkäufen mittels eigener Affiliate-Links Spenden für soziale Zwecke generiert. Der Use-Case sieht vor, dass die Extension im Hintergrund läuft und dem Nutzer anbietet, bei allen qualifizierenden Online-Einkäufen über einen Affiliate-Link weitergeleitet zu werden. Donista erhält dadurch von den Händlern eine kleine Provision. Diese Provision wird dann einer, vom Nutzer ausgewählten sozialen Einrichtung, gespendet.

Ihre Mission selbst beschreibt donista als: *"Wir verwandeln alltägliches Online-Shopping in eine Kraft für das Gute. donista macht es möglich, ohne Mehrkosten bei jedem Einkauf Spenden zu generieren – für nachhaltige, soziale und humanitäre Projekte."* [2]

In naher Zukunft soll diese Funktion nicht mehr nur über Partnershops ermöglicht werden, sondern auch über eigene Anwendungen & Apps, bei denen Nutzern direkt eine Produktsuche geboten wird. Diese Features sind bereits in der Konzeptions- und Entwicklungsphase. Um die Features rund um die Produktsuche nutzbar zu machen, benötigt es natürlich Produkte - hierfür steht ein Produktkatalog von ca. 3 Millionen Einträgen eines externen Datenlieferanten bereit.

Im Rahmen dieser Bachelorarbeit soll für donista eine semantische Produktsuche auf Basis von Vector Search-Technologie entwickelt werden. Dabei geht es über klassische Keyword-basierte Ansätze hinaus, indem die Bedeutung von Texten und Abfragen analysiert wird. Die Lösung muss für einen Produktkatalog von bis zu 5 Millionen Produkten skalierbar sein und sowohl technisch als auch qualitativ evaluiert werden.

Die konkrete Aufgabenstellung umfasst zwei Hauptziele:

1. **Technologie-Analyse und -Evaluation:** Untersuchung und Bewertung verfügbarer Technologien zur Implementierung einer semantischen Produktsuche, einschließlich der Analyse von Vektordatenbank-Lösungen, Embedding-Modellen und Evaluationsmethodiken für den E-Commerce-Kontext
2. **Entwicklung eines vollständigen semantischen Suchsystems:** Konzeption und Implementierung einer End-to-End-Lösung, die folgende Komponenten umfasst:
    - Integration und Transformation externer Produktdaten des Anbieters shopping24.com

- Kontinuierliche Datensynchronisation zur Aufrechterhaltung der Aktualität
- Automatisierte Embedding-Generierung und -Aktualisierung für semantische Vektorrepräsentationen
- Skalierbare API-Schnittstellen für semantische Suchanfragen und Produktempfehlungen
- Performance-Monitoring und Qualitätssicherung des Gesamtsystems

3 Theoretische Grundlagen

3.1 Grundlagen der Datenhaltung

Schon vor langer Zeit begannen Menschen, Daten zu speichern – lange vor der Erfindung von Computern, Datenbanken und elektronischer Datenverarbeitung. Bereits die Buchhalter des alten Ägyptens führten strukturierte Aufzeichnungen über die Leistung der landwirtschaftlichen Produktion, der Steuern und der königlichen Lagerhäuser. Schriftgelehrte zeichneten Inventare, Arbeiten und verschiedene Transaktionen auf Papyrus-, Ton- und Knochentafeln akribisch auf, um ihre begrenzten Ressourcen zu verfolgen und fundierte Entscheidungen zu treffen. Schon in der Neuzeit, aber immer noch weit vor dem Aufkommen von Computern, entwickelten Behörden, Krankenhäuser und Unternehmen durchdachte Systeme zur Datenhaltung [3, 4].

Betrachten wir beispielsweise die 1876 erstmals veröffentlichte Dewey-Dezimalklassifikation (DDC) [5] , ein Klassifizierungssystem für literarische Werke des Bibliothekars Melvil Dewey, so lassen sich auch hier Grundprinzipien erkennen, die in modernen Datenbanken nach wie vor von Relevanz sind, beispielsweise die hierarchische Gliederung, die systematische Codierung, die eine einzigartige (*unique*) Zuordnung ermöglicht, eine skalierbare Struktur und abstrakte Ordnung, welche die logische von der physischen Anordnung trennt.

3.1.1 Relationale Datenbanksysteme

Daten bilden die Basis fast aller Unternehmen und Organisationen. Diese Daten befinden sich oft in Form von Listen von Dingen. Für Datenbanken gilt das fundamentale Konzept, dass in jedem Unternehmen und jeder Organisation Dinge existieren, deren Daten persistiert werden müssen, und diese Dinge stehen in gewisser Weise miteinander in einer Beziehung. Um als Datenbank zu gelten, muss der Ort, an dem die Daten gespeichert werden, also nicht nur die Daten, sondern auch die Informationen über die Beziehungen zwischen diesen Daten enthalten. Die Idee hinter einer Datenbank ist, dass der Benutzer – entweder eine Person, die interaktiv mit der Datenbank arbeitet oder über ein Anwendungsprogramm – sich keine Gedanken über die Art und Weise machen muss, wie die Daten physisch auf der Festplatte gespeichert werden. Relationale Datenbanken, welche heutzutage den Großteil der in Benutzung befindlichen Datenbanken darstellen, wurden für Transaktionsprozesse entwickelt, also um in hohem Volumen

Daten zu schreiben und zu lesen und dabei die Integrität der Daten sicherzustellen [6, 7].

Bei einer relationalen Datenbank spricht man von einer Datenbank, deren logische Struktur allein aus einer Sammlung von Relationen (Beziehungen) besteht. In der mathematischen Mengenlehre ist eine Relation die Definition einer Tabelle mit Spalten (Attributen) und Zeilen (Tupeln). Die Definition der Datenbank legt dabei fest, welche Form die Daten in jeder Spalte und Zeile haben werden, enthält jedoch nicht die Daten selbst. Wenn Zeilen mit Daten enthalten sind, spricht man von einer Instanz einer Relation.

3.1.2 Das Problem der semantischen Suche in klassischen DBMS

Betrachten wir nun, was eigentlich damit gemeint ist, etwas zu suchen. Um inhaltlich nah am Thema der Arbeit zu bleiben, können wir dies am Beispiel eines Produktkatalogs tun. Für dieses Beispiel können wir diesen relativ simpel halten. Nehmen wir an, wir haben unseren Produktkatalog auch in einer relationalen Datenbank gespeichert, beispielsweise mit einem Schema, das eine einzigartige Produktnummer, den Namen des Produkts, die Kategorie, zu der das Produkt gehört, die Farbe des Produkts, eine Beschreibung und den Preis des Produkts enthält. Die Anwendung von Normalisierungen wird in diesem Beispiel der Einfachheit halber ignoriert.

id	**name**	**brand**	**description**	**category**	**color**	**price**
int	varchar	varchar	mediumtext	varchar	varchar	double

Eine Instanz dieser Relation könnte dann folgendermaßen aussehen:

id	**name**	**brand**	**description**	**category**	**color**	**price**
1	iPhone 12	Apple	Das neue Apple iPhone 12	Smartphones	schwarz	500.00

Sagen wir nun, ich bin auf der Suche nach einem neuen Smartphone. Welches Modell es genau werden soll, weiß ich noch nicht, nur dass es ein iPhone sein soll und in der Farbe Schwarz. Mittels SQL lassen sich Produkte, die in diese Parameter fallen, natürlich einfach ausgeben:

Listing 3.1: SQL-Abfrage für Apple-Produkte der Kategorie Smartphone

```
1 SELECT * from products WHERE product.brand='Apple' AND product.category='Smartphone' AND
      product.color='black'
```

Im echten Leben wird es natürlich keine E-Commerce-Plattform geben, bei denen die Nutzer direkt SQL-Abfragen schreiben und ausführen, um Produkte zu finden. Wie kommt man also von dem, was der Nutzer sucht, zu einer Abfrage, die in der Datenbank ausgeführt werden kann?

Generell basieren die meisten Anwendungen auf der in der Abbildung 3.1 vereinfacht dargestellten mehrschichtigen Architektur. Diese folgt dem bewährten Client-Server-Datenbankmodell mit klarer Trennung der Verantwortlichkeiten. Der typische Ablauf einer Produktsuche erfolgt dabei über mehrere Schichten: Der Nutzer interagiert mit der Client-Anwendung, diese sendet `HTTP`-Anfragen an die API-Schicht, welche die Anfrage über Service-Funktionen verarbeitet und mittels Datenbankconnector in entsprechende SQL-Abfragen übersetzt. Die Ergebnisse durchlaufen denselben Pfad in umgekehrter Richtung zurück zum Nutzer.

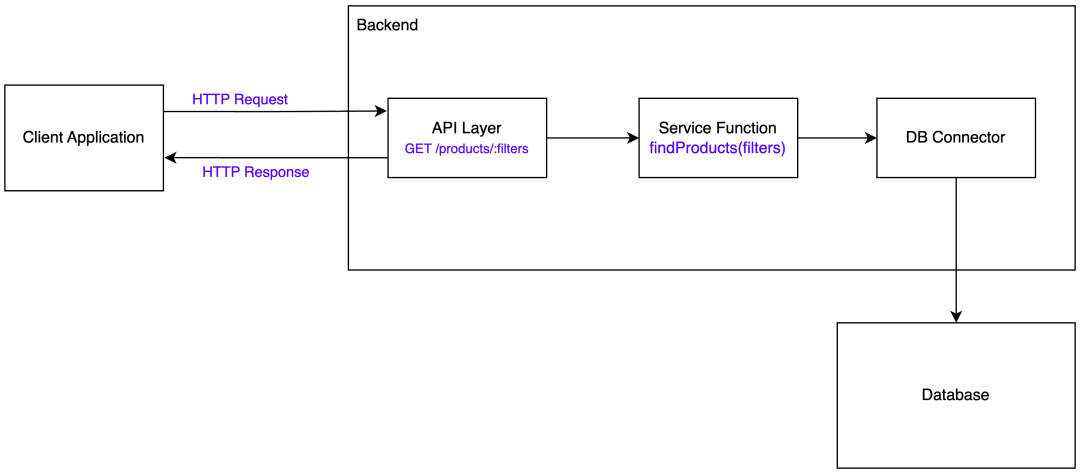


Abbildung 3.1: Beispielhafte Architektur

Dem Nutzer wird über die Client Application also ein Interface bereitgestellt, welches die Suchparameter erfasst. Diese werden dann an das Backend übermittelt, wo eine Servicefunktion, normalerweise unter Verwendung eines ORM, die Suchparameter in eine SQL-Abfrage verwandelt. Betrachten wir aber beispielhaft die Landing Pages einiger namhafter Online-Shops 3.2, 3.3, 3.4, lässt sich direkt erkennen: Statt der Möglichkeit, Filter in verschiedenen Kategorien wie Hersteller, Farbe oder Kategorie anzugeben, gibt es hauptsächlich nur eine primäre Eingabemethode, um nach Produkten zu suchen: ein Text-Input.

Abbildung 3.2: Amazon Landing Page

Abbildung 3.3: Saturn Landing Page

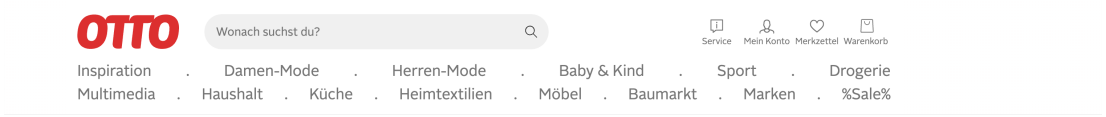


Abbildung 3.4: OTTO Landing Page

Statt direkt in bereits festgelegten Kategorien Filter anzuwenden, ist die primär angestrebte Nutzerinteraktion, mit einem oder mehreren Suchbegriffen in einer Freitextsuche nach dem Produkt zu suchen, welches ihn interessiert. Hierbei stellt sich nun die Frage, wie man daraus letztendlich eine SQL-Abfrage generiert. Der einfachste Weg wäre natürlich, den Suchbegriff des Nutzers als Parameter mitzuschicken und in die Abfrage zu integrieren. Eine Möglichkeit hierfür wäre die Verwendung des `LIKE`-Operators.

Listing 3.2: SQL-Abfrage für iPhone-Produkte

```
1 SELECT * from products WHERE product.name LIKE 'Apple iPhone 12, schwarz'
```

In diesem Beispiel wird der `LIKE`-Operator für exakte String-Übereinstimmung verwendet, also würde diese Abfrage alle Produkte zurückgeben, bei denen der Wert der Spalte `name` exakt `Apple iPhone 12, schwarz` beträgt. In der Realität ist dies natürlich nicht der Fall – schauen wir uns beispielsweise alle Einträge für das, was in unserem Beispiel die Spalte `products.name` ist, das Apple iPhone 12 im Onlineshop von Saturn an (https://www.saturn.de/de/search.html?query=iphone%2012):

- APPLE IPHONE 12 128GB BLACK 128 GB Schwarz Dual SIM
- APPLE iPhone 12 Pro 128 GB Graphit Dual SIM
- APPLE iPhone 12 128 GB Blau Dual SIM
- APPLE IPHONE 12 64GB BLUE 64 GB Blau Dual SIM
- APPLE IPHONE 12 MINI 128GB BLUE 128 GB Blau Dual SIM
- APPLE IPHONE 12 128GB GREEN 128 GB Grün Dual SIM
- APPLE IPHONE 12 128GB PURPLE 128 GB Violett Dual SIM
- APPLE iPhone 12 Pro 128 GB Pazifikblau Dual SIM
- APPLE iPhone 12 Mini 128 GB Weiß Dual SIM

- APPLE IPHONE 12 MINI 128GB BLACK 128 GB Schwarz Dual SIM

Wie sich direkt erkennen lässt, ist die Benennung der Produkte nur grob systematisch: es folgt zwar immer auf den Herstellernamen die Modellbezeichnung (`APPLE iPhone 12`), allerdings ist die Groß- und Kleinschreibung inkonsistent, es gibt noch die Produktvariation `Pro` und `Mini/MINI`, die Angaben für Speicherkapazität sind unterschiedlich (Wert und Einheit zusammengeschrieben und getrennt) und die Produktfarbe ist in manchen Fällen auf Deutsch und in manchen wiederum auf Englisch.

Da die Produktnamen keinem konsistenten Schema folgen müssen und die Nutzer auch nicht vorhersagen können oder wissen müssen, welchem Schema Produktnamen in der Relation folgen, wird hier eine gewisse Flexibilität benötigt: Approximate string matching, am bekanntesten als *fuzzy string searching* oder auf Deutsch auch *unscharfe Suche*.

Während mit der oben gezeigten SQL-Abfrage 3.2 keine Ergebnisse gefunden werden, könnten wir diese Abfrage mit sogenannten Wildcards und einer leicht veränderten Struktur optimieren.

Nehmen wir an, in unserem Backend existiert eine generische Funktion `split()`, wie es sie in vielen Programmiersprachen gibt. Diese rufen wir auf unserem Suchbegriff auf, siehe folgenden Beispielcode:

Listing 3.3: split-Funktion

```
1 const search_string = 'Apple iPhone 12, schwarz'
2 const arr = search_string.split(' ').replace(',', '')
```

Das Array `arr` sieht nun folgendermaßen aus: `['Apple', 'iPhone', '12', 'schwarz']`. Statt den Suchbegriff direkt in der Abfrage auszuführen, teilen wir diesen in seine einzelnen Wörter auf und selektieren alle Einträge in der Datenbank, bei denen jede Zeile der Spalte `products.name` ein Auftreten aller einzelnen Wörter enthält.

Listing 3.4: SQL-Abfrage für iPhone-Produkte mit Wildcards

```
1 SELECT * from products
2 WHERE product.name LIKE '%Apple%'
3 AND product.name LIKE '%iPhone%'
4 AND product.name LIKE '%12%'
5 AND product.name LIKE '%schwarz%'
```

Am Beispiel der Apple iPhone 12 Modelle von Saturn-Onlineshop würden wir dafür bereits folgende Produkte als Ergebnisse erhalten:

- APPLE IPHONE 12 128GB BLACK 128 GB Schwarz Dual SIM
- APPLE IPHONE 12 MINI 128GB BLACK 128 GB Schwarz Dual SIM

Für dieses einfache Beispiel ist es so relativ trivial, eine sehr vereinfachte Textsuche zu implementieren. Was hier allerdings passiert, ist allein der Vergleich von Mustern (Pattern-Matching). Was ist nun aber, wenn das Muster des Suchbegriffs nicht direkt mit den Werten in der Tabelle übereinstimmt? Nehmen wir an, dass statt mit produktspezifischen Parametern wie `Apple iPhone 12, schwarz` mit anwendungsspezifischen Wörtern gesucht wird, beispielsweise `Günstiges Smartphone mit guter Kamera`.

3.1.3 Erweiterte Textsuche: Volltextindizes und statistische Relevanz-Bewertung

Die aus einfachen Pattern-Matching-Ansätzen resultierenden Limitierungen führten zur Entwicklung verschiedener Technologien für die Textsuche in Datenbanken. Ein erster bedeutender Fortschritt war die Einführung von Volltextindizes, die auf sogenannten Inverted Indexes basieren. Diese Datenstrukturen erstellen für jedes vorkommende Wort eine Liste aller Dokumente oder Datensätze, in denen es auftritt – ähnlich einem Stichwortverzeichnis in einem Buch. PostgreSQL bietet beispielsweise mit Volltextsuche solche Möglichkeiten, die deutlich performanter sind als LIKE-basierte Abfragen und gleichzeitig linguistische Verbesserungen wie Stemming (Reduktion auf Wortstämme) und Stop-Word-Filterung ermöglichen.

Für die Behandlung von Tippfehlern und ungenauen Suchbegriffen wurde das Konzept der Trigram-basierten Suche entwickelt, das in PostgreSQL durch die pg_trgm-Erweiterung implementiert ist. Trigrams zerlegen Wörter in überlappende Drei-Zeichen-Sequenzen und bewerten die Ähnlichkeit zwischen Begriffen basierend auf der Anzahl gemeinsamer Trigrams. Dies ermöglicht eine Art Fuzzy Search, die auch bei Rechtschreibfehlern noch relevante Ergebnisse liefert.

Spezialisierte Suchmaschinen wie Elasticsearch und Apache Solr, welche beide auf der Apache Lucene-Bibliothek basieren, brachten weitere Verbesserungen mit sich. Diese Systeme bieten nicht nur performante Volltextsuche für große Datenmengen, sondern implementieren auch ausgeklügelte Relevanz-Scoring-Algorithmen. Der weit verbreitete TF-IDF (Term Frequency-Inverse Document Frequency) Algorithmus bewertet die Relevanz eines Dokuments basierend darauf, wie häufig ein Suchbegriff im Dokument vorkommt, gewichtet gegen seine Seltenheit in der gesamten Dokumentensammlung. Eine Weiterentwicklung stellt der BM25-Algorithmus dar, der als probabilistisches Ranking-Verfahren gilt und heute als Standard in vielen Suchmaschinen eingesetzt wird.

Diese Ansätze brachten erhebliche Verbesserungen für strukturierte Suchanfragen mit sich und lösten viele praktische Probleme der traditionellen SQL-basierten Suche. Sie ermöglichen es, auch bei großen Datenmengen performant zu suchen, Tippfehler zu tolerieren und Ergebnisse nach Relevanz zu sortieren. Dennoch stoßen all diese Verfahren bei der Erfassung semantischer Zusammenhänge an ihre fundamentalen Grenzen. Während sie statistisch bewerten können, wie relevant ein Begriff für ein Dokument ist, verstehen sie nicht die Bedeutung von Begriffen. Ein Nutzer, der nach „Laptop für Videobearbeitung“ sucht, würde keine Ergebnisse für einen „High-Performance-Computer für Content Creation“ erhalten, obwohl beide Begriffe semantisch ähnliche Konzepte beschreiben.

3.2 Vektordatenbanken als Infrastruktur

In Fällen, wo der Nutzer nicht genau weiß, was er sucht, sondern nur, welche Eigenschaften das Gesuchte mit sich bringen soll, ist der oben beschriebene Ansatz wenig effektiv. Ein weiteres Problem, welches sich hier ergibt, ist die Vernachlässigung von Kontextbezügen. Nehmen wir an, es handelt sich hier nicht um einen deutschen Onlineshop, sondern einen im englischsprachigen Raum. Wenn ein Nutzer nun eine weniger präzise Suche durchführt, beispielsweise `Apple Device`, existiert hier kein semantischer Kontext, die Suche kann also nicht wissen, ob es um Produkte des US-amerikanischen Technologieunternehmens Apple Inc. geht oder um den Apfel als Pflanzengattung der Kernobstgewächse.

Was wir also benötigen, ist ein Ansatz, welcher nicht die Muster von Zeichenkettenfolgen vergleicht, sondern die intrinsische Bedeutung dieser sprachlichen Zeichen versteht. Dieses Problem gehört in den Bereich des Natural Language Processing (NLP) und wird durch Worteinbettungen ermöglicht.

Bevor wir uns direkt der Funktionsweise von Worteinbettungen widmen, sollten wir zuerst die theoretischen Grundlagen betrachten, die diese ermöglichen.

3.2.1 Grundkonzepte und Architektur

Was sind Vektordatenbanken?

Bei Vektordatenbanken sprechen wir von Datenbanken, die in der Lage sind, Daten in Form hochdimensionaler Vektoren zu speichern. Bei diesen Daten handelt es sich um

mathematische Repräsentationen von Eigenschaften und Attributen. Die Dimensionalität der Vektoren kann, abhängig von der Granularität der Daten, Hunderte Dimensionen betragen [8, 9].

Betrachten wir dazu ein einfaches Beispiel: Wir haben eine Vektordatenbank mit geospatialen Daten 3.5 `points_of_interest`, welche für jedes punkthafte Geo-Objekt die jeweiligen Koordinaten als 2-dimensionalen Vektor speichert.

points_of_interest	location
poi_1	(2,4)
poi_2	(-3,2)
poi_3	(4,-1)
poi_4	(-2,-2)

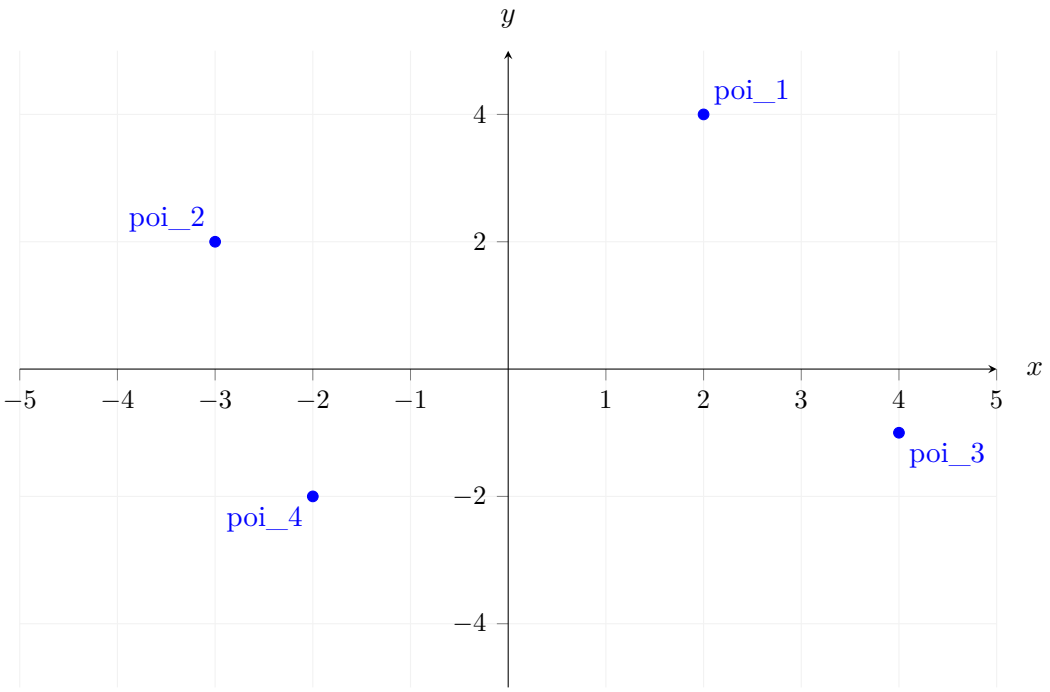


Abbildung 3.5: Visualisierte Geoobjekte

Um beispielsweise den nächstgelegenen Punkt zu einem gegebenen Standort zu finden, berechnet die Vektordatenbank die Distanz zwischen dem Abfragevektor und jedem gespeicherten Vektor in der Datenbank. Dieses Verfahren ist allgemein unter dem Begriff k-nearest-neighbor searching problem bekannt. Ziel dabei ist es, die k nächstgelegenen Punkte in einem Datensatz $X \subset \mathbb{R}^D$, der n Punkte enthält, zu einem Abfragepunkt $q \in \mathbb{R}^D$ zu finden – beispielsweise unter Verwendung der euklidischen Distanz [10]. Eine

gebräuchliche Definition der euklidischen Distanz für zwei Vektoren $\vec{a} = (x_1, y_1)$ und $\vec{b} = (x_2, y_2)$ lautet:

$$d(a, b) = (x_2 - x - 1)^2 + (y_2 - y1)^2$$

Nehmen wir an, der Abfragepunkt befindet sich am Punkt (1,1). Die Datenbank berechnet dann die Distanz zu allen gespeicherten Punkten und kann so den nächstgelegenen Point-of-Interest effizient bestimmen 3.6. Dieser Vorgang wird auch als Nearest Neighbor Search (NN) bezeichnet.

