Automatisierte Generierung von Datenpipelines

MASTERARBEIT

Choro Ulan Uulu

Eingereicht am 7. Mai 2024



Friedrich-Alexander-Universität Erlangen-Nürnberg Technische Fakultät, Department Informatik Professur für Open Source Software

Betreuer:

Johannes Jablonski, M. Sc. Prof. Dr. Dirk Riehle, M.B.A.



Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 7. Mai 2024

Lizenz

Diese Arbeit unterliegt der Creative Commons Attribution 4.0 International Lizenz (CC BY 4.0), https://creativecommons.org/licenses/by/4.0/

Erlangen, 7. Mai 2024

Anerkennungen

Ich möchte mich bei Johannes Jablonski, Philip Heltweg und Georg Schwarz für die schnelle und hilfreiche Betreuung meiner Masterarbeit bedanken.

Ich erkennen die Hilfe von ChatGPT bei der Verbesserung des Schreibens und ausschließlich des Schreibens dieses Artikels an.

Abbildungen wurden dank Draw.io generiert.

Abschließend möchte ich mich bei meiner Familie und meinen Freunden, insbesondere bei meiner Schwester, für die Unterstützung beim Korrekturlesen bedanken.

Abstract

The necessity of repeatedly writing code that requires little creative thinking can be perceived as monotonous and can consume time that clients might otherwise use for more important tasks. This process can be optimized through automation using a code generator that produces starter code.

In this paper, the methods of Scaffolding, Reinforcement Learning, and customized ChatGPT for code generation are analyzed. The focus is on comparing the speed of code generation and scalability. For this purpose, the system architectures of the solutions are implemented after comprehensive literature research, and the results collected by the evaluator are subsequently assessed. The evaluation criteria include the number of correctly generated Java models, the number of successfully generated SQLite and PostgreSQL databases, and the speed of code generation. The results show that using the latest technologies is not always advantageous. Scaffolding proves to be simpler in implementation and maintenance and generates code faster. Overall, this work provides a comprehensive overview of the possibilities for code generation.

Zusammenfassung

Die Notwendigkeit, wiederholt Code zu schreiben, der wenig kreatives Denken erfordert, kann als eintönig empfunden werden und Zeit in Anspruch nehmen, die Kunden für wichtigere Aufgaben nutzen könnten. Dieser Prozess lässt sich durch die Automatisierung mittels eines Codegenerators, der Startercode erzeugt, optimieren.

In dieser Arbeit werden die Methoden Scaffolding, Reinforcement Learning und benutzerdefiniertes ChatGPT zur Codegenerierung analysiert. Der Fokus liegt auf einem Vergleich der Geschwindigkeit der Codegenerierung und der Erweiterbarkeit. Hierfür werden die Systemarchitekturen der Lösungen nach umfassender Literaturrecherche implementiert, und die vom Evaluator gesammelten Ergebnisse werden anschließend bewertet. Die Bewertungskriterien umfassen die Anzahl korrekt erzeugter Java-Modelle, die Anzahl erfolgreich generierter SQLiteund PostgreSQL-Datenbanken sowie die Geschwindigkeit der Codegenerierung. Die Ergebnisse zeigen, dass der Einsatz der neuesten Technologien nicht immer von Vorteil ist. Scaffolding erweist sich als einfacher in der Implementierung und Wartung und generiert Code schneller. Insgesamt bietet diese Arbeit einen umfassenden Überblick über die Möglichkeiten der Codegenerierung.

Inhaltsverzeichnis

1	Ein	führung	1
2	Ver	wandte Arbeiten	3
	2.1	Jayvee	3
	2.2	Methoden der Codegenerierung	4
	2.3	Scaffolding	5
	2.4	Reinforcement Learning	6
		2.4.1 Q-Learning	7
		2.4.2 Deep Q-Networks	8
		2.4.3 "Epsilon-greedy"-Strategie	10
	2.5	ChatGPT	10
3	Δnf	Corderungen	13
J	3.1	Funktionale Anforderungen an Scaffolding	14
	$3.1 \\ 3.2$	Qualitätsanforderungen an Scaffolding	15
	$\frac{3.2}{3.3}$	Funktionale Anforderungen an Reinforcement Learning	16
	3.4	Qualitätsanforderungen an Reinforcement Learning	17
	3.5	Anforderungen an das benutzerdefinierte ChatGPT	18
			01
4		hitektur	21
	4.1	Evaluator	22
	4.2	Scaffolding	23
	4.3	Reinforcement Learning	25
		4.3.1 Reinforcement Learning Generation 1	25
		4.3.2 Reinforcement Learning Generation 2	27
	4.4	ChatGPT	28
5	Des	ign und Implementierung	31
	5.1	Evaluator	31
	5.2	Ermittlung der Spaltennamen und Spaltendatentypen	33
		5.2.1 Ermittlung der Spaltennamen und Spaltendatentypen Ge-	2.2
		neration 1	33

			Ermittlung der Spaltennamen und Spaltendatentypen Ge-	2.4
	5.3		neration 2	$\frac{34}{35}$
			ling	ან 38
	5.4		rement Learning	
	5.5		erdefiniertes ChatGPT	42
			Versuch 1	42
		5.5.2	Versuch 2	44
6	Eva	luation		47
	6.1	Automa	atisierte Code-Evaluierung	47
	6.2	Scaffold	ding-Ansatz	49
		6.2.1	Funktionale Anforderungen an Scaffolding	49
		6.2.2	Qualitätsanforderungen Anforderungen an Scaffolding	53
	6.3	Reinfor	cement-Learning-Ansatz	57
		6.3.1	Funktionale Anforderungen an Reinforcement Learning	57
		6.3.2	Qualitätsanforderungen an Reinforcement Learning	62
	6.4	Evaluat	tion der Anzahl der importierten Zeilen	64
	6.5	Anforde	erungen an das benutzerdefinierte ChatGPT	66
	6.6	Vergleie	ch der Ansätze	67
		6.6.1	Vergleich der Geschwindigkeit zwischen Ansätzen	67
		6.6.2	Vergleich der Erweiterbarkeit	68
7	Fazi	\mathbf{t}		69
$\mathbf{A}_{\mathbf{I}}$	ppen	dices		73
	A	Dialoge	e mit ChatGPT	75
		A.1	Dialog 1	75
		A.2	Dialog 2	76
	В		te Bibliotheken	78
Lii	terat	lirverz <i>e</i>	pichnis	79

Abbildungsverzeichnis

2.1	Workflow von DQN (Doshi, 2020)	9
4.1	Generische Architektur des Codegenerators	21
4.2	Flussdiagramm des Evaluierungssystems	23
4.3	Architektur des Scaffolding-Ansatzes	24
4.4	Flussdiagramm des Scaffolding-Ansatzes	25
4.5	Architektur des RL-Ansatzes	26
4.6	Aktivitätsdiagramm des RL-Ansatzes, dargestellt anhand eines Pro-	
	zesses	27
4.7	Aktivitätsdiagramm des RL-Ansatzes, dargestellt anhand mehre-	
	rer Prozesse	29
5.1	Sequenzdiagramm des Evaluierungssystems	33
5.2	Sequenzdiagramm des Scaffolding-Ansatzes	36
6.1	Beispiel einer XLSX-Datei, die nicht von der Ursprungs- in die	
	Zieldatenbank importiert werden konnte	55
6.2	Vergleich der Trainingszeiten zwischen der 1. und 2. Generation	
	des RL-Ansatzes	60

Tabellenverzeichnis

3.1	Funktionalle Anforderungen an die 1. Generation des Scaffolding-Ansatzes	1
3.2	Funktionalle Anforderungen an die 2. Generation des Scaffolding-	1
3.3	Ansatzes	10
3.4	Qualitätsanforderungen an die 2. Generation des Scaffolding-Ansatzes	10
3.5	Funktionalle Anforderungen an die 1. Generation des RL-Ansatzes	1
3.6	Funktionalle Anforderungen an die 2. Generation des Reinforcement	
	Learning (RL)-Ansatzes	1
3.7	Qualitätsanforderungen an die 1. Generation des RL-Ansatzes	18
3.8	Qualitätsanforderungen an die 2. Generation des RL-Ansatzes	18
6.1	Zusammenfassung der erfüllten funktionalen Anforderungen an die	
	1. Generation des Scaffolding-Ansatzes	4
6.2	Vergleich der 1. und 2. Generation des Scaffolding-Ansatzes in Be-	
	$zug\ auf\ die\ erzeugten\ Jayvee-Modelle\ und\ SQLite-Datenbanken .$	4
6.3	Vergleich der Anzahl der abgelesenen Zeilen aus der CSV- und	
	XLSX-Ursprungsdatenbanken in SQLite-Zieldatenbanken zwischen	٦
	der 1. und 2. Generation des Scaffolding-Ansatzes	5
6.4	Zusammenfassung der erfüllten funktionalen Anforderungen an die	_
	2. Generation des Scaffolding-Ansatzes	5
6.5	Vergleich der 1. und 2. Generation des Scaffolding-Ansatzes in Be-	_
	zug auf die erzeugten Jayvee-Modelle und Postgres-Datenbanken .	5
6.6	Zusammenfassung der erfüllten Qualitätsanforderungen an die 1.	_
	Generation des Scaffolding-Ansatzes	5
6.7	Vergleich der 1. und 2. Generation des Scaffolding-Ansatzes hin-	
	sichtlich der Gesamtgeschwindigkeit bei der Erzeugung von SQLite-	
	Zieldatenbank aus 6 CSV- und 6 XLSX-Ursprungsdatenbanken	5°

6.8	Die Geschwindigkeit der 2. Generation des Scaffolding-Ansatzes	
	bei der Erzeugung von SQLite-Zieldatenbanken für größere Da-	
	tenmengen	55
6.9	Zusammenfassung der erfüllten Qualitätsanforderungen an die 2.	
	Generation des Scaffolding-Ansatzes	56
6.10	Zusammenfassung der erfüllten funktionalen Anforderungen an die	
	1. Generation des RL-Ansatzes	57
6.11	Vergleich der 1. und 2. Generation des RL-Ansatzes in Bezug auf	
	die erzeugten Jayvee-Modelle und SQLite-Datenbanken	57
6.12	Zusammenfassung der erfüllten funktionallen Anforderungen an	
	die 2. Generation des RL-Ansatzes	59
6.13	Geschwindigkeitsvergleich der Codegenerierung für Modelle, die	
	mit 12, 1.000 und 10.000 Datenbanken in der 2. Generation trai-	
	niert werden	60
6.14	Geschwindigkeitsvergleich der Codegenerierung für Modelle, trai-	
	niert mit 3 bzw. 300 Episoden in der zweiten Generation	61
6.15	Anzahl der getesteten Ursprungsdatenbanken ohne Spaltennamen	61
	Vergleich der 1. und 2. Generation des RL-Ansatzes in Bezug auf	
	die erzeugten Jayvee-Modelle und Postgres-Datenbanken	62
6.17	Zusammenfassung der erfüllten Qualitätsanforderungen an die 1.	
	Generation des RL-Ansatzes	62
6.18	Zusammenfassung der erfüllten Qualitätsanforderungen an die 2.	
	Generation des RL-Ansatzes	63
6.19	Vergleich der Geschwindigkeit der Ansätze zur Codegenerierung	
	Scaffolding und RL	68
	O Company of the comp	

Akronyme

ACE Automatisierte Code-Evaluierung

LM Language Model

LLM Large Language Model

RLHF Reinforcement Learning from Human Feedback

DRL Deep Reinforcement Learning

DQL Deep Q-Learning

DQN Deep Q-Networks

QL Q-Learning

QN Q-Network

TN Target Network

QV Q-Value

RL Reinforcement Learning

Sc Scaffolding

RC Retargierbare Codegenerierung

CLI Command-Line-Interface

AR Aktionsraum

ER Experience Replay

EG "Epsilon-greedy"-Strategie

DSL Domänenspezifische Sprache

1 Einführung

Die Codegenerierung spielt eine zunehmend wichtige Rolle im Software Engineering. Diese Arbeit beschäftigt sich mit der Codegenerierung, einem Prozess, bei dem Code durch ein Skript erzeugt wird, das der Nutzer durch Parameter steuern kann. Dieses Konzept ist seit den Anfängen des Software Engineerings bekannt und zielte darauf ab, repetitive Codierungsaufgaben zu automatisieren und auf Skripte zu übertragen. Codegenerierung trägt dazu bei, Zeit und Ressourcen zu sparen, sodass Entwickler sich auf Herausforderungen konzentrieren können, bei denen menschliche Problemlösungsfähigkeiten im Vordergrund stehen, statt sich mit wiederkehrenden Codieraufgaben zu beschäftigen.

In dieser Masterarbeit wird automatische Codegenerierung genutzt, um Nutzern Startercode für Datenpipelines bereitzustellen. Dieser Code basiert auf Jayvee-Modellen, die speziell für die Aufbereitung von Daten in Zieldatenbanken entwickelt wurden. Die generierten Modelle können bei Bedarf angepasst werden. Ziel ist es, für jede valide bereitgestellte Ursprungsdatenbank (CSV oder XLSX) ein entsprechendes Jayvee-Modell zu erstellen. Aus diesen Modellen werden Zieldatenbanken wie SQLite und PostgreSQL (Postgres) generiert. Im Rahmen dieser Arbeit werden verschiedene Methoden der automatischen Codegenerierung analysiert und miteinander verglichen. Zusätzlich wird analysiert, wie sich die spezifischen Eigenschaften der Zieldatenbanken (SQLite, Postgres) auf die Anzahl der erzeugten Zieldatenbanken auswirkt.

In den letzten Jahren haben insbesondere die Ansätze zur Codegenerierung wie Scaffolding, Reinforcement Learning (RL) und ChatGPT erhebliche Aufmerksamkeit erlangt. Diese Technologien wurden für diese Masterarbeit ausgewählt, da sie eine schnelle Codegenerierung sowie eine einfache Implementierung und Wartung bieten und zudem hochrelevant sind. Scaffolding beschleunigt den Entwicklungsprozess durch den Einsatz vordefinierter Strukturen. Reinforcement Learning ermöglicht adaptive Lösungen, die sich dynamisch anpassen, und ChatGPT revolutioniert die Codegenerierung durch den Einsatz fortschrittlicher maschineller Lernverfahren.

Diese Ansätze werden im Hinblick auf ihre Eignung für die automatische Co-

degenerierung sorgfältig untersucht und bewertet. Nach der Analyse verwandter Arbeiten (Kapitel 2) zu diesen Methoden wurden Codegenerierungs-Ansätze wie folgt entwickelt: zuerst wurden passende Architekturen (Kapitel 4) entworfen, darauf basierend die Methoden implementiert (Kapitel 5) und schließlich evaluiert (Kapitel 6).

Die Entwicklung in zwei Generationen wird hervorgehoben, um den Fortschritt deutlich nachzuverfolgen und die Ergebnisse effektiv zu vergleichen.

Die Anforderung an die 1. Generation besteht darin, ein Modell zu entwickeln, das einen Link zu einer CSV-Datei, die Daten und zwingend Spaltennamen im Header enthält, empfängt und daraus Startercode generiert, wobei alle Spaltendatentypen als 'text' gespeichert werden.

Für die 2. Generation besteht zusätzlich die Anforderung, XLSX-Dateien, die entweder klare Spaltennamen aufweisen oder unmittelbar mit Daten beginnen, als Ergänzung zu CSV-Dateien als Ursprungsdatenbanken einzusetzen. Zudem soll neben SQLite auch Postgres als weitere Zieldatenbank integriert werden.

In dieser Arbeit, werden folgende Forschungsfragen beantwortet:

- 1. Welcher Ansatz eignet sich besonders zur Erstellung von Startercodes für Domänenspezifische Sprache (DSL) Jayvee?
- 2. Welcher Ansatz generiert schnellen und sinnvoll nutzbaren Code? Dies bedeutet, dass der Jayvee-Code darauf ausgelegt ist, eine Zieldatenbank effizient zu erzeugen oder das Erzeugen einer solchen Datenbank anstrebt.

Die Ergebnisse werden von einem Evaluator bewertet, der sowohl das erzeugte Modell als auch die durch die Ausführung dieses Modells generierten Daten beurteilt. Die Bewertungskriterien der Ansätze umfassen die Geschwindigkeit der Codegenerierung und die Erweiterbarkeit. Zusätzlich wurde in der finalen Version der Codegenerierungsansätze, die in dieser Masterarbeit als 2. Generation definiert ist, die Anzahl der korrekt erzeugten Jayvee-Modelle sowie die Anzahl erfolgreich generierter SQLite- und Postgres-Datenbanken getestet, einschließlich der korrekten Erkennung von Spaltennamen und Spaltendatentypen. Im Rahmen des RL-Ansatzes werden zusätzlich die Trainingszeiten zwischen den Generationen verglichen sowie der Einfluss der Anzahl der Datenbanken auf die Codegenerierungsgeschwindigkeit untersucht.

Diese Masterarbeit versteht sich als "Engineering Thesis", die traditionelle ingenieurwissenschaftliche Arbeit durchführt und präsentiert, wobei der Schwerpunkt meist auf Software-Design und -Implementierung liegt (Professorship for Open-Source Software, o. D).

2 Verwandte Arbeiten

Dieses Kapitel bietet eine Zusammenfassung und Diskussion relevanter Literatur für diese Masterarbeit. Es beginnt mit einer Darstellung der vorangegangenen Forschung im Bereich der Codegenerierung. Abschnitt 2.3 widmet sich Scaffolding, seinen Definitionen sowie früheren Projekten, die Scaffolding für die Codeerstellung eingesetzt haben. Abschnitt 2.4 beschreibt die Funktionsweise von Reinforcement Learning, definiert seine Grundlagen und untersucht frühere Forschungen, die diese Methode zur Codegenerierung verwendet haben. Abschließend wird in Abschnitt 2.5 erläutert, wie die Codegenerierung mit ChatGPT funktioniert, warum kein Large Language Model (LLM) trainiert wird, und es werden Studien aufgeführt, die ChatGPT oder LLMs für die Codegenerierung verwenden.

2.1 Jayvee

In der vorliegenden Arbeit wird Code für Jayvee, eine DSL zur automatischen Verarbeitung von Datenpipelines, generiert (JValue Contributors, 2024). Eine Datenpipeline ist eine Struktur, die es ermöglicht, Daten aus Ursprungsdatenbanken zu extrahieren, zu bereinigen und für Zieldatenbanken aufzubereiten. Eine DSL ist eine Programmiersprache, die speziell für die Lösung von Problemen in einem bestimmten Anwendungsbereich entwickelt wird. Jayvee verfügt über eine eigene Syntax, die mithilfe verschiedener Blocktypen Pipelines definiert. Diese Blöcke lassen sich wie folgt kategorisieren:

- "Extractor"-Blöcke, die Daten aus den Ursprungsdatenbanken extrahieren. Beispielsweise gibt es Extractors für CSV- und XLSX-Ursprungsdatenbanken.
- "Transformator"-Blöcke, die für die Transformation der Daten zuständig sind.
- "Loader"-Blöcke, die die Daten in die Zieldatenbanken laden. Beispielsweise existieren Loader für SQLite- (Hipp, 2020) und Postgres-Zieldatenbanken (Group, 2018).

2.2 Methoden der Codegenerierung

Die Generierung von Code ist ein vielschichtiges Forschungsfeld mit einer Vielzahl an Methoden (Sun et al., 2020) (Yang et al., 2021), (Le et al., 2022). Es existieren zahlreiche grundlegende Arbeiten zu den Prinzipien und Methoden der Codegenerierung (Dehaerne et al., 2022; Herrington, 2003).

Es gibt folgende klassische Ansätze für die Codegenerierung:

- 1. Retargierbare Codegenerierung (RC): Bei RC wird der Codegenerator so implementiert, dass er angepasst werden kann, um Code für neue Umgebungen zu erzeugen. Es gibt drei Hauptarten von RC(Ganapathi et al., 1982): (i) Interpretive Codegeneratoren. (ii) Pattern matching Codegeneratoren. (iii) Table-driven Codegeneratoren.
- 2. Scaffolding: Ein Ansatz, die Muster nutzt, um auf Basis von Parametern Code zu generieren. Ähnlich wie RC zielt Scaffolding jedoch darauf ab, Startcode zu generieren, während RC verwendet wird, um den Code flexibler für verschiedene Plattformen und Umgebungen zu gestalten. In dieser Arbeit wird der Scaffolding-Ansatz eingesetzt, da dieser im Vergleich zu anderen Verfahren als besonders intuitiv angesehen wird.

Die folgenden Methoden gelten derzeit als führende moderne Machine Learning-Methoden im Bereich der Codegenerierung:

- 1. Reinforcement Learning (RL): Ein Ansatz des maschinellen Lernens, der Agenten durch Versuch und Irrtum trainiert (Achiam, 2018a). Beispielsweise könnte der Agent eine Aktion auswählen und eine bestimmte Belohnung erhalten, oder er könnte eine andere Aktion wählen und zu einem späteren Zeitpunkt insgesamt mehr Belohnungen im Vergleich zur ersten Aktion erhalten. Auf diese Weise lernt der Agent. Diese Methode wird angewandt, wenn keine Datenbank mit Einträgen in Frage-Antwort-Form verfügbar ist, wie es in dieser Arbeit der Fall ist. Eine ausführliche Diskussion erfolgt in Abschnitt 2.4.
- 2. Deep Reinforcement Learning (DRL): DRL ist eine Weiterentwicklung von RL, die neuronale Netze nutzt. Ein bekanntes Beispiel hierfür ist das Programm AlphaZero, das herausragende Schachspieler besiegen konnte (Silver, 2017). Forscher entwickelten ein Modell, das ein Language Model (LM) als Actor-Netzwerk zur Generierung von Code und ein weiteres LM als Critic-Netzwerk zur Bewertung dieses Codes verwendet. Die Ergebnisse dieser Bewertungen werden genutzt, um das Actor-Modell kontinuierlich zu verbessern (Le et al., 2022). Weiterhin wird DRL eingesetzt, um Datenpipelines für gängige Programmiersprachen zu entwickeln (Yang et al., 2021).

