Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Eignung von Large Language Models (LLMs) zur Generierung von Python Code zur Datenanalyse

Bachelorarbeit

Name des Studiengangs
Wirtschaftsinformatik

Fachbereich 4

vorgelegt von
Maurice Krüger

Datum:
Berlin, 23. 01. 2025

Erstgutachter: Prof. Dr.-Ing. Ingo Claßen
Zweitgutachter: Prof. Dr. Axel Hochstein

Abstract

Diese Bachelorarbeit untersucht die Eignung moderner Large Language Models (LLMs) für die automatisierte Generierung von Python-Code im Kontext typischer Datenanalyseaufgaben. Dabei wird zunächst ein Überblick über die theoretischen Grundlagen von LLMs und der Programmiersprache Python gegeben. Anschließend werden in einer empirischen Untersuchung Codebeispiele durch den Marktführer ChatGPT mit dem Sprachmodell GPTo1 erzeugt und mit manuell geschriebenen Skripten verglichen. Die Bewertung erfolgt anhand mehrerer Kriterien wie Korrektheit, Performanz sowie Verständlichkeit des generierten Codes. Abschließend werden Empfehlungen für den praktischen Einsatz von LLMs in der Datenanalyse abgeleitet sowie ein Ausblick auf zukünftige Entwicklungen gegeben.

Inhaltsverzeichnis

1 Einleitung **3**
1.1 Problemstellung und Forschungsfragen 3
1.2 Relevanz der Thematik 4
1.3 Zielsetzung 4
1.4 Aufbau der Arbeit 4

2 Grundlagen **4**
2.1 Einführung Large Language Models 5
2.2 Einführung Python 6
2.2.1 Bedeutung und Bibliotheken 6
2.2.2 Typische Schritte einer Datenanalyse 6
2.3 Einführung automatisierte Code-Generierung 6
2.3.1 Funktionsweise und Vorteile 6
2.3.2 Herausforderungen und Grenzen 7

3 LLMs in der Programmierung – aktueller Stand **7**
3.1 Überblick und Vergleich von verschiedenen LLMs 7
3.2 Einsatzgebiete von LLMs in der Programmierung 9
3.3 Vergangene Studien und Arbeiten zur Code-Generierung 9

4 Methodik **11**
4.1 Vorgehensweise der Untersuchung 11
4.2 Testfälle der Datenanalyse 12
4.2.1 Testfall 1 12
4.2.2 Testfall 2 12
4.2.3 Testfall 3 12
4.2.4 Testfall 4 12
4.2.5 Testfall 5 12
4.3 Auswertungskriterien 13
4.4 Verwendete Tools 13

5 Auswertung der Python-Code-Generiung zur Datenanalyse durch LLMs **13**

6 Fazit und Ausblick **13**

7 Literaturverzeichnis **14**

1 Einleitung

1.1 Problemstellung und Forschungsfragen

Die schnelle Entwicklung von Large Language Models (LLMs), wie zum Beispiel ChatGPT, hat in den letzten Jahren sowohl im privaten als auch im beruflichen Bereich viel Aufmerksamkeit erregt. Ursprünglich wurden LLMs hauptsächlich zur Lösung alltäglicher Probleme und der Verarbeitung und Erzeugung menschlicher Sprache eingesetzt, doch zunehmend zeigt sich, dass sie auch Programmiercode in verschiedenen Sprachen erstellen können. Besonders in der Programmiersprache Python – einer weit verbreiteten Sprache für Datenanalyse und Machine Learning – sind die Fortschritte in der automatisierten Code-Generierung durch LLMs bereits bemerkenswert[1, 2].

Aktuelle Forschungsarbeiten konzentrieren sich auf die systematische Bewertung von solch generierten Codes, um Fehlerquellen und Qualitätsmerkmale zu bemessen. Die Bereitstellung öffentlicher Evaluierungsdatensätze und -frameworks, wie etwa *HumanEval*[2] oder *EvalPlus*[3], ermöglicht standardisierte Vergleichsstudien verschiedener LLMs. Dies eröffnet neue Anwendungsfelder im Bereich der Datenanalyse: Anstatt den Code manuell zu schreiben, könnten Nutzer in Zukunft lediglich ihre Anforderungen in natürlicher Sprache formulieren und das Modell würde diese für den Nutzer umsetzen[4].

Vor diesem Hintergrund stellt sich die Frage, ob und inwiefern LLMs tatsächlich qualitativ hochwertigen Python-Code für datenanalytische Aufgaben erzeugen können und wie dieser Code im Vergleich zu manuell geschriebenen Code abschneidet. Auch die möglichen Grenzen dieser automatisierten Generierung, wie etwa in Bezug auf Performanz, Wartbarkeit oder Fehlerraten, sind hierbei von großer Bedeutung[5].