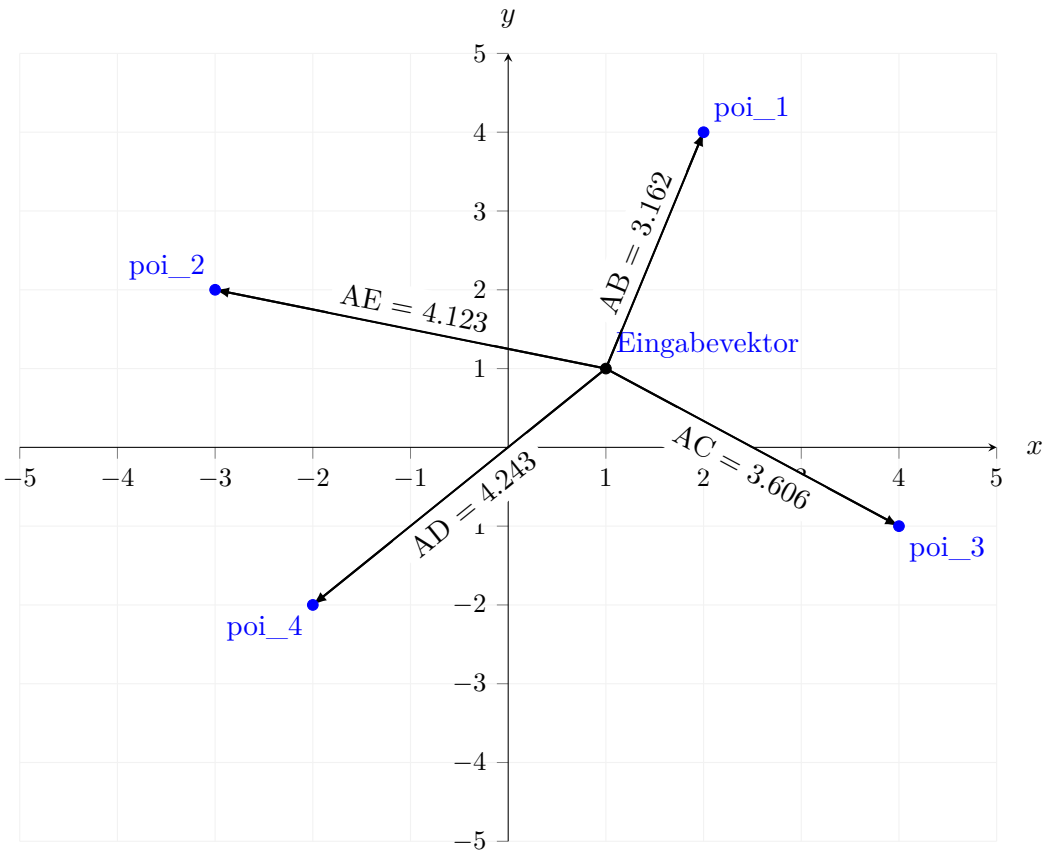


Abbildung 3.6: Visualisierte Geoobjekte mit berechneten Distanzen mittels Euklidischer Distanz

3.2.2 Distanzfunktionen

Im obigen Beispiel wurde bereits veranschaulicht, wie mittels euklidischer Distanz aus einer Liste von Vektoren die kürzeste Distanz zu einem Abfragevektor ermittelt werden kann. Vektordatenbanken unterstützen normalerweise eine Vielzahl von Distanzfunktionen – auf die relevantesten wird im Folgenden eingegangen.

Kosinus-Ähnlichkeit

Die Kosinus-Ähnlichkeit misst den eingeschlossenen Winkel θ zwischen zwei Vektoren in einem mehrdimensionalen Raum. Der mögliche Wertebereich, der die Ähnlichkeit beschreibt, liegt hierbei zwischen -1 und 1 wobei gilt:

- -1: genau entgegengerichtete Vektoren
- 0: die Vektoren sind orthogonal zueinander
- 1: genau gleichgerichtete Vektoren

Die Formel hierfür lautet:

$$\cos(\theta) = \frac{a \cdot b}{\|a\| \cdot \|b\|}$$

Die Kosinus-Ähnlichkeit hat eine besondere Relevanz im Einsatz in Vektordatenbanken, da sie unabhängig von der Größe (engl.: *magnitude*) des Vektors ist. Sie misst also nur die Richtung der Vektoren, nicht deren Länge, was die Kosinus-Ähnlichkeit besonders für hochdimensionale Vektorräume geeignet macht.

Skalarprodukt

Beim Skalarprodukt werden zwei Vektoren miteinander multipliziert. Das Skalarprodukt zeigt sowohl die Ausrichtung als auch die Magnitude zweier Vektoren an. Es ist negativ, wenn die Vektoren in unterschiedliche Richtungen ausgerichtet sind, positiv, wenn die Vektoren in dieselbe Richtung ausgerichtet sind, und null, wenn die Vektoren orthogonal zueinander stehen. Berechnet wird es mit folgender Formel:

$$a \cdot b = \|a\| \cdot \|b\| \cdot \cos(\theta)$$

Im Gegensatz zur Kosinus-Ähnlichkeit ist das Skalarprodukt magnitudenabhängig – längere Vektoren führen zu größeren Werten, selbst bei gleicher Ausrichtung. Dies macht es besonders geeignet für Anwendungen, wo sowohl die Richtung als auch die Stärke der Vektoren relevant sind.

Quadrierte Euklidische Distanz

Die quadrierte euklidische Distanz berechnet den Abstand zwischen zwei Vektoren, indem sie die Summe der quadrierten Differenzen der entsprechenden Vektorkomponenten bildet. Der Abstand kann einen beliebigen Wert zwischen Null und Unendlich annehmen. Ist der Abstand gleich Null, sind die Vektoren identisch. Je größer der Abstand ist, desto weiter sind die Vektoren voneinander entfernt.

Die Formel hierfür ist:

$$d^2(a,b) = \sum_{i=1}^{n}(a_i - b_i)^2$$

Die quadrierte euklidische Distanz steht in direktem Zusammenhang zur euklidischen Distanz, siehe Beispiel in 3.2.1. Durch das Weglassen der Quadratwurzel ist die quadrierte Variante rechnerisch effizienter, da die Quadratwurzel-Operation vermieden wird. Da beide Metriken die gleiche Reihenfolge der Ähnlichkeiten liefern, wird in performancekritischen Anwendungen oft die quadrierte Version bevorzugt.

Manhattan

Die Manhattan-Metrik (auch Taxicab-Distanz genannt) berechnet den Abstand zwischen zwei Vektoren als Summe der absoluten Differenzen ihrer Komponenten. Der Name leitet sich vom Straßennetz Manhattans ab, bei dem sich ein Taxi nur entlang von rechtwinkligen Straßen bewegen kann und nicht geradlinig durch Gebäude hindurch.

Die Formel dafür lautet:

$$d_{\text{Manhattan}}(a,b) = \sum_{i=1}^{n}|a_i - b_i|$$

Hamming

Die Hamming-Distanz misst die Anzahl der Positionen, an denen sich zwei Vektoren gleicher Länge unterscheiden. Einfacher ausgedrückt berechnet sie, wie viele einzelne Operationen notwendig sind, um einen Vektor in den anderen umzuwandeln. Sie wird hauptsächlich für kategorische oder binäre Daten verwendet.

3.2.3 Curse of Dimensionality

Das einfache Beispiel aus 3.2.1 erklärt grundlegend, wie eine Vektordatenbank aufgebaut ist und wie eine Suche auf Basis eines Abfragevektors durchgeführt werden kann. In der Praxis sind die Vektoren, abhängig von den verwendeten Embedding-Modellen, jedoch normalerweise Hunderte Dimensionen tief – bekannte Modelle, wie `sentence-transformers/all-MiniLM-L6-v2` [11], liefern beispielsweise 384-dimensionale Vektoren.

Da Vektordatenbanken in der Praxis also keine 2-dimensionalen, sondern hochdimensionale Vektoren verarbeiten müssen, treten eine Reihe von Phänomenen in diesen Räumen auf und verursachen Probleme, die in niedrigdimensionalen Räumen nicht existieren. Diese Phänomene werden üblicherweise unter der Bezeichnung Curse of dimensionality (dt.: *Fluch der Dimensionalität*) zusammengefasst.

Mit steigender Dimensionalität explodiert das Volumen des Vektorraums, was dazu führt, dass die Punkte in diesem auch viel weiter auseinanderliegen. Ein direktes Resultat daraus ist, dass die Punkte gleich unähnlich erscheinen und sinnvolle Nachbarschaftsbeziehungen verloren gehen, obwohl diese potenziell semantisch ähnlich sind [12].

Welche Implikationen entstehen dadurch für den praktischen Einsatz von Vektordatenbanken? Schauen wir uns das Beispiel aus 3.2.1 an, bei dem unter Einsatz der Formel für die euklidische Distanz der präzise nächstgelegene Vektor in Relation zum Abfragevektor berechnet wird. Hierbei handelt es sich allerdings um einen Ansatz, der zu den Brute-Force-Methoden gehört, es muss also für jeden existierenden Vektor die euklidische Distanz ermittelt werden. Während dies in unserem einfachen Beispiel mit 2-dimensionalen Vektoren vertretbar ist, steigt der Rechenaufwand bei zunehmender Dimensionalität der Vektoren linear an.

Um die praktische Auswirkung des Curse of Dimensionality zu verdeutlichen, lohnt es sich, einen Blick auf den Rechenaufwand bei der Berechnung der euklidischen Distanz zu werfen. Während bei 2-dimensionalen Vektoren lediglich zwei Subtraktionen, zwei Quadrierungen und eine Addition notwendig sind, steigt die Anzahl der Operationen bei 512-dimensionalen Vektoren auf über 1.000.

Genauer gesagt, müssen 512 Subtraktionen und 512 Quadrierungen durchgeführt und anschließend 511 Werte aufsummiert werden, bevor abschließend die Quadratwurzel gezogen wird. Diese Berechnung muss für jeden einzelnen Vergleich mit einem gespeicherten Vektor erfolgen. In einem realistischen Szenario mit Millionen von Vektoren

führt dies zu einem erheblichen Rechenaufwand, der klassische exakte Suchverfahren (*Brute-Force*) schnell ineffizient werden lässt.

Betrachten wir die Zeitkomplexität, haben wir hier einen Aufwand von $\mathcal{O}(n \cdot d)$, wobei n die Anzahl der gespeicherten Vektoren und d die Dimensionalität des Vektorraums ist.

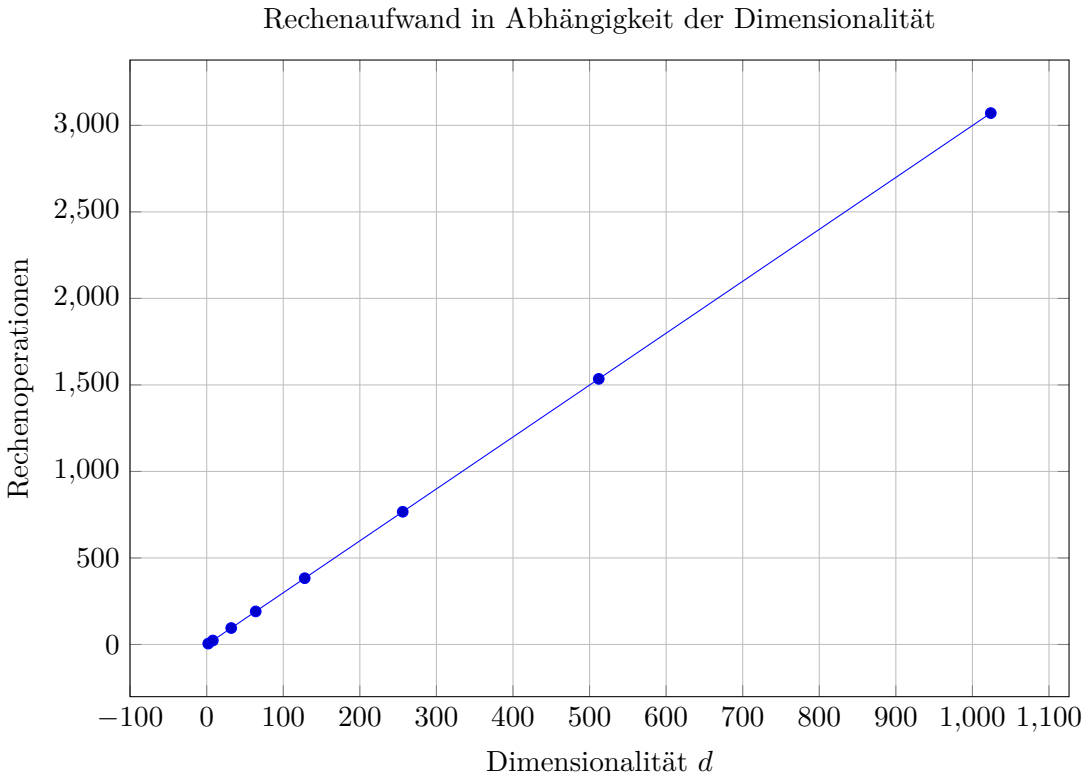


Abbildung 3.7: Lineares Wachstum der Rechenoperationen bei steigender Dimensionalität

3.2.4 Suchalgorithmen und Indexierungsstrategien

Dieser naive Ansatz, mittels Brute-Force-Methoden eine Suche durchzuführen, welche im Berechnungsaufwand linear skaliert, ist in der Praxis für Anwendungen mit mehreren Millionen DAU natürlich wenig vorteilhaft. Zur Mitigation dieser Problematik wurden verschiedene Ansätze und Algorithmen entwickelt, die im Folgenden erläutert werden.

Approximate Nearest Neighbor (ANN)

Mit Approximate Nearest Neighbor bezeichnet man einen Algorithmus, der versucht, einen Datenpunkt in einem Datensatz zu finden, der sich sehr nah am Abfragepunkt befindet, ohne zwingend der exakt nächste Punkt zu sein.

In einer formellen Notation bedeutet dies also: Ein ANN-Algorithmus liefert für eine Abfrage q einen Punkt p' mit $d(q, p')(1 +) \times d(q, p*)$, wobei $\varepsilon > 0$ der Approximationsfaktor und $p*$ der exakte nächste Nachbar ist [10].

Im Gegensatz zu NN-Algorithmen, wie die bereits in 3.2.1 beschriebene Brute-Force-Methode mittels euklidischer Distanz, die versuchen, den exakt nächsten Punkt zu finden, akzeptieren ANN-Algorithmen eine kontrollierte Approximation, also eine Übereinstimmung, die als *nah genug* bewertet wurde (den Approximationsparameter erfüllen).

Um zu verstehen, wie alternative Indexierungsstrategien und Suchalgorithmen funktionieren, empfiehlt es sich, auf die Kompromisse einzugehen, die notwendig sind, um hohe Effizienz bei großer Datenmenge zu ermöglichen. Wie wir bereits erfahren haben, liefern ANN-Algorithmen nicht garantiert den exakten nächsten Nachbarn, sondern einen ungefähr nächsten. Bei dem ersten Kompromiss, der hier bedacht werden sollte, handelt es sich um die Genauigkeit. Ein aggressiv optimierter Algorithmus bietet schnellere Geschwindigkeiten, während dieser gleichzeitig die Genauigkeit reduziert; es können also bessere Treffer existieren, die verpasst werden. Je schneller ein Ergebnis geliefert werden soll, desto eher wird die Genauigkeit reduziert und umgekehrt.

Eine konkrete Form dieses Kompromisses mit Fokus auf die Suchqualität ist Recall vs. Latency. Mit *Recall* werden die richtigen (also tatsächlich nächsten) Treffer, die gefunden wurden, bezeichnet. Eine *Recall*-Rate von 100% würde ein Ergebnis bedeuten, das identisch zu dem der *Brute-Force*-Methode ist. *Latency* bedeutet hierbei nur, wie schnell die Anfrage beantwortet wurde. Die Herausforderung ist hier also, einen Kompromiss zwischen der Präzision der Ergebnisse und einer vertretbaren Antwortzeit zu finden.

Konkrete Algorithmen

HNSW

Grob zusammengefasst erstellt HNSW (Hierarchical Navigable Small-Worlds) eine Graphenstruktur aus mehreren Schichten, bei der jede Schicht ein vereinfachtes, navigierbares Netz darstellt. Man kann sich dies vereinfacht als eine digitale Karte eines Straßennetzes vorstellen: Auf einer geringen Zoomstufe lassen sich die Zusammenhänge von großen Städten und Autobahnen erkennen. Bei Veränderung der Zoomstufe lassen sich dann die Verbindungen der einzelnen Stadtteile erkennen, bis sich schließlich die Verbindungen zwischen den einzelnen Gebäuden eines Bereichs erkennen lassen.

Nach einem ähnlichen Prinzip konstruiert HNSW verschiedene Schichten mit variablen Erreichbarkeitsstufen auf. Durch diese Struktur lassen sich Daten auch in hochdimensionalen Datenräumen schnell lokalisieren.

IVF_FLAT

IVF_FLAT (Inverted File with Flat compression) implementiert ein clusterbasiertes Indexierungsverfahren, das sich vereinfacht als ein großes Lagersystem vorstellen lässt: Der gesamte Datenraum wird in verschiedene Lagerbereiche (Cluster) unterteilt, wobei jeder Bereich um einen zentralen Referenzpunkt (Centroid) organisiert ist. Ähnlich wie Waren in einem Lager nach Kategorien sortiert und in entsprechenden Bereichen gelagert werden, ordnet IVF_FLAT jeden Vektor dem Cluster zu, dessen Zentrum ihm am ähnlichsten ist.

Bei einer Suchanfrage muss das System nicht das gesamte "Lager"durchsuchen, sondern identifiziert zunächst die vielversprechendsten Lagerbereiche anhand der Ähnlichkeit zu deren Zentren. Anschließend wird nur in diesen ausgewählten Clustern nach den exakten Nachbarn gesucht. Der FLAT-Zusatz bedeutet dabei, dass die Vektoren innerhalb der Cluster unverändert und unkomprimiert gespeichert werden, was eine hohe Suchgenauigkeit bei moderatem Speicherbedarf ermöglicht. Diese Methode ist besonders effektiv bei großen Datenmengen, da sie die Suchzeit erheblich reduziert, ohne die Präzision der Ergebnisse zu beeinträchtigen.

Dies sind nur zwei Beispiele für Indexierungsansätze. In der Praxis implementieren Anbieter von Vektordatenbanken ein breites Spektrum an verschiedenen Kategorien wie Floating Vector Indexes, Binary Vector Indexes, Sparse Vector Indexes, Scalar Indexes und GPU-enabled Indexes.

3.3 Embeddings und Vektorrepräsentationen

3.3.1 Allgemeine Embedding-Konzepte

Unter Embeddings versteht man eine Transformation, die es Maschinen ermöglicht, komplexe Objekte und Konzepte der realen Welt in numerischer Form zu verstehen und zu verarbeiten. Ein Embedding ist eine Vektorrepräsentation von Daten, welche die semantische Beziehung in dem sich befindlichen Vektorraum erhält. Diese Transformation ermöglicht, dass im resultierenden Vektorraum ähnliche Objekte näher beieinander liegen als unähnliche Objekte.

Um dieses Konzept einfach zu veranschaulichen, können wir eine Ableitung auf Basis des Beispiels aus 3.2.1 verwenden. Hierfür vernachlässigen wir die Geospatialität, sondern weisen jedem Objekt stattdessen eine Bedeutung zu, hier mittels einer Kategorie. Da dieses Beispiel rein illustrativ ist, kann ignoriert werden, dass Embeddings normalerweise nicht in Form 2-dimensionaler Vektorrepräsentationen existieren.

point_of_interest	location
poi_1	House
poi_2	Tree
poi_3	Lake
poi_4	Bush
poi_5	Church
poi_6	River

Über die Transformation in ein Embedding können mittels der Vektorrepräsentationen nun die semantische Bedeutung und die Relationen der Objekte untereinander anhand der Nähe oder Entfernung im Vektorraum erkannt werden.

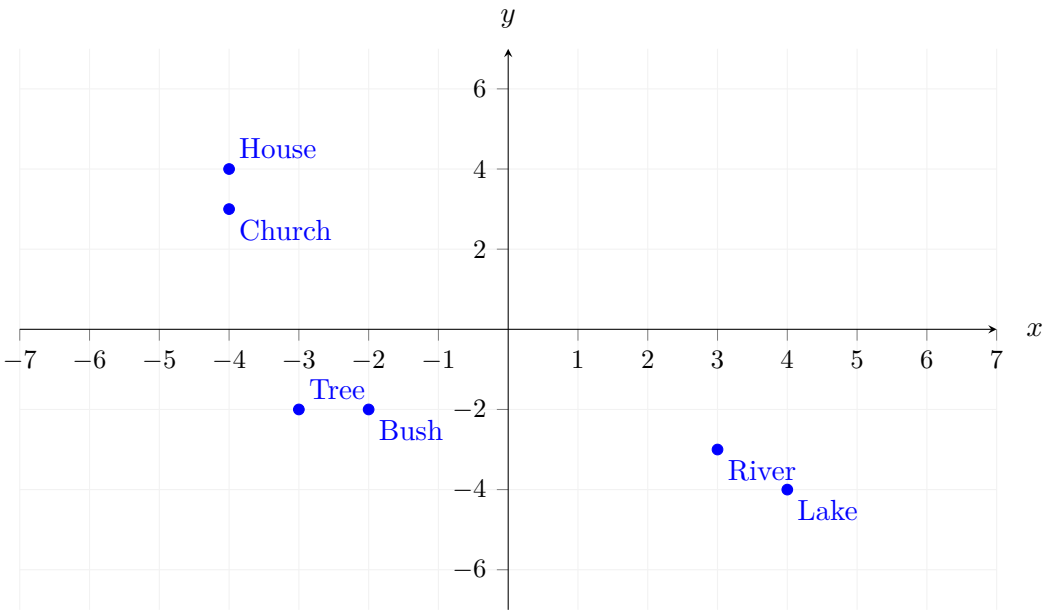


Abbildung 3.8: Objekte als Vektorrepräsentationen in einem 2-dimensionalen Vektorraum

3.3.2 Spezifische Embedding-Typen

Text-Embeddings

Die Evolution von Text-Embeddings zeigt eine kontinuierliche Entwicklung von einfachen statistischen Ansätzen hin zu komplexen Architekturen, die tiefere semantische Beziehungen erfassen können. Diese lassen sich grob in 3 Kategorien unterteilen:

- Word-level
- Sentence-level
- Document-level

Word-level Embeddings wie Word2Vec, GloVe und FastText bilden die erste Generation moderner Word-Embeddings. Diese sind in der Lage, die semantische Bedeutung von Wörtern in Vektorrepräsentationen zu transformieren. Ein berühmtes Beispiel hierfür ist

$$\text{king} - \text{man} + \text{woman} = \text{queen}$$

welches die Fähigkeit demonstriert, semantische Analogien mittels Vektorarithmetik zu erfassen [13]. Eine Limitation von Word-level Embeddings ist jedoch der fehlende Kontextbezug, welcher schon im Abschnitt 3.2 erwähnt wurde: Das Wort *Apple*, unabhängig, ob es um Produkte des US-amerikanischen Technologieunternehmens Apple Inc. geht oder um den Apfel als Pflanzengattung der Kernobstgewächse, erhält die gleiche Repräsentation.