3. Transformers: Die Autoren Sun et al., 2020 nutzen die Transformer-Architektur, um grammatikalische Regeln von Programmiersprachen vorherzusagen. Ein berühmtes Beispiel hierfür ist ChatGPT, ein LLM, das mit einer Datenbank trainiert wird, die häufig Daten in Form von Fragen und Antworten enthält. Es wird schließlich mit Reinforcement Learning from Human Feedback (RLHF) verfeinert. RLHF ist eine spezielle Form von RL, die menschliche Bewertungen nutzt, um die Ergebnisse zu evaluieren und Belohnungen an das Modell zurückzugeben.

2.3 Scaffolding

In dieser Arbeit wird das Konzept des Scaffoldings verwendet. Scaffolding definiert präzise einen Erzeugungsalgorithmus, der die Codegenerierung ermöglicht. Der so erzeugte Code kann später für spezifische Anwendungsfälle angepasst werden. Diese Methode ermöglicht die schnelle Erzeugung von allgemeinem Code für Datenbanksysteme wie SQLite und Postgres. Die Arbeit erläutert die Implementierung dieses Ansatzes.

Ein bekanntes Beispiel für Scaffolding ist die Erstellung von Starter-Projekten im Angular Framework. Um in Angular ein Projekt zu erstellen, verwendet man den Befehl ng new 'Projekt-Name'. Der angegebene Projektname wird in eine vordefinierte Vorlage eingefügt, woraufhin das Projekt einschließlich Ordnerstruktur und Klassennamen automatisch erstellt wird.

Ein weiteres Beispiel ist Yeoman (Yeoman, 2023). Yeoman kann Ordnerstrukturen und Startercode generieren.

Scaffolding erweist sich als nützlich, wenn umfangreicher Boilerplate-Code vorhanden ist. Boilerplate-Code bezeichnet Code, der an vielen Stellen eingefügt wird, ohne dass wesentliche Änderungen notwendig sind (Laine, 2019). In dieser Arbeit zielt man darauf ab, Startercode für eine große Anzahl von Datenbanken zu generieren. Startercode dient dazu, die Daten für die jeweilige Datenbankart (z.B. SQLite, Postgres) zu initialisieren. Er ist oft repetitiv und kann automatisch mit Scaffolding erstellt werden. Der Nutzer hat zudem die Möglichkeit, diesen Code anzupassen.

Der Scaffolding-Ansatz sollte folgenden Komponenten umfassen:

- Generator: Eine Anwendung, die speziell für Scaffolding-Zwecke entwickelt wird, oft in Form einer Webanwendung oder eines Command-Line-Interface (CLI).
- 2. Vorlagen: Muster des Startercodes (CSV-, XLSX-, SQLite- und Postgres-Startercodes), der generiert werden soll. Diese Muster können jederzeit erweitert werden.

2.4 Reinforcement Learning

RL wird aufgrund seiner Aktualität und Relevanz im Bereich des maschinellen Lernens ausgewählt. Jayvee verfügt aufgrund ihrer relativen Neuheit noch nicht über umfangreiche Beispielcodes. Weil uns keine Daten zur Verfügung stehen, ist ein Ansatz basierend auf Deep Learning nicht umsetzbar. Aus diesem Grund wird für RL entschieden, da RL die Erstellung eines Modells zur Codegenerierung ohne vorhandene Daten ermöglicht.

Der Ansatz umfasst folgende Komponenten:

- 1. Ein Agent ist eine Einheit, die seine Umgebung wahrnehmen, interpretieren, mit ihr interagieren und durch Versuch und Irrtum lernen kann (Hashemi-Pour, 2023).
- 2. Eine Aktion ist ein Schritt, der in der Umgebung des Agenten angewendet werden kann.
- 3. Die Umgebung ist die Welt, in welcher der Agent existiert und mit der er interagiert. Die Umgebung gibt, je nach der vom Agenten gewählten Aktion, eine Rückmeldung in Form einer positiven oder negativen Zahl, die zur kumulativen Belohnung hinzugefügt wird.
- 4. Der Zustand ist die aktuelle Beschreibung der Umgebung.
- 5. Aktionsraum (AR) ist der Raum aller möglichen Aktionen.
- 6. Die Strategie ist das gelernte Verhalten des Agenten (Lemke, 2023).
- 7. Eine Episode beschreibt eine Sequenz von Aktionen, die zu einer Belohnung führen. Die Episode endet nicht, solange keine Belohnung erreicht wird.

In dieser Arbeit wird RL zur Codegenerierung eingesetzt. Ein Agent wird konzipiert und eine Umgebung entwickelt, um diesen Agenten zu trainieren.

Nach Achiam, 2018b gibt es zwei Typen von RL-Algorithmen:

- 1. Modellbasiert: Der Agent hat Zugang zum Modell der Umgebung, was bedeutet, dass es eine Funktion gibt, die vorhersagt, wie die Umgebung auf Aktionen reagieren wird.
- 2. Modellfrei: Algorithmen, die keinen Zugang zum Modell der Umgebung haben.

Da in dieser Arbeit kein Zugang zum Modell der Umgebung besteht, wird ein modellfreier Algorithmus ausgewählt. Nach Achiam, 2018b werden Modellfreie Algorithmen wie folgt unterteil:

- Q-Learning (QL) ist ein modellfreier Algorithmus, der Tabellen verwendet, um für jedes Aktion-Zustand-Paar eine zukünftige Belohnung zu speichern.
- Policy Optimization ist ein modellfreier Algorithmus, der darauf abzielt, die Entscheidungsstrategie eines Lernagenten zu verbessern. Eine weiterführende Erläuterung wird in dieser Arbeit nicht vorgenommen, da sie nicht den Fokus der Masterarbeit bildet.

Für die 1. Generation haben wir uns für Deep Q-Learning (Deep Q-Learning (DQL)) entschieden, eine fortgeschrittene Form von Q-Learning (QL), da es verbesserte Lernfähigkeiten bietet, ohne erheblich mehr Speicher zu verbrauchen und einfach zu implementieren bleibt. In der 2. Generation planen wir, Deep Q-Networks (Deep Q-Networks (DQN)) (Mnih et al., 2013) zu verwenden, um die Vorteile einer weiterentwickelten Verarbeitung zu nutzen. In Zukunft könnte die Anwendung von Policy Optimization mit mehr Ressourcen getestet und die Leistung mit DQN verglichen werden.

Um DQN zu verstehen, wird zunächst QL diskutiert (Watkins & Dayan, 1992).

2.4.1 Q-Learning

QL ist ein modellfreier Algorithmus aus dem Bereich des RL, der Tabellen nutzt, um Zustände, Aktion und Belohnungen zu speichern. Basierend auf diesen Werten wird die Aktion gewählt. Ziel ist es, die erwartete Rendite nach der Formel 2.1 zu maximieren (Decker, 2023). Im Verlauf des Trainings werden diese Werte aktualisiert, sodass der Agent die optimale Strategie lernt. Diese Tabellen können sehr groß werden, was sie ineffizient macht, da die Größe der Tabelle exponentiell mit der Anzahl der Zustände und Aktionen wächst.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right)$$
 (2.1)

Dabei bezeichnet:

- $Q(s_t, a_t)$ den Q-Value (QV),
- s den aktuellen Zustand,
- r die erhaltene Belohnung,
- a die ausgewählte Aktion,
- α die Lernrate (zwischen 0 und 1),
- t den aktuelle Zeitpunkt,
- γ den Diskontfaktor (zwischen 0 und 1).

DQL ist eine Form von QL, bei der statt einer Tabelle ein neuronales Netz, das Q-Network (QN) genannt wird, verwendet wird (Luu, 2024). QN wird folgendermaßen genutzt:

- 1. Ein Zustand wird eingegeben und Werte zurückgegeben.
- 2. Ein sequenzielles Modell wird verwendet, da stets eine Eingabe vorhanden ist. Ein sequenzielles Modell ist ein Modell, bei dem es immer eine Eingabe und eine Ausgabe gibt.

DQL repräsentiert eine fortschrittliche Form des QL, entwickelt um spezifische Limitationen von QL, wie ineffiziente Speichernutzung, zu adressieren (Luu, 2024). Anders als QL, das auf Tabellen basiert, nutzt DQL neuronale Netze, um optimale Handlungen zu bestimmen. Ein weiteres zentrales Element von DQL ist das Konzept des Experience Replay (ER), welches Paare von Aktionen, Belohnungen und Zuständen sammelt. Diese dienen als Trainingsdaten für das neuronale Netz. Diese Methode ist entscheidend, da das Netzwerk ohne eine Vielfalt an Trainingsdaten nicht konvergieren würde, sprich es könnte nicht die optimale Strategie erlernen. Das Hauptziel von DQL ist es, eine Konvergenz des Netzes zu erreichen.

DQL wird in der 1. Generation der Implementierung verwendet.

2.4.2 Deep Q-Networks

Eine Weiterentwicklung von DQL ist DQN, ein Algorithmus aus dem Bereich des RL. Der Hauptunterschied zwischen DQL und DQN in dieser Arbeit besteht darin, dass DQN ein sogenanntes Target Network (TN) verwendet, eine Kopie des QN, das dazu dient, das Modell zu stabilisieren (Doshi, 2020). Der Algorithmus wird nach Doshi, 2020 schrittweise erklärt:

- 1. Zunächst wird eine Aktion basierend auf der "Epsilon-greedy"-Strategie (EG), erklärt in 2.4.3, ausgewählt.
- 2. Wie in Abbildung 2.1 dargestellt, speichert der Experience Replay (ER) die Rückmeldungen bestehend aus der ausgewählten Aktion, dem Zustand vor und nach der Aktion sowie der erhaltenen Belohnung für eine Reihe von Aktionen.
- 3. Die Berechnung des vorhergesagten QV erfolgt dann. Ein Datenbatch wird so ausgewählt, dass es eine Mischung aus verschiedenen Werten enthält. Dieses Batch wird in das QN eingespeist, das den Zustand vor der Aktion und die Aktion nimmt und den QV für diese spezifische Aktion vorhersagt. Vorhergesagte QV repräsentieren die erwartete Rendite, die erzielt wird, wenn die Entscheidung getroffen wird, diese Aktion zu nutzen.

- 4. Dasselbe Batch wird in das TN eingespeist, allerdings mit dem Unterschied, dass statt des Zustands vor der Aktion der Zustand nach der Aktion eingegeben wird, und der QV vorhergesagt wird, der dann als Ziel-QV bezeichnet wird. Die Ziel-QV werden nach Decker, 2023 mit der Formel 2.2 berechnet.
- 5. Anschließend werden die QN, die Ziel-QV und die beobachtete Belohnung verwendet, um den Verlust zu berechnen. Dieser Verlust dient zum Training des QN. Das TN wird nach N Schritten durch Kopieren der Gewichte des QN aktualisiert.

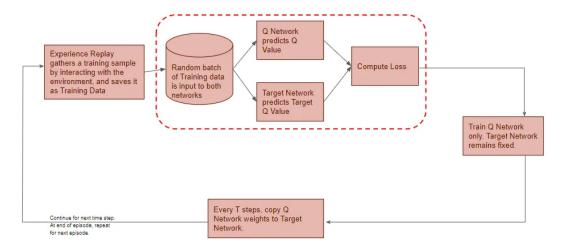


Abbildung 2.1: Workflow von DQN (Doshi, 2020)

Q-Target =
$$r + \gamma \max_{a'} Q(s', a'; \theta^-)$$
 (2.2)

wobei:

- r die erhaltene Belohnung,
- γ der Diskontfaktor (zwischen 0 und 1),
- $Q(s_t, a_t)$ ist der vorhersagte QV nach dem Zustand s',
- s' der nächste Zustand,
- a' ist eine mögliche Aktion nachdem Zustand s',
- θ^- die Gewichte des TN

Dieser Algorithmus wird in der 2. Generation der Implementierung verwendet.

Die Auswahl der Aktionen erfolgt gemäß der EG (Sutton & Barto, 2018).

2.4.3 "Epsilon-greedy"-Strategie

Dieser Punkt beschreibt den Prozess, der verhindert, dass der Agent stets dieselben Aktionen wählt. Es ist wichtig, dass der Agent entscheiden kann, ob er eine gelernte Aktion oder eine zufällige Aktion wählt, basierend darauf, ob er eine sofortige Belohnung ausnutzen oder alternative Wege erkunden möchte, die kurzfristig zu Sanktionen führen können, aber langfristig vorteilhafter sein könnten. Da simultanes Experimentieren und Absichern nicht möglich ist, besteht ein notwendiger Kompromiss zwischen Ausnutzung und Exploration. Ziel ist es, ein Gleichgewicht zwischen beiden zu finden (Baeldung, 2023).

Gemäß der Formel 2.3 (Decker, 2023) wird die Wahl der Aktion implementiert:

$$a = \begin{cases} \operatorname{argmax}_{a} Q(s, a; \theta) & \text{mit Wahrscheinlichkeit } 1 - \epsilon, \\ \operatorname{zuf\"{a}llige Aktion} & \operatorname{mit Wahrscheinlichkeit } \epsilon. \end{cases}$$
 (2.3)

- 1. Wenn eine zufällig ausgewählte Zahl kleiner oder gleich ϵ ist, wird die Explorationsstrategie gewählt. Falls ϵ null ist, findet keine Exploration statt. Ist ϵ gleich eins, erfolgt ausschließlich Exploration. Üblicherweise wird ϵ nahe null gesetzt (Baeldung, 2023). Es existieren jedoch viele Methoden zur Auswahl von ϵ .
- 2. Andernfalls wird die vom neuronalen Netz vorgeschlagene Aktion gewählt.

2.5 ChatGPT

Zum Vergleich mit Scaffolding und RL wird ChatGPT herangezogen. Es gibt Versuche, LLM zur Codegenerierung einzusetzen (Mülder, 2023). Da nur wenige Beispiele für korrekt erzeugten Jayvee-Code vorhanden sind (fünf Beispiele in der Dokumentation) und aus den in Mülder, 2023 genannten Gründen, wird in dieser Arbeit kein LLM trainiert, sondern ein angepasstes ChatGPT-Modell entwickelt.

Es gibt verschiedene Methoden, ChatGPT zu trainieren:

- 1. Zero-Shot: Diese Methode ermöglicht es, basierend auf initialem Wissen eine Antwort von ChatGPT oder einer LLM zu erhalten, ohne dass der Nutzer das Modell trainiert.
- 2. One-Shot: Diese Methode ermöglicht es, Wissen von ChatGPT abzufragen, indem zusätzliche Anweisungen und Kontext bereitgestellt werden. Der Nutzer gibt mehr Kontext oder ein Beispiel dessen, was er von einer LLM erwartet.

In Jain et al., 2023 wird gezeigt, wie man ChatGPT mittels One-Shot beibringt, Code in einer DSL zu generieren.

3. Few-Shot: Diese Methode ermöglicht es, Wissen von ChatGPT iterativ abzufragen. Falls zum Beispiel die LLM die Frage falsch beantwortet, korrigiert der Nutzer die Antwort und gibt zusätzliche Beispiele sowie Kontext.

3 Anforderungen

Das Ziel dieser Masterarbeit ist die Entwicklung eines Codegenerierungssystems für Jayvee. Dabei werden drei Ansätze untersucht: Codegenerierung durch "Scaffolding", "Reinforcement Learning" und den Einsatz eines "benutzerdefinierten ChatGPTs". Um diese Ansätze umzusetzen, ist es zunächst erforderlich, die Anforderungen zu definieren.

Die Masterarbeit wird nach einem agilen Ansatz verfasst, da die Anforderungen zu Beginn unklar waren und sich in Absprache mit den Betreuern im Laufe der Entwicklung weiterentwickelt haben. Die Anforderungen, die in diesem Kapitel aufgelistet sind, werden fortlaufend gemeinsam mit den Betreuern besprochen und präzisiert.

Es liegen sowohl funktionale als auch qualitätsbezogene Anforderungen vor.

"Eine funktionale Anforderung ist eine Anforderung bezüglich eines Ergebnisses eines Verhaltens, das von einer Funktion eines Systems bereitgestellt werden soll" (Pohl, 2011).

"Eine Qualitätsanforderung ist eine Anforderung, die sich auf ein Qualitätsmerkmal bezieht, das nicht durch funktionalle Anforderungen abgedeckt wird" (Pohl, 2011).

Da natürliche Sprache nicht das ideale Mittel zur Definition von Anforderungen ist, müssen entsprechende Methoden implementiert werden, um sicherzustellen, dass die Anforderungen eine hohe Qualität aufweisen.

Gemäß Zowghi und Gervasi, 2002 existieren zwei Kriterien zur Bewertung der Qualität von Anforderungen:

1. Interne Vollständigkeit: Dies bedeutet, dass die Anforderungen korrekt sind. Korrekte Anforderungen sind sowohl vollständig als auch konsistent. Konsistenz impliziert, dass keine internen Widersprüche vorliegen. Vollständigkeit bedeutet, dass die Anforderungen alle Aspekte der Software abdecken. Wenn beide Bedingungen erfüllt sind, gilt die Software als intern vollständig. In der Praxis wird Vollständigkeit jedoch oft durch die Erfüllung spe-

zifischer Geschäftsziele definiert, die für den Kunden von Bedeutung sind.

2. Externe Vollständigkeit: Sie gewährleistet, dass alle Informationen, die zur Definition der Herausforderungen erforderlich sind, in der Spezifikation enthalten sind.

Im Kapitel 3 definierten Anforderungen wird angestrebt, Widersprüche zu vermeiden und Konsistenz zu erreichen. Die implementierten Ansätze erfüllen die vorgegebene Aufgabe, Code zu generieren, wodurch die internen Vollständigkeitskriterien erfüllt sind.

Während der gesamten Masterarbeit wird ein kontinuierlicher Austausch mit den Betreuern gepflegt, um die Anforderungen zu diskutieren und festzulegen, die in Kapitel 3 dokumentiert sind. Dies zielt auf die externe Vollständigkeit ab.

3.1 Funktionale Anforderungen an Scaffolding

Die Anforderungen an die 1. Generation des Scaffolding-Ansatzes sind in Tabelle 3.1 dargestellt.

Da CSV eines der gängigsten Formate für Datenbanken ist, ist es entscheidend, dieses Format für die Generierung von Jayvee-Code zu implementieren. Ebenso ist die Unterstützung von SQLite als Zieldatenbank wichtig. SQLite wird als Ausgangspunkt gewählt, weil es die meisten Beispiele für korrekte Jayvee-Pipelines bietet. Ein weiterer wichtiger Punkt ist das Auslesen der Spaltennamen, um Datenverluste zu vermeiden. Da die Erkennung der Datentypen zunächst nicht implementiert war, entschied man sich dafür, alle Daten vorerst als "text" zu speichern.

Anforderungen	ID
Der Ansatz unterstützt ausschließlich den CSVExtractor.	FA_S1_Gen1
Der Ansatz sollte in der Lage sein, ein korrektes Jayvee-Modell zu	FA_S2_Gen1
erstellen, um eine SQLite-Datenbank zu generieren.	
Der Ansatz soll genau die verfügbaren Spaltennamen auslesen können.	FA_S3_Gen1
Der Ansatz sollte in der Lage sein, sämtliche Datenspalten-Typen als	FA_S4_Gen1
"Text" zu speichern. In der 1. Generation wird davon ausgegangen, dass	
alle CSV-Ursprungsdatenbanken Spaltennamen enthalten.	

Tabelle 3.1: Funktionalle Anforderungen an die 1. Generation des Scaffolding-Ansatzes

Die Anforderungen an die 2. Generation sind in Tabelle 3.2 dargestellt.

Die 2. Generation des Scaffolding-Ansatzes erweitert die Unterstützung zusätzlich zu CSV-Datenbanken um XLSX-Datenbanken, da diese zu den gängigsten Datenbankformaten gehören. Zudem soll eine leichtgewichtige Lösung implementiert

werden, die passende Spaltennamen für Spalten ohne vorhandene Bezeichnungen generiert, da viele Datenbanken keine Spaltennamen aufweisen. Aufgrund der Popularität und der Tatsache, dass es sich bei Postgres um ein Open-Source-Datenbankmanagementsystem handelt, wird ebenfalls beschlossen, dass dieses System unterstützt werden soll. Schließlich sollte ein Weg eingeführt werden, der die richtigen Datentypen für Spalten auswählt, da die Daten oft nicht einheitlich sind und die Wahl eines unpassenden Datentyps dazu führen kann, dass Daten, die diesem Typ nicht entsprechen, von Jayvee nicht importiert werden.

Anforderungen	ID
Erweiterte Unterstützung von Datenquellen: Neben CSV-Dateien wird	FA_S1_Gen2
nun auch die Unterstützung für XLSX-Datenquellen angeboten. Es wird	
vorausgesetzt, dass die XLSX-Dateien über klare Header verfügen und	
die Daten entsprechend darunter angeordnet sind. Zusätzlich wird die	
Verarbeitung von XLSX-Dateien unterstützt, die unmittelbar mit Daten	
beginnen.	
Der Ansatz sollte auch in der Lage sein, Dateien zu verarbeiten, selbst	FA_S2_Gen2
wenn diese keine Spaltennamen enthalten.	
Der Ansatz soll ebenfalls PostgresLoader unterstützen.	FA_S3_Gen2
Der Scaffolding-Ansatz sollte fähig sein, alle von Jayvee unterstützten	${ m FA_S4_Gen2}$
Datenspalten-Typen zu identifizieren.	

Tabelle 3.2: Funktionalle Anforderungen an die 2. Generation des Scaffolding-Ansatzes

3.2 Qualitätsanforderungen an Scaffolding

Die Qualitätsanforderungen an die 1. Generation des Scaffolding Generators sind in Tabelle 3.3 aufgelistet.

Performanz beschreibt, wie schnell der Ansatz Ursprungsdatenbanken in Zieldatenbanken überführt. Dies ist besonders wichtig, da es die Entwicklungszeit sparen kann. Daher ist es entscheidend, dass der Ansatz performant ist. Skalierbarkeit beschreibt, wie der Ansatz mit einer höheren Auslastung umgeht. Dies ist von Bedeutung, weil der Ansatz effektiv die verfügbaren Ressourcen nutzen sollte. Ein weiterer wichtiger Aspekt ist die Nutzung öffentlicher Bibliotheken. Eine öffentliche Bibliothek wird kostenlos verbreitet und kann von den Nutzern ohne Bedenken über rechtliche Einschränkungen modifiziert werden. Sie sind kosteneffektiv, weil sie kostenlos verbreitet werden, sie sind flexibel und sicher, weil oft viele Nutzer die Bibliothek verwenden und Verbesserungsvorschläge einbringen können.

Die Qualitätsanforderungen an die 2. Generation des Scaffolding Generators sind in Tabelle 3.4 aufgelistet.

Anforderungen	ID
Der Ansatz sollte performant sein, also Daten möglichst schnell	QA_S1_Gen1
verarbeiten können. Zudem soll der Ansatz skalierbar sein und in der	
Lage sein, aus 1.000 CSV-Dateien aus einer Datenbank funktionsfähige	
SQLite-Datenbanken zu generieren.	
Der Ansatz sollte ausschließlich öffentlich zugängliche Bibliotheken mit	QA_S2_Gen1
gängigen Open-Source-Lizenzen verwenden.	

Tabelle 3.3: Qualitätsanforderungen an die 1. Generation des Scaffolding-Ansatzes

Erweiterbarkeit beschreibt die Leichtigkeit, mit der neue Funktionen hinzugefügt oder bestehende modifiziert werden können. Daher ist es wichtig, dass der geschriebene Code für den Ansatz leicht erweiterbar werden kann.

Es ist ebenso entscheidend, dass der Ansatz einfach zu nutzen ist. Der Entwickler sollte keine Schwierigkeiten bei der Nutzung haben, da sonst wertvolle Zeit darauf verwendet wird, den Umgang mit dem Ansatz zu erlernen.

Anforderungen	ID
Der Code des Ansatzes sollte möglichst erweiterbar sein. Der Code soll	QA_S1_Gen2
modernen Entwicklungstechniken folgen, sinnvolle Namen verwenden	
und erweiterbar sein.	
Der Ansatz soll einfach zu nutzen sein. Durch den Terminal sollte der	QA_S2_{en2}
Nutzer mit der Eingabe einer Einweisung und einem Link zur	
CSV-Datei den Generator starten können.	

Tabelle 3.4: Qualitätsanforderungen an die 2. Generation des Scaffolding-Ansatzes

3.3 Funktionale Anforderungen an Reinforcement Learning

Die funktionalen Anforderungen an die 1. Generation des RL-Ansatzes sind in Tabelle 3.5 aufgelistet.

Zunächst war es wichtig, das Funktionsprinzip von RL zu verstehen und wie es speziell für die Codegenerierung von Jayvee-Modellen angewendet werden könnte. Daher wird nicht explizit verlangt, dass alle Randfälle abgedeckt werden, jedoch sollte der generierte Code valide sein. Zudem wird für die 1. Generation gefordert, dass die Spalten mindestens als "text" gespeichert werden, da andernfalls die produzierten Jayvee-Modelle nicht funktionieren würden.

Die funktionalen Anforderungen an die 2. Generation des RL-Ansatzes sind in Tabelle 3.6 aufgelistet.

Anforderungen	ID
Der Ansatz sollte in der Lage sein einen validen Code-Snippet, welches	FA_RL1_Gen1
nicht zwingend alle Randfälle abdecken muss generieren.	
Der Ansatz sollte in der Lage sein, sämtliche Datenspalten-Typen als	FA RL2 Gen1
"text" zu speichern. In der 1. Generation wird davon ausgegangen, dass	
die CSV-Dateien Spaltennamen enthalten.	

Tabelle 3.5: Funktionalle Anforderungen an die 1. Generation des RL-Ansatzes

In der 1. Generation wird das Modell auf einem einzelnen Prozess trainiert, was die Nutzung mehrerer Kerne im Prozessor ineffizient macht. Daher wird für die 2. Generation verlangt, das Training auf mehrere Kerne zu verteilen. In der 1. Generation des RL-Ansatzes wird angenommen, dass keine Datenbanken existieren, die keine Spaltennamen haben. Deshalb wird in der 2. Generation die Implementierung einer Funktion gefordert, die fehlende Spaltennamen generiert. Wie bereits in Abschnitt 3.1 erwähnt, ist XLSX eines der gängigsten Formate zur Datenspeicherung. Daher ist es wichtig, dass in der 2. Generation des RL XLS-XInterpreter unterstützt werden. Aus demselben Grund ist es ebenfalls wichtig, zusätzlich zu SQLite auch Postgres zu unterstützen. Aufgrund der gleichen Überlegungen, die für den Scaffolding-Ansatz der 2. Generation gelten, war es wichtig, die Erkennung von Spaltendatentypen zu implementieren.

Anforderungen	ID
Statt die Umgebung auf einen Prozessor laufen zu lassen, werden	FA_RL1_Gen2
mehrere Umgebungen auf mehreren Kernen laufen.	
Verbesserte Fehlerbehandlung und Flexibilität: Fähigkeit, Dateien ohne	FA_RL2_Gen2
Spaltennamen zu verarbeiten.	
Erweiterte Unterstützung von Datenquellen: Neben CSV-Dateien wird	FA_RL3_Gen2
nun auch die Unterstützung für XLSX-Datenquellen angeboten. Es wird	
vorausgesetzt, dass die XLSX-Dateien über klare Header verfügen und	
die Daten entsprechend darunter angeordnet sind. Zusätzlich wird die	
Verarbeitung von XLSX-Dateien unterstützt, die unmittelbar mit Daten	
beginnen.	
Erweiterte Ladekapazitäten: Unterstützung für das Laden von Daten in	FA_RL4_Gen2
Postgres-Datenbanken, zusätzlich zu SQLite.	
Der Ansatz sollte fähig sein, die Datentypen der Spalten korrekt zu	FA_RL5_Gen2
identifizieren.	

Tabelle 3.6: Funktionalle Anforderungen an die 2. Generation des RL-Ansatzes

3.4 Qualitätsanforderungen an Reinforcement Learning

Die Qualitätsanforderungen an die 1. Generation des RL-Ansatzes sind in Tabelle 3.7 aufgelistet.

Wie bereits in Abschnitt 3.2 beschrieben wird, sind Erweiterbarkeit und Flexibilität von entscheidender Bedeutung, was hier nicht erneut detailliert wird. Es ist wesentlich, klarzustellen, wie der RL-Ansatz zu nutzen ist. Deshalb wird gefordert, die Logik für das Trainieren des Modells und die Generierung von Jayvee-Modellen mithilfe der trainierten Modelle zu trennen.

Anforderungen	ID
Erweiterbarkeit und Flexibilität: Fähigkeit, neue Funktionen und	QA_RL1_Gen1
Verbesserungen leicht zu integrieren.	
Nutzererfahrung: Klare und zugängliche Schnittstelle für Training und	QA_RL2_Gen1
Bewertung.	

Tabelle 3.7: Qualitätsanforderungen an die 1. Generation des RL-Ansatzes

Die Qualitätsanforderungen an die 2. Generation sind in Tabelle 3.8 aufgelistet.

Ähnlich wie bei der 1. Generation soll die 2. Generation weiterhin entwickelbar und leicht zu nutzen sein. Der Generator soll durch die Eingabe eines Links gestartet werden können, was den Start durch einen Evaluator erleichtert oder die Automatisierung ermöglicht. Dies ist besonders wichtig, wenn man mehr als 1.000 Links hat, für die SQLite oder Postgres Datenbanken generiert werden sollen. In solchen Fällen möchte man nicht jeden Link einzeln eingeben, sondern den Prozess durch eine Schleife automatisieren.

Anforderungen	ID
Der Code des Ansatzes sollte möglichst erweiterbar sein. Der Code soll	QA_RL1_Gen2
modernen Entwicklungstechniken folgen, sinnvolle Namen verwenden	
und erweiterbar sein.	
Der Ansatz soll einfach zu nutzen sein. Durch den Terminal sollte der	QA_RL2_Gen2
Nutzer mit der Eingabe einer Einweisung und einem Link zur	
CSV-Datei den Generator starten können.	

Tabelle 3.8: Qualitätsanforderungen an die 2. Generation des RL-Ansatzes

3.5 Anforderungen an das benutzerdefinierte Chat-GPT

ChatGPT wird nicht von mir implementiert, sondern durch Prompts gesteuert. Prompting, ein verbreitetes Verfahren in der Welt der LLMs, ermöglicht es Menschen, auf spezifische Weise Antworten auf ihre Fragen von einem LLM wie ChatGPT zu erhalten. Daraus ergeben sich spezifische funktionale Anforderungen:

FA C1: Der Ansatz muss in der Lage sein, validen Jayvee-Code zu generieren.

 ${\bf FA_C2} :$ Der Ansatz soll Eingaben in Form von CSV- und XLSX-Datenbanken akzeptieren.

Zu den Qualitätsanforderungen gehört:

 $\mathbf{QA}_{-}\mathbf{C1} :$ Der Ansatz soll einen freundlichen Ton wahren.

4 Architektur

In diesem Kapitel wird die Architektur der Implementierung erörtert. Zunächst erfolgt eine Beschreibung der generischen Architektur. Anschließend wird detaillierter auf die Codegenerierung eingegangen, die mittels Scaffolding und Reinforcement Learning erfolgt.

Die Architektur in dieser Arbeit wird wie folgt definiert: "Die Software-Architektur eines Systems umfasst die Beschreibung der Software-Struktur oder -Strukturen des Systems, einschließlich der Software-Komponenten, deren sichtbaren Eigenschaften und die Beziehungen zwischen ihnen." (Bass, 2003) In dieser Masterar-

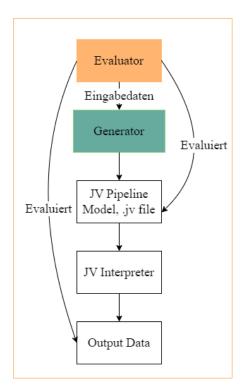


Abbildung 4.1: Generische Architektur des Codegenerators

beit werden zwei Arten von Architekturen beschrieben:

- 1. Generische Architektur Hierbei wird eine Übersicht präsentiert, die die allgemein verfolgte Struktur erklärt.
- 2. Nicht-Generische Architektur Diese beschreibt detailliert die Architektur, einschließlich der Interaktionen zwischen den Komponenten.

Wie in Abbildung 4.1 dargestellt, lässt sich die gesamte generische Systemarchitektur in zwei Hauptkomponenten unterteilen:

- Der Evaluator entnimmt Daten aus der DuckDB-Datenbank, die Links zu Datenbanken enthält, startet die Generatoren und beurteilt deren Ergebnisse. Diese Daten stammen aus einem Projekt, das am Lehrstuhl für Open Source Software der FAU entwickelt wurde und unterschiedliche Datenbanken aus diversen Quellen enthält.
- Der Generator hingegen erstellt Code und führt diesen aus.

Diese Architektur ist sinnvoll, da sie eine klare Aufteilung zwischen dem Evaluator und dem Generator vorsieht. Dies ermöglicht die Entwicklung und den Austausch verschiedener Generatoren sowie das Sammeln von Statistiken über die Anzahl der Ursprungsdatenbanken, die Anzahl generierter Jayvee-Modelle, die Anzahl der Zieldatenbanken und die Geschwindigkeit der Codegenerierung.

4.1 Evaluator

In diesem Abschnitt wird die Architektur des Evaluators erläutert. Wie in Abbildung 4.2 ersichtlich, beginnt der Evaluator zunächst mit der Abfrage der Parameter. Anschließend erfolgt die Löschung von Modellen und Datenbanken aus vorherigen Durchläufen, um einen sauberen Startzustand für die aktuelle Evaluation zu gewährleisten. Darauf folgend werden Statistiken aus früheren Durchläufen ausgelesen, um kontinuierliche Analysen und Verbesserungen zu ermöglichen. Sollte "sc" als Parameter vorhanden sein, wird der Scaffolding-Ansatz angewendet:

- 1. Anschließend werden die Daten aus DuckDB geladen.
- 2. Danach wird das Jayvee-Modell mittels Scaffolding erstellt und ausgeführt.
- 3. Am Ende werden Statistiken gesammelt und Evaluationsdiagramme erstellt

Wenn der Parameter "rl" übergeben wird, kommt RL für die Codegenerierung zum Einsatz.

- 1. Zunächst wird das trainierte Modell ausgelesen.
- 2. Das ausgelesene Modell wird verwendet, um die Jayvee-Modelle zu generieren.

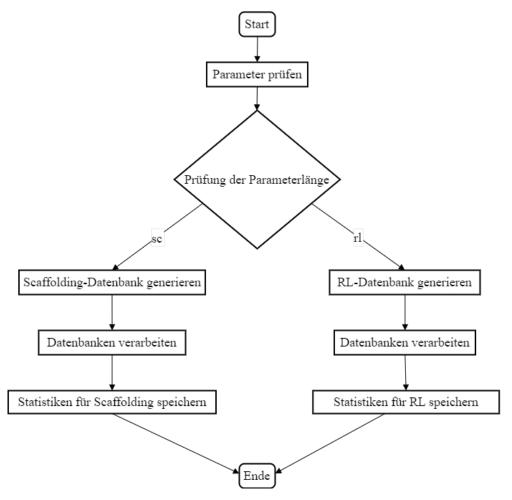


Abbildung 4.2: Flussdiagramm des Evaluierungssystems

3. Zum Abschluss werden die Statistiken gesammelt.

Der Algorithmus für den Start und die Sammlung der Ergebnisse aus dem Generator ist für beide Ansätze gleich, was das einfache Hinzufügen neuer Funktionalitäten ermöglicht.

4.2 Scaffolding

In diesem Schritt wird die Architektur des Scaffolding Generators beschrieben, wie sie in Abbildung 4.3 dargestellt ist.

Der Scaffolding Generator wird durch den Evaluator gestartet, wobei ein Link als Eingabe dient.

Die Architektur beider Generationen ist wie folgt aufgebaut:

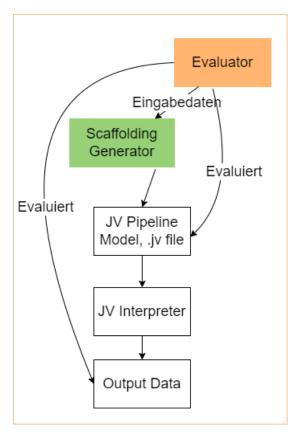


Abbildung 4.3: Architektur des Scaffolding-Ansatzes

- 1. Zuerst werden die Daten abgerufen.
- 2. Anschließend wird der Code mittels Scaffolding-Technik generiert, was detailliert in Kapitel 5 erläutert wird.
- 3. Daraufhin werden Dateipfade erstellt, unter denen die Daten gesichert werden.
- 4. Informationen zur Beschreibung der CSV-Datei, einschließlich der Anzahl der Zeilen, der URL und des Namens, werden gespeichert.
- 5. Abschließend wird die SQLite-Datenbank aus dem durch Scaffolding generierten Code erstellt.

Die Architektur wird gemäß der generischen Architektur aufgebaut. Dies ermöglicht die Integration des Scaffolding-Generators in den Evaluator.

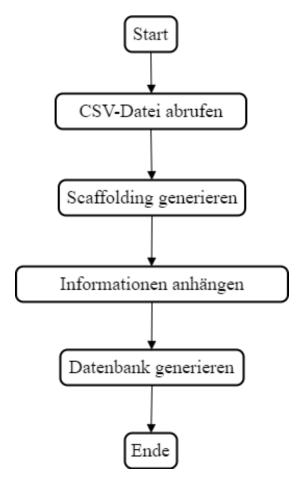


Abbildung 4.4: Flussdiagramm des Scaffolding-Ansatzes

4.3 Reinforcement Learning

Dieser Abschnitt widmen sich der Architektur, dem Flussdiagramm und dem Aktivitätsdiagramm des Reinforcement-Learning-Ansatzes.

4.3.1 Reinforcement Learning Generation 1

Der Reinforcement-Learning-Ansatz, dargestellt in Abbildung 4.5, setzt sich aus den folgenden Hauptkomponenten zusammen:

- Trainingsumgebung: In dieser wird das Modell mit einer umfangreichen Menge an Datenpunkten trainiert. Anschließend wird das trainierte Modell auf der Festplatte gespeichert.
- RL-Generator: Der RL-Generator erhält einen Link als Eingabe und liest das auf der Festplatte gespeicherte Modell aus.

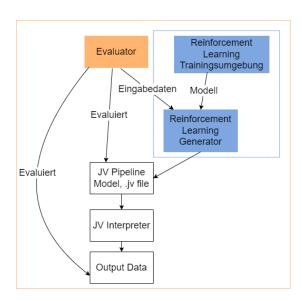


Abbildung 4.5: Architektur des RL-Ansatzes

Hier wird die Architektur der einzelnen Komponenten genauer betrachtet.

Die Trainingsumgebung der 1. Generation funktioniert, wie in der Abbildung 4.6 dargestellt, folgendermaßen:

- 1. Zunächst werden Daten aus DuckDB abgefragt. In diesem Schritt werden die Datensätze selektiert und heruntergeladen, wobei zunächst ausschließlich CSV-Daten bezogen werden. Es wird überprüft, ob alle erforderlichen Ordner vorhanden sind.
- 2. Anschließend werden die Umgebung und der Agent (Modell) initialisiert. Der Agent interagiert mit der Umgebung und lernt aus dieser Interaktion. Als Algorithmus für den Agenten wird das Deep Q-Network ausgewählt (siehe Punkt 2.4.2). Die Umgebung ist zuständig für Rückmeldungen und das Training des Agenten. Sie führt das Modul "train_evaluator" aus, das für die Bewertung des vom Agenten erzeugten Codes zuständig ist. Es beurteilt die Validität des Codes und informiert die Umgebung, welche wiederum den Agenten informiert. Aus diesen Erfahrungen lernt der Agent.
- 3. Das Modell wird solange trainiert, bis eine festgelegte Anzahl von Episoden erreicht ist (siehe Kapitel 5).
- 4. Schließlich wird das Modell gespeichert.

In der 1. Generation des RL-Generators werden folgende Prozesse durchgeführt:

1. Zunächst erfolgt das Laden und Lesen der Daten. Dabei werden Dateien über einen bereitgestellten Link heruntergeladen und die Spaltennamen ausgelesen.

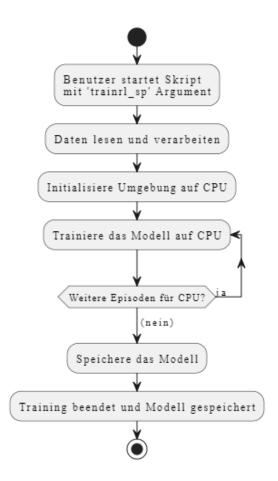


Abbildung 4.6: Aktivitätsdiagramm des RL-Ansatzes, dargestellt anhand eines Prozesses

- 2. Anschließend wird die Umgebung für das trainierte Modell initialisiert, indem sie mit einem Initialcode versehen wird.
- 3. Daraufhin werden die erlernten Aktionen ausgewählt und implementiert.
- 4. Danach wird die Datei erstellt und ausgeführt.
- 5. Abschließend werden Statistiken zu den Ergebnissen gesammelt.

Diese Architektur macht Sinn, da sie die Funktionalitäten auf Module aufteilt, wobei jedes Modul seinen eigenen Zweck hat.

4.3.2 Reinforcement Learning Generation 2

In der 2. Generation der Architektur und des Trainingsprozesses werden wesentliche Verbesserungen vorgenommen, um die Effizienz und Flexibilität zu erhöhen. Die Modifikationen umfassen die Integration moderner Algorithmen und die Er-

weiterung der Trainingsumgebung, wie folgt auf Abbildung 4.7 dargestellt:

- 1. Der erste Schritt, die Initialisierung des Prozesses, bleibt unverändert wie in der 1. Generation.
- 2. Im 2. Schritt wird eine wichtige Änderung realisiert: Der ursprünglich verwendete Algorithmus wird durch eine vorimplementierte Lösung aus Bibliothek "Stable Baselines3" ersetzt. Diese Neuerung ermöglicht eine effektivere Integration von Funktionen für die Interaktion mit multiplen Umgebungen. Im Gegensatz zur 1. Generation, die nur eine einzige Umgebung unterstützt, ermöglicht diese Generation die Handhabung mehrerer Umgebungen simultan. Diese Erweiterung auf eine 1-zu-n-Beziehung zwischen dem Algorithmus und den Umgebungen wird in der beigefügten Abbildung verdeutlicht und zielt darauf ab, die Trainingsdauer erheblich zu verkürzen.
- 3. Abschließend wird, analog zur 1. Generation, das trainierte Modell gespeichert. Dieser Schritt markiert den Endpunkt des Trainingsprozesses, wobei die erzielten Fortschritte und Optimierungen gesichert werden.

Diese Anpassungen in der 2. Generation führen zu einer deutlichen Verbesserung des Trainingsprozesses, indem sie die Flexibilität erhöhen und die Zeit für das Training reduzieren. Die Implementierung einer vorimplementierten Lösung aus Bibliothek "Stable Baselines3" und die Fähigkeit, mit mehreren Umgebungen zu interagieren, sind zentrale Elemente dieser Fortschritte. In der 2. Generation des RL-Generators werden die folgenden verbesserten Prozesse durchgeführt:

- 1. Das Laden und Lesen der Daten findet wie zuvor statt, mit dem Unterschied, dass nun auch die Datentypen der Spalten erkannt werden. Sollten keine Spaltennamen vorhanden sein, werden neue generiert.
- 2. Die Initialisierung der Umgebung für das trainierte Modell erfolgt durch das Aufbringen eines Initialcodes.
- 3. Die Auswahl der erlernten Aktionen basieren auf den in der Bibliothek "Stable Baselines3" implementierten Algorithmen (siehe Abschnitt 5.4).
- 4. Die Erstellung und Ausführung der Datei bleibt ebenfalls unverändert.
- 5. Die Sammlung von Statistiken über die Ergebnisse bildet erneut den Abschluss des Prozesses.