Sentence-level Embeddings adressieren die Kontextlimitationen von Word-level Ansätzen durch die Generierung von Repräsentationen für ganze Sätze. Sentence-BERT (SBERT) revolutionierte diesen Bereich durch eine Siamese-Network-Architektur, die vortrainierte BERT-Modelle für effiziente Satzvergleiche modifiziert. Während BERT für die Suche nach dem ähnlichsten Satzpaar in 10.000 Sätzen etwa 65 Stunden benötigt, reduziert SBERT dies auf ~5 Sekunden für die Embedding-Generierung plus ~0,01 Sekunden für Kosinus-Ähnlichkeitsberechnungen [14].

Document-level Embeddings erweitern den Ansatz auf ganze Dokumente und längere Textpassagen. Doc2Vec erweitert Word2Vec um Dokument-Tokens, die während des Trainings mitlernen. BERT-basierte Ansätze für Dokumente nutzen oft hierarchische Strukturen oder Sliding-Window-Techniken, um die 512-Token-Limitierung zu überwinden [15]. Bei E-Commerce-Produktbeschreibungen, die oft mehrere hundert Wörter umfassen, erfordern diese Ansätze spezielle Segmentierungsstrategien. Die Hauptherausforderung liegt in der Erhaltung semantischer Kohärenz über längere Textpassagen hinweg, ohne wichtige lokale Informationen zu verlieren.

Multimodale Embeddings

Vision-Language Models wie CLIP (Contrastive Language-Image Pre-training) ermöglichen es, Text und Bilder in einen gemeinsamen Embedding-Raum zu projizieren. CLIP ist auf 400 Millionen Text-Bild-Paaren trainiert worden und lernt, semantisch verwandte Texte und Bilder nahe beieinander zu positionieren. ALIGN von Google verfolgt einen ähnlichen Ansatz mit noch größeren Datensätzen. Diese multimodalen Ansätze eröffnen Möglichkeiten für Cross-Modal Retrieval, bei dem beispielsweise Bilder durch Textbeschreibungen gefunden werden können oder umgekehrt.

3.4 Stand der Technik

Die semantische Produktsuche hat sich in den letzten Jahren als wichtiger Bestandteil moderner E-Commerce-Plattformen etabliert. Im Gegensatz zu traditionellen Keyword-

basierten Suchverfahren ermöglicht sie ein tieferes Verständnis der Nutzerintention durch die Analyse semantischer Zusammenhänge. Diese Entwicklung wurde maßgeblich durch Fortschritte im Bereich der Vektorrepräsentationen und maschinellen Lernverfahren vorangetrieben.

3.4.1 Vergleich existierender Vector Search-Technologien

Der Markt für Vector Search-Technologien lässt sich grundsätzlich in drei Kategorien unterteilen: Search-as-a-Service-Angebote, Managed-Vector-Datenbanken und Self-Managed Vector-Search-Lösungen. Jede dieser Kategorien adressiert unterschiedliche Anforderungen hinsichtlich Kontrolle, Skalierbarkeit und Implementierungsaufwand.

Search-as-a-Service

Search-as-a-Service-Lösungen stellen vollständig verwaltete Dienste dar, die über einfache API-Zugänge verfügen und häufig mit grafischen Benutzeroberflächen ausgestattet sind. Der Fokus liegt auf schneller Integration und geringem Wartungsaufwand. Anbieter wie Algolia NeuralSearch und Typesense Search bieten API-basierten Zugriff mit hohem Abstraktionsgrad, sodass sich Entwickler nicht um die zugrundeliegende Infrastruktur kümmern müssen.

Diese Ansätze zeichnen sich durch ihre schnelle Integrierbarkeit und den geringen Wartungsaufwand aus. Gleichzeitig bringen sie jedoch bedeutende Einschränkungen mit sich: Die Kontrolle über die Infrastruktur ist minimal, die Flexibilität bei der Wahl von Metriken oder Index-Tuning stark begrenzt. Zudem besteht das Risiko eines Vendor Lock-ins, und die Kosten können bei zunehmender Skalierung erheblich steigen.

Managed Vector-Databases

Managed-Vector-Databases repräsentieren spezialisierte Anbieter, die sich ausschließlich auf Vektorsuche konzentrieren und diese als vollständig verwaltete Datenbanken bereitstellen. Anbieter wie Pinecone, Weaviate, Zilliz und Qdrant Cloud bieten native Unterstützung für Vektoren mit optimierter Skalierbarkeit und Performance. Der Zugriff erfolgt typischerweise über REST-, gRPC- oder SDK-Schnittstellen.

Diese Lösungen sind explizit für hohe Skalierbarkeit und ANN-Suche optimiert und bieten vielfältige Indexierungsstrategien wie HNSW, Product Quantization oder IVF-PQ. Die Abhängigkeit vom jeweiligen Anbieter und der eingeschränkte SQL-Zugriff,

da es sich nicht um klassische relationale Datenbanksysteme handelt, stellen jedoch potenzielle Nachteile dar.

Self-Managed Vector Search

Self-managed Vector Search-Lösungen umfassen sowohl lokale als auch self-hosted Optionen, die oft als Erweiterungen klassischer Datenbanksysteme oder als spezialisierte Engines implementiert werden. Zu den prominenten Vertretern gehören pgvector für PostgreSQL, FAISS von Meta für Embedding Search in Python und C++, sowie Annoy von Spotify und Hnswlib. Auch einige der zuvor genannten Anbieter wie Qdrant und Milvus stehen als self-hosted Optionen zur Verfügung.

Der wesentliche Vorteil dieser Ansätze liegt in der vollständigen Kontrolle über Infrastruktur, Speicher und Indexierung, was sie besonders für datenschutzkritische oder On-Premises-Anwendungen attraktiv macht. Sie können nahtlos mit bestehenden Datenbanken kombiniert werden und verursachen keine laufenden Cloud-Kosten. Andererseits erfordern sie einen höheren Wartungsaufwand, da Infrastruktur und Skalierung eigenständig gelöst werden müssen, und sind meist nicht so unmittelbar einsatzbereit wie Cloud-basierte Alternativen.

4 Konzeption und Methodik

Dieses Kapitel entwickelt das konzeptionelle Fundament für die semantische Produktsuche. Ausgehend von den in Kapitel 2 definierten Aufgabenstellungen wird hier das System technologieunabhängig entworfen und die methodische Herangehensweise erläutert.

Die hier entwickelten Konzepte bilden die Grundlage für die in Kapitel 5 beschriebene technische Implementierung und stellen sicher, dass die Lösung sowohl den fachlichen Anforderungen entspricht als auch technisch fundiert umsetzbar ist.

4.1 Anforderungsanalyse

Wie schon im Kapitel 2 erläutert wurde, ist das Ziel, eine Produktsuche zu implementieren, die unter Verwendung von Vektordatenbanken und Word Embeddings für eine Abfrage in natürlicher Sprache relevante Produkte ermittelt. Hierbei ist für die Konzeption des Systems auch relevant, dass für die Produktdaten der externen Datenlieferanten keine Infrastruktur bereitstand, um diese zu importieren und kontinuierlich synchron zu halten. Dementsprechend beinhaltet die Konzeption und später auch die Implementierung hier einen Ansatz, der dieses Problem bearbeitet. Da diese Produktsuche kein reines Demo-Projekt ist, sondern innerhalb der donista-Infrastruktur integriert zur Anwendung kommen soll, gibt es einige Anforderungen, die beachtet werden müssen.

4.1.1 Systemkontext und Randbedingungen

Das zu entwickelnde System muss sich in die bestehende Unternehmensinfrastruktur integrieren. Dies führt zu folgenden Randbedingungen:

Technische Randbedingungen

- **Datenbankkompatibilität:** Integration in bestehende relationale Datenbankinfrastruktur
- **Cloud-Kompatibilität:** Betrieb in verwalteter Cloud-Umgebung
- **Keine externen Abhängigkeiten:** Vermeidung zusätzlicher Managed Services

Betriebliche Randbedingungen

- **Verfügbarkeit:** 24/7 Betrieb erforderlich
- **Skalierbarkeit:** Verarbeitung von mehreren Millionen Datensätzen
- **Wartbarkeit:** Integration in bestehende DevOps-Prozesse

Infrastrukturelle Anforderungen:

- **Konsistente Technologie:** Das System, dass die Produktsuche bereitstellt, muss vollständig mit PostgreSQL kompatibel sein, um nahtlos in die bestehende Infrastruktur integriert werden zu können
- **Cloud-Integration:** Die Lösung muss mit den in GCP gehosteten PostgreSQL-Instanzen funktionieren
- **Keine externen Abhängigkeiten:** Um die bestehende donista-Infrastruktur zu erweitern, sollte auf externe Managed-Vektor-Datenbanken-Anbieter wie Pinecone oder Weaviate, verzichtet werden.

4.1.2 Dynamische Produktdaten

Die Datenbasis der semantischen Produktsuche bildet der über eine REST-API bereitgestellte Produktkatalog von `shopping24.com`. Im Gegensatz zu statischen Datenbeständen unterliegt dieser Produktkatalog kontinuierlichen Veränderungen.

- **Produktverfügbarkeit:** Neue Produkte werden hinzugefügt, bestehende Produkte können entfernt werden
- **Attributänderungen:** Produktbeschreibungen, Preise, Kategorisierungen und weitere Metadaten können sich jederzeit ändern
- **Datenqualität:** Korrekturen von Produktinformationen oder Ergänzungen fehlender Attribute erfolgen fortlaufend

Diese Dynamik der Datenquelle stellt besondere Anforderungen an das zu entwickelnde System:

- **Datensynchronisation:** Das System muss Änderungen in der externen Datenquelle erkennen und lokal nachvollziehen können

- **Embedding-Aktualisierung:** Bei Änderungen semantisch relevanter Produktattribute (Titel, Beschreibung, Kategorie) müssen die entsprechenden Vektorrepräsentationen neu generiert werden
- **Konsistenz:** Während Synchronisationsprozessen muss die Suchfunktionalität weiterhin verfügbar bleiben, ohne inkonsistente Ergebnisse zu liefern
- **Performance:** Die Aktualisierungsprozesse dürfen die Antwortzeiten der Suchfunktion nicht beeinträchtigen

Das System muss daher über einen robusten Mechanismus zur kontinuierlichen Datensynchronisation verfügen, der sowohl die Aktualität der Produktdaten als auch die Performance der semantischen Suche gewährleistet.

4.2 Auswahl und Bewertung von Embedding-Modellen

Die Qualität der Ergebnisse der semantischen Produktsuche hängt zu großen Teilen auch von der Wahl des Embedding-Modells ab. Im Folgenden wird darauf eingegangen, wie für meinen spezifischen Anwendungsfall relevante Modelle herangezogen und evaluiert werden können.

4.2.1 Recherche verfügbarer Modelle

Die Suche nach geeigneten Embedding-Modellen konzentrierte sich auf etablierte Plattformen:

huggingface.co

Als primäre Quelle diente der Hugging Face Model Hub, der eine umfangreiche Sammlung vortrainierter Modelle mit detaillierten Metriken und Dokumentationen bereitstellt. Die Suche fokussierte sich auf Modelle der Kategorie `Sentence Similarity`.

MTEB Leaderboard

Das Massive Text Embedding Benchmark (MTEB) bietet eine standardisierte Evaluierung verschiedener Embedding-Modelle über eine Vielzahl von Aufgaben und Sprachen hinweg: https://huggingface.co/spaces/mteb/leaderboard

4.2.2 Auswahlkriterien

- **Sprachunterstützung:** Da der Produktkatalog deutschsprachige Produktbeschreibungen enthält, war die Unterstützung der deutschen Sprache ein Ausschlusskriterium. Priorisiert wurden multilingualen Modelle oder speziell für Deutsch optimierte Modelle.
- **Dimensionalität:** Die Vektordimensionalität beeinflusst direkt den Speicherbedarf und die Suchgeschwindigkeit. Für den geplanten Umfang von bis zu 5 Millionen Produkten wurde ein Kompromiss zwischen Ausdrucksstärke und Effizienz angestrebt.
- **Modellgröße:** Aufgrund der Deployment-Anforderungen wurden Modelle bevorzugt, die einen guten Trade-off zwischen Qualität und Ressourcenverbrauch bieten. Modelle im MTEB Leaderboard mit mehreren GB Speicherbedarf wurden ausgeschlossen.

4.2.3 Kandidatenmodelle

Zur Auswahl von Kandidatenmodellen wurde das bereits im Abschnitt 4.2.1 Plattform huggingface.co und das dort bereitgestellte MTEB-Leaderboard als Referenz genutzt. Modelle wurden nun nach folgendem Prinzip ausgewählt:

Kategorie 1: Verbreitung

Zum einen wurde ein Modell gewählt, welches eine sehr hohe Verbreitung hat. Hierfür wurden auf huggingface.co die Modelle nach folgenden Kriterien gefiltert:

- Task: Sentence Similarity
- Sortierung: Most downloads

Das erste Modell, welches hier explizit als multilingual gekennzeichnet wurde, ist mit 11.196.191 Downloads im letzten Monat (Stand Juni 2025) `sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2`.

Kategorie 2: Ranking

Insgesamt sollen 3 Modelle verglichen werden – für die weiteren 2 Modelle wurde nun das MTEB-Leaderboard zur Auswahl hinzugezogen. Da die Implementierung der Anwendung aktuell nur für Modelle unterstützt, die über das Python-Modul `sentence-transformers` geladen werden können, wurde hier der Filter *Should be sentence-transformers compatible* aktiviert.

Im ersten Augenblick ist es verlockend, direkt das Modell mit dem höchsten Ranking zu verwenden, es sollte aber bedacht werden, dass diese Arbeit die Implementierung in Form eines Python-Microservice behandelt, der Suchanfragen über eine REST-API verarbeitet. Zum Zeitpunkt der Abfrage steht `Qwen3-Embedding-8B` auf Platz 1 des MTEB-Leaderboards. Vergleicht man nun aber einige Parameter dieses Modells mit `paraphrase-multilingual-MiniLM-L12-v2`, welches wir schon für unsere Anwendung ausgewählt haben, ist direkt die massive Differenz an RAM-Bedarf ersichtlich:

- `paraphrase-multilingual-MiniLM-L12-v2`: 449 MB
- `Qwen3-Embedding-8B`: 28866 MB

Für die gleichen Aufgaben hat `Qwen3-Embedding-8B` also einen 64-fachen RAM-Bedarf im Vergleich zum leichteren `paraphrase-multilingual-MiniLM-L12-v2`.

4.3 Anforderungsspezifikation

4.3.1 Akteure und Use-Cases

Akteure

- donista-System
- Endnutzer
- donista-Admins/DevOps
- s24.com API
- Scheduler

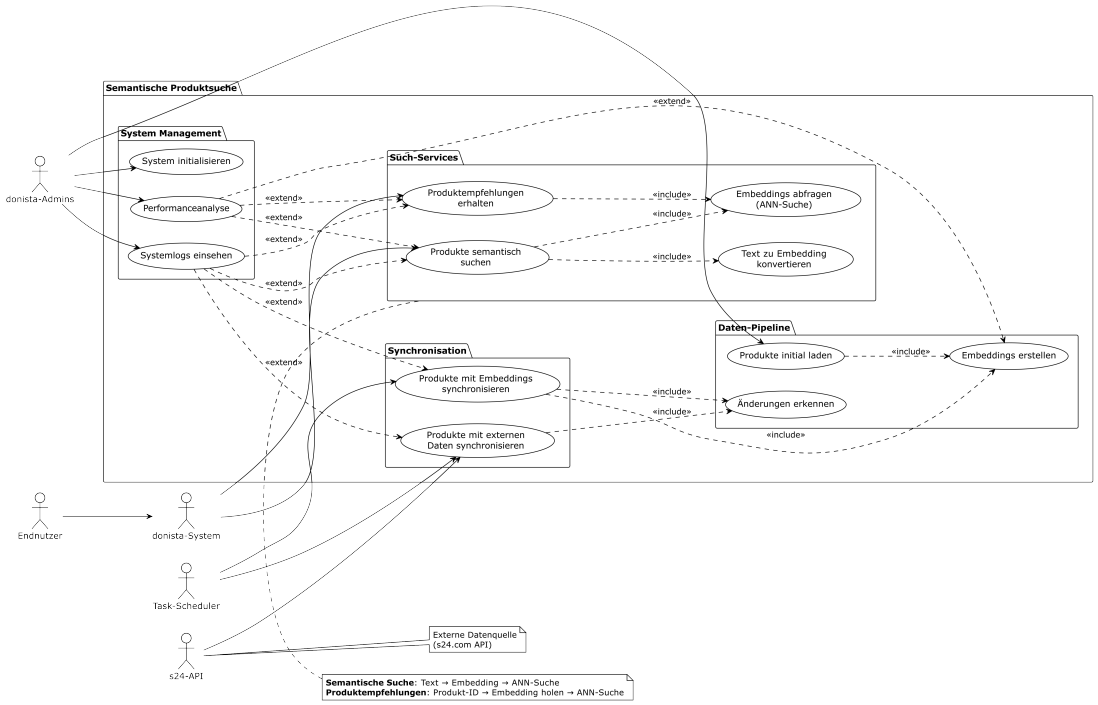


Abbildung 4.1: Use-Case-Diagramm

Use-Cases

- **UC1:** Semantische Produktsuche durchführen
- **UC2:** Produktempfehlungen auf Basis semantischer Relevanz erhalten
- **UC3:** Produktdaten und Embeddings initial laden
- **UC4:** Produktdaten synchronisieren
- **UC5:** Embeddings synchronisieren
- **UC6:** Logs einsehen

4.3.2 Anforderungsbestimmung

Eine Anforderung in der Softwareentwicklung (*„requirement“*) beschreibt eine Eigenschaft oder Leistung, die ein Produkt, System oder ein Prozess erfüllen muss. Es gibt verschiedene Methoden zur Klassifizierung von Anforderungen, wobei die gängigste die

Einteilung in funktionale und nicht-funktionale Anforderungen ist. Hierbei bestimmen funktionale Anforderungen, welche Aufgaben ein Produkt erfüllen soll. In der Literatur gibt es keine einheitliche Definition für eine nichtfunktionale Anforderung (englisch: *„non-functional requirement“*, NFR). Allgemein akzeptiert ist aber, dass sie die funktionale Anforderung übersteigt. Die nichtfunktionalen Anforderungen spezifizieren die Qualität der erwarteten Systemleistung und werden oft als Rahmenbedingungen und Eigenschaften der Qualität interpretiert.

Funktionale Anforderungen

- **FA1 – Semantische Suche**: Das System muss natürlichsprachige Suchanfragen verarbeiten und semantisch relevante Produkte zurückgeben
- **FA2 – Datenintegration**: Das System muss Produktdaten aus externen Quellen importieren und verarbeiten können
- **FA3 – Datensynchronisation**: Das System muss Änderungen in externen Datenquellen erkennen und lokal synchronisieren
- **FA4 – Embedding-Generierung**: Das System muss aus Produktdaten Vektorrepräsentationen generieren
- **FA5 – API-Bereitstellung**: Das System muss Suchfunktionalität über eine programmatische Schnittstelle bereitstellen

Nicht-funktionale Anforderungen

- **NFA1 – Performance**: Suchanfragen müssen in unter 2 Sekunden beantwortet werden
- **NFA2 – Skalierbarkeit**: Das System muss bis zu 5 Millionen Produkte verarbeiten können
- **NFA3 – Verfügbarkeit**: Das System muss eine Verfügbarkeit von 99.9% gewährleisten
- **NFA4 – Genauigkeit**: Die semantische Suche muss eine Precision@5 von mindestens 0.8 erreichen
- **NFA5 – Ressourceneffizienz**: Embedding-Generierung darf maximal 4 Stunden für Vollsynchronisation benötigen

4.4 Systemarchitektur

4.4.1 Architekturmuster

Das System folgt einer Microservice-Architektur mit klarer Trennung von Datenverarbeitung und Bereitstellung:

- **Schichtenarchitektur**: Trennung von Datenhaltung, Verarbeitung und Präsentation
- **Separation of Concerns**: Getrennte Verantwortlichkeiten für verschiedene Geschäftsprozesse
- **Event-Driven Architecture**: Asynchrone Verarbeitung von Datenänderungen

4.4.2 Systemkomponenten

Das System besteht aus folgenden Hauptkomponenten:

1. **Datenerfassung**: Importiert und transformiert externe Produktdaten
2. **Embedding-Generierung**: Erstellt Vektorrepräsentationen aus Produktdaten
3. **Datensynchronisation**: Überwacht Datenänderungen und koordiniert Updates
4. **Suchfunktionalität**: Führt semantische Ähnlichkeitssuche durch
5. **API**: Stellt REST-Schnittstelle für externe Anwendungen bereit

4.4.3 Services

Aus den aufgeführten Use-Cases und entsprechenden Anforderungen lassen sich 5 Services ableiten, die das System bereitstellen muss:

data-loader

Aus den Anforderungen und der Bereitstellung wissen wir bereits, dass unser System in der Lage sein muss, mindestens 3 Millionen Produkte verarbeiten zu können – dazu gehört auch, initial beim ersten Aufsetzen des Systems diese 3 Millionen Produkte strukturiert in die donista-Datenbank zu bewegen. Um hierbei nicht an die API-Limits des externen Datenlieferanten gebunden zu sein, ist es die Aufgabe des Service `data-loader` diese Produktdaten von einer lokalen Kopie des Produktkatalogs von shopping24.com in die Relation `products` einzufügen.

Dieser sollte dabei folgende Anforderungen erfüllen:

- Der `data-loader`-Service muss in der Lage sein, Produktdaten von einer lokalen Datei zu lesen und diese entsprechend dem Schema der `product`-Relation in die Datenbank einzufügen.
- Der `data-loader`-Service muss mindestens in der Lage sein, `JSON`-Dateien lesen zu können; erweiternd bietet es sich an, wenn dieser auch `CSV` verarbeiten kann.
- Der `data-loader`-Service ist unabhängig von den anderen Prozessen des Systems, das soll bedeuten:
  - Es kann nur der `data-loader`-Service ausgeführt werden und dieser erstellt allein die `products`-Relation ohne weitere Services zu beeinflussen.
  - Der Rest des Systems kann unabhängig vom `data-loader` betrieben werden: wurde die `products`-Relation auf anderem Wege erstellt, kann das restliche System, das die semantische Suche bereitstellt, betrieben werden ohne vom `data-loader` abhängig zu sein.
- Der Prozess des erfolgreichen Verarbeitens (Erstellung des Produkts in der Datenbank) und der des fehlgeschlagenen Verarbeitens (Produkt konnte nicht erstellt werden) sollte nachverfolgbar sein, beispielsweise unter Verwendung eines Logging-Systems [16].

embedding-generator

Betrachten wir das Prinzip von *Separation of concerns* [17, 18], so sind Produktinformationen und die Produkt-Embeddings zwei verschiedene *concerns* – zwar werden die Produkt-Embeddings aus Daten des Produktkatalogs erstellt, jedoch werden diese

für unterschiedliche Zwecke verwendet und befinden sich dementsprechend auch in verschiedenen Relationen: die Produktinformationen in `products` und die Embeddings in `product_embeddings`.

Die Rolle des `embedding-generator`-Service ist es, initial beim Aufsetzen des Systems, aus der `products`-Relation, die aus den externen Daten erstellt wurde, alle Embeddings der relevanten Produktdaten zu erstellen.

Dieser sollte dabei folgende Anforderungen erfüllen:

- Der `embedding-generator` erstellt aus allen Produkten die Vektorrepräsentationen in seiner separaten Relation.
- Für jedes Produkt exisitiert pro verwendetem Embedding-Modell maximal eine Vektorrepräsentation.
- Der `embedding-generator`-Service kann nicht starten, bevor der `load-products`-Service mit der Verarbeitung abgeschlossen hat.
- Die einzige Ausnahme hierfür ist, wenn, der im obigen Abschnitt 4.4.3 beschriebene Fall eintritt, dass die Produkte bereits auf anderem Wege angelegt wurden und nicht durch den `data-loader` erstellt werden müssen.
- Der Prozess des erfolgreichen Verarbeitens (Erstellung des Embeddings in der Datenbank) und der des fehlgeschlagenen Verarbeitens (Embedding konnte nicht erstellt werden) sollte nachverfolgbar sein, beispielsweise unter Verwendung eines Logging-Systems.

external-product-sync

Wie wir schon wissen, können sich die Produktdaten des externen Datenlieferanten jederzeit ändern. Um die Anpassungen in unserem System zu erkennen und Produkte zu aktualisieren, implementieren wir einen Service `external-product-sync`, welcher diese Aufgaben übernimmt.