4.4 ChatGPT

ChatGPT wird in dieser Arbeit nicht programmiert oder aufgebaut, sondern lediglich gepromptet. Deshalb wird seine Architektur an dieser Stelle nicht weiter erörtert.

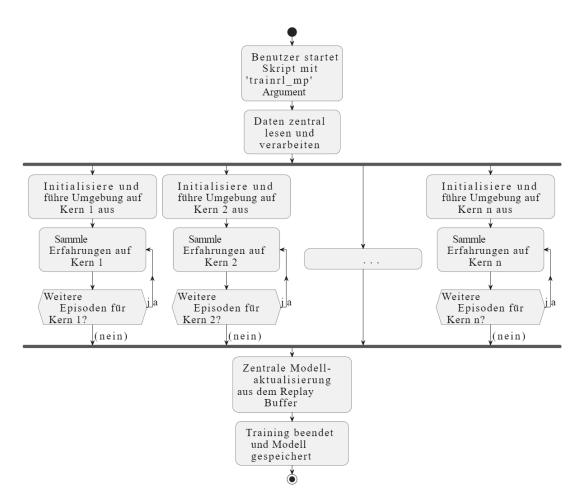


Abbildung 4.7: Aktivitätsdiagramm des RL-Ansatzes, dargestellt anhand mehrerer Prozesse

5 Design und Implementierung

Hier werden die Implementierungsdetails beschrieben. Typescript in Verbindung mit Node.js wird als Steuerungssprache verwendet, während für das Reinforcement Learning Python zum Einsatz kommt.

5.1 Evaluator

In diesem Abschnitt wird die Implementierung des Evaluators detaillierter betrachtet. Die Fehlerbehandlung spielt eine entscheidende Rolle, da Methoden implementiert werden müssen, um Fehler zu erkennen und sie dem Benutzer zu signalisieren. In einem solchen Fall wird der Prozess mit dem Statuscode "1" beendet. Dieses Prinzip findet durchgängig in der Lösung Anwendung, da es Softwareentwicklern das Handling von Fehlern erleichtert.

Die Masterarbeit thematisiert die asynchrone Implementierung der Ansätze. Asynchronität ermöglicht es, mehrere Prozesse simultan zu bearbeiten. TypeScript wird so programmiert, dass es wartet, bis eine Funktion eine Rückmeldung gibt, bevor die nächste Funktion gestartet wird, um den Code lesbarer zu gestalten und einen synchronen Ausführungsfluss zu unterstützen.

- 1. Zunächst wird die Eingabe des Benutzers als String gespeichert. Diese Eingabe dient dazu, zu entscheiden, welche Methode evaluiert werden soll.
- 2. Wie in Abbildung 5.1 dargestellt, werden für eine saubere Evaluation die Daten der letzten Ausführung gelöscht. Dies wird durch den Einsatz von fs/promises ermöglicht, einer API, die asynchrone, auf Promises basierende Funktionen bereitstellt, die unter anderem Zugriff auf das Dateisystem ermöglichen. Ein Platzhalterobjekt wird erstellt und später festgelegt. Es wird gewartet, bis die Funktion, die für das Löschen von Ordnern zuständig ist, abgeschlossen ist.
- 3. Anschließend werden die Statistiken aus vorherigen Durchläufen geladen. Die Daten sind im JSON-Format gespeichert, wofür das JSON-Paket benötigt wird, um die Daten auszulesen. JSON, kurz für JavaScript Object

Notation, ist eine leichtgewichtige Sprache, die für den Datenaustausch entwickelt wird. Aufgrund der geringen Datenmenge, der Plattformunabhängigkeit und der einfachen Parsbarkeit wird JSON für die Sammlung von Statistiken gewählt.

4. Danach werden Prozesse beschrieben, die beim Laden von Datenbanken aus DuckDB erfolgen. Hierzu wird eine spezielle Bibliothek verwendet, die das Abrufen von Daten aus einer DuckDB-Datenbank ermöglicht.

Mit einer Abfrage werden die URLs ermittelt, über die auf die Datenbanken zugegriffen und diese heruntergeladen werden können. In der 1. Generation beschränkt man sich auf das Holen von CSV-Datenbanken, da keine anderen Formate unterstützt werden. Es ist ebenfalls möglich, die Anzahl der herunterzuladenden Datenbanken individuell anzupassen.

Zur Verarbeitung der Rückmeldung wird erneut der API Promise eingesetzt. Dabei gibt es zwei Arten von Rückmeldungen:

- 4.1 "err": Tritt ein Fehler auf, wird dieser gespeichert und ausgegeben.
- 4.2 "res": Bei einer erfolgreichen Rückmeldung wird diese weiterverarbeitet:
 - 4.2.1 Zuerst wird die Rückmeldung verarbeitet, wobei Generatoren mithilfe von Evaluatorn gestartet werden. Er werden erneut Promises eingesetzt, um die Generatoren zu initiieren. Die Kindprozesse werden in eine asynchrone Form umgewandelt. Obwohl dieser Ansatz die Ausführungsgeschwindigkeit des Codes reduzieren kann, erleichtert er die Fehlerbehandlung. Zusätzlich wird die Ausführungszeit sowie die Anzahl korrekt generierter Jayvee-Modelle und Datenbanken gespeichert.
 - 4.2.2 Abschließend werden die Daten über die durch die Generatoren erstellten Datenbanken ausgelesen und die Verbindung zur Datenbank geschlossen.

Es gibt zwei mögliche Parameter: "sc" für Scaffolding und "rl" für Reinforcement Learning.

- 1. Zuerst wird der Umgang mit dem Parameter "sc" erklärt.
- 2. Anschließend wird dargelegt, was geschieht, wenn der Parameter "rl" auftritt.

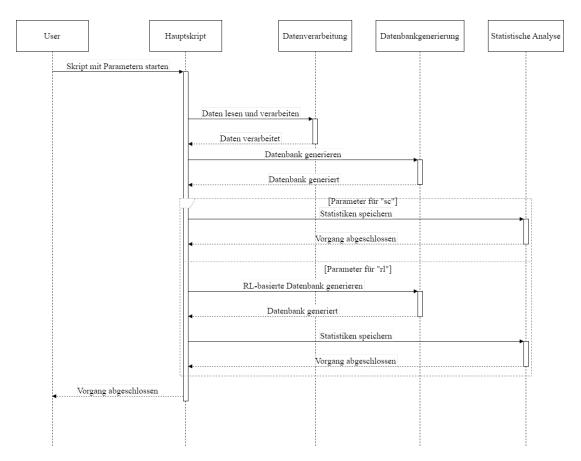


Abbildung 5.1: Sequenzdiagramm des Evaluierungssystems

5.2 Ermittlung der Spaltennamen und Spaltendatentypen

5.2.1 Ermittlung der Spaltennamen und Spaltendatentypen Generation 1

Die Daten in der DuckDB-Datenbank sind mangelhaft gepflegt. Einige haben keine Spaltennamen, andere weisen Einträge auf, die sich von den übrigen unterscheiden. Wenn Jayvee eine Spalte ausliest, erwartet es, dass alle Daten in dieser Spalte einheitlich sind. Daten, die dem erwarteten Spaltendatentyp nicht entsprechen, werden nicht importiert, was dazu führt, dass die Zieldatenbank (z.B. SQLite, Postgres) weniger Zeilen aufweist als die Ursprungsdatenbank (z.B. CSV, XLSX). Daher ist es wichtig, ein System zu implementieren, das die Spaltennamen und Spaltendatentypen zuverlässig erkennt.

Es gibt zwei Hauptansätze zur Implementierung der Erkennung von Spaltendatentypen:

- Einsatz von Deep Learning: Ein Anwendungsbeispiel für Deep Learning ist die Methode 'Sherlock' (Hulsebos et al., 2019). Da die genaue Erkennung von Spaltendatentypen jedoch nicht im Fokus dieser Arbeit steht, wird eine alternative Methode gewählt.
- Anwendung statistischer Methoden: Diese Methoden bestimmen beispielsweise den Datentyp, der am besten zu der Mehrheit der Werte in einer Spalte passt. Eine effiziente und schnelle öffentlichee Bibliothek, das diesen Ansatz nutzt, ist "MessyTables". Obwohl es Nachteile gibt, liegt der Schwerpunkt dieser Arbeit nicht auf der absolut korrekten Erkennung von Spaltendatentypen.

In der 1. Generation von RL wird die öffentliche Bibliothek "MessyTables" verwendet, um die Spaltennamen aus CSV-Dateien zu extrahieren. Der Prozess läuft wie folgt ab:

- 1. Zunächst wird die Datenbank in das passende für "MessyTables" Format konvertiert.
- 2. Diese Datenbank wird dann in die Funktion von "MessyTables" eingegeben, die die Spaltennamen erkennt.
- 3. Anschließend wird die Funktion von "MessyTables" genutzt, um die Spaltendatentypen zu bestimmen.

Da Scaffolding bereits zu einem früheren Zeitpunkt entwickelt wird und sich die 1. Generation ausschließlich auf CSV-Datenbanken mit Spaltennamen konzentrierte, wird in der 1. Generation die Bibliothek "Axios" verwendet, um Daten und Spaltennamen herunterzuladen. Die Datentypen werden in der 1. Generation des Scaffolding-Ansatzes standardmäßig als "text" gespeichert und nicht weiter spezifiziert.

5.2.2 Ermittlung der Spaltennamen und Spaltendatentypen Generation 2

In der 2. Generation ist es wichtig, in beiden Ansätzen die Datentypen zu erkennen, XLSX-Dateien zu unterstützen und Postgres als Zieldatenbank zu verwenden.

Das Python-Skript zur Erkennung von Spaltendatentypen funktioniert für Scaffolding und RL ähnlich und verläuft wie folgt:

- 1. Bei einer CSV-Datei:
 - 1.1 Zuerst werden mithilfe von "MessyTables" Paare aus Spaltennamen und Spaltendatentypen erstellt.

- 1.2 Wenn die Zieldatenbank SQLite ist, werden die Daten unverändert gelassen, da in SQLite Integer mit 0, 1, 2, 3, 4, 6 oder 8 Bytes gespeichert werden können.
- 1.3 Wenn Postgres als Zieldatenbank ausgewählt wird, müssen die Daten angepasst werden, weil in Postgres die Integer eine kleinere Größe haben. Falls die Länge eines Integer-Werts in der Zelle größer als 4 Bytes ist, wird die Spalte als "text" gespeichert.

Dies ist besonders wichtig, da in Postgres ein integer 4 Bytes groß ist, während in SQLite ein integer bis zu 8 Bytes groß sein kann. Wenn die integer-Werte zu groß sind, können SQLite-Datenbanken weiterhin funktionieren, während Postgres-Datenbanken dies nicht tun. Falls die Zahl größer als 4 Bytes ist, erfolgt kein Import der Daten in die Postgres-Datenbank. Für größere Zahlen muss der Datentyp bigint verwendet werden, der jedoch von Jayvee nicht unterstützt wird. Um diesen Spaltendatentyp für Postgres lesbar zu machen, wird der Datentyp als "text" im Jayvee-Modell gespeichert.

1.4 Die erkannten Datentypen und Spaltennamen werden ausgegeben, sodass der Generator sie auslesen kann.

2. Bei einer XLSX-Datei:

- 2.1 Zuerst werden die XLSX-Blätter nacheinander als Pandas Dataframes eingelesen, weil Pandas interne Funktionen nutzt, um die Spaltendatentypen korrekt zu erkennen.
- 2.2 Danach werden die Spaltendatentypen ausgelesen und ausgegeben, sodass der Generator diese Daten auslesen kann.

Im Fall von XLSX-Daten könnte zwar auch "MessyTables" verwendet werden, jedoch unterstützt diese Bibliothek nur XLS-Dateien, die eine begrenzte Anzahl von Zeilen aufweisen. Die Erkennung von Spaltendatentypen in XLSX-Dateien erfolgt daher durch die Nutzung von Pandas.

5.3 Scaffolding

Es gibt zwei Wege einen Generator zu implementieren:

• Entwicklung eines Servers: Man könnte eine REST API entwickeln, an die Befehle vom Evaluator gesendet werden. Der Generator würde dann als Server diese Befehle ausführen und darauf antworten. Diese Methode könnte die Verteilung der Generierung unterstützen, jedoch liegt dies nicht im Fokus dieser Arbeit.

• Direktes Starten des Generators: Der Generator wird durch die Eingabe von Parametern wie URL und Zieldatenbank vom Evaluator gestartet. Die Kommunikation erfolgt über das Ausgeben von JSON-Objekten. Diese Methode wird in dieser Arbeit verwendet, da sie zuverlässig ist, einfaches Debugging ermöglicht und die Bewertung des Modells erleichtert. Im RL-Ansatz wird ebenfalls diese Methode verwendet.

Die Generation erfolgt sequenziell, da dies das Debugging der Ansätze für die Codegenerierung wesentlich erleichtert. Eine Auseinandersetzung mit Parallelverarbeitung ist nicht erforderlich, da sie nicht Gegenstand dieser Arbeit ist.

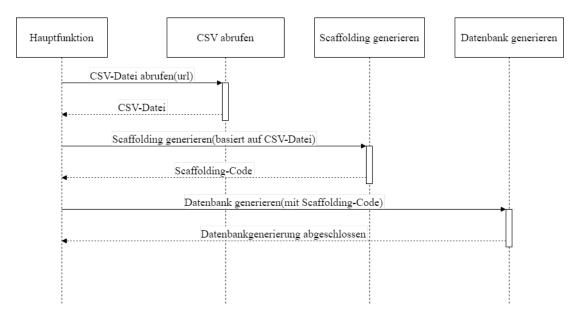


Abbildung 5.2: Sequenzdiagramm des Scaffolding-Ansatzes

Die 1. Generation des Generators besteht aus den folgenden Teilen:

- 1. Herunterladen der CSV-Zeilen und Auslesen der Spaltennamen.
- 2. Die Erstellung des Scaffoldings basiert auf den ausgelesenen Daten, wobei abhängig von den Nutzereingaben spezifische Codeabschnitte generiert werden. Der Prozess beginnt mit der Implementierung einer Pipeline für CSV-Dateien. In zukünftigen Generationen ist geplant, Unterstützung für weitere Dateitypen zu integrieren. Nach dieser Initialphase werden verschiedene Blöcke hinzugefügt, um die Funktionalität weiter zu differenzieren und zu erweitern.
- 3. Speichern des Jayvee-Modells in einem Ordner.
- 4. Ausführen des Modells zur Erstellung einer SQLite-Datenbank.

Die 2. Generation unterstützt folgendes:

- 1. Herunterladen der CSV- und XLSX-Datenbanken und Auslesen der Spaltennamen.
- 2. Erkennung der Spaltendatentypen.
- 3. Das Jayvee-Modell wird basierend auf den ausgelesenen Daten erstellt, wobei sowohl CSV- als auch XLSX-Dateien unterstützt werden. Nach "If"-Anweisungen wird bestimmt, welche Extractors hinzugefügt werden. Abhängig von der Dateierweiterung werden spezifische Codeteile hinzugefügt: Für CSV-Dateien werden CSV-spezifische Codeteile, und für XLSX-Dateien entsprechend XLSX-spezifische Codeteile integriert.
- 4. Speichern des Jayvee-Modells in einem Ordner.
- 5. Ausführen des Modells zur Erstellung SQLite- und Postgres-Datenbanken.

Der produzierte Code sieht folgend aus:

```
pipeline Bevoelkerungsstruktur2005bewegungraeumlichPipeline {
    Bevoelkerungsstruktur2005bewegungraeumlichExtractor
         -> Bevoelkerungsstruktur2005bewegungraeumlichTextFileInterpreter
         -> Bevoelkerungsstruktur2005bewegungraeumlichCSVInterpreter
         -> Bevoelkerungsstruktur2005bewegungraeumlichTableInterpreter
         -> Bevoelkerungsstruktur2005bewegungraeumlichLoader;
block\ Bevoelkerungsstruktur 2005 bewegung raeumlich Extractor\ of type\ Http Extractor\ \{a,b,c\}
           https://geo.sv.rostock.de/download/opendata/bevoelkerungsstruktur_2005/bevoelkerungsstruktur_
    2005_bewegung_raeumlich.csv";
block Bevoelkerungsstruktur2005bewegungraeumlichTextFileInterpreter oftype TextFileInterpreter { }
block Bevoelkerungsstruktur2005bewegungraeumlichCSVInterpreter oftype CSVInterpreter {
    enclosing:
block Bevoelkerungsstruktur2005bewegungraeumlichTableInterpreter oftype TableInterpreter {
    columns: [
       "stadtbereich_code" oftype text,
"stadtbereich_bezeichnung" oftype text,
       "anzahl_zuzuege" oftype text,
"zuzuege_je_1000" oftype text,
"anzahl_wegzuege" oftype text,
"wegzuege_je_1000" oftype text,
       "wanderungssaldo" oftype text,
"wanderungssaldo_je_1000" oftype text
  1:
    table: "Bevoelkerungsstruktur2005bewegungraeumlich"; file: "./Bevoelkerungsstruktur2005bewegungraeumlich.sqlite";
```

Um neue Funktionen hinzuzufügen, gibt es zwei Schritte:

- 1. Erweiterung der Codegenerierungslogik: Um neue Funktionen zu unterstützen, sollte die Möglichkeit bestehen, den Ansatz zu erweitern. Beispielsweise wird die Codegenerierung aktuell mit if-else-Bedingungen gesteuert.
- 2. Anpassung der Logik für Ursprungsdatenbanken: Wenn es um die Integration von Ursprungsdatenbanken geht, muss auch die Logik für das Importieren der Daten angepasst werden. Jedes Datenbankformat organisiert Daten auf eine eigene Weise und erfordert eine spezifische Behandlung.

5.4 Reinforcement Learning

In diesem Abschnitt wird die Codegenerierung durch Reinforcement Learning (RL) erörtert.

Zudem verfügt der Agent über einen Zustand in Form einer Matrix mit zwei Werten:

- 1. Die Anzahl der Zeilen im aktuellen Code.
- 2. Ein binärer Wert, der angibt, ob der letzte Zustand erfolgreich war.

Das Training erfolgt in Episoden, wobei eine Episode alle Aktionen umfasst, bis das erste valide Jayvee-Modell erreicht wird.

Die richtige Wahl der Episodenanzahl ist entscheidend. Es gibt sehr viele Faktoren, die die Episodenanzahl beeinflussen können (Sutton & Barto, 2018). In dieser Arbeit werden zwei Methoden getestet:

- Geringe Anzahl an Episoden (3 Episoden), große Anzahl an (CSV-, XLSX-) Datenbanken.
- Relativ große Anzahl an Episoden (200-300 Episoden), kleine Anzahl an (CSV-, XLSX-) Datenbanken.

Des Weiteren war die Entwicklung einer geeigneten Umgebung erforderlich. Zunächst wird versucht, eine Umgebung eigenständig zu entwickeln, was sich jedoch als zu komplex erwies. Daher wird beschlossen, das von OpenAI entwickelte Bibliother "Gym" zu verwenden. OpenAI hat die Wartung des Projekts eingestellt, und es wurde von der Farama Foundation übernommen, die es nun unter dem Namen "Gymnasium" weiterführt. Diese Version wird in dieser Arbeit verwendet. Bibliothek "Gymnasium" bietet eine Schnittstelle, die die Entwicklung einer Umgebung und eines Agenten erleichtert.

Es ist ebenfalls entscheidend zu bestimmen, wie viele verschiedene Aktionen der Agent durchführen kann. Es gibt zwei primäre Ansätze zur Entscheidungsfindung:

- Viele kleine Aktionen: Ein Beispiel hierfür sind Aktionen wie das Einfügen von "url" oder schreiben von "TableInterpreter". Dieser Ansatz wird in dieser Arbeit getestet, führt jedoch dazu, dass zu viele Aktionen (insgesamt 25) generiert werden. Dies ist problematisch, da der Agent sehr viele Kombinationsmöglichkeiten (25! = 1.551121e + 25) durchlaufen muss, um zu lernen, wie valides Jayvee-Code geschrieben wird.
- Eine geringe Anzahl größerer Aktionen: Hierbei wird statt einzelner Zeichen ein gesamter Block als eine Aktion behandelt, beispielsweise "write block CSVInterpreter". Dies reduziert die Anzahl der Aktionen auf

fünf, was erheblich effizienter ist, da der Agent nicht mehr 1.551121e + 25, sondern nur 120 Kombinationsmöglichkeiten (5!) durchlaufen muss.

Die 1. Generation des Trainingsprozesses besteht aus folgenden Schritten:

- 1. Herunterladen der CSV-Zeilen und Auslesen der Spaltennamen.
- 2. Für jede CSV-Datei wird eine separate Umgebung sowie ein Agent erstellt. Die Umgebung folgt der "Gymnasium"-Interface-Spezifikation.
- 3. Einsatz eines selbst implementierten Agenten, der Deep Q-Learning (DQN) verwendet. DQN ist eine RL-Methode, die den Zustand als Eingabewert nimmt, diesen in ein neuronales Netzwerk einspeist, Rückmeldungen aus der Umgebung erhält und daraus lernt.
- 4. Falls bereits ein älterer Agent existiert, wird dieser geladen.
- 5. Das Training beginnt.

Der Trainingsprozess ist wie folgt aufgebaut:

- 1. Die Datei wird in einen initialen Zustand versetzt.
- 2. Der Agent erfasst den Zustand.
- 3. Mit dem Befehl

```
np.reshape(current_state, [1, agent.state_size])
```

wird der Zustand in die passende Form gebracht, wobei 1 die Anzahl der gleichzeitig an das Modell übergebenen Eingaben (Datenpunkte) repräsentiert.

- 4. Der Agent durchläuft eine Schleife, um mit der Umgebung zu interagieren, die endet, sobald ein valides Jayvee-Modell erreicht wird. Dieser Prozess beinhaltet mehrere Schritte:
 - 4.1 Der Agent wählt eine Reihe von Aktionen basierend auf den berechneten Q-Werten aus. Je nach Ursprungs- oder Zieldatenbank wird eine bestimmte Anzahl an Aktionen ausgewählt, die als Evaluationsintervall bezeichnet wird. Nachdem diese Zahl erreicht ist, werden die Aktionen an die Umgebung weitergeleitet.
 - 4.2 Die Umgebung verarbeitet diese Aktionen, gibt den nächsten Zustand und die Belohnung für die Aktionen zurück, und zeigt an, ob die Episode beendet ist.
 - 4.3 Anschließend wird der Zustand erneut "reshaped", um eine konsistente Form für die Erwartungen des neuronalen Netzes zu gewährleisten.
 - 4.4 Das Ergebnis wird im Speicher des Agenten gespeichert.

- 4.5 Sobald der Speicher des Agenten ausreichend Daten enthält, wird der Lernprozess gestartet.
- 5. Nach jeder Serie von 10 Eingaben (Ursprungsdatenbanken) wird das RL-Modell gespeichert. Diese Frequenz der Speicherung soll an die Anzahl der verarbeiteten Datenbanken angepasst werden. Da die Testphase mit 20 Datenbanken begonnen hat, wird die Zahl 10 gewählt, um sicherzustellen, dass im Falle von Fehlern nach den ersten 10 Datenbanken zumindest ein Modell vorhanden ist, das auf diesen 10 Datenbanken trainiert wird.

Falls "evaluate" und ein Link als Parameter übergeben wird:

- 1. Zuerst wird sichergestellt, dass ein Verzeichnis, in dem das Modell gespeichert wird, existiert.
- 2. Anschließend wird das Modell durch Ausführung bewertet.
- 3. Es wird eine Rückmeldung gegeben, ob das Modell gültig ist oder nicht.

Der letzte Fall ist darauf ausgelegt, das Training des Modells zu unterstützen. Der RL-Generator setzt dies um und wartet auf eine Rückmeldung.

Für die 2. Generation des Prozesses werden zusätzliche Verbesserungen eingeführt:

- 1. Statt einer einzigen Umgebung werden mehrere Umgebungen auf verschiedene Kerne verteilt, um die Trainingsgeschwindigkeit zu erhöhen.
- 2. Die Datentypen der Spalten werden korrekt erkannt.
- 3. Es gibt Unterstützung für Daten ohne Spaltennamen, wobei neue Namen für die Spalten generiert werden.
- 4. Es wird Unterstützung für Postgres-Datenbanken hinzugefügt.
- 5. Es wird Unterstützung für das Einlesen von XLSX-Dateien bereitgestellt.
- 6. Die Anzahl der Aktionen hat sich durch die Unterstützung von XLSX und Postgres auf neun erhöht. Daher wird ein weiterer Ansatz verfolgt, um die Anzahl der Aktionen zu reduzieren. Wenn Datenbanken für Postgres generiert werden sollen, werden die für SQLite spezifischen Aktionen als bereits genutzt markiert. Das Modell muss dann diese Aktionen nicht mehr durchlaufen. Dasselbe Verfahren wird angewandt, wenn SQLite als Zieldatenbank ausgewählt wird: Die Aktionen für Postgres werden als bereits verwendet markiert.
- 7. Anstatt DQN eigenständig zu implementieren, wird die DQN-Implementierung aus Bibliothek "Stable Baselines3" genutzt. Ursprünglich wurde Bibliothek "Stable Baselines3" entwickelt, um das Lesen und Verstehen wissenschaftlicher RL-Papiere zu erleichtern. Es gab viele unterschiedliche Ansät-

ze, dieselben Algorithmen zu entwickeln, und Bibliothek "Stable Baselines3" vereinfachte die Reproduktion dieser Ergebnisse. Zudem bietet Bibliothek "Stable Baselines3" eine einfache Methode, die Umgebungen auf mehrere Kerne zu verteilen und die Rückmeldungen in DQN zu verarbeiten.

Diese Verbesserungen zielen darauf ab, die Effizienz und Flexibilität des Trainingsprozesses zu steigern, die Kompatibilität mit verschiedenen Datenquellen zu erhöhen und die Implementierung des Deep Q-Learning-Verfahrens zu optimieren.

Es ist wichtig, den Nutzern das Hinzufügen neuer Ursprungs- und Zieldatenbanken zu erleichtern. Um eine Ursprungs- oder Zieldatenbank hinzuzufügen, müssen folgende Schritte durchgeführt werden:

- 1. Neue Aktionen zur bestehenden Aktionsliste hinzufügen: Abhängig von der spezifischen Ursprungs- oder Zieldatenbank, müssen relevante Aktionen hinzugefügt werden.
- 2. Diese neuen Aktionen in der Umgebung definieren: Die Umgebung muss über die neu verfügbaren Aktionen informiert werden, damit sie entsprechend genutzt werden können.
- 3. Das Evaluationsintervall anpassen: Das Intervall für die Bewertung der Aktionen muss basierend auf der Art der Datenbank angepasst werden. Beispielsweise erfordern CSV-Dateien fünf Aktionen, während für XLSX-Dateien sechs benötigen.
- 4. Die Liste der genutzten Aktionen aktualisieren: Es muss sichergestellt werden, dass die Liste der verwendeten Aktionen die neuen Aktionen einschließt, um Konsistenz und Genauigkeit im Trainingsprozess zu gewährleisten. Alternativ kann man sie auch unmarkiert lassen, was jedoch mehr Trainingsdaten und Zeit erfordert, da der Agent mehr Optionen durchgehen muss.
- 5. Die Logik für den Import anpassen: Wenn es sich um Aktionen für Ursprungsdatenbanken handelt, muss die Importlogik entsprechend angepasst werden.
- 6. Das Modell neu trainieren: Das Modell muss über die Hinzufügung neuer Aktionen informiert werden, damit es seine Strategien entsprechend anpassen und mit den neuen Aktionen effektiv umgehen kann.

Durch die Durchführung dieser Schritte wird gewährleistet, dass die neuen Aktionen nahtlos in die bestehenden Prozesse integriert und effektiv genutzt werden können.

5.5 Benutzerdefiniertes ChatGPT

In diesem Abschnitt wird die Codegenerierung mittels eines benutzerdefinierten ChatGPT erläutert. Zum Zeitpunkt dieser Masterarbeit wurde eine neue Funktion von ChatGPT vorgestellt, die es den Benutzern erlaubt, ein benutzerdefiniertes ChatGPT zu kreieren und zu teilen. Ein solches benutzerdefiniertes ChatGPT ist eine auf einen bestimmten Anwendungsfall maßgeschneiderte Version des originalen ChatGPT.

Das Vorgehen gestaltete sich wie folgt:

- 1. Ein Dialog mit ChatGPT wird initiiert, in dem genau festgelegt wird, welche Aufgaben ChatGPT übernehmen soll.
- 2. Bei etwaigen Unklarheiten stellt ChatGPT Rückfragen und speichert eine Zusammenfassung des Dialogs, die nach Bedarf konsultiert werden kann.

5.5.1 Versuch 1

Im Anhang A.1 ist das Gespräch dokumentiert, das mit ChatGPT geführt wird.

Anschließend wird versucht, die ersten 16 Zeilen einer CSV-Datei zu kopieren und einzufügen, was zu dem folgenden Ergebnis führt:

```
// Load CSV block
block LoadCSV {
   type: "LoadFile",
   properties: {
      URL: "path/to/your/csv/file.csv"
// Parse CSV block
block ParseCSV {
   type: "ParseCSV",
   properties: {
      headers: true,
       delimiter:
// Data transformations (if necessary)
// Example: Transforming POINT data to geometric types
block TransformGeolocation {
   type: "TransformData",
  type: "Iransformulata",
properties: {
  operation: "convertToPoint",
  sourceFields: ["Vehicle Location"],
  targetField: "Geolocation"
// Load to SQLite block
block LoadToSQLite
   type: "LoadDatabase",
      databaseType: "SQLite",
databaseURL: "path/to/your/database.db",
       table: "ElectricVehicles",
schema: {
    "VIN": "TEXT",
          "County": "TEXT",
"City": "TEXT",
"State": "TEXT",
         "State": "TEXT",
"Postal Code": "INTEGER",
"Model Year": "INTEGER",
"Make": "TEXT",
"Model": "TEXT",
```

Der Code erwies sich aus folgenden Gründen als ungültig:

- 1. Am Anfang des Codes fehlt die Pipeline.
- 2. Die Blocktypen sind nicht angegeben.

Anschließend wird versucht, das Vorwissen einzubringen. Eine Word-Datei wird erstellt, in welche die Cars-Datenbank¹ sowie der entsprechende Jayvee-Code zusammen mit Anmerkungen eingefügt werden.

Zusammenfassend lässt sich feststellen, dass bei Einreichung derselben Aufgabe bei ChatGPT ein verbessertes Ergebnis erzielt wird:

```
\verb|pipeline| ElectricVehicleDataPipeline| \{
   Extract CSV data from a local or web source
LocalCSVExtractor -> CSVInterpreter;
// Define the transformation and loading steps
CSVInterpreter
-> CleanDataWriter
-> ElectricVehicleTableInterpreter
-> ElectricVehicleDataLoader;
// Block to extract CSV data from a file
block LocalCSVExtractor of type FileExtractor {
     path: "path_to_your_csv_file.csv";
// Block to interpret the CSV data
block CSVInterpreter of type CSVInterpreter {
   enclosing: '"';
// Block to clean and prepare data if necessary
block CleanDataWriter oftype CellWriter {
     // Define any data cleaning or preparation steps here
// Block to interpret cleaned data as a table block ElectricVehicleTableInterpreter oftype TableInterpreter {
     ck ElectricVehicleTableInterprete:
header: true;
columns: [
   "VIN" oftype text,
   "County" oftype text,
   "City" oftype text,
   "State" oftype text,
   "Postal Code" oftype text,
   "Model Year" oftype integer,
   "Make" oftype text,
            "Make" of type text,
"Model" of type text,
```

 $^{^{1}} gist.githubusercontent.com/noamross/e5d3e859aa0c794be10b/raw/b999fb4425b54c63cab088c0ce2c0d6ce961a563/cars.csv$

```
"Electric Vehicle Type" oftype text,

"CAFV Eligibility" oftype text,

"Electric Range" oftype integer,

"Base MSRP" oftype integer,

"Legislative District" oftype integer,

"DOL Vehicle ID" oftype integer,

"Vehicle Location" oftype text,

"Electric Utility" oftype text,

"2020 Census Tract" oftype integer

];
}

// Block to load the table into a SQLite database
block ElectricVehicleDataLoader oftype SQLiteLoader {
  table: "Electric_vehicles.sqlite";
}
```

Dieses Mal wird festgestellt, dass ein "HttpExtractor" fehlt und im "CellWriter" die Befehle "write" und "at" nicht vorhanden sind.

5.5.2 Versuch 2

Beim zweiten Versuch wird auf Deutsch mit ChatGPT kommuniziert. Die Beispiele werden nicht im Chatfenster eingegeben, sondern direkt als Vorwissen integriert (siehe Gespräch im Anhang A.2). Dennoch wird ein Gespräch mit ChatGPT geführt, um Aspekte wie den Gesprächston, den Namen des Generators und das Verfahren bei Rückfragen zu erörtern. Hier findet man den Link zum Generator². Es ist zu beachten, dass ein anderer Autor aufgelistet ist, weil die Zugangsdaten einer anderen Person gehören.

Hier sind die Anforderungen aufgelistet, die ChatGPT aus der Unterhaltung extrahiert hat:

- 1. Verständnis der Datenbankstruktur: Jayvee muss in der Lage sein, eine gegebene Datenbankstruktur zu verstehen und relevante Informationen daraus abzuleiten. Dies beinhaltet die Interpretation von Tabellen, Spalten, Datentypen und Beziehungen innerhalb der Datenbank.
- Generierung von DSL-Code: Basierend auf der Analyse der Datenbankstruktur generiert Jayvee Code für domänenspezifische Sprachen. Der generierte Code sollte effektiv und funktionsreich sein, um die Anforderungen der Ziel-DSL zu erfüllen.
- 3. Berücksichtigung von Richtlinien und Beschränkungen: Der generierte Code muss spezifischen Richtlinien und Beschränkungen der Ziel-DSL entsprechen. Jayvee achtet darauf, dass der Code optimal auf die Ziel-Datenbank abgestimmt ist und wählt zwischen SQLite und Postgres als Ziel-Datenbanksysteme.
- 4. Präzision und Maßschneiderung: Jayvee liefert maßgeschneiderte und präzise Antworten, die auf die individuellen Anforderungen und den Kontext

²chat.openai.com/g/g-2kKhbof2A-jvcodecraft

der Nutzeranfrage zugeschnitten sind. Dies umfasst die Bereitstellung von klaren und präzisen Anweisungen sowie das Vorschlagen effizienter Lösungen.

- 5. Formeller Kommunikationston: Um Professionalität und Klarheit zu gewährleisten, verwendet Jayvee einen formellen Ton in der Kommunikation mit den Nutzern.
- 6. Klärung fehlender Informationen: Falls notwendige Informationen fehlen, fragt Jayvee zunächst nach, bevor er mit der Generierung des Codes fortfährt. Dies stellt sicher, dass die generierten Lösungen den Anforderungen der Nutzer entsprechen.

Als Fazit lässt sich ziehen, dass ChatGPT durch diesen Ansatz das Erstellen von Jayvee-Modellen erlernen konnte. Der dabei generierte Code war allerdings nicht fehlerfrei, konnte jedoch auf Nachfrage verbessert oder manuell angepasst werden. Als zusätzliche Anforderung könnte das Abrufen von Daten aus einer URL betrachtet werden. Hierfür müsste jedoch eine API entwickelt werden, die die Daten für ChatGPT abruft. Dies liegt leider außerhalb des Umfangs dieser Masterarbeit.

5. Design und Implementierung

6 Evaluation

In diesem Kapitel wird untersucht, wie effektiv die vorgeschlagenen Ansätze die gestellten Herausforderungen bewältigen.

Zunächst wird in Kapitel 6 der Evaluator, seine Funktionsweise sowie die verwendeten Evaluierungsmetriken detailliert beschrieben. Abschnitt 6.6 diskutiert, inwieweit die Anforderungen über verschiedene Generationen hinweg erfüllt werden, um die Forschungsfragen zu beantworten. Anschließend wird ein Vergleich dieser Ansätze untereinander vorgenommen.

Der Evaluator erfasst Daten wie:

- Dauer der Codegenerierung für einzelne Datenpunkte in einer Datenbank.
- Anzahl generierter Jayvee-Modelle.
- Anzahl erstellter SQLite- und Postgres-Datenbanken.

Die Trainingsdauer des RL-Modells wird separat erfasst, da das Training unabhängig von der Evaluierung erfolgt und der Evaluator lediglich die Generatoren ausführt.

Die Bewertung wurde auf Dell Inspiron 14 Plus 7420 durchgeführt, der wie folgt konfiguriert ist:

Prozessor: Intel(R) Core(TM) i7-12700H der 12. Generation mit 14 Kernen Arbeitsspeicher: 16 GB Betriebssystem: Windows 11 Home

6.1 Automatisierte Code-Evaluierung

In dieser Arbeit wird das Automatisierte Code-Evaluierung (ACE)-System (Rahman, 2018) verwendet, um das entwickelte Jayvee-Modell zu evaluieren. ACE steht für ein Bewertungssystem zur Code-Evaluierung, das aus den folgenden Teilen besteht:

• Codeauslesung,

- Code-Evaluierung,
- Berechnung der Auswertungspunktzahl (evaluation score) des Codes.

Das ACE-System wird für diese Masterarbeit speziell angepasst, um als Evaluator zu fungieren. Es dient dazu, sowohl das Jayvee-Modell als auch die daraus resultierende Datenbank auf Qualität zu überprüfen und zu testen, ob die Anforderungen korrekt implementiert werden. Der Evaluator führt folgende Aufgaben aus:

- 1. Verwaltung der Generatoren: Dem Evaluator stehen zwei Arten von Generatoren zur Verfügung: Scaffolding und RL. Je nach Bedarf können entweder bis zu 1.000 Scaffolding-Generatoren nacheinander aktiviert werden, wobei die Anzahl flexibel angepasst werden kann, oder alternativ können RL-Generatoren eingesetzt werden.
- 2. Sammlung der Ergebnisse:
 - 2.1 Die Zieldatenbanken werden auf ihre Korrektheit überprüft, indem sie ausgeführt und Daten aus ihnen abgefragt werden. Zieldatenbanken, die erfolgreich Daten zurückgeben, gelten als korrekt. Alle korrekten Zieldatenbanken werden aufgezählt und gespeichert.
 - 2.2 Der Evaluator erfasst Informationen über die Anzahl der eingegebenen Ursprungsdatenbanken und der produzierten Jayvee-Modelle.
 - 2.3 Es werden Informationen bezüglich der Geschwindigkeit der Codegenerierung der verschiedenen Ansätze gesammelt.
- 3. Verwaltung des Grafikgenerators: Nach der Ausführung der Generatoren wird ein separater Grafikgenerator gestartet, der die Evaluationsergebnisse visualisiert.

In diesem Kapitel werden die gesammelten Ergebnisse dargelegt und es erfolgt ein Vergleich zwischen den beiden Generationen der Ansätze sowie untereinander:

- 1. Die Anzahl der korrekt erzeugten Zieldatenbanken wird in dieser Arbeit nicht verglichen, da die Logik zur Erkennung der Spaltendatentypen für beide Ansätze gleich ist und bei beiden Ansätzen sichergestellt wird, dass das produzierte Jayvee-Modell die gleichen Informationen enthält.
- 2. Es wird die Anzahl der Ursprungsdatenbanken, der erzeugten Jayvee-Modelle und der Zieldatenbanken verglichen und erläutert.
- Die Performanz des Scaffolding-Ansatzes wird hinsichtlich der Geschwindigkeit mit dem RL-Ansatz verglichen. Zusätzlich erfolgt ein Vergleich der Ansätze bezüglich Erweiterbarkeit.

6.2 Scaffolding-Ansatz

6.2.1 Funktionale Anforderungen an Scaffolding

Eine Übersicht über die Erfüllung der Anforderungen, die in Abschnitt 3.1 aufgelistet sind, ist in Tabelle 6.1 gegeben.

ID	Anforderungen	
FA_S1_Gen1	Der Ansatz unterstützt ausschließlich den CSVExtractor	
FA_S2_Gen1	Der Ansatz sollte in der Lage sein, ein korrektes Jayvee-Modell	
	zu erstellen, um eine SQLite-Datenbank zu generieren.	
FA_S3_Gen1	Der Ansatz soll genau die verfügbaren Spaltennamen auslesen	ja
	können.	
FA_S4_Gen1	Der Ansatz sollte in der Lage sein, sämtliche	ja
	Datenspalten-Typen als "text" zu speichern. In der 1.	
	Generation wird davon ausgegangen, dass alle	
	$\operatorname{CSV-Ursprungsdatenbanken}$ Spaltennamen enthalten.	

Tabelle 6.1: Zusammenfassung der erfüllten funktionalen Anforderungen an die 1. Generation des Scaffolding-Ansatzes

Somit erfüllte die 1. Generation des Scaffolding-Ansatzes alle Anforderungen.

Um zu prüfen, ob die Anforderung FA_S2_Gen1 erfüllt ist, werden im Evaluator zwölf Zeilen aus der DuckDB-Datenbank abgerufen. Jede Zeile enthält einen Link zu einer Datenbank. Es werden jeweils 6 CSV- und 6 XLSX-Ursprungsdatenbanken abgefragt. Tabelle 6.2 zeigt, dass in der 1. Generation insgesamt 6 aus 6 Jayvee-Modelle und 6 aus 6 SQLite-Datenbanken erzeugt werden. Dies ist darauf zurückzuführen, dass in der 1. Generation des Scaffolding-Ansatzes ausschließlich CSV-Datenbanken unterstützt werden, was auch den Anforderungen entsprach. Somit ist die Anforderung FA S1 Gen1 erfüllt.

Zur Bestätigung der Anforderung FA_S2_Gen1 wird analysiert, ob der Scaffolding-Ansatz wie geplant zur Produktion von 6 aus 6 SQLite-Datenbanken durch ein korrektes Jayvee-Modell geführt hat. Wie in Tabelle 6.2 dargestellt, trifft dies in dieser Arbeit zu. Somit ist die Anforderung FA_S2_Gen1 ebenfalls erfüllt.

Generation	Anzahl der	Anzahl der	Anzahl der
	angefragten	${f erzeugten}$	erzeugten SQLite-
	Datenbanken	${f Jayvee} ext{-}{f Modelle}$	Zieldatenbanken
1	6 CSV + 6 XLSX	$6~\mathrm{CSV}+0~\mathrm{XLSX}$	6 CSV + 0 XLSX
2	$6~\mathrm{CSV} + 6~\mathrm{XLSX}$	$6~\mathrm{CSV}+6~\mathrm{XLSX}$	6 CSV + 6 XLSX

Tabelle 6.2: Vergleich der 1. und 2. Generation des Scaffolding-Ansatzes in Bezug auf die erzeugten Jayvee-Modelle und SQLite-Datenbanken

Um die Korrektheit der Extraktion von Spaltennamen und Datenzeilen durch den

Scaffolding-Ansatz zu überprüfen, werden Daten aus den generierten Datenbanken gesammelt. Die Extraktion erfolgt mittels der externen Bibliothek "Axios" (siehe Abschnitt 5.2.1). Die genaue Implementierung wird in Kapitel 5 beschrieben. Nur CSV-Datenbanken mit Spaltennamen werden berücksichtigt, wodurch 86 Spalten, also 55%, in der 1. Generation ausgelesen werden (siehe Tabelle 6.3). In der 2. Generation werden 100% der Spaltennamen extrahiert. Somit ist auch die Anforderung FA S3 Gen1 erfüllt.