Dieser sollte dabei folgende Anforderungen erfüllen:

- Der Service muss in konfigurierbaren, regelmäßigen Abständen den Produktkatalog des externen Datenlieferanten kontrollieren und Änderungen erkennen. Änderungen sind in diesem Fall:
  - Ein neues Produkt wurde hinzugefügt.

  - Ein bestehendes Produkt wurde entfernt.
  - Ein bestehendes Produkt wurde aktualisiert.

- Da CRUD-Aktionen in der Synchronisierung von Produktkatalogen deutlich weniger ressourcenintensiv als in der Synchronisierung von Produkten zu deren entsprechenden Vektorrepräsentationen, bietet es sich an, den Prozess der Embedding-Aktualisierung auszulagern. Der `external-product-sync` synchronisiert also nur den internen Produktkatalog mit dem des externen Datenlieferanten.
- Alle CRUD-Prozesse des `external-product-sync`-Services sollten protokolliert werden und nachverfolgbar sein, beispielsweise unter Verwendung eines Logging-Systems.

internal-product-sync

Wie schon bereits aufgeführt, ist die Erstellung und Aktualisierung von Einträgen in `product_embeddings` deutlicher ressourcenintensiver als die Erstellung und Aktualisierung von reinen Produktdaten. Dementsprechend wurde die Produktsynchronisation in 2 verschiedene Services unterteilt:

- `external-product-sync`-Service synchronisiert den internen Produktkatalog mit dem des externen Datenlieferanten
- `internal-product-sync`-Service synchronisiert die Vektorrepräsentationen der Produkte mit dem internen Produktkatalog

Somit haben wir die Produktsynchronisierung in 2 verschiedene Services aufgeteilt, die je nach Ressourcenbedarf individuell skaliert werden können.

Die interne Produktsynchronisierung sollte dabei folgende Anforderungen erfüllen:

- Der Service muss in konfigurierbaren, regelmäßigen Abständen den internen Produktkatalog kontrollieren und die Änderungen entsprechend erkennen, um dann die Embeddings zu aktualisieren. Änderungen sind in diesem Fall:
  - Ein neues Produkt wurde hinzugefügt.
  - Ein bestehendes Produkt wurde entfernt.
  - Die Aktualisierung eines bestehendes Produkt verändert seine semantische Bedeutung und erfordert die Aktualisierung des Embeddings

  - Alle CRUD-Prozesse des `internal-product-sync`-Services sollten protokolliert werden und nachverfolgbar sein, beispielsweise unter Verwendung eines Logging-Systems.

api

Im vorherigen Abschnitt wurde bereits darauf eingegangen, dass das System als Microservice implementiert werden sollte, um unabhängig von der donista-Infrastruktur zu funktionieren und diese flexibel erweitern zu können. Um die Funktionen des Microservice an die Hauptanwendung bereitzustellen, bieten sich verschiedene Möglichkeiten an, beispielsweise REST oder gRPC.

Der `api`-Service sollte dabei folgende Anforderungen erfüllen:

- Bereitstellung eines Endpunkts zur Abfrage von Produkten mittels einer Anfrage in natürlicher Sprache
- Bereitstellung eines Endpunkts zur Rückgabe von Produktempfehlung auf Basis eines existierenden Produkts
- Bereitstellung einer API-Dokumentation, die anderen Entwicklern erklärt, nach welchem Prinzip Anfragen durchgeführt werden können, welche Parameter diese erwarten und was die Antwort zurückgibt
- Alle API-Anfragen sollten protokolliert werden und nachverfolgbar sein, beispielsweise unter Verwendung eines Logging-Systems.

4.4.4 Datenmodell

Aus der Anforderungsanalyse haben wir schon erfahren, welche Daten wir speichern müssen:

- Produktdaten aus dem Produktkatalog des externen Datenlieferanten
- Für jedes Produkt das entsprechende Embedding
- Für jeden CRUD-Prozess ein Aktivitätsprotokoll

Daraus ergeben sich 3 Relationen:

- `products`
- `product_embeddings`

- `logs`

ER-Diagram

Aus der Anforderungsanalyse haben wir schon erfahren, welche Daten wir speichern müssen:

- Produktdaten aus dem Produktkatalog des externen Datenlieferanten
- Für jedes Produkt das entsprechende Embedding
- Für jeden CRUD-Prozess ein Aktivitätsprotokoll

Entity-Relationship-Diagramme werden genutzt, um die Typisierung von Objekten und deren Beziehungen untereinander zu visualisieren. Betrachten wir unseren Anwendungsfall, haben wir 3 relevante Beziehungen:

- `products` und `product_embedding`
- `products` und `logs`
- `product_embeddings` und `logs`

Wie wir schon erfahren haben, sind `load-products` und `generate-embeddings` zwei getrennte Services. Auch trennt der Service `external-product-sync` und `internal-product-sync` dies voneinander; d.h. es können Produkte erstellt oder aktualisiert werden, die Erstellung oder Aktualisierung der dazugehörigen Embeddings kann aber aufgrund des unterschiedlichen Ressourcenbedarfs verzögert werden. Das bedeutet für unsere Beziehungen, dass wir hier eine *One-to-zero-or-one*-Beziehung haben: es muss also nicht zwingend jederzeit für jedes Produkt ein Embedding existieren, es darf allerdings maximal ein Embedding pro Produkt entstehen. Die Beziehung wird hier jeweils über die Primärschlüssel `products.id` und `product_embeddings.id` hergestellt.

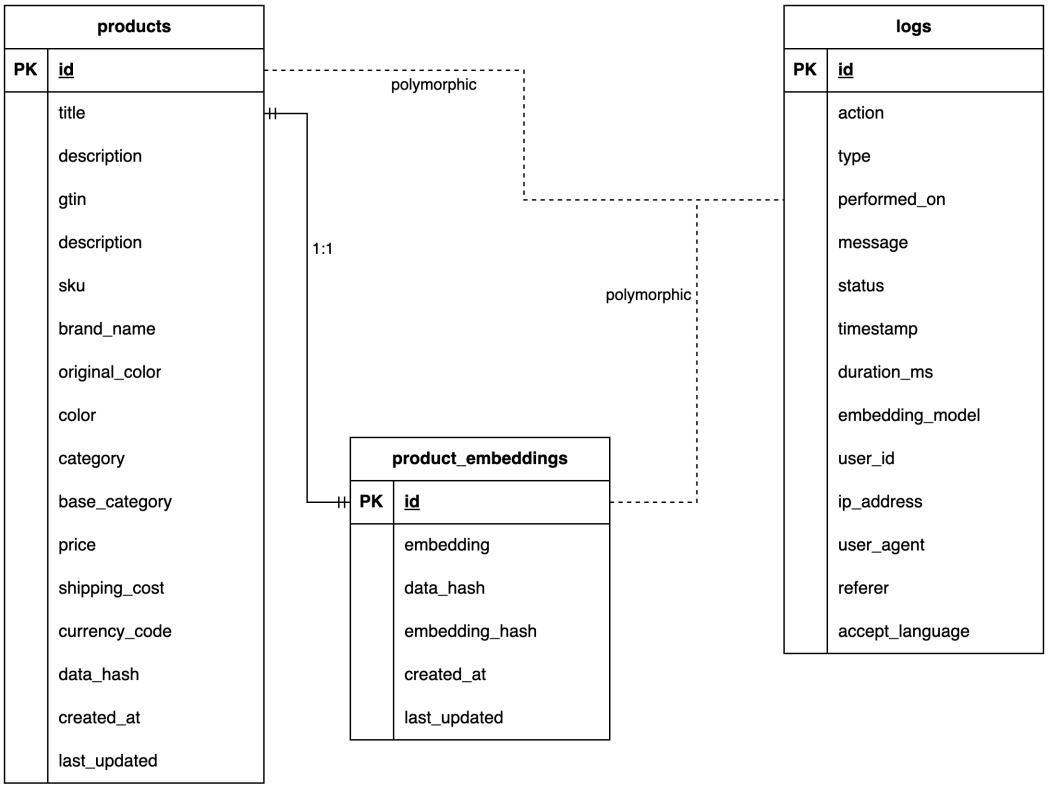


Abbildung 4.2: ER-Diagramm

4.4.5 Datenflüsse

Auf Basis der konzipierten Services und des Datenmodells lassen sich nun Datenflüsse visualisieren, die uns helfen zu verstehen, in welchen Prozessen und Schritten Daten abgefragt, synchronisiert und modifiziert werden.

data-loader

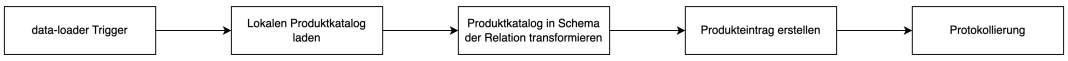


Abbildung 4.3: data-loader

embedding-generator

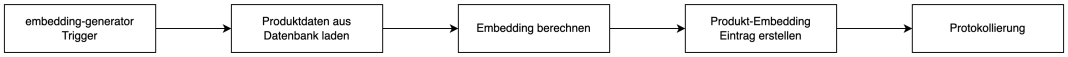


Abbildung 4.4: embedding-generator

external-product-sync

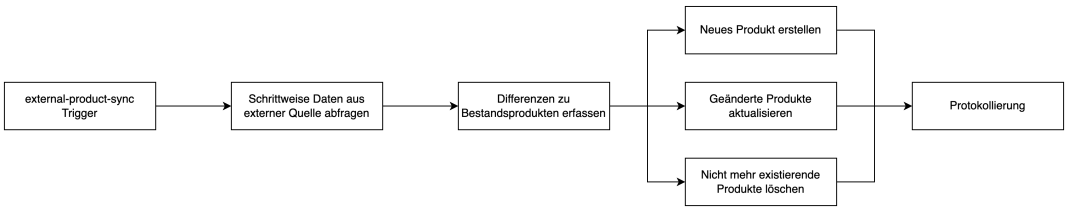


Abbildung 4.5: external-product-sync

internal-product-sync

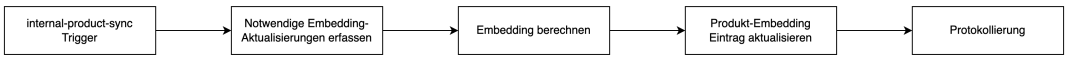


Abbildung 4.6: internal-product-sync

api

Wie bereits erwähnt, wurden in dieser Arbeit zwei Anwendungsfälle für den Einsatz einer semantischen Suche konzipiert:

- Natürliche Sprache zu Vektorrepräsentation
- Vektorrepräsentation zu Vektorrepräsentation

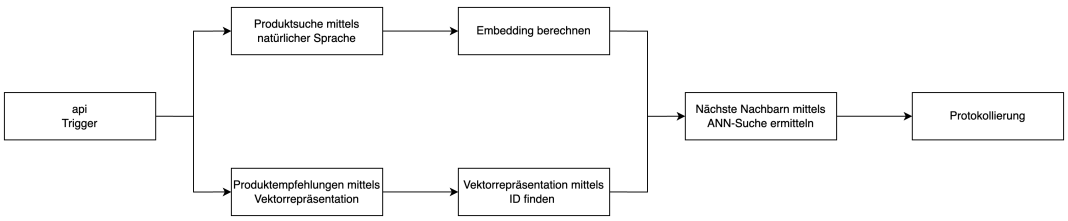


Abbildung 4.7: api

4.5 Evaluationsmethoden und Metriken

Die Konzeption über die Verwendung der Embedding-Modelle, der Datenmodellierung und Systemarchitektur verliert ihre Bedeutung, wenn die Daten, die unser System liefert, nicht gültig und fundiert sind. Die Funktion unseres Systems lässt sich nicht auf ein binäres *funktioniert* oder *funktioniert nicht* reduzieren – vielmehr ist es notwendig, Metriken zu implementieren, welche die semantische Güte unserer Ergebnisse validieren können. In diesem Abschnitt wird auf die konzeptionellen Grundlagen dieser Metriken eingegangen und was notwendig ist, um die Bereitstellung dieser Metriken implementieren zu können.

4.5.1 Einschränkungen

Um die Leistungsfähigkeit von Recommendation- und Ranking-Systemen zu optimieren, existieren verschiedene Ansätze auf unterschiedlichen Ebenen: Auf Systemebene durch Wahl der Programmiersprache, Parallelisierung und Hardware-Optimierung; auf Infrastrukturebene durch Indexoptimierung (z.B. HNSW-Parameter wie `M`, `efConstruction` und `efSearch` in pgvector); sowie auf Modellebene durch domänenspezifisches Fine-Tuning der Embedding-Modelle oder Dimensionalitätsreduktion mittels PCA oder UMAP.

In dieser Arbeit fokussiere ich mich bei der Evaluation nur auf den Vergleich der Leistung verschiedener vortrainierter Transformer-Modelle für die Verwendung in meiner semantischen Produktsuche.

4.5.2 Evaluierung der technischen Leistungsparameter

Bevor wir uns Evaluierungsmetriken für die semantische Qualität mittels Beurteilung eines binären Klassifikators widmen, bietet es sich an, zuerst die technischen Leistungsparameter unseres Systems zu betrachten. Die Qualität unserer Suchergebnisse kann noch so gut sein, führen die Implementierung der technischen Infrastruktur zu hohen Reaktionszeiten, so ist dieses System im E-Commerce-Kontext nicht verwertbar.

Eine Metrik, welche sich hierfür verwenden lässt, ist beispielsweise *Average Response Time (ART)*, welche folgend ermittelt wird:

$$\text{Average Response Time} = \frac{\text{Total time taken to respond}}{\text{Total number of responses}}$$

Diese Metrik lässt sich also verwenden, um die durchschnittliche Reaktionszeit zwischen einer Anfrage und einer Antwort zu ermitteln. Für unser System können wir diese Berechnung auf verschiedene Komponenten projizieren:

- Durschnittliche Reaktionszeit zwischen einer Suche in natürlicher Sprache und der Rückgabe der Ergebnisse
- Durschnittliche Reaktionszeit zwischen einer automatisierten Suche nach ähnlichen Produkten und der Rückgabe der Ergebnisse
- Durchschnittliche Dauer des erstmaligen Erstellens eines Embeddings
- Durschnittliche Dauer des aktualisieren eines Embeddings

Die Werte, die mittels dieser Metrik berechnet werden können, bieten bereits einen guten Einblick in die Leistung des implementierten Systems.

4.5.3 Evaluierung der semantischen Qualität der auf Vektorsuche basierenden Ergebnisse

Ein technisch so optimiertes System, das in der Lage ist, Suchanfragen im Millisekundenbereich zu beantworten, ist sicherlich attraktiv und anstrebenswert, hat allerdings keinen Wert, wenn die Ergebnisse der Suchanfragen nicht relevant sind. Parallel zu den im vorherigen beschriebenen Leistungsparametern benötigt es also auch Metriken, die die tatsächliche semantische Qualität der Suchergebnisse bewerten können, also Antworten auf Fragen wie:

- Wie viele der Suchergebnisse sind tatsächlich Relevant zu einer Suchanfrage?

- Wie hoch ist die Anzahl der relevanten Suchergebnisse aus der Menge aller für diese Suchanfrage relevanten Daten?
- Sind die als relevant bewerteten Suchergebnisse in ihrer Relevanz korrekt sortiert?

Die Antworten auf diese Fragen bieten Ranking- und Recommendations-Metriken [19, 20, 21, 22]. Dazu zählen beispielsweise:

- Precision@K
- Recall@K
- F1- und F-Beta-Score
- MAP@K
- MRR
- R-squared Score
- Pearson Coefficient
- Spearman Coefficient
- NDCG
- Kendall Tau

Es ist anzumerken, dass im Rahmen dieser Arbeit nicht alle der genannten Metriken implementiert werden, sondern sich auf eine Auswahl dieser beschränkt wurde. Dementsprechend wird auch nur auf diese im Folgenden genauer eingegangen.

Precision@K

Precision@K beschreibt das Verhältnis der korrekt identifizierten relevanten Elemente innerhalb der gesamten empfohlenen Elemente innerhalb der K-langen Liste. Einfach ausgedrückt, ermittelt P@K, wie viele der abgerufenen Elemente wirklich relevant sind.

$$\text{Precision at } K = \frac{\text{Number of relevant items in } K}{\text{Total number of items in } K}$$

Precision@K wird berechnet, indem die Anzahl der relevanten Elemente innerhalb der Top-k-Empfehlungen durch K dividiert wird.

Recall@K

Recall@K misst den Anteil der korrekt identifizierten relevanten Elemente in den Top-k-Empfehlungen in Relation zur Gesamtzahl der relevanten Elemente im Datensatz.

$$\text{Recall at } K = \frac{\text{Number of relevant items in } K}{\text{Total number of relevant items}}$$

Recall@K wird berechnet, indem die Anzahl der relevanten Elemente innerhalb der Top-k-Empfehlungen durch die Gesamtzahl der relevanten Elemente im gesamten Datensatz dividiert wird. Diese Metrik misst die Fähigkeit des Systems, alle relevanten Elemente in einem Datensatz abzurufen.

F-Beta-Score

Wie wir nun bereits erfahren haben, erfasst Precision@K wie genau das System bei seinen Empfehlungen ist, während Recall@K versucht, alle relevanten Elemente zu erfassen.

Der F-Beta-Score@K kombiniert diese beiden Metriken zu einem einzigen Wert und ermöglicht so eine ausgewogene Bewertung. Der Beta-Parameter erlaubt es, die Gewichtung von Recall relativ zu Precision anzupassen.

Hierbei gilt:

- Beta > 1: Priorisiert Recall
- Beta < 1: Bevorzugt Precision
- Beta $= 1$: Klassischer F1-Score, bei dem Precision und Recall gleichermaßen gewichtet ist

$$F_\beta = \frac{(1 + \beta^2) \times \text{Precision at K} \times \text{Recall at K}}{(\beta^2 \times \text{Precision at K}) + \text{Recall at K}}$$

MAP@K

MAP@K (Mean Average Precision) ist eine Qualitätsmetrik, die bewertet, wie gut ein Ranking- oder Empfehlungssystem relevante Elemente in den Top-K Ergebnissen liefert, wobei relevantere Elemente höher gewichtet werden, je weiter oben sie in der Rangliste stehen. Die Formel hierfür lautet:

$$\text{MAP@K} = \frac{1}{U} \sum_{u=1}^{U} \text{AP@K}_u$$

Parameter:

- K: Der gewählte Schwellenwert (Anzahl betrachteter Ergebnisse)
- U: Gesamtanzahl der Nutzer (bei Empfehlungssystemen) oder Suchanfragen (bei Informationsabruf) im evaluierten Datensatz
- AP: Average Precision für eine gegebene Rangliste

Im Gegensatz zu Precision@K und Recall@K berücksichtigt MAP@K explizit die Position relevanter Elemente in der Rangliste – ein relevantes Produkt auf Position 1 wird höher bewertet als dasselbe Produkt auf Position 10.

NDCG

Normalized Discounted Cumulative Gain (NDCG) ist eine Qualitätsmetrik für Rankings. Sie vergleicht Ranglisten mit einer idealen Reihenfolge, in der alle relevanten Elemente an der Spitze der Liste stehen. Wie auch bei den bereits behandelten Ranking-Metriken wird NDCG oft "@K"berechnet, wobei ein Schwellenwert für die betrachteten Top-Ergebnisse definiert wird.

- DCG misst die Gesamtrelevanz der Elemente in einer Liste mit einem Abschlag (Discount), der den abnehmenden Wert von Elementen weiter unten in der Liste berücksichtigt.
- NDCG-Werte liegen zwischen 0 und 1, wobei 1 eine Übereinstimmung mit der idealen Reihenfolge anzeigt und niedrigere Werte eine geringere Ranking-Qualität repräsentieren.

NDCG@K wird folgendermaßen berechnet:

$$\text{NDCG@K} = \frac{\text{DCG@K}}{\text{IDCG@K}}$$

Wobei DCG@K folgendermaßen berechnet wird:

$$\text{DCG@K} = \sum_{i=1}^{K} \frac{\text{rel}_i}{\log_2(i+1)}$$

IDCG@K sind die K-ersten Elemente einer idealen Reihenfolge der Elemente, gegen die ein Suchverfahren bzw. Empfehlungsalgorithmus ausgewertet wird.

4.6 Technologie-Evaluation und -Auswahl

Die Implementierung einer semantischen Produktsuche für große Datenmengen stellt spezialisierte Anforderungen an die technische Infrastruktur. Wie wir bereits wissen, müssen Vektordatenbanken Embeddings in Form hochdimensionaler Vektoren speichern.

Um eine nachvollziehbare und übertragbare Grundlage für technische Entscheidungen zu treffen, wird im Folgenden ein grobes Framework entwickelt, um entsprechende Technologieoptionen anhand objektiver Kriterien zu evaluieren.

4.6.1 Datenbankstrategie und Legacy-Intergration

Die erste Frage betrifft das bereits vorhandene Datenbanksystem. Verschiedene Szenarien erfordern unterschiedliche Strategien:

Bestehendes Datenbanksystem mit nativer Unterstützung

Einige Datenbanksysteme bieten bereits nativ Unterstützung für die Verwendung, beispielsweise MongoDB Atlas Vector Search. Der Vorteil liegt hier am minimalen Migrationsaufwand und der Förderung einer einheitlichen Infrastruktur. Es sollte allerdings berücksichtigt werden, dass die Implementierung eventuell nicht für alle Anwendungsfälle passend ist.

Bestehendes Datenbanksystem mit nativer Extension-Möglichkeiten

Auch wenn das verwendete Datenbanksystem keine Vektoren nativ unterstützt, existieren möglicherweise Erweiterungen, die dies möglich machen. Beispiele hierfür sind PostgreSQL in Kombination mit pgvector oder, wenn auch weniger ausgereift, MySQL mit mysql_vss.

Keine Unterstützung für Vektoren im Bestandssystem

Für den Fall, dass das aktuell verwendete Datenbanksystem keine Unterstützung für Vektoren bietet, ergeben sich grob zusammengefasst 3 Optionen:

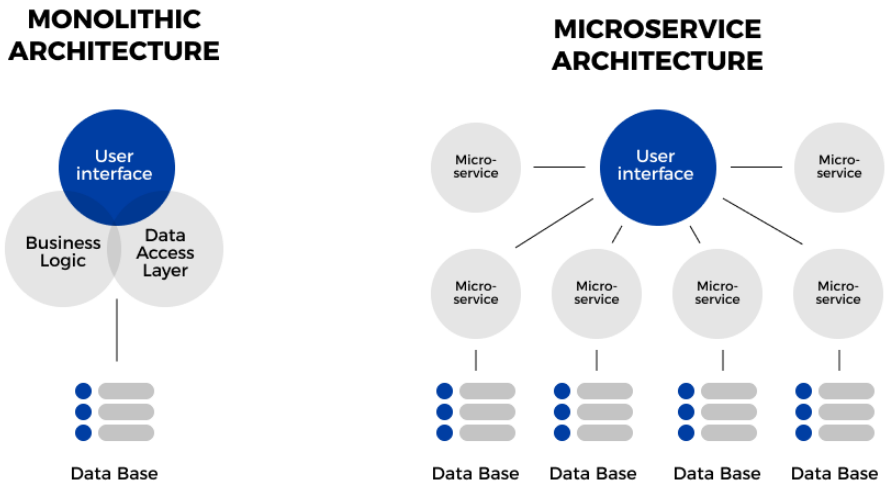
1. **Vollmigration**
   - Migration auf spezialisierte Vektor-Datenbank wie Milvus oder Qdrant
   - Vorteile: Optimale Performance durch spezialisierte Vektordatenbanken
   - Nachteile: Hoher Migrationsaufwand, möglicher Verlust bewährter Features
2. **Hybride Architektur**
   - Produktdaten bleiben in bestehender Datenbank, Embeddings in separater Vektordatenbank
   - Vorteile: Minimale Disruption bestehender Systeme
   - Nachteile: Daten-Synchronisation, Komplexität bei Relationen zwischen Daten und Embeddings, erhöhte Latenz
3. **Managed Vector-Services**
   - Anbieter wie Pinecone, Weaviate Cloud, GCP Vertex AI Vector Search
   - Vorteile: Minimaler Betriebs-Overhead, automatische Skalierung
   - Nachteile: Vendor Lock-in, Daten-Residency, laufende (deutlich höhere) Kosten

Monolithische Integration

Als monolithische Architektur versteht man das herkömmliche Modell einer Anwendung, die als eine komplette Einheit implementiert wurde, die in sich geschlossen und unabhängig von anderen Anwendungen ist. Ein Monolith bündelt dabei alle geschäftlichen Belange in sich [23].