Das Ziel der nächsten Anforderung bestand darin, das Scaffolding-Ansatz so anzupassen, dass es Datentypen als Text speichert. Zu diesem Zweck wird der Beispielcode in 6.1 durch Scaffolding generiert.

```
block Electric Vehicle Table Interpreter of type
   TableInterpreter {
    header: true;
    columns: [
        "VIN" of type text,
        "County" of type text,
        "City" of type text,
        "State" of type text,
        "Postal Code" of type text,
        "Model Year" of type text,
        "Make" of type text,
        "Model" of type text,
        "Electric Vehicle Type" of type text,
        "CAFV Eligibility" of type text,
        "Electric Range" of type text,
        "Base MSRP" of type text,
        "Legislative District" of type text,
        "DOL Vehicle ID" oftype text,
        "Vehicle Location" of type text,
        "Electric Utility" of type text,
        "2020 Census Tract" of type text
    ];
```

Code 6.1: Die 1. Generation des Scaffolding-Ansatzes Ansatzes speichert Spaltendatentypen als "text"

Die erzeugten Datenbanken speichern die Datentypen standardmäßig als "text", anstatt spezifischere Typen wie "decimal", "integer" und ähnliche zu verwenden. In der 1. Generation werden 44% der Zeilen abgelesen, da die XLSX-Dateien nicht berücksichtigt werden. Dies entspricht der Erfüllung der Anforderung FA_S4_-Gen1.

Generation	Anzahl der	Anzahl der	Anzahl der
	erzeugten	abgelesenen Zeilen	abgelesenen Spalten
	Zieldatenbanken		
1	$6~\mathrm{CSV} + 0~\mathrm{XLSX}$	653 aus 1497 (44%)	86 aus 157 (55%)
2	$6~\mathrm{CSV} + 6~\mathrm{XLSX}$	$1496 \text{ aus } 1502 \ (99\%)$	157 aus 157 (100%)

Tabelle 6.3: Vergleich der Anzahl der abgelesenen Zeilen aus der CSV- und XLSX-Ursprungsdatenbanken in SQLite-Zieldatenbanken zwischen der 1. und 2. Generation des Scaffolding-Ansatzes

Die Zusammenfassung der Erfüllung der Anforderungen an die 2. Generation ist in Tabelle 6.4 aufgelistet.

ID	Anforderungen	
FA_S1_Gen2	Erweiterte Unterstützung von Datenquellen: Neben	
	CSV-Dateien wird nun auch die Unterstützung für	
	XLSX-Datenquellen angeboten. Es wird vorausgesetzt, dass die	
	XLSX-Dateien über klare Header verfügen und die Daten	
	entsprechend darunter angeordnet sind. Zusätzlich wird die	
	Verarbeitung von XLSX-Dateien unterstützt, die unmittelbar	
	mit Daten beginnen.	
FA_S2_Gen2	Der Ansatz sollte auch in der Lage sein, Dateien zu	ja
	verarbeiten, selbst wenn diese keine Spaltennamen enthalten.	
FA_S3_Gen2	FA S3 Gen2 Der Ansatz soll ebenfalls PostgresLoader unterstützen.	
FA_S4_Gen2	Der Ansatz sollte fähig sein, alle von Jayvee unterstützten	
	Datenspalten-Typen zu identifizieren.	

Tabelle 6.4: Zusammenfassung der erfüllten funktionalen Anforderungen an die 2. Generation des Scaffolding-Ansatzes

Die Anforderung FA_S1_Gen2 führte zur Implementierung von XLSX-Dateiunterstützung in der 2. Generation. Tabelle 6.2 zeigt, dass im Vergleich zur 1. Generation des Scaffolding-Ansatzes aus 6 XLSX-Ursprungsdatenbanken korrekt 6 aus 6 Jayvee-Modelle erzeugt werden. Insgesamt werden 12 aus 12 Zieldatenbanken erzeugt, da die CSV-Ursprungsdatenbanken mitgezählt werden. Auf diese Weise wird die Anforderung FA S1 Gen2 erfüllt.

Fehlen Spaltennamen in einer CSV- oder XLSX-Datei, verwendet das System automatisch die erste Zeile als Spaltennamen. Tabelle 6.15 belegt, dass alle Ursprungsdatenbanken ohne Spaltennamen korrekt ausgelesen werden. Damit ist auch die Anforderung FA S2 Gen2 auch erfüllt.

Der Scaffolding-Ansatz in der 2. Generation kann ebenfalls aus CSV- und XLSX-Datenbanken validen Jayvee-Code für Postgres-Datenbanken generieren. Er verwendet dabei den Postgres-Loader-Blocktyp, in den man eigene Daten eingeben kann. In Tabelle 6.5 sieht man, dass auch Jayvee-Code zur Erzeugung einer Postgres-Datenbank generiert werden kann.

Generation	Anzahl der	Anzahl der	Anzahl der
	angefragten	${f erzeugten}$	erzeugten Postgres-
	Datenbanken	Jayvee-Modelle	Zieldatenbanken
1	6 CSV + 0 XLSX	6 CSV + 0 XLSX	6 CSV + 0 XLSX
2	$6~\mathrm{CSV} + 6~\mathrm{XLSX}$	$6~\mathrm{CSV} + 6~\mathrm{XLSX}$	$4~\mathrm{CSV} + 5~\mathrm{XLSX}$

Tabelle 6.5: Vergleich der 1. und 2. Generation des Scaffolding-Ansatzes in Bezug auf die erzeugten Jayvee-Modelle und Postgres-Datenbanken

Bei der Generierung von Datenbanken aus dem erzeugten Jayvee-Code treten zwei Limitationen auf: Zum einen ist ein Integer in Postgres 4 Bytes groß, während in SQLite ein Integer bis zu 8 Bytes groß sein kann. In Postgres führt dies beim Import von Zahlen, die größer als 4 Bytes sind, zu Problemen, wenn die Daten als Integer importiert werden. Postgres verfügt über den Datentyp bigint, der bis zu 8 Bytes große Integer speichern kann. Zum anderen besteht die Herausforderung darin, dass Jayvee den bigint-Datentyp nicht unterstützt. Um diesen Datentyp für Postgres lesbar zu machen, werden die Daten gemäß Abschnitt 5.2.2 als "text" gespeichert.

Diese Zwischenlösung ermöglicht den Import und die Speicherung der Daten. Zukünftig sollte eine Erweiterung von Jayvee erfolgen, um den Datentyp 'bigint' zu unterstützen, der größere Zahlen verarbeiten kann. Dies wird durch Tabelle 6.5 illustriert, wo ersichtlich ist, dass aufgrund dieser Limitation jeweils 4 von 6 CSV und 5 von 6 XLSX in Postgres-Zieldatenbanken korrekt generiert wurden.

Die Erkennung von Spaltendatentypen birgt auch Nachteile. Pandas erkennt nicht immer die Datentypen korrekt, wodurch manche Spalten falsch klassifiziert werden. Zum Beispiel wird die Spalte "barrierefrei" in der Datenbank "Hörgeräteakustiker" fälschlicherweise als "decimal" erkannt, obwohl sie "boolean" sein sollte. Dies führte zur Konvertierung von 4 von 6 CSV- und 5 von 6 XLSX-Ursprungsdatenbanken, insgesamt zu 9 aus 12 Postgres-Zieldatenbanken. Somit ist die Anforderung FA S3 Gen2 auch erfüllt.

In dieser Arbeit setzt der Scaffolding-Ansatz der 2. Generation auf eine hybride Lösung, die sowohl die Bibliothek "MessyTables" als auch selbst entwickelte Algorithmen unter Verwendung der Bibliothek "Pandas" zur Erkennung von Spaltendatentypen integriert. Dies markiert eine deutliche Weiterentwicklung im Vergleich zur 1. Generation, in der lediglich die Bibliothek "Axios" zum Einsatz kam. Die detaillierte Implementierung wird in Kapitel 5 beschrieben.

Die Bibliothek "MessyTables" wird bei einer CSV-Datenbank eingesetzt, um Spaltennamen zu identifizieren, insbesondere in Fällen, wo die erste Zeile Datenbankinformationen statt Spaltennamen enthält. Zudem dient "MessyTables" dazu, die Datentypen der Spalten zu bestimmen, indem es geeignete Heuristiken zur Aus-

 $^{^{1}}$ geo.sv.rostock.de/download/opendata/hoergeraeteakustiker/hoergeraeteakustiker.xlsx

wahl der richtigen Zeile und Statistiken zur Erkennung der Datentypen verwendet (The Open Knowledge Foundation Ltd, 2013).

Für XLSX-Datenbanken wird die Datei zunächst in einen "Pandas" DataFrame geladen, um die automatische Datentypenerkennung zu nutzen.

Wie in Punkt 5.2.1 dargelegt, verzichtet diese Arbeit auf den Einsatz fortgeschrittener Methoden zur Spaltendatentyperkennung durch maschinelles Lernen, etwa dem Sherlock-Modell des MIT für Deep Learning. Der Schwerpunkt liegt stattdessen auf der schnellen Generierung von Vorlagen, nicht auf der Entwicklung perfekter Jayvee-Modelle. Deshalb werden "MessyTables" für CSV-Dateien genutzt und "Pandas" für XLSX-Dateien gewählt. Somit ist die Anforderung FA_S4_Gen2 auch erfüllt.

Zusammenfassend sind alle Anforderungen an den Scaffolding-Ansatz korrekt umgesetzt worden.

6.2.2 Qualitätsanforderungen Anforderungen an Scaffolding

In Tabelle 6.6 ist ersichtlich, inwiefern die Anforderungen an die 2. Generation des Scaffolding-Ansatzes erfüllt werden.

ID	ID Anforderungen	
QA_S1_Gen1	Der Ansatz sollte performant sein, also Daten möglichst schnell	
	verarbeiten können. Zudem soll der Ansatz skalierbar sein und	
	in der Lage sein, aus 1.000 CSV-Dateien aus einer Datenbank	
	funktionsfähige SQLite-Datenbanken zu generieren.	
QA_S2_Gen1	Der Ansatz sollte keine nicht öffentlichen Bibliotheken	ja
	verwenden, sondern nur solche mit gängingen Open-Source	
	Lizenzen.	

Tabelle 6.6: Zusammenfassung der erfüllten Qualitätsanforderungen an die 1. Generation des Scaffolding-Ansatzes

Um die Performanz und Skalierbarkeit des Scaffolding-Ansatzes zu überprüfen, wird getestet, wie schnell er Ursprungsdatenbanken in Zieldatenbanken umwandelt. Tabelle 6.7 zeigt, dass der Scaffolding-Ansatz die Jayvee-Modelle in 12 Sekunden erstellt. Da der Ansatz größtenteils auf If-Anweisungen basiert, gibt es keinen direkten Weg, die Performanz weiter zu steigern. Die Dateigröße beeinflusst ebenfalls die Geschwindigkeit. Außerdem gilt: Je größer die Anzahl der Spalten, desto langsamer erfolgt die Umwandlung einer Ursprungsdatenbank in eine Zieldatenbank.

Der Scaffolding-Ansatz ist auch skalierbar. In Tabelle 6.8 sind die Ergebnisse für größere Datenmengen dargestellt. Es wird getestet, wie viele Zieldatenbanken aus 1.000 angeforderten Ursprungsdatenbanken (500 CSV- und 500 XLSX-Datenbanken) erstellt werden können. Insgesamt werden 1.003 Jayvee-Modelle

und 919 SQLite-Zieldatenbanken produziert, wobei der Prozess insgesamt 1,19 Stunden dauerte.

Generation	Anzahl der erzeugten SQLite-Zieldatenbanken	Verbrauchte Zeit in Sekunden
1 2	$egin{array}{ll} 6~\mathrm{CSV} &+ 0~\mathrm{XLSX} \ 6~\mathrm{CSV} &+ 6~\mathrm{XLSX} \end{array}$	12,03 29.03

Tabelle 6.7: Vergleich der 1. und 2. Generation des Scaffolding-Ansatzes hinsichtlich der Gesamtgeschwindigkeit bei der Erzeugung von SQLite-Zieldatenbank aus 6 CSV- und 6 XLSX-Ursprungsdatenbanken

Es werden 1.003 Jayvee-Modelle produziert, da jedes Blatt in der XLSX-Datei einzeln als Datenbank betrachtet wird. Der Scaffolding-Ansatz hat 919 Zieldatenbanken erstellt, da einige Datenbanken aufgrund ihrer Größe übersprungen werden; wenn der Download länger als fünf Sekunden dauert, wird dieser unterbrochen. In der DuckDB-Datenbank existieren Links, die zu nicht mehr existierenden Datenbanken führen. Zudem weisen viele XLSX-Ursprungsdatenbanken eine unpassende Struktur auf, wie in Abbildung 6.1 dargestellt. Dateien, die nicht analysiert werden können, werden als unstrukturierte Dateien mit Präambel bezeichnet, weil sie häufig Titelinformationen oder Text enthalten, der nicht direkt mit den Daten zusammenhängt. Die Herausforderungen, die häufig bei öffentlichen Datensätzen auftreten, werden in der Entwicklung des Jayvee-Modells berücksichtigt. Dieses Modell lässt sich zukünftig anpassen, indem es beispielsweise durch Jayvee-Tools wie den "RowDeleter" oder den "TextLineDeleter" erweitert wird. Dadurch kann der generierte Jayvee-Code effektiv genutzt werden, um trotz dieser Herausforderungen mit den Daten zu arbeiten.

In Tabelle 6.8 wird dargestellt, dass aus einem Datensatz von 50.000 CSV- und 50.000 XLSX-Ursprungsdatenbanken insgesamt 10.442 Jayvee-Modelle generiert wurden. Im Vergleich dazu wurden aus einer kleineren Menge von 5.000 CSV- und 5.000 XLSX-Datenbanken 13.040 Jayvee-Modelle abgeleitet. Dieses überraschende Ergebnis kann durch die Anwendung eines LIMIT zur Begrenzung der Anzahl der generierten Modelle erklärt werden. Die Variation in den Datensätzen, bedingt durch unterschiedliche Größen der Ursprungsdatenbanken (1.000, 10.000, 100.000), führt zu unterschiedlichen Ergebnissen. Eine konsistentere Datengewinnung hätte durch die Anwendung einer ORDER BY-Klausel erreicht werden können. Diese wurde jedoch nicht verwendet, um die Komplexität der Abfragen zu reduzieren und eine pragmatischere Lösung zu bevorzugen. Dies stellt eine bewusste Entscheidung dar, um Effizienz über absolute Konsistenz zu priorisieren.

²https://jvalue.github.io/jayvee/docs/user/block-types/RowDeleter/

³https://jvalue.github.io/jayvee/docs/user/block-types/TextLineDeleter/

1. Baugenehmigungen ¹ im Wohn- und Nichtwohnbau in Schleswig-Holstein im August 2015					st 2015
			Wohnungen		
KREISFREIE STADT Kreis	Wohn- gebäude	Nichtwohn- gebäude	insgesamt (einschl. durch Baumaß- nahmen entstanden)	darunter N Wohnge- bäuden mit 1 und 2 Wohnungen	Wohnge- bäuden mit 3 und mehr Wohnungen
	Anzahl	Anzahl	Anzahl	Anzahl	Anzahl
FLENSBURG	8	6	18	4	14
KIEL	13	2	5	4	_
LÜBECK	18	7	9	2	4
NEUMÜNSTER	2	_	4	_	4

Abbildung 6.1: Beispiel einer XLSX-Datei, die nicht von der Ursprungs- in die Zieldatenbank importiert werden konnte

Es besteht keine Begrenzung bezüglich der Anzahl an Datenbanken, die erzeugt werden können - es ist möglich, Millionen zu generieren. Dies erfordert jedoch erhebliche Zeit und Rechenkapazitäten und unterliegt den sprachlichen Einschränkungen von TypeScript. Zusätzlich beeinflusst die Geschwindigkeit der Internetverbindung die Performanz, da die CSV- und XLSX-Dateien zunächst heruntergeladen werden müssen. Potenzielle Verbesserungen werden in Kapitel 7 ausführlich diskutiert. Deswegen ist die Anforderung QF_S1_G1 erfüllt.

Anzahl der CSV- und XLSX- Ursprungs- datenbanken	Anszahl der erzeugten Jayvee- Modelle	Anzahl der erstellten Zieldaten- banken	Verbrauchte Zeit in Stunden
$500~\mathrm{CSV} + 500$	1.003	919	1,18
XLSX			
$5.000~{ m CSV} + 5.000$	13.040	1.181	5,45
XLSX			
$50.000~\mathrm{CSV}~+$	10.442	2.185	107,82
50.000 XLSX			

Tabelle 6.8: Die Geschwindigkeit der 2. Generation des Scaffolding-Ansatzes bei der Erzeugung von SQLite-Zieldatenbanken für größere Datenmengen

Der Überblick über die verwendeten Bibliotheken ist im Anhang B aufgelistet. Alle haben öffentlichen Lizensen. So ist die Anforderung QF_S2_G1 erfüllt.

Somit wird gezeigt, dass alle Qualitätsanforderungen an die 2. Generation des Scaffolding-Ansatzes erfüllt sind.

In Tabelle 6.9 wird dargestellt, ob die Anforderungen an die 2. Generation des Scaffolding-Ansatzes erfüllt sind.

ID	${\bf An forderungen}$	Erfüllt
QA_S1_Gen2	Der Code des Ansatzes sollte möglichst erweiterbar sein. Der	ja
	Code des Ansatzes soll modernen Entwicklungstechniken	
	folgen, sinnvolle Namen verwenden und erweiterbar sein	
QA_S2_Gen2	Der Ansatz soll einfach zu nutzen sein. Durch den Terminal	ja
	sollte der Nutzer mit der Eingabe einer Einweisung und einem	
	Link zur CSV-Datei den Generator starten können.	

Tabelle 6.9: Zusammenfassung der erfüllten Qualitätsanforderungen an die 2. Generation des Scaffolding-Ansatzes

Um die in Abschnitt 5.3 beschriebene Erweiterbarkeit des Scaffolding-Ansatzes zu verdeutlichen, werden zwei wesentliche Schritte unternommen.

Erstens wird in Code 6.2 gezeigt, wie die Funktionalität des Scaffolding-Ansatzes erweitert wird, um Postgres zu unterstützen. Hierfür ist das Hinzufügen einer "else if"-Bedingung notwendig, die die Generierung von Postgres-spezifischem Code ermöglicht.

Zweitens ist es entscheidend, die Logik für das Importieren der Ursprungsdatenbanken sowie die Erkennung der Spaltendatentypen anzupassen. Dies ist notwendig, da in der 1. Generation ausschließlich CSV-Ursprungsdatenbanken unterstützt werden. Die Methode zum Importieren von XLSX-Ursprungsdatenbanken wird bereits in Abschnitt 5.2.2 erläutert. Somit ist die Anforderung QF_S1_G2 erfüllt, wie im Text hervorgehoben.

```
if (dbType == "sqlite") {
    code += ` \ n `
    code += '
    block ${pipelineName + sheetName}Loader oftype
       SQLiteLoader {
        table: "${pipelineName + sheetName}";
        file: "./${pipelineName + sheetName}.sqlite";
    } '
} else if (dbType == "postgres") {
    code += ' \ n'
    code += 
    block ${pipelineName + sheetName}Loader oftype
       PostgresLoader
        table: "${pipelineName + sheetName}";
        port: 5432;
        username: "postgres";
        password: "password";
        database: "postgres";
        host: "localhost";
```

```
} '
```

Code 6.2: Erweiterung des Scaffolding-Ansatzes um Postgres-Funktionalität

Um sicherzustellen, dass der Scaffolding-Ansatz benutzerfreundlich ist, wird er so implementiert, dass er durch Eingabe von Parametern gesteuert wird. Der Benutzer gibt einen Link und den Typ der Zieldatenbank (z.B. SQLite oder Postgres) ein. Auf diese Weise wird die Anforderung QF_S2_G2 erfüllt.

6.3 Reinforcement-Learning-Ansatz

6.3.1 Funktionale Anforderungen an Reinforcement Learning

In Tabelle 6.10 wird eine Zusammenfassung der Erfüllung der funktionalen Anforderungen an RL-Ansatz präsentiert.

ID	${\bf An for derungen}$	Erfüllt
FA_RL1_Gen1	Der Ansatz sollte in der Lage sein, ein valides Jayvee-Code zu	ja
	generieren, das nicht unbedingt alle Randfälle abdecken muss.	
FA_RL2_Gen1	Der Ansatz sollte in der Lage sein, sämtliche	ja
	Datenspalten-Typen als "text" zu speichern. In der 1.	
	Generation wird davon ausgegangen, dass die CSV-Dateien	
	Spaltennamen enthalten.	

Tabelle 6.10: Zusammenfassung der erfüllten funktionalen Anforderungen an die 1. Generation des RL-Ansatzes

Der RL-Ansatz in der 1. Generation produziert validen Jayvee-Code, der SQLite-Datenbanken generiert. In Tabelle 6.11 ist ersichtlich, dass 6 Jayvee-Modelle für 6 CSV-Ursprungsdatenbanken mit RL erstellt werden. Somit ist die Anforderung FA_RL1_Gen1 erfüllt.

Generation	Anzahl der	Anzahl der	Anzahl der
	angefragten	${f erzeugten}$	erzeugten SQLite-
	Datenbanken	Jayvee-Modelle	Zieldatenbanken
1	6 CSV + 6 XLSX	6 CSV + 0 XLSX	6 CSV + 0 XLSX
2	$6~\mathrm{CSV} + 6~\mathrm{XLSX}$	$6~\mathrm{CSV} + 6~\mathrm{XLSX}$	$6~\mathrm{CSV} + 6~\mathrm{XLSX}$

Tabelle 6.11: Vergleich der 1. und 2. Generation des RL-Ansatzes in Bezug auf die erzeugten Jayvee-Modelle und SQLite-Datenbanken

In der 1. Generation des RL-Ansatzes werden die Spaltendatentypen analog zum Scaffolding-Ansatz als "text" gespeichert, wie der Code 6.3 demonstriert. Zusätzlich zeigt Tabelle 6.11, dass aus 6 Ursprungsdatenbanken 6 Zieldatenbanken erzeugt werden. Daher ist die Anforderung FA RL2 Gen1 erfüllt.

```
block Electric Vehicle Table Interpreter of type
   TableInterpreter {
    header: true;
    columns:
        "VIN" of type text,
        "County" of type text,
        "City" of type text,
        "State" of type text,
        "Postal Code" oftype text,
        "Model Year" of type text,
        "Make" of type text,
        "Model" of type text,
        "Electric Vehicle Type" of type text,
        "CAFV Eligibility" of type text,
        "Electric Range" of type text,
        "Base MSRP" of type text,
        "Legislative District" of type text,
        "DOL Vehicle ID" oftype text,
        "Vehicle Location" of type text,
        "Electric Utility" of type text,
        "2020 Census Tract" oftype text
    ];
```

Code 6.3: Die 1. Generation des RL-Ansatzes speichert Spaltendatentypen als Text

Die Zusammenfassung der Erfüllung der Anforderungen an die 2. Generation ist in Tabelle 6.12 aufgelistet.

Im Laufe der 2. Generation wird der RL-Ansatz zur Nutzung mehrerer Prozessoren umgestaltet. Wie in Kapitel 5 erläutert, wird das Modell so entwickelt, dass es die Rechenressourcen einer CPU optimal ausnutzt. Es verwendet die Bibliothek "Gymnasium", welches Ressourcen besonders effizient nutzt.

Zusätzlich wird die Bibliothek "Stable Baselines3" eingesetzt, um das Training von DQN-Verfahren zu beschleunigen. Wie in Abbildung 6.2 dargestellt, hat sich die Trainingszeit signifikant verringert. Zur Überprüfung wird eine lineare Regression durchgeführt. Die Steigung beträgt für die 1. Generation 349 Sekunden pro Ursprungsdatenbank und für die 2. Generation 9 Sekunden pro Ursprungsdatenbank, was eine erhebliche Verbesserung darstellt. Eine Einschränkung besteht jedoch darin, dass die Arbeit nicht zwischen verschiedenen Knoten verteilt wird, sondern lediglich auf die Kerne eines einzelnen Prozessors beschränkt ist. Die Anforderung FA_RL1_Gen2 ist somit erfüllt.

ID	${\bf An for derungen}$	Erfüllt
FA_RL1_Gen2	Statt die Umgebung auf einen Prozessor laufen zu lassen,	ja
	werden mehrere Umgebungen auf mehreren Kernen laufen	
FA_RL2_Gen2	Verbesserte Fehlerbehandlung und Flexibilität: Fähigkeit,	ja
	Dateien ohne Spaltennamen zu verarbeiten	
FA_RL3_Gen2	Erweiterte Unterstützung von Datenquellen: Neben	ja
	CSV-Dateien wird nun auch die Unterstützung für	
	XLSX-Datenquellen angeboten. Es wird vorausgesetzt, dass die	
	XLSX-Dateien über klare Header verfügen und die Daten	
	entsprechend darunter angeordnet sind. Zusätzlich wird die	
	Verarbeitung von XLSX-Dateien unterstützt, die unmittelbar	
	mit Daten beginnen.	
FA_RL4_Gen2	Erweiterte Ladekapazitäten: Unterstützung für das Laden von	ja
	Daten in Postgres-Datenbanken, zusätzlich zu SQLite	
FA_RL5_Gen2	Der Ansatz sollte fähig sein, die Datentypen der Spalten	ja
	korrekt zu identifizieren	

Tabelle 6.12: Zusammenfassung der erfüllten funktionallen Anforderungen an die 2. Generation des RL-Ansatzes

Es wird ebenfalls untersucht, wie sich die Anzahl der Trainingsdaten auf die Codegenerierungszeit des Reinforcement-Learning-Ansatzes auswirkt. Hierzu wird jedes Modell (mit 12, 1.000 und 10.000 Trainingsdatensätzen) verwendet, um dreimal zwölf Datenbanken zu erstellen. Der Durchschnitt dieser Durchläufe wird berechnet. Wie in Tabelle 6.13 ersichtlich, hat sich die Geschwindigkeit erhöht.

Zudem werden die Standardabweichungen berechnet, wie in Abbildung 6.13 dargestellt. Das Training des Modells mit 10.000 Datensätzen hat nicht nur die Geschwindigkeit erhöht, sondern auch die Konsistenz verbessert, da die Standardabweichung lediglich 0,51 Sekunden beträgt. Dies zeigt eine verbesserte Stabilität und Vorhersagbarkeit der Trainingsdauer, was besonders bei größeren Datensätzen von Vorteil ist.

Es sollte berücksichtigt werden, dass die Daten nicht bereinigt sind. Viele URLs in der DuckDB-Datenbank führen zu fehlerhaften Daten und die Strukturen sind oft unzureichend (z. B. unstrukturierte Dateien mit Präambeln). Jede Aktion, die das Modell basierend auf diesen Daten auswählt, führte zu schlechteren Ergebnissen, was die Performanz potenziell mindern könnte.

Für zukünftige Forschungen könnte die Entwicklung einer umfassend bereinigten Datenbank in Erwägung gezogen werden, um deren Einfluss auf die Geschwindigkeit der Codegenerierung zu untersuchen. Zudem besteht die Möglichkeit, das Modell mit einer deutlich größeren Anzahl an Datensätzen, beispielsweise bis zu 100.000, zu trainieren, um die Auswirkungen auf die Geschwindigkeit zu beobachten.

Im Rahmen dieser Arbeit wird untersucht, wie sich eine geringe Anzahl an Daten-

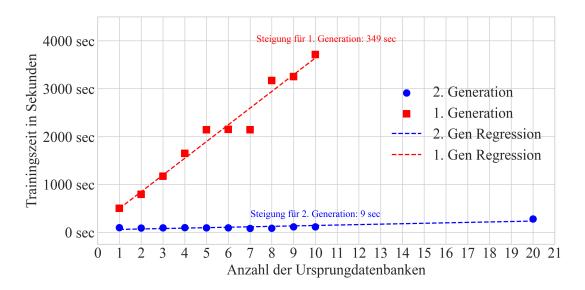


Abbildung 6.2: Vergleich der Trainingszeiten zwischen der 1. und 2. Generation des RL-Ansatzes

Anzahl der	Anzahl an	Durchschnitt-	Standard-
Datenpunkte,	Episoden	liche	abweichung
die für das		Geschwindig-	
Training des		keit der	
Modells		Codegenerie-	
verwendet		rung für 6 CSV	
werden		+ 6 XLSX	
6 CSV + 6	3	94,46 Sekunden	1,89 Sekunden
XLSX = 12			
500 CSV + 500	3	94,31 Sekunden	2,5 Sekunden
XLSX = 1.000			
$5.000~\mathrm{CSV}~+$	3	91,61 Sekunden	0,51 Sekunden
5.000 XLSX =			
10.000			

Tabelle 6.13: Geschwindigkeitsvergleich der Codegenerierung für Modelle, die mit 12, 1.000 und 10.000 Datenbanken in der 2. Generation trainiert werden

banken (6 CSV und 6 XLSX) sowie eine große Anzahl an Episoden (300 Episoden) auf die Geschwindigkeit auswirken. Wie in Tabelle 6.14 ersichtlich, hat sich die Performanz und Konsistenz bei 300 Episoden verschlechtert. Daher wird empfohlen, bei Vorhandensein vieler Ursprungsdatenbanken eine größere Menge an Trainingsdatenbanken und eine geringere Anzahl an Episoden zu wählen.

Der in der Tabelle 6.15 dargestellte Abschnitt verdeutlicht, dass Datenbanken,

Anzahl der	Anzahl an	Durchschnitt-	Standard-
Datenpunkte,	Episoden	liche	abweichung
die für das		Geschwindig-	
Training des		keit der	
Modells		Codegenerie-	
verwendet		rung für 6 CSV	
werden		+ 6 XLSX	
6 CSV + 6	3	94,46 Sekunden	1,89 Sekunden
XLSX = 12			
6 CSV + 6	300	105,04 Sekunden	5,26 Sekunden
XLSX = 12			

Tabelle 6.14: Geschwindigkeitsvergleich der Codegenerierung für Modelle, trainiert mit 3 bzw. 300 Episoden in der zweiten Generation

die keine Spaltennamen enthalten, ebenfalls unterstützt werden. Speziell zu Demonstrationszwecken wurden bei zwei Datenbanken die Spaltenüberschriften entfernt - konkret handelt es sich um zwei CSV- und zwei XLSX-Dateien. Auf Basis dieser modifizierten Datenbanken wurden jeweils vier Jayvee-Modelle sowie vier SQLite-Zieldatenbanken erstellt. Somit ist die Anforderung FA_RL2_Gen2 ebenfalls erfüllt.

	Typ der Ursprungsdaten- bank	Anzahl der angefragten Datenbanken	Anzahl der erzeugten Jayvee-Modelle	Anzahl der erzeugten SQLite-
	bank	Davenbanken	Jay vee Modelle	Zieldatenbanken
Ī	CSV	2	2	2
	XLSX	2	2	2

Tabelle 6.15: Anzahl der getesteten Ursprungsdatenbanken ohne Spaltennamen

Genauso wie bei der 2. Generation von Scaffolding wird die Unterstützung von XLSXInterpreter eingebaut. In Tabelle 6.11 sehen wir, dass korrekte Jayvee-Modelle erzeugt werden. Es werden, ähnlich wie bei Scaffolding, 6 CSV- und 6 XLSX-Ursprungsdatenbanken aus der DuckDB-Datenbank entnommen. In der 1. Generation sieht man nur 6, da dort nur CSVs unterstützt werden, während in der 2. Generation 12 aus 12 SQLite-Datenbanken generiert werden, weil dort auch XLSX-Datenbanken unterstützt werden. Somit ist die Anforderung FA RL3 Gen2 ebenfalls erfüllt.

Wie bei der 2. Generation von Scaffolding wird die Unterstützung von Postgres-Loader implementiert. Wie in Tabelle 6.16 ersichtlich, wurden erfolgreich 12 von 12 korrekten Jayvee-Modellen erstellt. Zudem zeigt sich, dass aufgrund der im Punkt 6.2.1 diskutierten Einschränkungen jeweils 4 von 6 CSV und 5 von 6 XLSX in Postgres-Zieldatenbanken korrekt generiert wurden. Somit ist die Anforderung FA RL4 Gen2 ebenfalls erfüllt.

	Generation	Anzahl der	Anzahl der	Anzahl der
		${f ange fragten}$	${f erzeugten}$	erzeugten Postgres-
		Datenbanken	Jayvee-Modelle	Zieldatenbanken
Ì	1	6 CSV + 6 XLSX	$6~\mathrm{CSV} + 0~\mathrm{XLSX}$	6 CSV + 0 XLSX
Ì	2	$6~\mathrm{CSV} + 6~\mathrm{XLSX}$	$6~\mathrm{CSV} + 6~\mathrm{XLSX}$	$oxed{4~ ext{CSV} + 5~ ext{XLSX}}$

Tabelle 6.16: Vergleich der 1. und 2. Generation des RL-Ansatzes in Bezug auf die erzeugten Jayvee-Modelle und Postgres-Datenbanken

Ähnlich wie beim Scaffolding-Ansatz wird eine Kombination aus Bibliothek "MessyTables" und einer eigenen Implementierung verwendet, um die Datentypen der Spalten zu erkennen. Das Verfahren zur Erkennung der Spaltendatentypen wird analog zum Scaffolding implementiert und wird detailliert in Abschnitt 6.4 erläutert. Somit ist die Anforderung FA_RL5_Gen2 ebenfalls erfüllt.

6.3.2 Qualitätsanforderungen an Reinforcement Learning

In Tabelle 6.17 sind die Qualitätsanforderungen an den RL-Ansatz aufgelistet.

ID	Anforderungen	Erfüllt
QA_RL1_Gen1	Erweiterbarkeit und Flexibilität: Fähigkeit, neue Funktionen	ja
	und Verbesserungen leicht zu integrieren	
QA RL2 Gen1	Nutzererfahrung: Klare und zugängliche Schnittstelle für	ja
	Training und Bewertung	

Tabelle 6.17: Zusammenfassung der erfüllten Qualitätsanforderungen an die 1. Generation des RL-Ansatzes

Der Ansatz wird so konzipiert, dass das Hinzufügen neuer Funktionen einfach ist. Eine detaillierte Erklärung befindet sich im Abschnitt 5.4. Um beispielsweise Unterstützung für Postgres zu integrieren, sollten die folgenden wesentlichen Schritte durchgeführt werden:

- 1. Eine neue Aktion zur Aktionsliste und Umgebung hinzufügen. Das Modell muss wissen, welche Aktionen es auswählen kann.
- 2. Die Liste der genutzten Aktionen aktualisieren. Der Nutzer gibt Heuristiken an das Modell, um die Effizienz des Lernens zu steigern. Wenn der Nutzer beispielsweise Postgres als Zieldatenbank auswählt, wird SQLite als "bereits verwendete Aktion" gesetzt, sodass das Modell diese Aktion nicht mehr wählen darf.

3. Das Modell trainieren. Das Modell muss die neuen Aktionen ausprobieren. Wenn das Modell nicht weiß, dass eine neue Funktion hinzugefügt wird, wird es diese auch nicht nutzen können.

Deswegen ist die Anforderung QA RL1 Gen1 erfüllt.

Der Nutzer kann über das Terminal einen Link und eine Zieldatenbank auswählen. Der Evaluator stellt anschließend eine Verbindung zur URL her und ermittelt automatisch vom Server, ob es sich um eine CSV- oder XLSX-Datei handelt. Es ist zudem wichtig, dass der Evaluator die erfassten Parameter wie URL, Ursprungsdatenbank und Zieldatenbank eindeutig an den Generator weitergibt. Daher ist auch die Qualitätsanforderung QA RL2 Gen1 erfüllt.

Zusammenfassend sind alle Qualitätsanforderungen an die 2. Generation des RL erfüllt.

In Tabelle 6.18 sind alle erfüllten Qualitätsanforderungen an die 2. Generation des RL-Ansatzes aufgelistet

ID	${\bf An for derungen}$	Erfüllt
QA_RL1_Gen2	Der Code des Ansatzes sollte möglichst erweiterbar sein. Der	ja
	Code soll modernen Entwicklungstechniken folgen, sinnvolle	
	Namen verwenden und erweiterbar sein.	
QA_RL2_Gen2	Der Ansatz soll einfach zu nutzen sein. Durch den Terminal	ja
	sollte der Nutzer mit der Eingabe einer Einweisung und einem	
	Link zur CSV-Datei den Generator starten können.	

Tabelle 6.18: Zusammenfassung der erfüllten Qualitätsanforderungen an die 2. Generation des RL-Ansatzes

Um die Erweiterbarkeit des Codes zu demonstrieren, werden mehrere Schritte durchgeführt.

Erstens müssen neue Aktionen hinzugefügt werden. Beispielsweise muss man bei Postgres, wie in Code 6.4 dargestellt, diese und andere Postgres-bezogene Aktionen zur Liste der Aktionen hinzufügen. Anschließend muss das Modell neu trainiert werden.

```
file . write ("\n")
```

Code 6.4: Erweiterung des RL-Ansatzes um Postgres-Funktionalität

Zweitens muss man, ähnlich wie beim Scaffolding-Ansatz, die Logik für das Importieren der Ursprungsdatenbanken anpassen.

Eine detaillierte Erklärung, wie neue Funktionalitäten hinzugefügt werden, findet sich in Abschnitt 5.4. Damit ist die Qualitätsanforderunge QA_RL1_Gen2 umgesetzt.

Den RL-Ansatz kann man ebenso wie den Scaffolding-Ansatz über das Terminal starten. Dies bietet aus denselben Gründen Vorteile. Somit ist die Anforderung QA_RL2_Gen2 ebenfalls erfüllt.

6.4 Evaluation der Anzahl der importierten Zeilen

Die Bewertung der importierten Zeilen aus CSV- oder XLSX-Datenbanken ist wichtig, da Jayvee nur die Zeilen importiert, die den unter "columns" gelisteten Typen entsprechen; alle anderen Zeilen werden übersprungen. Zwischen Scaffolding und RL wird hierbei nicht unterschieden, da beide Ansätze dieselbe Methode verwenden, um Spaltennamen und Datentypen zu erfassen. Der Vergleich erfolgt zwischen den Generationen, da in der 1. Generation keine Logik zur Erkennung von Spaltendatentypen existierte, während in der 2. Generation eine Kombination aus "MessyTables" und eigens entwickelter Logik verwendet wird.

Tabelle 6.3 zeigt, dass alle Zeilen aus den CSV-Ursprungsdatenbanken erfolgreich importiert werden. Dies geschieht, da keine spezifischen Datentypen erkannt werden und folglich alle Daten als "text" abgespeichert werden. XLSX-Datenbanken werden hingegen nicht berücksichtigt, da die erste Generation der Ansätze diese nicht unterstützt.

Die 2. Generation hat auch nicht alle Zeilen in die Zieldatenbank importiert. Dies liegt daran, dass nun nicht mehr pauschal 'text' als Spaltendatentyp verwendet wird, sondern die erkannten Datentypen angewendet werden. Dies wird durch Code 6.5 verdeutlicht. Eine manuelle Überprüfung der ursprünglichen CSV-Datenbank wird durchgeführt, um zu bestätigen, dass die Spaltendatentypen den erkannten Datentypen entsprechen. Die Überprüfung ergab, dass die erkannten Datentypen korrekt waren.

```
block Bevoelkerungsstruktur2010altersql_statemen
tTableInterpreter oftype TableInterpreter {
    header: true;
    columns: [
```

```
"stadtbereich code" oftype text,
    "stadtbereich\_bezeichnung" \ of type \ text \; ,
    "durchschnittsalter" oftype decimal,
    "jugend quotient" \ of type \ decimal \ ,
    "altenquotient" \ of type \ decimal \ ,
    "anzahl juenger 3" oftype integer,
    "anteil juenger 3" oftype decimal,
    "anzahl 3 6" oftype integer,
    "anteil 3 6" of type decimal,
    "anzahl 6 15" oftype integer,
    "anteil 6 15" of type decimal,
    "anzahl 15 25"
                    oftype integer,
    "anteil 15 25"
                    oftype decimal,
    "anzahl 25 35"
                    oftype integer,
    "anteil 25 35"
                    oftype decimal,
    "anzahl 35 45"
                    oftype integer,
    "anteil 35 45"
                    oftype decimal,
    "anzahl 45 55"
                    oftype integer,
    "anteil 45 55"
                    oftype decimal,
    "anzahl 55 65"
                    oftype
                           integer,
    "anteil 55 65"
                    oftype decimal,
    "anzahl 65 75"
                    oftype integer,
    "anteil_65_75"
                    oftype decimal,
    "anzahl 75 aelter" oftype integer,
    "anteil 75 aelter" oftype decimal
];
```

Code 6.5: Erkannte Spaltendatentypen

Die Analyse der XLSX-Datei konzentrierte sich auf die Überprüfung der Datentypen. Beispielsweise wird die Spalte "stadtbereich_code", die verschiedene Buchstaben enthält, als "text" gespeichert. Ebenso wird die Spalte "stadtbereich_bezeichnung", die unterschiedliche Stadtbereiche in Textform auflistet, ebenfalls als "text" gespeichert. Die Spalte "anzahl_juenger_3", die ausschließlich ganze Zahlen beinhaltet, wird als "integer" erkannt. "anteil_55_65", mit einer Mischung aus ganzen und rationalen Zahlen, werden als "decimal" gespeichert.

Die Spaltendatentypen wirken sich auf die Anzahl der importierten Zeilen aus. Dies liegt daran, dass die CSV- und XLSX-Datenbanken oft inkonsistente Werte in den Spalten aufweisen, und Jayvee importiert nur jene Zeilen in die Zieldatenbank, die den in dem Jayvee-Modell aufgelisteten Datentypen entsprechen. Zum Beispiel:

Alter:

6

2

Peter

In diesem Fall, wenn in der Spalte "Alter" in Jayvee der Datentyp "integer" festgelegt ist, wird der Wert "Peter" nicht importiert.

Da die Daten in der 2. Generation der Scaffolding-Ansatzes durch das Ausfiltern der nicht übereinstimmenden Spaltendatentypen konsistenter sind, wird es empfohlen. Es sollte jedoch berücksichtigt werden, dass Spalten gemischte Werte enthalten könnten, zum Beispiel:

2,6

2,7

2

In solchen Fällen wird der Spaltendatentyp als "decimal" gespeichert, weil die Mehrheit "decimal" ist, und der Wert "2" übersprungen wird.

Aus Sicht der Datenqualität ist daher die 2. Generation vorzuziehen. Wenn jedoch das Ziel ist, alle verfügbaren Daten zu importieren, bietet die 1. Generation Vorteile.

6.5 Anforderungen an das benutzerdefinierte Chat-GPT

Wie bereits in Abschnitt 5.5 erwähnt, produziert der ChatGPT-Ansatz validen Code, der jedoch aufgrund der fehlenden API zum Herunterladen von CSV- und XLSX-Ursprungsdatenbanken über einen Link nicht ausführbar ist. Wenn jedoch ein Nutzer explizit einen Link bereitstellt, der in die Anfrage eingebettet ist, fügt der Ansatz diesen Link zum erzeugten Jayvee-Modell hinzu. Somit ist die Anforderung "Der Ansatz muss in der Lage sein, validen Jayvee-Code zu generieren" (FA C1) erfüllt.