Microservice

Im Gegensatz dazu ist ein Microservice eine Architektur, bei der eine Reihe unabhängig bereitgestellter Services über ihre eigene Geschäftslogik verfügen und jeweils ein bestimmtes Belangen erfüllen. Aktualisierungen, Tests, Deployments und Skalierungen erfolgen separat in jedem einzelnen Service. [23]



Quelle: [24]

Abbildung 4.8: Monolith und Microservice

4.6.2 Programmiersprachen-Ökosystem

Die Wahl der Technologie für Embedding-Generierung ist stark abhängig vom bestehenden Tech-Stack bzw. der Programmiersprache. Betrachten wir einige Beispiele:

Python-Ökosystem

- Sentence-Transformers, Transformers (HuggingFace), pyTorch, spaCy
- Vorteile: Größtes ML-Ökosystem, neueste Modelle verfügbar
- Nachteile: Performance bei hohem Durchsatz, GIL-Limitierungen

Cross-Platform Ansätze

- ONNX Runtime
- Vorteile: Konsistenz mit bestehendem Stack
- Nachteile: Begrenztes Modell-Angebot, Konvertierungs-Overhead

Cloud-basierte Embedding-Services

- OpenAI Embeddings API, Cohere, Google Vertex AI
- Keine lokale ML-Infrastruktur, neueste Modelle
- Nachteile: API-Kosten, Latenz, Datenschutz

5 Implementierung

In diesem Kapitel wird die technische Umsetzung der semantischen Produktsuche detailliert erläutert. Die Implementierung erfolgte als eigenständiger Microservice in Python, der über definierte Schnittstellen mit der bestehenden Kotlin-basierten donista-Plattform interagiert.

5.1 Systemarchitektur und Design

5.1.1 Erweiterung durch Microservice:

Die semantische Produktsuche wurde als eigenständiger Microservice konzipiert, der:

- Unabhängig vom Hauptsystem deployed werden kann
- Über REST-APIs mit dem Monolithen kommuniziert
- Die gleiche PostgreSQL-Instanz nutzt
- Horizontal skalierbar ist

5.1.2 Technologieentscheidung

Bevor wir genauer auf die Implementierung eingehen, sollten wir betrachten, wieso die semantische Produktsuche in Python umgesetzt wurde, und nicht, wie der Rest des donista-Systems, in Kotlin. Die Wahl von Python als Implementierungssprache basierte auf mehreren technischen und praktischen Überlegungen.

Machine Learning Ökosystem

Python bietet das ausgereifteste Ökosystem für Machine Learning und NLP-Anwendungen:

- Sentence-Transformers: Native Python-Bibliothek mit direktem Zugriff auf vortrainierte Modelle
- Hugging Face Integration: Nahtlose Integration mit dem Hugging Face Model Hub
- NumPy/SciPy: Effiziente Vektoroperationen ohne zusätzliche Abstraktionsschichten

Vermeidung von Modellkonvertierung

Bei einer Kotlin-Implementierung wären folgende zusätzliche Schritte notwendig gewesen:

- Konvertierung der PyTorch-Modelle nach ONNX
- Integration einer ONNX-Runtime in Kotlin, beispielsweise KInference
- Potentielle Kompatibilitätsprobleme und Performance-Einbußen
- Erschwerter Modellwechsel bei Updates

Entwicklungsgeschwindigkeit

- Rapid Prototyping durch High-Level-Abstraktion
- Umfangreiche Dokumentation und Community-Support
- Theoretisch dynamischer Wechsel zwischen verschiedenen Modellen ohne großen Aufwand

Microservice-Architektur

Für Microservices gilt eigentlich das Prinzip „require nothing“ – alles, was zum Betrieb des Microservice notwendig ist, sollte im Microservice selbst enthalten sein. Während die semantische Suche zwar ihr eigenes Modell und Schema hat, welches unabhängig von der restlichen Infrastruktur existiert, ist es jedoch geplant, auf die gleiche Datenbank wie der Rest der donista-Anwendung zuzugreifen.

5.1.3 Technologie-Stack

Als Referenz für die weitere Erläuterung der Implementierung wird hier kurz darauf eingegangen, welche Technologien in der Implementierung verwendet werden:

- PostgreSQL – Relationale Datenbank
- pgvector – Vector similarity search für PostgreSQL
- Python – Programmiersprache
- NumPy – Python-Bibliothek für Verarbeitung mehrdimensionaler Arrays

- sentence-transformers – Python-Bibliothek für die einfache Verwendung von Embedding Modellen
- SQLAlchemy – ORM
- FastAPI – Webframework zur Erstellung von (HTTP-basierten) APIs mit Python
- Uvicorn – Webserver für Python
- Docker – Containervirtualisierung
- pytest – Testframework für Python

5.1.4 Entscheidung für pgvector als Vektordatenbank

Wie mehrfach erwähnt wurde, beruht die sonstige Infrastruktur von donista auf PostgreSQL-Datenbanken. Mit der Wahl von pgvector kann so, auch wenn der Microservice in einer anderen Programmiersprache als der Rest der Infrastruktur implementiert ist, eine nahtlose Integration in die bestehende Datenbankumgebung umgesetzt werden und verringert so die Systemkomplexität. Auch wenn dies grundsätzlich ein Microservice-Prinzip verletzt, bietet pgvector mit PostgreSQL einen entscheidenden Vorteil. Andere Komponenten der donista-Infrastruktur können auf eine zentrale Produkt-Relation zugreifen, ohne dass diese aufwendig zwischen verschiedenen Datenbanksystemen synchron gehalten werden muss.

5.2 Definition der Datenmodelle

Im Kapitel Konzeption und Methodik 4 schon erfahren, welche Relationen in unserer Anwendung relevant sein werden. In der Anwendung werden die Objekte als Klassen unter Verwendung des ORM `SQLAlchemy` [25, 26] modelliert. Unter Berücksichtigung der Konzeption und Anforderungsanalyse benötigen wir also:

- Das `product`-Modell, welches alle direkt Produkt-relevanten Daten enthält
- Das `product_embdding`-Modell, welches das zum Produkt gehörende Embedding speichert
- Ein `log`-Modell, zur Protokollierung aller relevanten Datenbanktransaktionen

Zusätzlich zu diesen Modellen werden im späteren Verlauf noch weitere Modelle zur Implementierung von Ranking- und Recommendations-Metriken benötigt; diese werden in ihrem relevanten Abschnitt genauer erläutert.

Bevor wir die Modelle selbst implementieren können, benötigen wir als Anforderung von SQLAlchemy eine gemeinsame Basisklasse `declarative_base()` [27] von der alle Modelle erben können. Dies wird folgendermaßen umgesetzt:

Listing 5.1: Definition der gemeinsamen Basisklasse in app/models/base.py

```
"""SQLAlchemy Base definition"""
from sqlalchemy.orm import declarative_base
Base = declarative_base()
```

Nun können wir unter Verwendung dieser Basisklasse das erste Modell erstellen, hier am Beispiel des `product`-Modells erläutert:

Listing 5.2: Definition des Product-Modell in app/models/product.py

```
"""Product model definition"""

from sqlalchemy import Column, DateTime, Float, String, Text

from app.models.base import Base


class Product(Base):
    """Product model representing a product in the database"""

    __tablename__ = "products"

    id = Column(String(255), primary_key=True)
    title = Column(Text)
    description = Column(Text)
    gtin = Column(String(255))
    sku = Column(String(255))
    brand_name = Column(String(255))
    original_color = Column(String(255))
    color = Column(String(255))
    category = Column(String(255))
    base_category = Column(String(255))
    price = Column(Float)
    shipping_cost = Column(Float)
    currency_code = Column(String(3))
    data_hash = Column(String(32))
    created_at = Column(DateTime)
    last_updated = Column(DateTime)
```

```
29
30      def __repr__(self):
31          return f"<Product(id='{self.id}', title='{self.title[:20]}...')>"
```

5.3 Besonderheiten des ProductEmbedding-Modells

Interessant ist es, noch einen Blick auf die Definition des Modells zu werfen, welche die Vektorrepräsentationen enthalten wird. Zweck dieser Arbeit ist es unter anderem auch, qualitative und quantitative Leistungsparameter zu erfassen und zu vergleichen. Hierfür ist es notwendig, parallel verschiedene Embedding-Modelle benutzen zu können. Hierfür wurde ein flexibel skalierbares System zur dynamischen Erweiterung durch verschiedene Embedding-Modelle implementiert.

Hier wurde eine abstrakte Basisklasse `ProductEmbeddingBase` implementiert, welche alle Attribute enthält, die über alle spezifischen Produkt-Embedding-Relationen gleich sein werden:

Listing 5.3: Definition der Basisklasse in app/models/embedding/ProductEmbeddingBase.py

```
1  """Product embedding base model definition"""
2
3  from sqlalchemy import Column, DateTime, ForeignKey, String
4
5  from app.models.base import Base
6
7
8  class ProductEmbeddingBase(Base):
9      """Abstract base class for all embedding models"""
10
11     __abstract__ = True
12
13     id = Column(String(255), ForeignKey("products.id"), primary_key=True)
14     data_hash = Column(String(32))
15     embedding_hash = Column(String(32))
16     created_at = Column(DateTime)
17     last_updated = Column(DateTime)
18
19     def __repr__(self):
20         return f"<ProductEmbedding(id='{self.id}')>"
```

Unter Verwendung einer Fabrikmethode `create_embedding_model` werden unter Verwendung der abstrakten Basisklasse dann die tatsächlichen Relationen mit Verwendung der spezifischen Anforderungen des Embedding-Modells erstellt:

Listing 5.4: Definition der Fabrikmethode in app/models/embedding/embedding_factory.py

```python
def create_embedding_model(table_name: str, dimensions: int):
    """Dynamically creates an embedding model class using type()."""

    # Sanitize table name for SQL compatibility
    clean_table_name = sanitize_table_name(table_name)

    if clean_table_name != table_name:
        print(f"Sanitized table name: '{table_name}' -> '{clean_table_name}'")

    # Clean table name for class name (remove special characters)
    class_name = f"ProductEmbedding_{clean_table_name}"

    # Define class attributes
    class_attrs = {
        "__tablename__": clean_table_name,
        "__module__": __name__,
        "embedding": Column(Vector(dimensions)),
        # Store original config for reference
        "_original_table_name": table_name,
        "_dimensions": dimensions,
        # Override the repr to include the table name
        "__repr__": lambda self: f"<{class_name}(id='{self.id}')>",
    }

    # Create the class using type()
    ProductEmbeddingClass = type(
        class_name, (ProductEmbeddingBase,), class_attrs
    )

    return ProductEmbeddingClass
```

Wie wir sehen, wird für das Attribut `embedding` kein konventioneller Datentyp von SQLAlchemy bzw. PostgreSQL genutzt, sondern ein spezieller Datentyp `Vector`, welcher von `pgvector` exportiert wird und als Parameter die Dimensionalität akzeptiert.

5.4 Konfiguration der Datenbank-Schnittstelle

Als nächstes wird die Verbindung zur Datenbank selbst benötigt. Diese wurde ebenfalls mithilfe von `SQLAlchemy` implementiert und stellt zwei zentrale Funktionen bereit (siehe Listing 5.5):

Die Funktion `get_engine()` erstellt und konfiguriert eine `SQLAlchemy Engine` [28], welche als zentrale Schnittstelle zur Datenbank fungiert [28]. Die Engine verwaltet den

Connection Pool und stellt die grundlegende Infrastruktur für alle Datenbankoperationen zur Verfügung. Sie wird primär für Schema-Management-Operationen wie das Erstellen von Tabellen verwendet.

Die Funktion `get_session()` hingegen erstellt Session-Objekte [29], die als Arbeitskontext für ORM-Operationen dienen [29]. Sessions verwalten Transaktionen und ermöglichen es, Datenbankoperationen wie das Hinzufügen, Aktualisieren oder Abfragen von Objekten durchzuführen. Jede Session repräsentiert eine logische Arbeitseinheit mit der Datenbank.

Beide Funktionen implementieren umfassende Fehlerbehandlung für häufige Verbindungsprobleme wie fehlende Treiber, ungültige Connection Strings oder nicht erreichbare Datenbankserver. Die Datenbankverbindungsparameter werden dabei über Umgebungsvariablen konfiguriert.

Listing 5.5: Definition des Datenbank-Schnittstelle in app/database/engine.py

```
"""Database connection setup"""

import os
import sys

from dotenv import load_dotenv
from sqlalchemy import create_engine
from sqlalchemy.exc import ArgumentError, NoSuchModuleError, OperationalError
from sqlalchemy.orm import sessionmaker

load_dotenv()
CONNECTION_STRING = os.environ.get("CONNECTION_STRING")


def get_engine():
    """Connect to the database using the connection string from environment variables"""
    if not CONNECTION_STRING:
        print(
            "\033[91m> Error:\033[0m Database connection string not found "
            "in environment variables."
        )
        print(
            "> Please create a .env file with CONNECTION_STRING or set it "
            "in your environment."
        )
        print(
            "> Example: CONNECTION_STRING=postgresql://username:password@"
            "localhost:5432/vectordb"
        )
```

```
30          sys.exit(1)
31
32      try:
33          return create_engine(CONNECTION_STRING)
34      except NoSuchModuleError as e:
35          print(f"\033[91mError:\033[0m Missing database driver: {e}")
36          sys.exit(1)
37      except ArgumentError as e:
38          print(f"\033[91mError:\033[0m Invalid database connection string: {e}")
39          sys.exit(1)
40      except OperationalError as e:
41          print(
42              f"\033[91mError:\033[0m Could not connect to database server: {e}"
43          )
44          sys.exit(1)
45
46
47  def get_session():
48      """Create a new session for database operations"""
49      engine = get_engine()
50      session = sessionmaker(bind=engine)
51      return session()
```

5.5 Implementierung der Repository-Klassen

Mit den definierten `SQLAlchemy`-Modellen und der konfigurierten Datenbankschnittstelle sind die grundlegenden Komponenten für den Datenzugriff vorhanden. Für die Implementierung der konkreten Datenzugriffsmethoden orientieren wir uns am Repository Pattern.

Das Repository Pattern [30, 31] fungiert als Vermittlungsschicht zwischen der Geschäftslogik und der Datenzugriffsschicht. Es abstrahiert die Datenzugriffe durch eine einheitliche Schnittstelle und kapselt die Komplexität der zugrundeliegenden Persistierungstechnologie. Repositories stellen eine Kollektion von Domain-Objekten dar, als würden diese sich im Arbeitsspeicher befinden, wodurch komplexe Datenbankoperationen hinter einer verständlichen API verborgen werden.

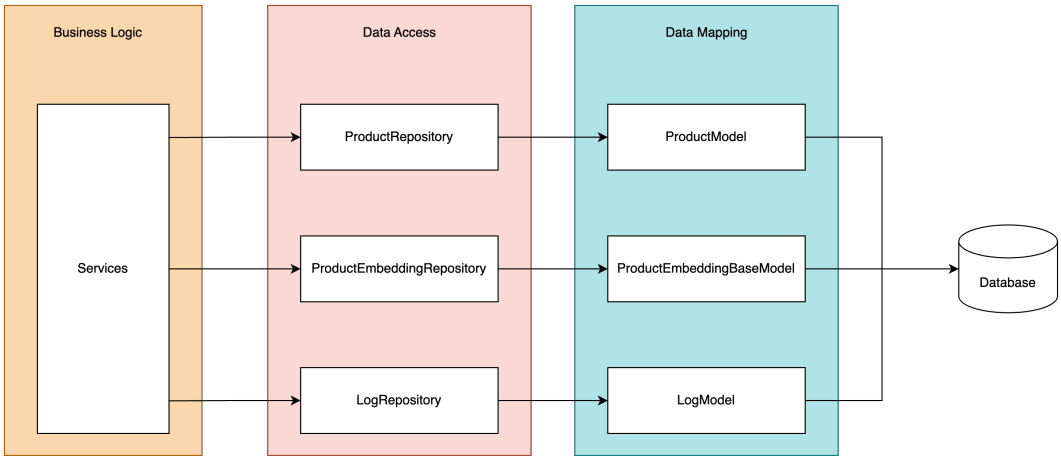


Abbildung 5.1: Anwendung des Repository-Pattern

5.6 Protokollierung von Systemoperationen

Entsprechend den in Abschnitt 4 definierten Anforderungen implementiert die Anwendung ein umfassendes Logging-System zur Nachverfolgbarkeit aller durchgeführten Aktionen. Die Implementierung der `logs`-Relation weist dabei eine besondere architektonische Eigenschaft auf, die sich von den klassischen Entity-Beziehungen unterscheidet.

5.6.1 Polymorphe Referenzierung in der Log-Architektur

Während die Beziehung zwischen `products` und `product_embeddings` über klassische Fremdschlüssel-Constraints realisiert wird, verwendet die `logs`-Tabelle ein polymorphes Referenzierungsschema. Anstelle dedizierter Foreign-Key-Beziehungen zu jeder referenzierten Entität wird das Attribut `performed_on` als generische String-Referenz implementiert.

Diese Designentscheidung ermöglicht es, Logging-Einträge für beliebige Entitäten zu erstellen, ohne die Datenbankstruktur bei jeder neuen loggbaren Entität erweitern zu müssen. Das `performed_on`-Feld kann dabei Primärschlüssel von Produkten, Embeddings oder anderen Geschäftsobjekten enthalten, wobei der Kontext durch die kombinierten Attribute `action` und `type` bestimmt wird.

5.6.2 Kategorisierung von Log-Einträgen

Das Logging-System implementiert eine hierarchische Kategorisierung durch Enumerationen:

- **LogAction:** Definiert übergeordnete Aktionskategorien (`API_REQUEST`, `EMBEDDING_GENERATION`, `DATABASE_OPERATION`, `SYNC`)
- **LogType:** Spezifiziert den konkreten Operationstyp (`SEARCH_QUERY`, `INSERT`, `UPDATE`)
- **LogStatus:** Kategorisiert das Operationsergebnis (`SUCCESS`, `ERROR`, `WARNING`, `INFO`)

Diese strukturierte Kategorisierung ermöglicht, effizientes Filtern und Analyse der Log-Einträge für verschiedene Anwendungsfälle wie Performance-Monitoring, Error-Tracking oder Audit-Trails zu implementieren.

Implementierung des `log`-Modells

Listing 5.6: Definition des log-Modells in app/models/log.py

```python
"""Log model definition"""

import enum
from app.models.base import Base
from sqlalchemy import Column, DateTime, Integer, String, Text


class LogAction(enum.Enum):
    API_REQUEST = "api_request"
    SYNC = "sync"
    EMBEDDING_GENERATION = "embedding_generation"
    DATABASE_OPERATION = "database_operation"


class LogType(enum.Enum):
    # API request types
    SEARCH_QUERY = "search_query"

    # Sync types
    PRODUCT_SYNC = "product_sync"

    # Embedding generation types
    EMBEDDING_GENERATION = "embedding_generation"
    EMBEDDING_UPDATE = "embedding_update"

```

```
    # Database operation types
    INSERT = "insert"
    UPDATE = "update"


class LogStatus(enum.Enum):
    SUCCESS = "success"
    ERROR = "error"
    WARNING = "warning"
    INFO = "info"


class Log(Base):
    """Log model representing a log entry in the database"""

    __tablename__ = 'logs'

    id = Column(String(255), primary_key=True)
    # high-level action (api_request, sync, embedding_generation)
    action = Column(String(255))
    type = Column(String(255)) # specific type of the action
    # entity/object ID the action is performed on
    performed_on = Column(String(255))
    # detailed information
    message = Column(Text)
    # success/error/warning
    status = Column(String(255))
    timestamp = Column(DateTime)
    # If you have user authentication
    user_id = Column(String(255), nullable=True)
    duration_ms = Column(Integer, nullable=True) # For performance tracking
    ip_address = Column(String(255), nullable=True) # For API requests
    user_agent = Column(String(255), nullable=True) # For API requests
    referer = Column(String(255), nullable=True) # For API requests
    accept_language = Column(String(255), nullable=True) # For API requests

    def __repr__(self):
        return f"<Log(id='{self.id}', action='{self.action}', type='{self.type}', status='{
            self.status}')>"
```

5.6.3 Vorteile und Trade-offs des polymorphen Designs

Das gewählte polymorphe Referenzierungsschema bietet mehrere Vorteile:

- **Flexibilität:** Neue Entitätstypen können ohne Schema-Änderungen geloggt werden, was die Erweiterbarkeit des Systems erhöht.
- **Vereinfachte Datenbankstruktur:** Es entfallen multiple Foreign-Key-Beziehungen und die damit verbundene Komplexität von NULL-Werten für nicht-relevante Referenzen.
- **Einheitliche Log-API:** Der `LoggingService` kann universell für alle Entitätstypen verwendet werden, ohne typspezifische Implementierungen.

Implikationen

Als Trade-off wird jedoch die referenzielle Integrität auf Datenbankebene aufgegeben, wodurch die Konsistenz primär durch die Anwendungslogik gewährleistet werden muss.

5.7 Service-Flows

Statt detailliert jede Klasse und Methode zu erläutern, ist es deutlich sinnvoller, wie die Abläufe der einzelnen Services strukturiert sind und welche Prozesse diese auslösen. Diese lassen sich grob in 3 Kategorien unterteilen:

- Aufsetzen/Setup der Anwendung
- Datensynchronisation
- Bereitstellung der Funktionen

5.7.1 Setup: app/initialize.py

Im ersten Schritt wird die pgvector-Extension in PostgreSQL aktiviert und alle Relationen in der Datenbank erstellt. Da dieser Prozess zu komplex ist um hier den kompletten Code dafür darzustellen, wird der Ablauf im Folgenden in einem Diagramm dargestellt.

Alle Embedding-Modelle, die verwendet werden sollen, werden in einer Konfigurationsdatei formuliert:

Listing 5.7: Konfiguration in app/config/embeddings.yml

```
# Embedding models configuration

# Use comments to select the active model
active_model: "sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2"
#active_model: "Alibaba-NLP/gte-multilingual-base"
#active_model: intfloat/multilingual-e5-base

# Models to generate tables for
models:
  - name: "sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2"
    dimensions: 384
    table_name: "product_embeddings_minilm"
  - name: "Alibaba-NLP/gte-multilingual-base"
    dimensions: 768
    table_name: "product_embeddings_gte"
  - name: "intfloat/multilingual-e5-base"
    dimensions: 768
    table_name: "product_embeddings_e5"
```

Hier werden alle Embedding-Modelle festgelegt, für die bei Initialisierung eine Relation mittels Vererbung aus der bereits angesprochenen abstrakten Basisklasse `ProductEmbeddingBase` und der Fabrikmethode `create_embedding_model`.

- `name`: Der Name des Embeddings-Modells.
- `dimension`: Die Dimensionen der durch das Modell erstellten Vektorrepräsentationen. Diese wird für den `Vector`-Datentyp in der Relation benötigt.
- `table_name`: Hier kann direkt der Name festgelegt werden, mit der die Relation in der Datenbank erstellt werden soll.