Die Anforderung "Der Ansatz soll Eingaben in Form von CSV- und XLSX-Datenbanken akzeptieren" (FA_C2) ist leider nicht erfüllt, da es Bugs in ChatGPT selbst gibt. Zudem nimmt die Eingabe von Dateien einen erheblichen Teil der Kontextlänge in Anspruch, was leider zu Funktionsstörungen führt. Die Kontextlänge ist die Menge an Text, die von einem LLM verarbeitet werden kann. Bei ChatGPT ist diese zum Zeitpunkt des Verfassens dieser Arbeit begrenzt. Dies könnte durch die Implementierung einer API zum Herunterladen von Datenban-

ken gelöst werden.

Die Qualitätsanforderung "Der Ansatz soll einen freundlichen Ton wahren" (QA_-C1) ist erfüllt, weil das ChatGPT-Programm so konfiguriert wird, dass es freundlich ist. Es wird nicht unfreundlich sein, solange es nicht explizit dazu aufgefordert wird.

Es wurden Tests durchgeführt, um herauszufinden, wie effektiv verschiedene Trainingsmethoden für die benutzerdefinierte Programmierung von ChatGPT sind:

Zunächst wurde die Zero-Shot-Methode angewandt, bei der ChatGPT direkt befragt wurde, um Jayvee-Code für eine Datenbank zu generieren. Die Ergebnisse waren jedoch nicht zufriedenstellend, da Python-Code statt Jayvee-Code erzeugt wurde. Jayvee ist eine relativ neue domänenspezifische Sprache (DSL), für deren Codegenerierung ChatGPT nicht speziell trainiert ist.

Anschließend wurde die One-Shot-Methode getestet, die relativ gute Ergebnisse lieferte. Es ist jedoch anzunehmen, dass der Few-Shot-Ansatz potenziell bessere Ergebnisse erzielen könnte.

In dieser Arbeit wurde die Few-Shot-Methode verwendet, da OpenAI die Möglichkeit bietet, durch ein Gespräch ein benutzerdefiniertes Modell mit spezifischem Vorwissen für die Codegenerierung zu erstellen.

6.6 Vergleich der Ansätze

Die Ansätze werden hinsichtlich ihrer Geschwindigkeit und Erweiterbarkeit verglichen.

6.6.1 Vergleich der Geschwindigkeit zwischen Ansätzen

Zunächst wird die Geschwindigkeit der Codegenerierung zwischen den Scaffoldingund RL-Ansätzen verglichen. Das benutzerdefinierte ChatGPT wird mit anderen Ansätzen nicht verglichen, da das erzeugte Modell nicht immer korrekt ist und manuelle Anpassungen erforderlich sind.

Tabelle 6.19 veranschaulicht, dass der Scaffolding-Ansatz wesentlich schneller Code generiert. Dies liegt daran, dass beim RL-Ansatz aufgrund der "Epsilongreedy"-Strategie immer unterschiedliche Antworten generiert werden. Es kann vorkommen, dass das Modell irrtümlich lernt, dieselbe Aktion mehrmals zu wiederholen, wenn es aus irgendeinem Grund falsch trainiert wird. Durch die Nutzung der "Epsilon-greedy"-Strategie wird gelegentlich eine andere Aktion gewählt, um die Exploration zu ermöglichen. Daher macht ein direkter Zeitvergleich wenig Sinn. Man kann auch in Zukunft versuchen, das RL-Modell mit größeren

Datenbanken zu trainieren und zu testen, wie sich das auf die Geschwindigkeit auswirkt.

Ansatz	Anzahl der erzeugten SQLite-Zieldatenbanken	Verbrauchte Zeit in Sekunden
Scaffolding	$6~\mathrm{CSV} + 6~\mathrm{XLSX}$	12,029
RL	$6~\mathrm{CSV} + 6~\mathrm{XLSX}$	88,903

Tabelle 6.19: Vergleich der Geschwindigkeit der Ansätze zur Codegenerierung Scaffolding und RL

6.6.2 Vergleich der Erweiterbarkeit

Im Vergleich zum Scaffolding-Ansatz erfordert der RL-Ansatz mehr Zeit für Erweiterungen. Beim Scaffolding-Ansatz sind zwei Schritte notwendig (siehe Abschnitt 5.3), um neue Funktionen hinzuzufügen:

- 1. Erweiterung der Codegenerierungslogik.
- 2. Anpassung der Logik für Ursprungsdatenbanken.

Während beim RL-Ansatz fünf Schritte erforderlich sind (siehe Abschnitt 5.4):

- 1. Neue Aktionen zur bestehenden Aktionsliste hinzufügen.
- 2. Diese neuen Aktionen in der Umgebung definieren.
- 3. Das Evaluationsintervall anpassen.
- 4. Die Liste der genutzten Aktionen aktualisieren.
- 5. Das Modell neu trainieren.

Beim Scaffolding-Ansatz ist anfänglich viel Aufmerksamkeit erforderlich, da für jede Zieldatenbank ein spezifischer Loader entwickelt werden muss. Soll eine neue Funktionalität für eine bestimmte Zieldatenbank hinzugefügt werden, muss diese Funktionalität entsprechend für jeden Ursprungsdatenbanktyp implementiert werden. Im Vergleich dazu erfordert der RL-Ansatz lediglich das Hinzufügen der neuen Funktionalität zur Liste der neuen und bereits genutzten Aktionen. Während der Entwicklung zeigte sich, dass es fehleranfälliger ist, neue Funktionen beim RL-Ansatz zu implementieren; daher ist der Scaffolding-Ansatz wesentlich einfacher und zuverlässiger zu erweitern.

Das benutzerdefinierte ChatGPT kann durch das Hochladen von Beispielen als Vorwissen erweitert werden, wodurch auch dieser Ansatz nicht zeitaufwendig zu erweitern ist.

7 Fazit

Zusammenfassend bietet diese Arbeit nun drei methodische Ansätze zur Generierung vielfältiger Datenbanken, wobei jeweils spezifische Stärken und Schwächen identifiziert wurden, die für zukünftige Verbesserungen und Anpassungen richtungsweisend sein können.

Das vorrangige Ziel dieser Masterarbeit war die Entwicklung der Ansätze zur Generierung von Startercode für Datenpipelines. Dazu wurden zunächst die erforderlichen Anforderungen definiert und implementiert. Um diese Ziele zu erreichen, wurde eine umfangreiche Literaturrecherche im Bereich der Codegenerierung durchgeführt. Basierend auf Kriterien wie Generierungsgeschwindigkeit, Erweiterbarkeit und Relevanz wurden drei Software-Ansätze ausgewählt und entwickelt.

Die Architektur des entwickelten Ansätze wurde in zwei Hauptmodule unterteilt: den Evaluator und den Generator. Der Evaluator wählt abhängig vom gewählten Ansatz einen Generator aus, der dann den Code basierend auf den vom Evaluator übermittelten Parametern generiert. Ein besonderes Merkmal des Reinforcement-Learning-Ansatzes ist ein zusätzliches Modul, das für das Training der RL-Modelle verantwortlich ist. Das trainierte Modell wird später vom Generator zur Codegenerierung verwendet. In der Implementierungsphase wurde die entworfene Architektur realisiert und anschließend wurden die verschiedenen Ansätze miteinander verglichen und evaluiert, um ihre Leistungsfähigkeit und mögliche Limitationen festzustellen. In dieser Arbeit wurde bewiesen, dass der Scaffolding-Ansatz besonders effektiv für die Aufgabe der automatischen Generierung von Jayvee-Datenpipelines ist, insbesondere weil er kein Training erfordert und dennoch eine angemessene Geschwindigkeit bei der Codegenerierung bietet. Außerdem lässt er sich leichter erweitern.

In dieser Arbeit wurde ebenfalls festgestellt, dass die 1. Generation der beiden Ansätze besser für die Importierung aller Zeilen geeignet ist, weil alle Datentypen als 'Text' gespeichert werden. Die 2. Generation der beiden Ansätze eignet sich besser, wenn die Daten konsistenter gemacht werden müssen, da sie die Datentypen der Spalten erkennt.

In der 2. Generation des Reinforcement-Learning-Ansatzes wurde beobachtet, dass die Trainingsgeschwindigkeit erheblich zunimmt, wenn das Training auf mehrere Kerne verteilt wird. Hierbei kamen moderne Algorithmen aus den Bibliotheken "Stable Baselines3" und "Gymnasium" zum Einsatz. Weiterhin wurde festgestellt, dass die Codegenerierungsgeschwindigkeit steigt, wenn das RL-Modell mit größeren Datensätzen trainiert wird.

Bei beiden Ansätzen wurden jedoch auch Limitationen erkannt, insbesondere bei der Verarbeitung von unstrukturierte XLSX-Datenbanken mit Präambel. Obwohl in solchen Fällen Jayvee-Code generiert wird, führt dieser nicht immer zu einer erfolgreichen Zieldatenbank. Mit Hilfe der Tools "TextLineDeleter" und "RowDeleter" lässt sich das Jayvee-Modell jedoch manuell anpassen und funktionsfähig machen.

ChatGPT erwies sich auch als nützlich für die Generierung einzelner Jayvee-Modelle. Allerdings erfordert die Nutzung mehr Aufmerksamkeit von den Benutzern, da ChatGPT keine URLs erkennen, keine Dateien verarbeiten kann und gelegentlich Fehler produziert.

Diese Arbeit trägt wesentlich zur Erleichterung der automatischen Generierung von Datenpipelines bei und erfüllt die gestellten Anforderungen, wenn auch mit einigen Limitationen.

Ausblick

Aufgrund zeitlicher Beschränkungen konnten nicht alle potenziellen Ideen in dieser Arbeit realisiert werden. Zukünftige Forschungen könnten sich unter anderem auf folgende Aspekte konzentrieren:

- Evaluation des Codes durch Syntaxüberprüfung während des Trainings.
- Weiterentwicklung der Techniken zur Erkennung von Spaltennamen. Ein Beispiel hierfür ist "Sherlock", ein Tool, das von Wissenschaftlern des MIT entwickelt wurde und maschinelles Lernen verwendet, um Spaltennamen zu interpolieren. Aufgrund des thematischen Fokus dieser Masterarbeit wird jedoch eine einfachere Lösung gewählt und die fortgeschrittene Erkennung von Spaltennamen nicht weiter verfolgt.
- Die Verteilung des Trainings auf verschiedene Kerne war sehr erfolgreich, beschränkte sich jedoch auf einen einzigen PC. Um den Trainingsprozess des RL-Ansatzes weiter zu beschleunigen, könnte man die Implementierung einer Unterstützung für Hochleistungsrechenzentren (HPCs) in Betracht ziehen. Derzeit bietet Stable Baselines3 keine Möglichkeit, Ressourcen zwischen Nodes zu verteilen; eine solche Erweiterung wäre eine sinnvolle Innovation.

Diese Erweiterung wurde in der aktuellen Arbeit nicht vorgenommen, da sie den festgelegten Umfang dieser Forschung überschreitet. Die Beschränkung auf den aktuellen Rahmen gewährleistete eine fokussierte Analyse und sicherte die Handhabbarkeit der Studie unter den gegebenen Ressourcen und zeitlichen Begrenzungen.

Anhänge

A Dialoge mit ChatGPT

A.1 Dialog 1

ChatGPT:Hi! I'll help you build a new GPT. You can say something like, "make a creative who helps generate visuals for new productsör "make a software engineer who helps format my code."

What would you like to make?

Ich: I want to create a code generator, that takes a csv database as input, and generates code in a dsl that has been newly written, i can give you some examples

ChatGPT:Great! You're looking to create a GPT that specializes in converting CSV databases into code for a newly developed DSL, and you're willing to provide examples to guide the process.

How about we call this GPT Code Crafter? Does that name work for you?

Ich: The DSLs name is "Jayvee". please put it in the name



ChatGPT:Here's the initial profile picture for Jayvee Code Generator, inspired by a futuristic theme of data transformation. Do you like it, or would you like any changes?

Ich: Yes, I like it

ChatGPT:Fantastic! Now, let's refine how Jayvee Code Generator will work.

First off, how should it handle the CSV data? For example, should it ask for specific details about the CSV structure or make assumptions based on common patterns?

Ich: It should make assumptions based on common patterns where possible, if an assuption cant be made, then it should a question, if there is no answer then take a guess

ChatGPT:Next, let's talk about the DSL, Jayvee. Could you describe its primary features or the key syntax elements that Jayvee Code Generator should prioritize in its code generation? This will help ensure it focuses on the most important aspects of the DSL when translating from CSV.

Hier habe ich Jayvee cars Beispiel¹ einkopiert

ChatGPT:Now that we have a clear understanding of Jayvee's DSL structure and its emphasis on data transformation pipelines, blocks, pipes, and properties, Jayvee Code Generator is ready to help you transform CSV databases into Jayvee code.

Would you like to give it a try in the preview, or is there anything else you'd like to adjust?

Ich habe versucht, eine CSV-Datei einzulesen, jedoch ohne Erfolg.

ChatGPT:It seems like there might have been a mistake, as you haven't provided any input or question. How can I assist you today?

A.2 Dialog 2

ChatGPT:Hi! I'll help you build a new GPT. You can say something like, "make a creative who helps generate visuals for new productsör "make a software engineer who helps format my code."

What would you like to make?

Ich: Ich benötige einen Generator, der Code für meine domänenspezifische Sprache (DSL) aus der Struktur einer Datenbank generiert.

ChatGPT:Ich schlage vor, wir nennen diesen GPT "DSL Code Wizard". Wie klingt das für dich?

lch: Bitte füge den Namen Jayvee zusäzlich hinzu

ChatGPT:Ich habe den Namen zu Jayvee DSL Code Wizard aktualisiert. Lass uns nun ein Profilbild für ihn erstellen.

Hier ist das Profilbild für den Jayvee DSL Code Wizard. Gefällt es dir oder möchtest du Änderungen vornehmen?

Hier habe ich ein altes Bild hochgeladen weil es mir besser gefallen hat.

¹https://jvalue.github.io/jayvee/docs/user/examples/cars/

Ich: Bitte benenne es "Jayvee Code Generator"

ChatGPT:Ich habe den Namen zu Jayvee Code Generator aktualisiert Jetzt, da wir den Namen festgelegt haben, lass uns die Details seines Verhaltens und seiner Fähigkeiten weiter verfeinern.

Was sollte Jayvee betonen oder vermeiden?

Ich: Falls Wissen fehlt, sollte zunächst nachgefragt werden

ChatGPT:Perfekt, Jayvee wird nun immer nachfragen, falls notwendige Informationen fehlen, bevor er fortfährt. Wie möchtest du, dass Jayvee mit dir spricht? Soll er eher formell oder informell sein, oder gibt es einen bestimmten Ton oder Stil, den du bevorzugst?

Ich: Formell

ChatGPT: Jayvee wird jetzt einen formellen Ton in der Kommunikation verwenden, um Professionalität und Klarheit zu gewährleisten. Fühle dich frei, Jayvee in der Vorschau auszuprobieren, die sich rechts im Dialog befindet. Wenn du irgendwelche Verfeinerungen oder zusätzliche Anpassungen hast, lass es mich wissen. Wie kann ich sonst noch helfen?

Nach dem Hochladen der zwei PDFs mit Vorwissen habe ich versucht, Jayvee-Code zu generieren. Dabei fügte ChatGPT einen Block "DataAnalysis"hinzu, der nicht zu Jayvee gehört. Außerdem wurde der "TextFileInterpreterßunächst vergessen, jedoch nach einer Rückfrage hinzugefügt. Darüber hinaus begann ChatGPT sofort mit der Generierung von Code für Postgres. Ich habe daraufhin angewiesen, immer nachzufragen, für welche Datenbank der Code geschrieben werden soll.

lch: Frage stets nach, ob der generierte Code für SQLite oder PostgreSQL bestimmt sein soll

ChatGPT: Jayvee wird nun immer nachfragen, ob der generierte Code für SQLite oder PostgreSQL bestimmt sein soll, um sicherzustellen, dass der Code optimal auf die Ziel-Datenbank abgestimmt ist. Gibt es noch weitere Anpassungen, die du vornehmen möchtest, oder weitere Details, die wir hinzufügen sollen?

B Genutzte Bibliotheken

- 1. Stable Baselines3
- 2. NumPy
- 3. Gymnasium
- 4. Pandas
- 5. sqlite3
- 6. axios
- 7. Matlib
- 8. openpyxl
- 9. MessyTables
- 10. DuckDB
- 11. PostgreSQL
- 12. keras

Literaturverzeichnis

- Achiam, J. (2018a). OpenAI Spinning Up. Abgerufen am 24. April 2024 von https://spinningup.openai.com/en/latest/index.html
- Achiam, J. (2018b). Part 2: Kinds of RL Algorithms. Abgerufen am 24. April 2024 von https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html
- Baeldung. (2023). Epsilon-Greedy Q-learning. Abgerufen am 24. April 2024 von https://www.baeldung.com/cs/epsilon-greedy-q-learning
- Bass, L. (2003). Software Architecture in Practice.
- Decker, J. (2023). Deep-Q-Networks: Eine Revolution im Reinforcement Learning. Abgerufen am 24. April 2024 von https://atlane.de/deep-q-networks-eine-revolution-im-reinforcement-learning/#Q-Wert-Update
- Dehaerne, E., Dey, B., Halder, S., De Gendt, S., & Meert, W. (2022). Code Generation Using Machine Learning: A Systematic Review. *IEEE Access*, 10, 82434–82455. https://doi.org/10.1109/ACCESS.2022.3196347
- Doshi, K. (2020). Reinforcement Learning Explained Visually (Part 5): Deep Q Networks, step-by-step. Abgerufen am 24. April 2024 von https://towardsdatascience.com/reinforcement-learning-explained-visually-part-5-deep-q-networks-step-by-step-5a5317197f4b
- Ganapathi, M., Fischer, C. N., & Hennessy, J. L. (1982). Retargetable Compiler Code Generation. ACM Comput. Surv., 14(4), 573–592. https://doi.org/10.1145/356893.356897
- Group, T. P. G. D. (2018). *Documentation PostgreSQL 10.3* (T. P. G. D. Group, Hrsg.).
- Hashemi-Pour, C. (2023). Reinforcement learning. Abgerufen am 24. April 2024 von https://www.techtarget.com/searchenterpriseai/definition/reinforcement-learning#:~:text=In%20general%2C%20a%20reinforcement%20learning, to%20train%20machine%20learning%20systems.
- Herrington, J. (2003). Code Generation in Action. Manning Publications Co.
- Hipp, R. D. (2020). SQLite. Abgerufen am 30. April 2023 von https://www.sqlite.org/index.html
- Hulsebos, M., Hu, K., Bakker, M., Zgraggen, E., Satyanarayan, A., Kraska, T., Demiralp, Ç., & Hidalgo, C. (2019). Sherlock: A Deep Learning Approach

- to Semantic Data Type Detection. Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery $\mathcal{E}\#38$; Data Mining.
- Jain, R., Ni, W., & Sunshine, J. (2023). Generating Domain-Specific Programs for Diagram Authoring with Large Language Models. Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, 70– 71. https://doi.org/10.1145/3618305.3623612
- JValue Contributors. (2024). *Introduction to Jayvee*. Abgerufen am 24. April 2024 von https://jvalue.github.io/jayvee/docs/user/intro/
- Laine, C. (2019). Using Scaffolding Tools to Generate Boilerplate Code. Abgerufen am 24. April 2024 von https://medium.com/it-dead-inside/using-scaffolding-tools-to-generate-boilerplate-code-d35c2705ac6e
- Le, H., Wang, Y., Gotmare, A. D., Savarese, S., & Hoi, S. C. H. (2022). CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho & A. Oh (Hrsg.), Advances in Neural Information Processing Systems (S. 21314–21328, Bd. 35). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2022/file/8636419dea1aa9fbd25fc4248e702da4-Paper-Conference.pdf
- Lemke, C. (2023). Reinforcement Learning: Kompakt erklärt. Abgerufen am 24. April 2024 von https://www.alexanderthamm.com/de/blog/einfacherklaert-so-funktioniert-reinforcement-learning/
- Luu, Q. T. (2024). Q-Learning vs. Deep Q-Learning vs. Deep Q-Network. Abgerufen am 6. Mai 2024 von https://www.baeldung.com/cs/q-learning-vs-deep-q-learning-vs-deep-q-network
- Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning.
- Mülder, A. (2023). Large Language Models for Domain-Specific Language Generation: How to Train Your Dragon.
- Pohl, K. (2011). Basiswissen Requirements Engineering. dpunkt Verlag.
- Professorship for Open-Source Software. (o. D). Structure and content. Abgerufen am 26. April 2024 von https://oss.cs.fau.de/theses/structure-content/
- Rahman, M. (2018). Exploring Automated Code Evaluation Systems and Resources for Code Analysis: A Comprehensive Survey.
- Silver, D. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.
- Sun, Z., Zhu, Q., Xiong, Y., Sun, Y., Mou, L., & Zhang, L. (2020). TreeGen: A Tree-Based Transformer Architecture for Code Generation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34 (05), 8984–8991. https://doi.org/10.1609/aaai.v34i05.6430
- Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction. A Bradford Book.

- The Open Knowledge Foundation Ltd. (2013). messytables: all your rows are belong to us. Abgerufen am 26. April 2024 von https://messytables.readthedocs.io/en/latest/
- Watkins, C. J. C. H., & Dayan, P. (1992). Technical Note Q-Learning. Mach. Learn., 8, 279–292. http://dblp.uni-trier.de/db/journals/ml/ml8.html# WatkinsD92
- Yang, J., He, Y., & Chaudhuri, S. (2021). AutoPipeline: Synthesize Data Pipelines By-Target Using Reinforcement Learning and Search. CoRR, abs/2106.13861. https://arxiv.org/abs/2106.13861
- Yeoman. (2023). Yeoman. Abgerufen am 24. April 2024 von yeoman.io
- Zowghi, D., & Gervasi, V. (2002). The Three Cs of Requirements : Consistency , Completeness , and Correctness. https://api.semanticscholar.org/CorpusID:18907695