Prozessablauf bei Initialisierung

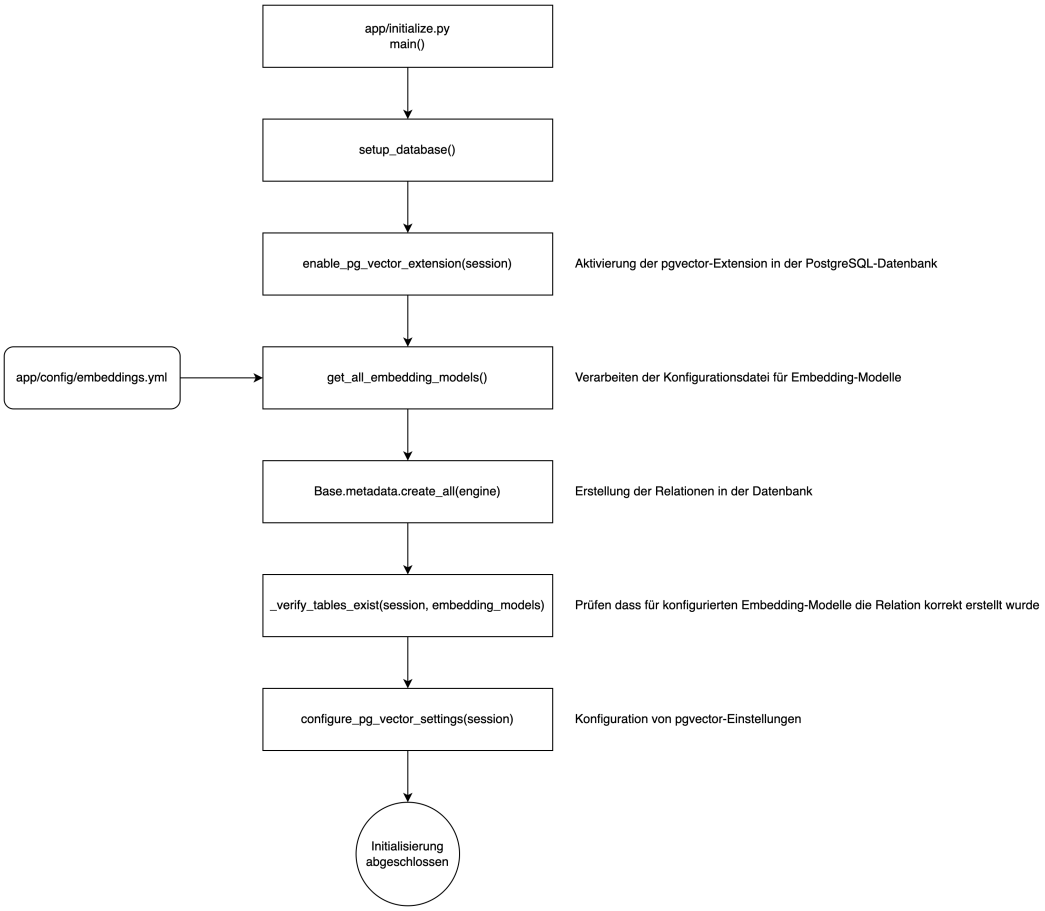


Abbildung 5.2: Prozessablauf bei Initialisierung

5.7.2 Setup: app/load_data.py

Der `load_data.py`-Service implementiert die initiale Datenakquisition aus externen JSONL-Dateien und stellt damit den Einstiegspunkt der gesamten Datenverarbeitungspipeline dar. Der Service führt die Grundbefüllung der Produktdatenbank durch, also die Relation `products`.

Wie bereits erwähnt, basiert der Datensatz auf dem Produktkatalog von shopping24. Die shopping24 Gesellschaft für multimediale Anwendungen mbH ist die Betreibergesellschaft mehrerer Produktsuchmaschinen und Shopping-Portale. Außer shopping24.de gehören die Shopping-Portale fashion24, smatch, yalook, living24 und discount24 zur shopping24 internet group.

Um auf Daten des Produktkatalogs zuzugreifen, stellt shopping24 eine REST-API bereit. Um für die erstmalige Befüllung der `products`-Relation nicht an API-Limits oder Netzwerkbeschränkungen gebunden zu sein, wurde, ebenfalls in Python, im Rahmen dieser Arbeit ein Scraper (`s24-fetcher`) implementiert, der den gesamten Produktkatalog in eine JSONL-Datei schreibt.

Datenverarbeitungs-Workflow

1. **Datei-Parsing:** Laden des Produktkatalogs im JSONL-Format
2. **Datenvalidierung und -transformation:** Konvertierung des Produktkatalogs im JSONL-Format zu strukturierten `Product`-Objekten
3. **Persistierung & Protokollierung** Sequenzielle Speicherung über das `ProductRepository` in der Datenbank und Protokollierung mittels `LogRepository`
4. **Logging des Fortschritts:** Echtzeitausgabe des Fortschritts und Durchsatzmetriken im CLI
5. **Abschlussbericht:** Überblick über verarbeitete Produkte und Durchsatzmetriken nach Abschluss

Skalierbarkeits- und Debug-Features

Für Entwicklungs- und Produktivumgebungen implementiert der Service konfigurierbare Verarbeitungsgrenzen über die Umgebungsvariable `DEBUG_MAX_ITEMS`, wodurch iterative Entwicklung und Performance-Tests ermöglicht werden.

5.7.3 Setup: app/generate_embeddings.py

Der `generate_embeddings.py`-Service implementiert die Embedding-Pipeline für die Transformation von Produkttextdaten zu semantischen Vektorrepräsentationen. Als zweiter Service in der Setup-Phase verarbeitet er die durch `load_data.py` bereitgestellten Produktdaten und erstellt die für die semantische Suche erforderlichen Embeddings.

Embedding-Generation-Prozess

1. **Gap-Detection:** Identifikation von Produkten ohne Embeddings
2. **Batch-Verarbeiten:** Konfigurierbare Batch-Größen für Verarbeitung großer Produktkataloge
3. **Feature-Extraktion:** Selektion semantisch relevanter Produktattribute (wie Titel, Beschreibung, Kategorie, Marke, Farbe) für Embedding-Generation
4. **Embedding-Generierung:** Anwendung des `EmbeddingService` mit dem konfigurierten Embedding-Modells
5. **Persistierung & Protokollierung** Sequenzielle Speicherung über das `ProductRepository` in der Datenbank und Protokollierung mittels `LogRepository`
6. **Logging des Fortschritts:** Echtzeitausgabe des Fortschritts und Durchsatzmetriken im CLI
7. **Abschlussbericht:** Überblick über verarbeitete in Embeddings verarbeitete Produkte und Durchsatzmetriken nach Abschluss

Die Implementierung gewährleistet, dass der Service idempotent ist und bei Unterbrechungen nahtlos fortsetzbar bleibt, da bereits verarbeitete Produkte automatisch übersprungen werden.

Von natürlicher Sprache zur Vektorrepräsentation

Die Implementierung der Funktion, die natürliche Sprache oder semantisch relevante Produktdaten unter Verwendung eines Embedding-Modells in eine Vektorrepräsentation transformiert sieht folgendermaßen aus:

Listing 5.8: EmbeddingService in app/services/embedding.py

```
"""Embedding generation service"""

import torch
from sentence_transformers import SentenceTransformer

from app.models.embedding import get_active_model_name


class EmbeddingService:
    """Service to generate embeddings for text data using Sentence Transformers"""

```

```
    # Get active model from the config
    active_model = get_active_model_name()

    # Check if CUDA is available for GPU acceleration
    cuda_available = torch.cuda.is_available()

    def __init__(self, model_name=active_model) -> None:
        if not model_name:
            raise ValueError("Missing active model name in config")

        self.model = SentenceTransformer(
            model_name,
            # Necessary for some models, Alibaba-NLP in this case
            trust_remote_code=True,
            device="cuda" if self.cuda_available else "cpu",
        )

    def get_embedding(self, text):
        """Generate embedding for a single text"""
        if isinstance(text, dict):

            # Combine relevant fields for products
            combined_text = f"Title: {text.get('title', '')}. "
            combined_text += f"Description: {text.get('description', '')}. "
            combined_text += f"Brand: {text.get('brand_name', '')}. "
            combined_text += (
                f"Original Color: {text.get('original_color', '')}. "
            )
            combined_text += f"Color: {text.get('color', '')}. "
            combined_text += f"Category: {text.get('category', '')}. "
            combined_text += (
                f"Base Category: {text.get('base_category', '')}. "
            )

            return self.model.encode(combined_text)
        else:
            # Handle plain text
            return self.model.encode(text)
```

Auch hier ist es wieder interessant, die einzelnen Schritte einzeln zu betrachten.

Listing 5.9: EmbeddingService in app/services/embedding.py

```
active_model = get_active_model_name()
```

Embeddings werden nicht gleichzeitig für alle Embedding-Modelle erstellt, sonder jeweils nur für das in der `embedding.yml`-Konfigurationsdatei als `active_model` festgelegt Modell.

Listing 5.10: EmbeddingService in app/services/embedding.py

```
cuda_available = torch.cuda.is_available()
```

Die Berechnung der Vektorrepräsentationen von Embedding-Modelle, die über das Python-Modul `sentence-transformers` geladen werden können, muss nicht zwingermaßen auf der CPU erfolgen sondern kann bei einer mit PyTorch-kompatiblen CUDA Version auch über die GPU durchgeführt werden. Die Methode `is_available()` gibt entsprechend `true` oder `false` zurück.

Listing 5.11: EmbeddingService in app/services/embedding.py

```
self.model = SentenceTransformer(
    model_name,
    # Necessary for some models, Alibaba-NLP in this case
    trust_remote_code=True,
    device="cuda" if self.cuda_available else "cpu",
)
```

Die Initialisierung des `SentenceTransformer`-Objekts.

Listing 5.12: EmbeddingService in app/services/embedding.py

```
def get_embedding(self, text):
    """Generate embedding for a single text"""
    if isinstance(text, dict):

        # Combine relevant fields for products
        combined_text = f"Title: {text.get('title', '')}. "
        combined_text += f"Description: {text.get('description', '')}. "
        combined_text += f"Brand: {text.get('brand_name', '')}. "
        combined_text += (
            f"Original Color: {text.get('original_color', '')}. "
        )
        combined_text += f"Color: {text.get('color', '')}. "
        combined_text += f"Category: {text.get('category', '')}. "
        combined_text += (
            f"Base Category: {text.get('base_category', '')}. "
        )

        return self.model.encode(combined_text)
    else:
        # Handle plain text
        return self.model.encode(text)
```

Die tatsächliche Erstellung der Vektorrepräsentation. Über die Abfrage `if isinstance(text, dict)` wird festgestellt ob:

- Produktdaten in Form eines `dict` übergeben werden, aus dessen semantisch relevanten Attributen ein String erstellt wird oder
- über die Suchfunktionalität direkt eine Anfrage in natürlicher Sprache übergeben wird.

In beiden Fällen wird die Vektorrepräsentation der Eingabe berechnet und zurückgegeben.

Index-Erstellung

Um die Dokumentation von pgvector zu zitieren: „Like other index types, it's faster to create an index after loading your initial data“ [32]. Der Embedding-Service wurde dementsprechend so konfiguriert, dass nach Abschluss der Berechnung aller Vektorrepräsentationen für das festgelegte Embedding-Modell der Index erstellt wird. Die Implementierung der Index-Erstellung wird im Folgenden erläutert:

Listing 5.13: Index-Erstellung

```
1
2  index_name = f"idx_{table_name}_hnsw_cosine"
3
4  Index(
5      index_name,
6      active_model.embedding,
7      postgresql_using="hnsw",
8      postgresql_with={"m": 16, "ef_construction": 64},
9      postgresql_ops={"embedding": "vector_cosine_ops"},
10 ).create(bind=session.bind, checkfirst=True)
```

Betrachten wir hier die relevanten Parameter:

- `index_name` und `active_model.embedding` legen fest, unter welchem Namen und für welche Relation der Index berechnet werden soll. Der Name sollte die verwendete Distanzfunktion widerspiegeln (hnsw_cosine für Cosine Distance).
- `postgresql_using="hnsw"` legt als ANN-Algorithmus HNSW fest, siehe hierfür den Abschnitt Approximate Nearest Neighbor 3.2.4.
- `postgresql_with={"m": 16, "ef_construction": 64}` legt die Werte für Maximum Connections per Node und Construction-Time Search Depth fest [33].

- `postgresql_ops={"embedding": "vector_cosine_ops"}` beschreibt, welche Distanzfunktion wir in der Implementierung der Suche verwenden. Hierbei ist wichtig, dass diese übereinstimmen: wird in der Suche `cosine_distance` bzw. der SQL-Operator `<=>` verwendet, muss bei Indexerstellung auch die passende Distanzfunktion angegeben werden, hier also `vector_cosine_ops`

Performance-Optimierungen

Für produktive Workloads implementiert der Service mehrere Performance-Features:

- **Batch-Processing:** Konfigurierbare Batch-Größen über `EMBEDDING_BATCH_SIZE` zur GPU/CPU-Optimierung
- **Error-Handling:** Einzelfehler-Isolation verhindert Batch-Failures bei problematischen Produktdaten
- **Progress-Tracking:** Echtzeit-Durchsatzmetriken

5.7.4 Synchronisation: app/external_data_sync.py

Der `external_data_sync.py`-Service implementiert die kontinuierliche Synchronisation zwischen der lokalen Produktdatenbank und dem sich verändernden Produktkatalog des externen Datenlieferanten..

Synchronisations-Konfigurations

Der Service implementiert eine flexible Konfiguration zur Unterstützung unterschiedlicher Umgebungsanforderungen:

1. **Debug-Modus (`DEBUG_SYNC_LOCAL=1`):** Datei-basierte Synchronisation aus lokalen JSONL-Dateien für Entwicklung und Testing
2. **Produktions-Modus (`DEBUG_SYNC_LOCAL=0`):** Konfiguration für Produktionsumgebung mit authentifizierter Verbindung an die API des externen Datenlieferanten

Der External Sync Service fungiert als kritischer Input für die nachgelagerte `internal_data_sync.py`-Pipeline. Durch Hash-basierte Überprüfung von Änderungen im Datensatz werden nur relevante Aktualisierungen an die, für Embedding-Erstellung und -Aktualisierung verantwortliche interne Datensynchronisation weitergeleitet, wodurch ressourcenintensive Berechnungen separat durchgeführt werden können und die Gesamtsystem-Performance optimiert wird. Die kontinuierliche Ausführung stellt sicher, dass semantische Suchergebnisse stets auf aktuellen Produktdaten basieren.

5.7.5 Synchronisation: app/internal_data_sync.py

Der `internal_data_sync.py`-Service implementiert die Synchronisationslogik zwischen Produktdaten und deren semantischen Vektorrepräsentationen und sorgt bei für die Semantik relevanten Änderungen für die Aktualisierung der entsprechenden Vektorrepräsentation.

5.7.6 Implementierung der Synchronisationslogik

Die Aufrechterhaltung der Datenkonsistenz ist ein integraler Bestandteil der Implementierung. Im Folgenden wird darauf eingegangen, wie dies umgesetzt wurde.

Verwendung von deterministischen Hashes

Betrachten wir die Datenmodelle genauer, erkennen wir das, jedem Produkt ein `data_hash` zugewiesen wird, sowie jeder Vektorrepräsentation eines Produkts jeweils ein `data_hash` und `embedding_hash`. Hierbei handelt es sich um eine Implementierung unter Verwendung von deterministischen MD5-Hashes:

Listing 5.14: Erstellung von MD5-Hashes in app/utils/hash.py

```
serialized = json.dumps(data, sort_keys=True)
return hashlib.md5(
    serialized.encode("utf-8"), usedforsecurity=False
).hexdigest()
```

Während MD5-Hashes schon längst nicht mehr für sicherheitsrelevante Aufgaben verwendet werden sollten, ist die Verwendung für die Implementierung in der Anwendung unbedenklich. Auch in Hinsicht auf Hash-Kollisionen müsste die Anwendung 2.615.321 Billionen Produkte hashen, um eine Kollisionsprobabilität von 1% zu erreichen.

Verwendung der Hashes

Die Hash-Funktion wird in der Synchronisationslogik für zwei verschiedene Hash-Typen verwendet:

- `data_hash`: Vollständiges Produktobjekt
- `embedding_hash`: Nur semantisch relevante Felder

Im ersten Schritt werden für alle Produkte, die in das System kommen, für jedes Produkt jeweils ein `data_hash` zusammen mit dem Produkt in der `products`-Relation gespeichert.

Im zweiten Schritt werden für alle Embeddings, die aus den gespeicherten Produkten berechnet werden, jeweils ein `data_hash` und ein `embedding_hash` erstellt.

Die Bedeutung für den `external-product-sync` ist nun, dass während der Synchronisation für jedes Produkt des externen Datenlieferanten der `data_hash` berechnet wird und daraus ersichtlich ist, ob sich das Produkt geändert hat. Ist dies der Fall, wird das Produkt inklusive `data_hash` aktualisiert.

Im nächsten Schritt müssen nun die Änderungen, die über den `external-product-sync` im System übernommen wurden, auch auf die Embeddings projiziert werden. Um diesen Vorgang erheblich zu beschleunigen und nicht für jedes Produkt Vektorrepräsentationen vergleichen zu müssen, führt der `internal-data-sync` daher mittels des, mit den Produkt-Embeddings gespeicherten `data_hash` einen einfachen `LEFT JOIN` aus, bei dem alle Embeddings erfasst werden, deren `data_hash`-Werte von den `products.data_hash`-Werten abweichen.

Dadurch wurde jetzt eine Liste an Produkten erstellt, die sich seit der letzten Synchronisierung mit dem Produktkatalog des externen Datenlieferanten geändert haben. Nun gilt nur noch zu erklären, bei welchen Produkten eine Änderung auch semantische Bedeutung hat. Hierfür erstellt der `internal-data-sync` nun für jedes veränderte Produkt einen temporären `embedding_hash` und vergleicht diesen mit dem gespeicherten `embedding_hash` – sind diese nicht identisch, wird die Vektorrepräsentation neu berechnet und aktualisiert.

5.7.7 Funktionsbereitstellung: app/search.py

Der `search.py`-Service implementiert die zentrale semantische Suchfunktionalität und repräsentiert den eigentlichen Kern der gesamten Anwendung. Als einer der zwei Services der Funktionsbereitstellungs-Kategorie, der direkte Endnutzer-Interaktionen verarbeitet, verarbeitet er die Transformation von natürlichsprachlichen Suchanfragen zu semantischen Vektoroperationen in Echtzeit.

1. **Embedding der Anfrage:** Transformation der natürlichsprachigen Suchanfrage zu einer Vektorrepräsentation
2. **Vector Similarity Search:** Ähnlichkeitsberechnung gegen alle Produkt-Embeddings mittels der von pgvector bereitgestellten Operatoren
3. **Ergebnis-Ranking:** Sortierung der Ergebnisse nach semantischer Ähnlichkeit mit konfigurierbaren Ergebnis-Limits
4. **Formartierung der Ergebnisse:** Strukturierte Aufbereitung der Produktdaten mit Ähnlichkeits-Scores für weitere Verwendung
5. **Performance Monitoring:** Vollständige Verfolgung der Anfrage mit Latenz-Metriken

5.7.8 Funktionsbereitstellung: app/recommendations.py

Der `recommendations.py`-Service implementiert einen komplementären Ansatz zur semantischen Suche durch Produkt-zu-Produkt-basierten Empfehlungen. Während `search.py` Suchanfragen in natürlicher Sprache verarbeitet, fokussiert dieser Service auf das klassische E-Commerce-Szenario *"Kunden, die dieses Produkt ansahen, interessierten sich auch für..."*.

Der Service zur Bereitstellung von ähnlichen Produkten auf Basis eines existierenden Produkts in der Datenbank kann als eine `low hanging fruit` betrachtet werden, also eine Funktion, die mit vergleichsweise geringem Implementierungsaufwand wertvolle Ergebnisse liefert.

Dadurch, dass das Produkt, für das ähnliche Produkte als Empfehlungen ermittelt werden sollen, bereits als Vektorrepräsentation vorliegt, kann hier eine einfache Ähnlichkeitssuche durchgeführt werden, ohne Embeddings berechnen zu müssen. Die `recommendations`-Funktion akzeptiert hierbei eine Produkt-ID, für die dann in Form von Empfehlungen die ähnlichsten Ergebnisse angezeigt werden.

5.7.9 Anwendung von Distanzfunktionen

Um Abfragen unter Verwendung von Distanzfunktionen durchzuführen, stellt pgvector eigene Operatoren bereit, welche die in Abschnitt 3.2.2 erläuterten Distanzfunktionen implementieren:

- `<->`: L2 Distance (Euklidische Distanz)
- `<#>`: Negative Inner Product (Negatives Skalarprodukt)
- `<=>`: Cosine Distance (1 - Kosinus-Ähnlichkeit)
- `<+>`: L1 Distance (Manhattan-Metrik)
- `<~>`: Hamming für binäre Vektoren
- `<%>`: Jaccard Similarity für binäre Vektoren

Da für die Implementierung der Arbeit das ORM SQLAlchemy verwendet wurde, können wir über das ebenfalls bereitgestellte Paket `pgvector-python` diese Funktionen einfach exportieren. Die praktische Anwendung der Ähnlichkeitssuche sieht dann folgendermaßen aus:

Listing 5.15: Ähnlichkeitssuche mittels Cosine Distance

```
def search_by_vector(self, embedding, limit=5):

    EmbeddingModel = get_active_model()
    embedding_array = np.array(embedding, dtype=np.float32)

    results = (
        self.session.query(
            Product,
            EmbeddingModel.embedding.cosine_distance(
                embedding_array
            ).label("distance"),
        )
        .join(EmbeddingModel, Product.id == EmbeddingModel.id)
        .order_by("distance")
        .limit(limit)
        .all()
    )

    return results
```

Betrachten wir genauer, was hier passiert:

```
EmbeddingModel = get_active_model()
```

Wie wir schon wissen, wurde die Infrastruktur der Anwendung so konfiguriert, dass mehrere Embedding-Modelle parallel betrieben werden können. Über diese Abfrage teilen wir der Suche mit, auf welcher Relation die Abfrage durchgeführt werden soll.

```
embedding_array = np.array(embedding, dtype=np.float32)
```

pgvector arbeitet intern mit float32-Datentypen, Embedding-Modelle können allerdings unterschiedliche Datentypen zurückgeben (float64, lists, tensors). Die Konvertierung stellt sicher, dass der Datentyp konsistent ist.

```
EmbeddingModel.embedding.cosine_distance(
    embedding_array
).label("distance"),
```

Hier findet mittels der Methode `cosine_distance()` nun die tatsächliche Ähnlichkeitssuche statt. Die Spalte mit den berechneten Distanzwerten wird dann als `distance` im Ergebnis erscheinen.

```
.order_by("distance")
```

In diesem Beispiel wird die bereits erläuterte Distanzfunktion `cosine_distance()` in Form der Implementierung von $1 - \text{Kosinus-Ähnlichkeit}$ verwendet. Das bedeutet, dass die Ergebnisse mit der kleinsten Entfernung den geringsten Distanzwert haben und dieser größer wird, umso weiter sich die Ergebnisse semantisch (und damit auch spatial) vom Abfragevektor entfernen. Da SQLAlchemy `ASC` als standardmäßig eine aufsteigende Sortierung verwendet, muss hier keine weitere Sortierung vorgenommen werden.

5.8 Deployment und Infrastruktur

Das System wurde nicht dafür entwickelt, um auf einem lokalen System zu leben, sondern in der technischen Infrastruktur von donista integriert zu werden, wo sie in echten Use-Cases zum Einsatz kommt. Software durchläuft Iterationsstufen, sich verändernde Anforderungen führen zu Modifikationen an der Implementierung, der Serverinfrastruktur oder dem Bereitstellungsprozess. Eine Iteration am in dieser Arbeit behandelten System darf keine Implikationen auf den laufenden Betrieb des Gesamtsystems haben und muss nach Abschluss der Iteration nahtlos in den laufenden Betrieb des Gesamtsystems übernommen werden. Diese Überlegungen gehören in den Bereich des DevOps und sind für einen Software-Lifecycle essenziell.

5.8.1 CI/CD

Deployment und CI/CD-Prozesse gehören nicht zum Kern der Aufgabenstellung dieser Arbeit, wurden jedoch als Teil einer vollständigen Implementierung konfiguriert. Im Folgenden wird darauf grob eingegangen.

Das Repository ist über `.github/workflows/main.yml` in verschiedene CI/CD-Stages konfiguriert:

1. **Code-Quality- und Code-Security-Checks** unter Verwendung von isort, black flake8, mypy, bandit, safety und pip-audit
2. **Unit- und Integration-Tests** mit pytest und pytest-cov

5.8.2 Docker Compose

Die Konfigurations-Datei für Docker Compose implementiert die gesamte Konzeption der Anwendung, sodass über einen Befehl das vollständige System aufgesetzt werden kann:

1. Initialisierung der Datenbank und pgvector
2. Erstellung der Produktdaten
3. Berechnung der Embeddings und Index-Erstellung
4. Kontinuierliche externe und interne Synchronisation in Intervallen
5. API-Bereitstellung für semantische Such- und Empfehlungsfunktionen

6 Evaluation und Ergebnisse

In diesem Kapitel werden sowohl die technischen als auch die semantischen Ergebnisse meiner Arbeit ausgewertet. Dies beinhaltet beispielsweise die Performance der Infrastruktur, Analyse der Logs bezüglich der Performance verschiedener Embedding-Modelle, als auch die Qualität der Ergebnisse von Anfragen wie Recall@k, Precision@k oder Mean Average Precision.

6.1 Performance-Analyse der Anwendung

6.1.1 Testumgebung

Die Analyse der Performance wurde auf folgender Hardware durchgeführt:

- **OS:** Windows 11
- **CPU:** AMD Ryzen 7 7800X3D 8x 4.20GHz So.AM5
- **GPU:** 12GB GeForce RTX 4070 SUPER Ventus 3X OC
- **RAM:** 32GB DDR5-6000 DIMM CL30
- **SSD:** 1TB NM790 M.2 2280

Bei der folgenden Auswertung der Performance der im Rahmen der Arbeit implementierten Anwendung sollte berücksichtigt werden, dass die Testumgebung eine höhere Leistungsfähigkeit hat als die in der Produktionsumgebung typischerweise verwendete Hardware.

Die Datenbank wird als Docker-Image bereitgestellt und verwendet PostgreSQL Version 17.5. Alle Embeddings wurden auf der GPU berechnet.

6.1.2 Verwendete Embedding-Modelle

Für eine vergleichende Auswertung der Anwendung werden die folgenden Modelle verwendet:

- `sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2`
- `Alibaba-NLP/gte-multilingual-base`

- `intfloat/multilingual-e5-base`

Insgesamt wurden pro Modell 1 Million Vektorrepräsentationen aus Produkten des shopping24.com-Katalogs erstellt und analysiert.

6.1.3 Durchführung

Über die implementierte Klasse `LogRepository` wird für jede Datenbanktransaktion der Zeitaufwand im Feld `duration_ms` in Millisekunden protokolliert. Wir können nun für alle Logs nach der Aktion `EMBEDDING_GENERATION` filtern. Zusätzlich bietet es sich noch an, die Performance getrennt nach `INSERT` und `UPDATE` zu evaluieren.

6.1.4 Performance-Metriken

Die Performance-Evaluierung wurde in `app/evaluate_performance.py` implementiert und liefert die im Folgenden erläuterten Metriken.

Total operations

Die Gesamtanzahl der Embedding-Generierung-Operationen, die für die Performance-Analyse herangezogen wurden. Diese Stichprobengröße ist entscheidend für die statistische Aussagekraft der Ergebnisse. Eine größere Anzahl von Operationen führt zu zuverlässigeren Metriken und reduziert die Wahrscheinlichkeit von Verzerrungen durch Ausreißer.

Min duration

Die kürzeste gemessene Ausführungsdauer einer Embedding-Generierung. Diese Metrik zeigt die optimale Performance des Systems unter idealen Bedingungen auf und kann als Baseline für die bestmögliche Ausführungsgeschwindigkeit dienen.

Max duration

Die längste gemessene Ausführungsdauer. Hohe Maximalwerte können auf Systemengpässe, Ressourcenkonflikte oder ineffiziente Verarbeitungsfälle hindeuten. Diese Metrik hilft bei der Identifikation von Performance-Anomalien.

Mean duration (Arithmetisches Mittel)

Der Durchschnittswert aller gemessenen Ausführungszeiten. Das arithmetische Mittel kann durch extreme Ausreißer beeinflusst werden und gibt daher die ”typische”Performance möglicherweise nicht akkurat wieder. Für die Kapazitätsplanung und Durchsatzberechnungen ist diese Metrik jedoch von zentraler Bedeutung.

Median (P50)

Der Wert, bei dem 50% aller Operationen schneller und 50% langsamer ausgeführt wurden. Der Median ist robuster gegenüber Ausreißern als das arithmetische Mittel und repräsentiert die ”typische”Benutzererfahrung besser. In asymmetrischen Verteilungen (wie sie bei Performance-Daten häufig auftreten) ist der Median oft aussagekräftiger als der Mittelwert.

Standard deviation (Standardabweichung)

Ein Maß für die Streuung der Ausführungszeiten um den Mittelwert. Eine hohe Standardabweichung deutet auf inkonsistente Performance hin, während eine niedrige Standardabweichung auf vorhersagbare, stabile Ausführungszeiten hindeutet. Die Standardabweichung ist wichtig für die Beurteilung der Systemzuverlässigkeit. Latenzmetriken (Perzentile)

P90 (90. Perzentil)

90% aller Operationen wurden in dieser Zeit oder schneller abgeschlossen. Diese Metrik gibt Aufschluss über die Performance für die große Mehrheit der Benutzer und wird häufig für Service Level Objectives (SLOs) verwendet.

P95 (95. Perzentil)

95% aller Operationen wurden in dieser Zeit oder schneller abgeschlossen. P95 ist ein Industriestandard für die Definition von Performance-Zielen, da es einen guten Kompromiss zwischen der Repräsentation der Benutzererfahrung und der Toleranz gegenüber seltenen Performance-Ausreißern darstellt.

P99 (99. Perzentil)

99% aller Operationen wurden in dieser Zeit oder schneller abgeschlossen. Diese Metrik zeigt die Performance im "Worst-CaseSzenario auf und ist relevant für die Bewertung der System-Robustheit. P99-Werte werden oft für kritische Systeme überwacht, da sie Aufschluss über seltene, aber potenziell problematische Performance-Degradationen geben.

Bedeutung der Perzentile:

Perzentile sind besonders wertvoll bei rechtsschiefen Verteilungen (wie sie bei Performance-Daten typisch sind), da sie unabhängig von extremen Ausreißern aussagekräftige Schwellenwerte definieren.

Average throughput (Durchschnittlicher Durchsatz)

Berechnet als 1000 / Mean duration (ms) und gibt die durchschnittliche Anzahl von Embedding-Generierungen pro Sekunde an. Diese Metrik ist entscheidend für die Kapazitätsplanung und die Bewertung der Systemeffizienz bei der Verarbeitung großer Datenmengen.

Coefficient of variation (Variationskoeffizient)

Berechnet als (Standardabweichung / Mittelwert) $\times$ 100%. Diese dimensionslose Kennzahl ermöglicht die Bewertung der relativen Variabilität unabhängig von der absoluten Größenordnung der Ausführungszeiten. Ein niedriger Variationskoeffizient ($<$20%) deutet auf konsistente Performance hin, während ein hoher Wert ($>$50%) auf unvorhersagbare Systemperformance hindeutet.

Outliers (Ausreißer)

Operationen, deren Ausführungszeit P99 überschreitet. Ausreißer können auf Systemanomalien, Ressourcenkonflikte oder ungewöhnliche Verarbeitungsfälle hindeuten. Die Analyse der Ausreißer-Rate hilft bei der Identifikation von Optimierungspotential und der Bewertung der Systemstabilität.

6.1.5 Ergebnisse

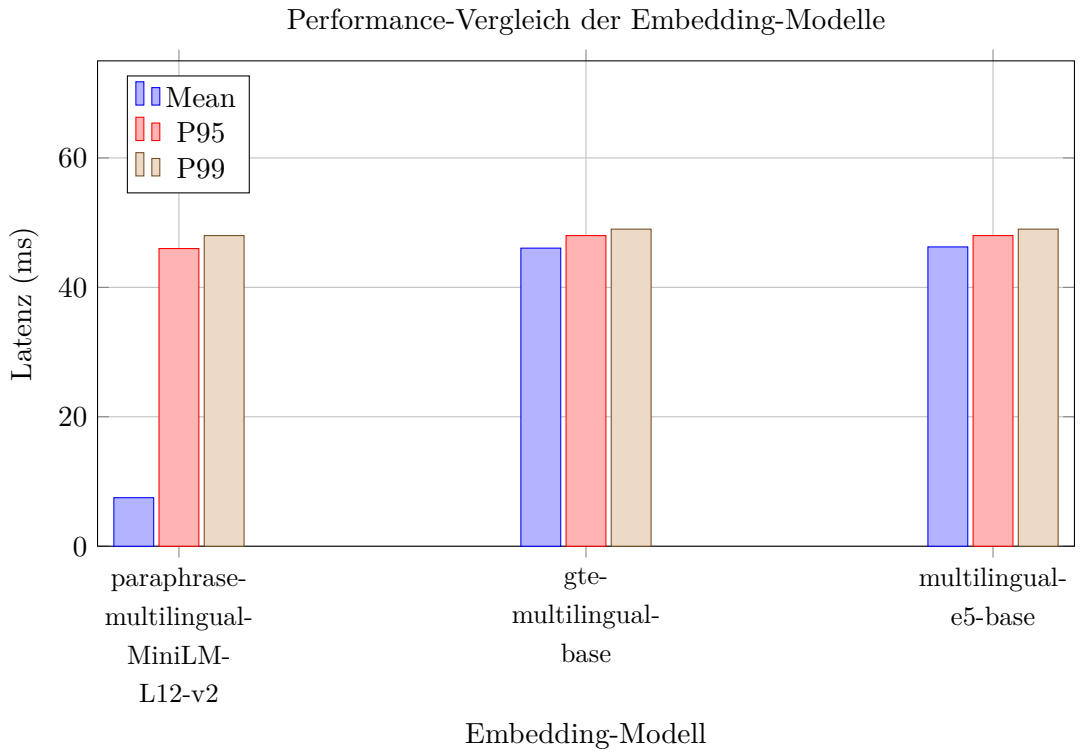


Abbildung 6.1: Latenz-Vergleich zwischen verschiedenen Embedding-Modellen

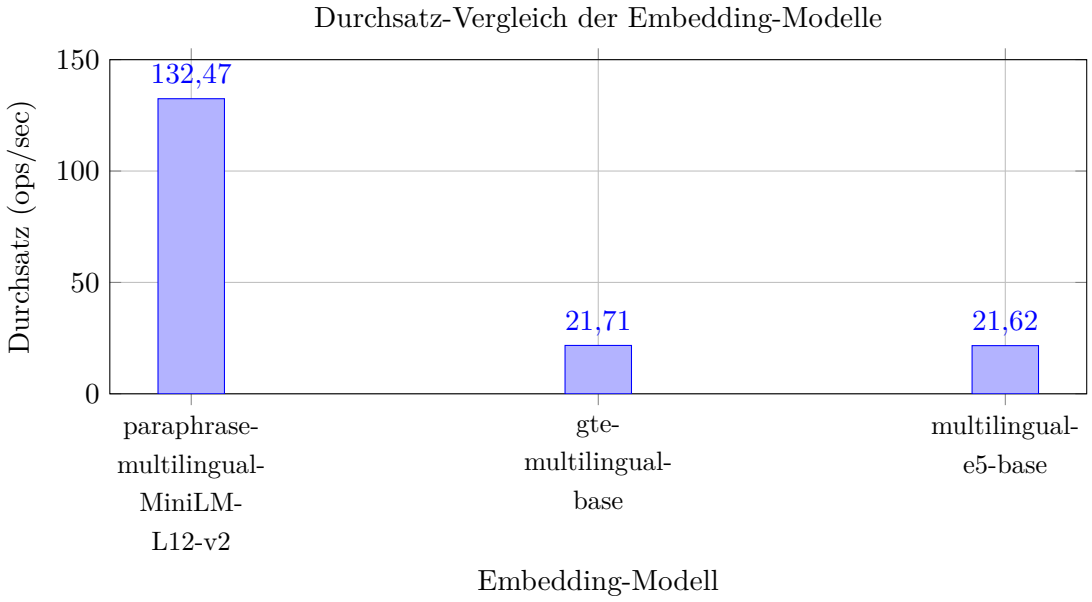


Abbildung 6.2: Durchsatz-Vergleich zwischen Embedding-Modellen

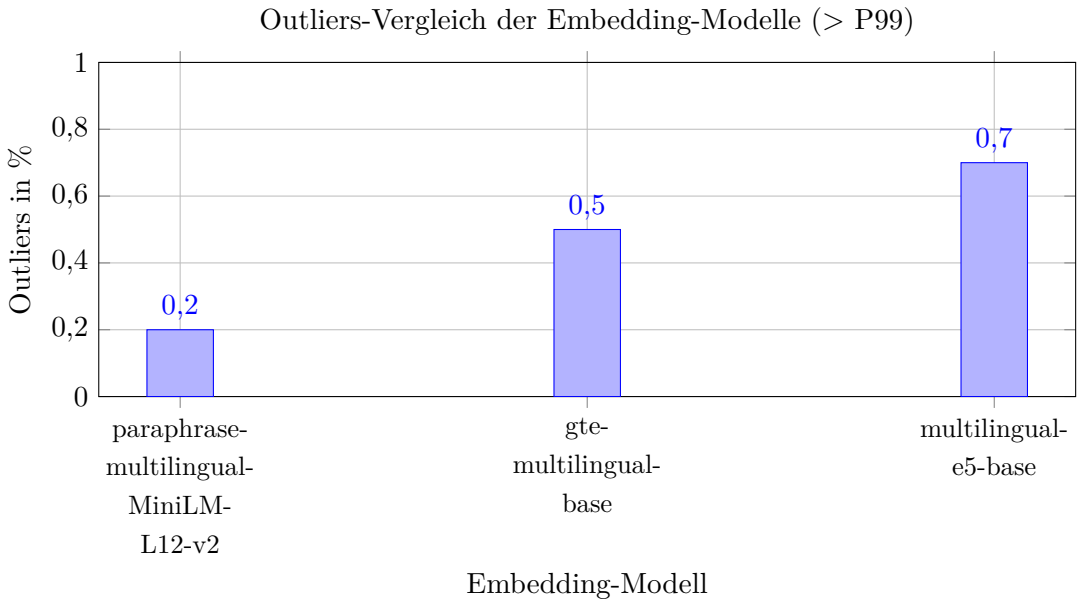


Abbildung 6.3: Vergleich der Outliers zwischen Embedding-Modellen

6.2 Qualitätsevaluation der Suchergebnisse

6.2.1 Testumgebung

Die Analyse der Performance wurde auf folgender Hardware durchgeführt:

- **OS:** Windows 11
- **CPU:** AMD Ryzen 7 7800X3D 8x 4.20GHz So.AM5
- **GPU:** 12GB GeForce RTX 4070 SUPER Ventus 3X OC
- **RAM:** 32GB DDR5-6000 DIMM CL30
- **SSD:** 1TB NM790 M.2 2280

Bei der folgenden Auswertung der Performance der im Rahmen der Arbeit implementierten Anwendung sollte berücksichtigt werden, dass die Testumgebung eine höhere Leistungsfähigkeit hat als die in der Produktionsumgebung typischerweise verwendete Hardware.

Die Datenbank wird als Docker-Image bereitgestellt und verwendet PostgreSQL Version 17.5. Alle Embeddings wurden auf der GPU berechnet.

6.2.2 Verwendete Embedding-Modelle

Für eine vergleichende Auswertung der Anwendung werden die folgenden Modelle verwendet:

- `sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2`
- `Alibaba-NLP/gte-multilingual-base`
- `intfloat/multilingual-e5-base`

Insgesamt wurden pro Modell 1 Million Vektorrepräsentationen aus Produkten des shopping24.com-Katalogs erstellt und analysiert.

6.2.3 Durchführung

Die Evaluation der Suchergebnisse unter Verwendung der implementierten Suche ist um einiges komplexer als die im vorherigen Abschnitt erfassten Metriken.

Vorbereitung

Die Bewertung der semantischen Suchqualität erfordert einen systematischen Ansatz zur Messung der Relevanz von Suchergebnissen. Anders als bei der Performance-Analyse, die objektive Metriken wie Latenz und Durchsatz verwendet, basiert die Qualitätsevaluation auf subjektiven Relevanzbewertungen von Suchergebnissen. Um aussagekräftige Metriken wie Precision, Recall und Mean Average Precision (MAP) zu berechnen, benötigt es einen Datensatz, der mögliche Suchanfragen, im Folgenden als Sample Queries betitelt, und Produkte mit einem Relevanzparameter, im Folgenden weiter Relevance Score aufgeführt, kombiniert. Dieser Datensatz wird im Folgenden weiter als Ground-Truth-Datensatz betitelt, der Prozess von gewichteter Bewertung als *annotierung* [34].

Der verwendete Relevance Score kann dabei ganzzahlige Werte von 0 bis 2 annehmen, wobei gilt:

- 0: Keine Relevanz
- 1: Teilweise relevant
- 2: Hochrelevant

Ein einfaches Beispiel einer Annotation wäre dann:

- Sample Query: *"bürobedarf arbeitsplatz"*
- Produkt: *"Vielseitiger Schreibtisch Organizer mit 5 Fächern, 2 Schubladen und multifunktionalen Aufbewahrungsmöglichkeiten - ideal für Bürobedarf, Schreibwaren und Schreibtisch-Accessoires"*
- Relevance-Score: 2 (Hochrelevant)

Implementierung des Annotations-Framework

Für die Implementierung wurde ein JSON `app/data/sample_queries.json` vorbereitet, welche mögliche Suchanfragen einer E-Commerce-Plattform enthält. Diese werden über `app/add_sample_queries.py` in die Relation `sample_queries` geladen.

Im nächsten Schritt wird über `app/create_gtd.py` ein interaktives CLI-Tool geladen, bei dem sequenziell zufällig ausgewählte Produkte und Sample Queries gegenübergestellt werden, deren Relevanz der Benutzer dann bewerten muss. Diese werden dann in die Relation `query_relevance` eingefügt, welche eine Many-to-Many-Beziehung zwischen Sample Queries und Produkten mit Relevanzbewertung darstellt. Für jede durchgeführte Annotation werden dementsprechend die Produkt-ID, die Sample-Query-ID und der Relevance Score gespeichert.

Implementierung des Evaluation der semantischen Suche

Über `app/evaluate_model.py` wird dann für jede Sample Query eine semantische Suche durchgeführt. Die Ergebnisse dieser Suchanfrage werden dann mit den subjektiv bewerteten Suchergebnissen aus dem Annotationsverfahren gegenübergestellt. Diese Gegenüberstellung erfolgt mit den in Abschnitt 4.5 erläuterten Ranking- und Recommendations-Metriken, spezifisch sind das Precision@K, Recall@K, F-Beta-Score und Mean Average Precision, die für alle Suchanfragen aggregiert ausgewertet werden.

6.2.4 Ergebnisse

Bei der Qualitätsevaluation der Suchergebnisse stieß ich letztendlich leider auf praktische Herausforderungen bei der Annotation der Daten. Auch wenn die komplette Infrastruktur zur Annotation der Daten und des darauf aufbauenden Systems zur Berechnung und Vergleich der verschiedenen Modelle inklusive Tests zur Validierung implementiert wurde, konnten in der mir verfügbaren Zeit keine statistisch aussagekräftigen Ergebnisse generiert werden.

Hierbei verwendete ich:

- 1 Millionen Produkt-Embeddings pro konfigurierten Modell
- 25 verschiedene Suchanfragen in natürlicher Sprache

Insgesamt habe ich knapp 7500 Annotationen erstellt, von diesen Annotationen hatten einzig 26 einen `relevance_score` von >= 1. 10 der 25 verwendeten Suchanfragen hatten nach 7500 Annotationen immer noch kein relevantes Produkt zugewiesen.

Methodische Schwächen der Annotationsstrategie

Zufällige Produktauswahl ohne Vorfilterung: Die implementierte Annotations-Pipeline verwendete eine zu primitive Strategie der rein zufälligen Produktauswahl ohne jegliche Vorfilterung nach Kategorien, Preisklassen oder semantischen Clustern. Dies führte zu einem extrem unausgewogenen Verhältnis zwischen relevanten und irrelevanten Query-Produkt-Kombinationen. Bei einem Produktkatalog von mehreren Millionen Einträgen und einer zufälligen Auswahl beträgt die Wahrscheinlichkeit für semantisch relevante Paarungen nur wenige Prozent, was die extrem niedrige Hit-Rate von 0,35% (26/7.500) erklärt.

Skalierungsprobleme des Single-Annotator-Ansatzes: Der Zeitaufwand für die manuelle Annotation durch einen einzelnen Bewerter erwies sich als nicht praktikabel.

Implikationen für die Modellbewertung

Das implementierte `app/evaluate_model.py`-Skript konnte aufgrund der unzureichenden Datengrundlage keine belastbaren Metriken wie Precision@K, Recall@K oder Mean Average Precision berechnen. Viele der Suchanfragen blieben ohne relevante Annotationen, was zu undefinierten Recall-Werten führte und eine vergleichende Bewertung der drei Embedding-Modelle unmöglich machte.

Funktionalität der implementierten Infrastruktur

Trotz der Evaluationslimitationen ist die implementierte Infrastruktur vollständig funktionsfähig und kann bei verfügbaren Annotationsdaten sofort für umfassende Modellvergleiche genutzt werden. Die entwickelten Komponenten für Embedding-Generierung,

Vektorsuche und Ranking-Metriken demonstrieren die technische Machbarkeit semantischer Produktsuche und stellen eine solide Grundlage für zukünftige Optimierungen dar.

7 Diskussion

7.1 Interpretation der Ergebnisse

7.1.1 Bewertung der quantitative Leistungsparameter

Die Performance-Analyse der drei Embedding-Modelle bei der Verarbeitung von jeweils 1 Million Produktdatensätzen zeigt deutliche Unterschiede in der Verarbeitungsgeschwindigkeit und -stabilität.

Durchsatzleistung

Das Modell `sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2` (384 Dimensionen) erreicht mit 132,47 Operationen/Sekunde die höchste Durchsatzrate und ist damit etwa 6-mal schneller als die beiden 768-dimensionalen Modelle. Die Modelle `Alibaba-NLP/gte-multilingual-base` und `intfloat/multilingual-e5-base` zeigen mit 21,71 bzw. 21,62 Operationen/Sekunde nahezu identische Durchsatzraten.

Verarbeitungslatenz

Bei der mittleren Verarbeitungszeit pro Embedding zeigt sich ein klarer Zusammenhang zur Dimensionalität:

- `paraphrase-multilingual-MiniLM-L12-v2`: 7,55 ms (Median: 3,00 ms)
- `Alibaba-NLP/gte-multilingual-base`: 46,06 ms (Median: 46,00 ms)
- `intfloat/multilingual-e5-base`: 46,25 ms (Median: 46,00 ms)

Die 768-dimensionalen Modelle benötigen etwa 6-mal länger für die Embedding-Generierung als das 384-dimensionale Modell.

Performance-Stabilität

Ein signifikanter Unterschied zeigt sich in der Konsistenz der Verarbeitungszeiten:

- Beide 768-dimensionalen Modelle: Sehr stabile Performance mit Variationskoeffizienten von 4,1% bzw. 4,3% und geringen Standardabweichungen (~2 ms)
- 384-dimensionales Modell : Hohe Variabilität mit einem Variationskoeffizient von 175,0% und einer Standardabweichung von 13,21 ms

Fazit der quantitativen Bewertung

Die Ergebnisse zeigen einen Zielkonflikt zwischen Geschwindigkeit und Dimensionalität: Das Modell `sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2` bietet zwar die schnellste Verarbeitungsgeschwindigkeit, dies ist allerdings einfach durch die niedrigere Dimensionalität der durch das Modell berechneten Vektorrepräsentationen bedingt.

Fortfahrend steht noch der hohe Variationskoeffizient von 175,0% zur Diskussion. Um auszuschließen, dass es sich hier um eine zufällige Anomalie handelt, sondern kontinuierlich die gleichen Ergebnisse liefert, habe ich die Berechnung der Embeddings mit dem Modell `sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2` für jeweils 20.000 Produkte 5 Mal wiederholt:

1. Durchlauf: Variationskoeffizient 54.7%
2. Durchlauf: Variationskoeffizient 40.1%
3. Durchlauf: Variationskoeffizient: 36.2%
4. Durchlauf: Variationskoeffizient: 74.7%
5. Durchlauf: Variationskoeffizient: 37.0%

Auch wenn Ausschläge bei geringeren Datensätzen natürlich einen größeren Einfluss auf den Variationskoeffizienten haben, lässt sich erkennen, dass dieser keine reine Anomalie war, sondern das Modell `sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2` hier tatsächlich eine deutlich höhere Variation vorzuweisen hat, als die beiden anderen verwendeten Modelle.

Abschließend lässt sich hier sagen, dass das 384-dimensionale Modell zwar deutlich schneller in der Berechnung war, dafür in der Gesamtperformance jedoch instabiler.

Allein die quantitativen Leistungsparameter sind jedoch nicht genug, um eine Entscheidung für oder gegen ein Embedding-Modell zu treffen. Daher werden im nächsten Schritt die qualitativen Leistungsparameter noch in die Bewertung einbezogen.

7.2 Limitationen und Herausforderungen

Die Entwicklung einer produktionsreifen semantischen Produktsuche bringt vielfältige technische und methodische Herausforderungen mit sich, von denen viele im Rahmen dieser dreimonatigen Arbeit nicht vollständig adressiert werden konnten. Die folgenden Limitationen verdeutlichen sowohl die Grenzen der entwickelten Lösung als auch Ansatzpunkte für weiterführende Forschung.

7.2.1 Technische Implementierungstiefe

Beschränkter Technologievergleich

Die ursprünglich geplante vergleichende Analyse verschiedener Vektordatenbanken (Milvus, Pinecone, Weaviate, Vertex AI) konnte aufgrund des zeitlichen Rahmens nicht umgesetzt werden. Ein aussagekräftiger Vergleich hätte die vollständige Implementierung, Konfiguration und Benchmarking mehrerer Systeme unter identischen Bedingungen erfordert, was den Umfang einer dreimonatigen Arbeit überschritten hätte. Die Fokussierung auf PostgreSQL mit pgvector ermöglichte zwar eine tiefere Auseinandersetzung mit einer Technologie, schränkt jedoch die Generalisierbarkeit der Erkenntnisse bezüglich der optimalen Technologiewahl ein.

Oberflächliche Optimierung

Die Implementierung kratzt nur an der Oberfläche der verfügbaren Optimierungsmöglichkeiten. Wesentliche Aspekte wie der systematische Vergleich von Distanzfunktionen (L2 Distance, Inner Product, Cosine Distance, L1 Distance, Hamming distance), Index-Strategien (IVFFlat, HNSW) und deren spezifisches Parameter-Tuning (z.B. HNSW-Parameter `ef_construction`, `m`) wurden nicht untersucht. Ebenso fehlt die Optimierung der Eingabedaten für die Embedding-Generierung, wie etwa die systematische Analyse verschiedener Feld-Kombinationen oder Preprocessing-Strategien.

Fehlende Domain-Spezifität

Für die verwendeten Embedding-Modelle wurde kein Fine-Tuning auf den domänenspezifischen E-Commerce-Kontext durchgeführt. Ein domänenspezifisches Training auf Produktdaten hätte potenziell erhebliche Qualitätsverbesserungen ermöglicht, war jedoch aufgrund der erforderlichen Datensätze, Rechenressourcen und Entwicklungszeit nicht realisierbar.

Code-Qualität und DX

Software darf nicht nur als Black Box allein die Funktionalität bereitstellen und nur für den verständlich sein, der diese implementiert hat, sondern, besonders im Hinblick auf moderne Ansätze in der Softwareentwicklung, für alle verständlich sein, die an der Entwicklung teilhaben. Mit Blick auf die Code-Qualität des von mir implementierten Systems würde ich folgenden Punkten in einem Refactoring gern noch mehr Aufmerksamkeit schenken:

- **Bessere Implementierung des Repository-Pattern:** Präzisere Unterteilung von Repository (Datenzugriff), Service-Klassen und Utility-Funktionen
- **Typisierung:** Python ist grundsätzlich zwar nicht typisiert, bietet aber Möglichkeiten um Typ-Annotationen umzusetzen, welche ich mit mehr Zeit noch besser umgesetzt hätte um die Anwendung noch robuster zu machen
- **Kommentare:** Für die meisten Methoden habe ich bereits Docstrings verwendet, um verständlich zu machen, was die Funktion einer Methode ist, welche Parameter sie akzeptiert und was diese zurückgibt, mit mehr Zeit hätte ich auch hier noch weitere Optimierungen vorgenommen
- **Auslagerung von Konfigurationen:** Um verschiedene Embedding-Modelle anhand der in Kapitel 6 erläuterten Ansätze zu analysieren, werden diese über `app/config/embeddings.yml` konfiguriert. Hier besteht möglicherweise Verbesserungsbedarf um Embedding- und Indexierungsparamter zu konfigurieren ohne direkt im Code arbeiten zu müssen.

7.2.2 Evaluationsmethodik und Datenqualität

Limitiertes Annotationssystem

Das implementierte System zur Erstellung des Ground Truth-Datensatzes basiert auf einer begrenzten Anzahl von Sample Queries und einem einzelnen Annotator, was die statistische Aussagekraft einschränkt. Für eine robuste Evaluation wären umfangreichere Datensätze mit mehreren unabhängigen Bewertern erforderlich, um subjektive Verzerrungen zu minimieren und Inter-Annotator-Agreement [35] zu messen.

Für produktive Evaluationen semantischer Suchsysteme sollten folgende methodische Verbesserungen implementiert werden:

- **Intelligente Vorfilterung:** Kategoriebasierte oder embeddings-gestützte Vorauswahl relevanter Produktkandidaten
- **Active Learning:** Gezielte Auswahl der informativsten Query-Produkt-Kombinationen zur Maximierung des Annotationsnutzens
- **LLM-basierte Annotation:** Automatisierte Bewertung großer Datensätze mittels Large Language Models mit stichprobenartiger manueller Validierung
- **Multi-Annotator-Setup:** Mehrere unabhängige Bewerter zur Erhöhung der Reliabilität und Messung des Inter-Annotator-Agreements

Skalierbarkeit der Evaluation

Moderne Ansätze zur Evaluation semantischer Suchsysteme nutzen zunehmend Large Language Models (LLMs) zur automatisierten Annotation großer Datensätze [36]. Solche Methoden könnten die manuelle Annotationsarbeit erheblich skalieren und konsistentere Bewertungen ermöglichen, waren jedoch nicht Teil dieser Arbeit.

Fehlende Baseline-Vergleiche

Die Evaluation beschränkt sich auf die Bewertung der semantischen Suche in Isolation, ohne systematischen Vergleich zu etablierten Suchmethoden wie BM25 oder Elasticsearch. Dies erschwert die Einordnung der erzielten Ergebnisse und die Bewertung des tatsächlichen Mehrwerts der semantischen Ansätze.

7.2.3 Produktions- und Skalierungsaspekte

Produktionsreife

Die Implementierung wurde unter realistischen Produktionsbedingungen nicht getestet. Aspekte wie Hochverfügbarkeit, Lastverteilung bei Millionen von Anfragen, Erfüllung von SLAs oder das Verhalten bei Hardware-Ausfällen blieben unberücksichtigt. Ebenso fehlen Konzepte für kontinuierliches Monitoring und automatische Qualitätsüberwachung.

Multimodale Limitationen

Die Beschränkung auf textuelle Embeddings ignoriert die Bedeutung visueller Informationen im E-Commerce-Kontext. Produktbilder sind oft entscheidend für Kaufentscheidungen, und multimodale Ansätze könnten die Suchqualität erheblich verbessern. Die infrastrukturellen Anforderungen, zusätzlich zu meiner Implementierung, noch mehrere Millionen Bilder zu verarbeiten und zu speichern, hätten bei weitem den zeitlichen Rahmen der Bearbeitung überschritten.

Business-Metriken

Die technische Evaluation über Precision und Recall bildet nicht die geschäftskritischen Metriken ab, die in produktiven E-Commerce-Systemen relevant sind, wie Click-Through-Rates, Conversion-Rates oder Revenue-per-Search. Hierfür wäre es unter Umständen empfehlenswert, A/B-Tests durchzuführen, um Insights zu erhalten, ob die semantische Suche tatsächlich zu einer höheren Conversion führt.

7.2.4 Generalisierbarkeit und Reproduzierbarkeit

Domänen-Spezifität

Die Evaluation basiert ausschließlich auf Daten von shopping24.com, was die Übertragbarkeit auf andere E-Commerce-Domains oder Produktkataloge einschränkt. Branchenspezifische Besonderheiten oder unterschiedliche Produktkategorien könnten erheblichen Einfluss auf die Modellperformance haben.

Reproduzierbarkeit

Die manuelle Komponente der Annotation und die nicht-deterministische Natur einiger Embedding-Modelle erschweren die exakte Reproduktion der Ergebnisse. Für wissenschaftliche Standards wären detailliertere Protokolle und deterministische Verfahren erforderlich.

8 Fazit und Ausblick

8.1 Beantwortung der Forschungsfragen

8.2 Semantische Suche für die donista-Plattform

Der in dieser Arbeit implementierte Microservice demonstriert eine Möglichkeit, semantische Suche für eine bestehende Infrastruktur zu implementieren und dabei mit der bestehenden PostgreSQL-Datenbank arbeiten zu können, ohne ein weiteres DBMS zur Infrastruktur hinzufügen zu müssen oder auf proprietäre bzw. Managed-Services zugreifen zu müssen. Über die Bereitstellung der Suchfunktionen mittels REST-API und der Docker-Konfiguration lässt sich dieser einfach in die Use-Cases der bestehenden Anwendung und der Deployment-Strategie integrieren.

Müsste ich Empfehlungen für die weitere Zukunft der semantischen Suche in der donista-Plattform abgeben, würde ich diese in folgende Schritte in Betracht ziehen:

1. **Parameter-Tuning**: Mit relativ wenig Aufwand bietet es sich an, die qualitative und quantitative Leistung von Suchanfragen mit verschiedenen Distanzfunktionen, verschiedenen ANN-Indizes und Parametern zu vergleichen. Erweiternd lassen sich zu jedem Embedding noch Spalten für Metadaten hinzufügen um mittels Metadaten-Extraktion ein Pre-Filtering der möglichen Ergebnisse durchzuführen.

2. **Auslagern der externen Produktsynchronisation**: Da die Synchronisation des Produktkatalogs in donista mit dem Produktkatalog des externen Datenlieferanten nur bedingt zusammenhängt, die gesamte Anwendung sowie der Microservice die gleiche Datenbank verwenden sowie die Gründe, die für die Implementierung in Python statt Kotlin geführt haben darauf keinen Einfluss haben, würde ich die externe Produktsynchronisation als erstes in die Hauptanwendung integrieren.

3. **Aktivierung von pgvectorscale**: Die Erweiterung pgvectorscale ist, eine von TimescaleDB entwickelte Ergänzung zu pgvector, die dessen Performance erheblich verbessert, beispielsweise durch einen spezifisch implementierte Index *StreamingDiskANN*

4. **Evaluierung zur direkten Integration**: Da die donista-Architektur als Monolith umgesetzt wurde, kann evaluiert werden, ob die semantische Suche nicht

direkt komplett in Kotlin implementiert wird. Soll der aktuelle Ansatz mit pgvector beibehalten werden, bietet pgvector mit `pgvector-java` auch Unterstützung für Kotlin, als Runtime für ONNX-Modelle bietet sich hier KInference an.

5. **Multimodalität**: Zusätzlich zu reinen Text-Embeddings bieten E-Commerce-Anwendung großes Potential, wenn zusätzlich auch Produktbilder in der semantischen Suche bzw. dem System für Produktempfehlungen integriert werden.

8.3 Ausblick

Vektordatenbanken sind inzwischen längst keine Nischentechnologie mehr und der generelle „AI-Hype“ bietet hier einen fruchtbaren Boden. Hierfür können wir allein aus 2023 und 2024 die Investitionen in Vektordatenbanken und verwandte Technologieunternehmen betrachten:

- Pinecone: 100 Millionen USD Series B Funding bei einer Valuation von 750 Millionen USD [37]
- Weaviate: 50 Millionen USD Investition bei einer Valuation von 200 Millionen USD [38, 39]
- Qdrant: 28 Millionen USD Series A Funding [40]
- Chroma: 18 Millionen USD Funding bei einer 75 Millionen USD Bewertung [41, 42]

Doch auch proprietäre Datenbanksysteme wie MongoDB, Oracle, Redis, Azure und Couchbase bieten inzwischen Implementierungen von Vektordatenbanken in ihrer Produktpalette – ebenso haben Suchmaschinen wie Elasticsearch, OpenSearch und Algolia ihre Funktionen um Vektorsuche erweitert.

Die Implementierung von Vektordatenbanken in produktiven Umgebungen offenbart erhebliche technische Komplexitäten, die in der aktuellen Diskussion oft unterrepräsentiert sind. Speicheranforderungen skalieren linear mit der Datenmenge (die PostgreSQL-Datenbank dieser Arbeit hat einen Speicherbedarf von 21 GB, davon knapp über 19 GB für die Embeddings und Indizes), während Indexing-Strategien wie HNSW Trade-offs zwischen Suchgeschwindigkeit, Speicherverbrauch und Suchqualität erfordern.

Die Integration semantischer Suche in bestehende E-Commerce-Infrastrukturen stellt zusätzliche Herausforderungen dar: Latenzanforderungen, Datenkonsistenz bei häufigen Produktaktualisierungen und die Notwendigkeit multimodaler Suchanfragen (Text, Bild, Produktattribute) erfordern sorgfältige Architekturentscheidungen.

8.3.1 Abschließend

Vektordatenbanken stellen eine substanzielle Erweiterung traditioneller Suchparadigmen dar, deren kommerzielle Machbarkeit jedoch von realistischen Leistungserwartungen und sorgfältiger technischer Implementierung abhängt. Die in dieser Arbeit demonstrierte Implementierung mit pgvector zeigt das Potenzial für verbesserte Nutzererfahrungen im E-Commerce, verdeutlicht jedoch auch die Notwendigkeit weiterer Forschung zur Skalierung, Personalisierung und multimodalen Ansätzen.

Ebenso wenig wie jedes Unternehmen „AI“ um des Willens willen in ihre Prozesse implementieren muss, nur um sich als ein „AI-first“ Unternehmen platzieren zu können, sind auch Vektordatenbanken nicht gezwungenermaßen die notwendige Lösung für jede Situation, in der Daten gespeichert werden müssen. Vektordatenbanken stellen dabei keine allumfassende Lösung für Datenpersistierung dar, sondern haben ihre Stärken speziell in einer Aufgabe: Speicherung und Abfrage von hochdimensionalen Vektoren. Damit sollten diese also nicht als Ersatz von bestehenden DBMS gesehen werden, sondern als eine Lösung für eine Limitation in eben diesen, ähnlich wie es NoSQL-Datenbanken bereits waren.

Literatur

[1] Maris Fessenden. *What Was the First Thing Sold on the Internet?* Smithsonian Magazine. Section: Smart News, Smart News History & Archaeology. url: https://www.smithsonianmag.com/smart-news/what-was-first-thing-sold-internet-180957414/ (besucht am 01. 07. 2025).

[2] *donista - Shop online, donate and save the world.* donista - Shop online, donate and save the world. url: https://donista.org (besucht am 31. 05. 2025).

[3] Kristi L. Berg, Tom Seymour und Richa Goel. „History Of Databases“. In: *International Journal of Management & Information Systems (IJMIS)* 17.1 (31. Dez. 2012), S. 29–36. issn: 2157-9628, 1546-5748. doi: 10.19030/ijmis.v17i1.7587. url: https://clutejournals.com/index.php/IJMIS/article/view/7587 (besucht am 24. 06. 2025).

[4] Michael Hudson und British Museum, Hrsg. *Creating economic order: record-keeping, standardization, and the development of accounting in the Ancient Near East; a colloquium held at The British Museum, November 2000.* International Scholars Conference on Ancient Near Eastern Economies Vol. 4. Bethesda, Md: CDL, 2004. 354 S. isbn: 978-1-883053-85-7.

[5] M. P. Satija. *The Theory and Practice of the Dewey Decimal Classification System.* Google-Books-ID: 5mZEAgAAQBAJ. Elsevier, 30. Sep. 2013. 329 S. isbn: 978-1-78063-404-3.

[6] Jan L. Harrington. *Relational Database Design and Implementation.* Google-Books-ID: yQgfCgAAQBAJ. Morgan Kaufmann, 15. Apr. 2016. 714 S. isbn: 978-0-12-849902-3.

[7] Kevin Driscoll. „From Punched Cards to ”Big Data: A Social History of Database Populism“. In: (2012). Publisher: Scholarworks @ Umass Amherst. doi: 10.7275/R5B8562P. url: https://openpublishing.library.umass.edu/cpo/article/id/36/ (besucht am 24. 06. 2025).

[8] *What is a Vector Database & How Does it Work? Use Cases + Examples | Pinecone.* url: https://www.pinecone.io/learn/vector-database/ (besucht am 18. 05. 2025).

[9] *Vector Databases: Tutorial, Best Practices & Examples.* Nexla. url: https://nexla.com/ai-infrastructure/vector-databases/ (besucht am 18. 05. 2025).

[10] Ting Liu u. a. „An Investigation of Practical Approximate Nearest Neighbor Algorithms“. In: *Advances in Neural Information Processing Systems*. Bd. 17. MIT Press, 2004. url: https://proceedings.neurips.cc/paper_files/paper/2004/hash/1102a326d5f7c9e04fc3c89d0ede88c9-Abstract.html (besucht am 29. 05. 2025).

[11] *sentence-transformers/all-MiniLM-L6-v2 · Hugging Face*. 5. Jan. 2024. url: https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2 (besucht am 29. 05. 2025).

[12] Need a VectorDB for Your GenAI Apps?Zilliz Cloud is a managed vector database built on Milvus perfect for building GenAI applications Try Free. *What is the curse of dimensionality and how does it affect vector search?* url: https://milvus.io/ai-quick-reference/what-is-the-curse-of-dimensionality-and-how-does-it-affect-vector-search (besucht am 29. 05. 2025).

[13] Kawin Ethayarajh. *Rotate King to get Queen: Word Relationships as Orthogonal Transformations in Embedding Space*. 5. Sep. 2019. doi: 10.48550/arXiv.1909.00504. arXiv: 1909.00504[cs]. url: http://arxiv.org/abs/1909.00504 (besucht am 01. 07. 2025).

[14] Nils Reimers und Iryna Gurevych. *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. 27. Aug. 2019. doi: 10.48550/arXiv.1908.10084. arXiv: 1908.10084[cs]. url: http://arxiv.org/abs/1908.10084 (besucht am 27. 06. 2025).

[15] Ming Ding u. a. „CogLTX: Applying BERT to Long Texts“. In: *Advances in Neural Information Processing Systems*. Bd. 33. Curran Associates, Inc., 2020, S. 12792–12804. url: https://proceedings.neurips.cc/paper_files/paper/2020/hash/96671501524948bc3937b4b30d0e57b9-Abstract.html (besucht am 27. 06. 2025).

[16] *Logging*. In: *Wikipedia*. Page Version ID: 236085822. 3. Aug. 2023. url: https://de.wikipedia.org/w/index.php?title=Logging&oldid=236085822 (besucht am 24. 06. 2025).

[17] Hafedh Mili, Amel Elkharraz und Hamid Mcheick. *(PDF) Understanding separation of concerns*. ResearchGate. url: https://www.researchgate.net/publication/244446574_Understanding_separation_of_concerns (besucht am 24. 06. 2025).

[18] *Separation of concerns*. In: *Wikipedia*. Page Version ID: 1289713997. 10. Mai 2025. url: https://en.wikipedia.org/w/index.php?title=Separation_of_concerns&oldid=1289713997 (besucht am 24. 06. 2025).

[19] Daniel Valcarce u. a. „Assessing ranking metrics in top-N recommendation“. In: *Information Retrieval Journal* 23.4 (1. Aug. 2020), S. 411–448. issn: 1573-7659. doi: 10.1007/s10791-020-09377-x. url: https://doi.org/10.1007/s10791-020-09377-x (besucht am 25. 06. 2025).

[20] *Precision and recall at K in ranking and recommendations*. url: https://www.evidentlyai.com/ranking-metrics/precision-recall-at-k (besucht am 25. 06. 2025).

[21] *Mean Average Precision (MAP) in ranking and recommendations*. url: https://www.evidentlyai.com/ranking-metrics/mean-average-precision-map (besucht am 25. 06. 2025).

[22] *Normalized Discounted Cumulative Gain (NDCG) explained*. url: https://www.evidentlyai.com/ranking-metrics/ndcg-metric (besucht am 25. 06. 2025).

[23] Atlassian. *Microservices und monolithische Architektur im Vergleich*. Atlassian. url: https://www.atlassian.com/de/microservices/microservices-architecture/microservices-vs-monolith (besucht am 23. 06. 2025).

[24] Bartosz Czerniakowski. *Monolithic architecture vs microservices*. Cloudflight. 13. Jan. 2020. url: https://www.cloudflight.io/en/blog/monolithic-architecture-vs-microservices/ (besucht am 23. 06. 2025).

[25] *SQLAlchemy*. url: https://www.sqlalchemy.org (besucht am 24. 06. 2025).

[26] *ORM Quick Start — SQLAlchemy 2.0 Documentation*. url: https://docs.sqlalchemy.org/en/20/orm/quickstart.html#declare-models (besucht am 24. 06. 2025).

[27] *Basic Use — SQLAlchemy 1.3 Documentation*. url: https://docs.sqlalchemy.org/en/13/orm/extensions/declarative/basic_use.html (besucht am 24. 06. 2025).

[28] *Engine Configuration — SQLAlchemy 2.0 Documentation*. url: https://docs.sqlalchemy.org/en/20/core/engines.html (besucht am 24. 06. 2025).

[29] *Using the Session — SQLAlchemy 2.0 Documentation*. url: https://docs.sqlalchemy.org/en/20/orm/session.html (besucht am 24. 06. 2025).

[30] Martin Fowler. *Patterns of Enterprise Application Architecture*.

[31] jamesmontemagno. *Designing the infrastructure persistence layer - .NET*. url: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design (besucht am 24. 06. 2025).

[32] *pgvector/pgvector*. original-date: 2021-04-20T21:13:52Z. 28. Juni 2025. url: https://github.com/pgvector/pgvector (besucht am 28. 06. 2025).

[33] Need a VectorDB for Your GenAI Apps?Zilliz Cloud is a managed vector database built on Milvus perfect for building GenAI applications Try Free. *What are the key configuration parameters for an HNSW index (such as M and efConstruction/efSearch), and how does each influence the trade-off between index size, build time, query speed, and recall?* url: https://milvus.io/ai-quick-reference/what-are-the-key-configuration-parameters-for-an-hnsw-index-such-as-m-and-efconstructionefsearch-and-how-does-each-influence-the-tradeoff-between-index-size-build-time-query-speed-and-recall (besucht am 28. 06. 2025).

[34] *What is Ground Truth in Machine Learning? | Domino Data Lab*. url: https://domino.ai/data-science-dictionary/ground-truth (besucht am 24. 06. 2025).

[35] *Inter Annotator Agreement (IAA) | forTEXT*. url: https://fortext.net/ueber-fortext/glossar/inter-annotator-agreement-iaa (besucht am 24. 06. 2025).

[36] Kasra Hosseini u. a. *Retrieve, Annotate, Evaluate, Repeat: Leveraging Multimodal LLMs for Large-Scale Product Retrieval Evaluation*. 18. Sep. 2024. doi: 10.48550/arXiv.2409.11860. arXiv: 2409.11860[cs]. url: http://arxiv.org/abs/2409.11860 (besucht am 24. 06. 2025).

[37] Ron Miller. *Pinecone drops $100M investment on $750M valuation, as vector database demand grows*. TechCrunch. 27. Apr. 2023. url: https://techcrunch.com/2023/04/27/pinecone-drops-100m-investment-on-750m-valuation-as-vector-database-demand-grows/ (besucht am 29. 06. 2025).

[38] *Weaviate - 2025 Company Profile & Team - Tracxn*. 17. Juni 2025. url: https://tracxn.com/d/companies/weaviate/__6vJGlhlx8N1NQ5WqqS3DhNPZ8g9-UVgpCmtvJfycvOk (besucht am 29. 06. 2025).

[39] Weaviate. *Weaviate Raises $50 Million Series B Funding to Meet Soaring Demand for AI Native Vector Database Technology*. url: https://www.prnewswire.com/news-releases/weaviate-raises-50-million-series-b-funding-to-meet-soaring-demand-for-ai-native-vector-database-technology-301803296.html (besucht am 29. 06. 2025).

[40] Andre Zayarni Co-Founder CEO \&. *Announcing Qdrant's $28M Series A Funding Round - Qdrant*. url: https://qdrant.tech/blog/series-a-funding-round/ (besucht am 29. 06. 2025).

[41] *Chroma raises $18M seed round*. url: https://www.trychroma.com/blog/seed (besucht am 29. 06. 2025).

[42] Stephanie Palazzolo. *Vector database Chroma scored $18 million in seed funding at a $75 million valuation. Here's why its technology is key to helping generative AI startups.* Business Insider. url: https://www.businessinsider.com/vector-database-startup-chroma-raises-seed-funding-generative-artificial-intelligence-2023-4 (besucht am 29. 06. 2025).

A Anhang

A.1 Inhalt des beigefügten Datenträgers