Daraus ergibt sich die zentrale **Hauptforschungsfrage**:

> *Inwieweit eignen sich Large Language Models (LLMs) zur Durchführung gängiger Datenanalyseaufgaben in Python, und wie schneidet dieser Code im Vergleich zu manuell geschriebenen Code hinsichtlich Effizienz, Korrektheit und Wartbarkeit ab?*

Zur weiteren Strukturierung dieser Hauptfrage werden mehrere Unterfragen hinzugezogen:

- **Qualität & Korrektheit:** Wie qualitativ hochwertig ist dieser generierte Code hinsichtlich Syntax und Implementierung von Analyseaufgaben (z. B. Datenfilterung, Modellierung)?
- **Effizienz & Performanz:** Inwieweit entspricht der automatisch erzeugte Code modernen Standards bezüglich Laufzeit und Ressourcenverbrauch?
- **Wartbarkeit & Verständlichkeit:** Wie gut lässt sich der generierte Code verstehen, dokumentieren und erweitern?
- **Einsatzgebiete & Grenzen:** Für welche spezifischen Aufgaben in der Datenanalyse ist der Einsatz von LLMs sinnvoll und wo stoßen diese an ihre Grenzen?

1.2 Relevanz der Thematik

Die Fähigkeit, Programmiercode mit Hilfe von LLMs zu erstellen, könnte die Entwicklungsprozesse erheblich beschleunigen und neue Nutzergruppen anziehen, die bisher wenig Erfahrung mit Programmierung hatten. Besonders in der Datenanalyse können viele Arbeitsschritte – vor allem wiederkehrende Aufgaben, wie das Erstellen von Standard-Pipelines zur Datenbeschaffung- und bereinigung – automatisiert werden. Gleichzeitig gibt es jedoch Herausforderungen in Bezug auf Performanz, Wartbarkeit und Transparenz.

1.3 Zielsetzung

Das Ziel dieser Arbeit ist es, herauszufinden, wie gut moderne LLMs (Large Language Models) für die automatische Code-Generierung in der Datenanalyse mit Python geeignet sind. Dafür wird in einem Experiment Code von einem LLM generiert und mit manuell geschriebenem Code verglichen. Der Vergleich basiert auf Kriterien wie Korrektheit, Performance und Wartbarkeit. Auf Basis der Ergebnisse werden Empfehlungen für den Einsatz von LLMs in der Praxis gegeben und deren Grenzen diskutiert. Zum Schluss wird ein Ausblick darauf gegeben, wie sich diese Technologie in Zukunft weiterentwickeln könnte und welche Auswirkungen das auf Aufgaben in der Datenanalyse haben könnte[2, 3].

1.4 Aufbau der Arbeit

Nach dieser Einleitung (Kapitel 1) folgt in Kapitel 2 eine Darstellung der **Grundlagen**, dazu gehört eine Einführung in Large Language Models, die Programmiersprache Python und die automatisierte Code-Generierung. Kapitel 3 gibt einen Überblick über den aktuellen Stand der Forschung, in dem verschiedene LLM-Modelle, Publikationen und Evaluationstechniken vorgestellt werden. Darauf aufbauend wird in Kapitel 4 die **Methodik** der Arbeit erläutert. Kapitel 5 enthält dann die **Auswertung** der gewonnenen Daten sowie den Vergleich von durch ein LLM generierten und manuell geschriebenen Code. Kapitel 6 fasst die Ergebnisse zusammen, beantwortet die Forschungsfragen und gibt einen **Ausblick** auf weitere Entwicklungen. Schließlich enthält Kapitel 7 den **Anhang**, einschließlich Literaturverzeichnis und relevanter Dokumentationen.

2 Grundlagen

Im folgenden Kapitel werden die theoretischen und technischen Grundlagen vorgestellt, die für das Verständnis dieser Arbeit notwendig sind. Abschnitt 2.1 beschäftigt sich mit den Large Language Models, ihrer Funktionsweise und ihrer Bedeutung in der Code-Generierung. Danach wird in Abschnitt 2.2 das Potenzial der Programmiersprache Python für die Datenanalyse erläutert, bevor Abschnitt 2.3 das Konzept der automatisierten Code-Generierung behandelt.

2.1 Einführung Large Language Models

Bei großen Sprachmodellen (LLMs) handelt es sich um KI-Systeme, die mithilfe moderner Deep-Learning-Methoden entwickelt wurden, um natürliche Sprache zu verstehen, selbst Texte zu generieren und mit Kontext zu interpretieren[6]. Diese Modelle basieren auf Transformer-Architekturen, welche einen Self-Attention-Mechanismus nutzt. Dieser Mechanismus ermöglicht es dem Modell Beziehungen zwischen verschiedenen Wörtern oder Tokens in einer Eingabe zu erkennen, wobei es nicht von Bedeutung ist, an welcher Position diese stehen. Die Transformer-Architektur unterscheidet sich desweiteren von früheren Architekturen, wie zum Beispiel RNNs (Recurrent Neuronal Networks), dadurch, dass sie auf die rekursive Verarbeitung der Tokens verzichtet und stattdessen alle Tokens parallel verarbeitet, wodurch die LLMs deutlich effizienter auf Eingaben reagieren können. Diese Leistung der LLMs korreliert jedoch stark mit der größe der Modelle und der Menge an Trainingsdaten, was dazu führt, dass besonders gute Modelle einen deutlich höheren Ressourcenbedarf und eine deutlich größere Menge an Trainingsdaten benötigen[7].

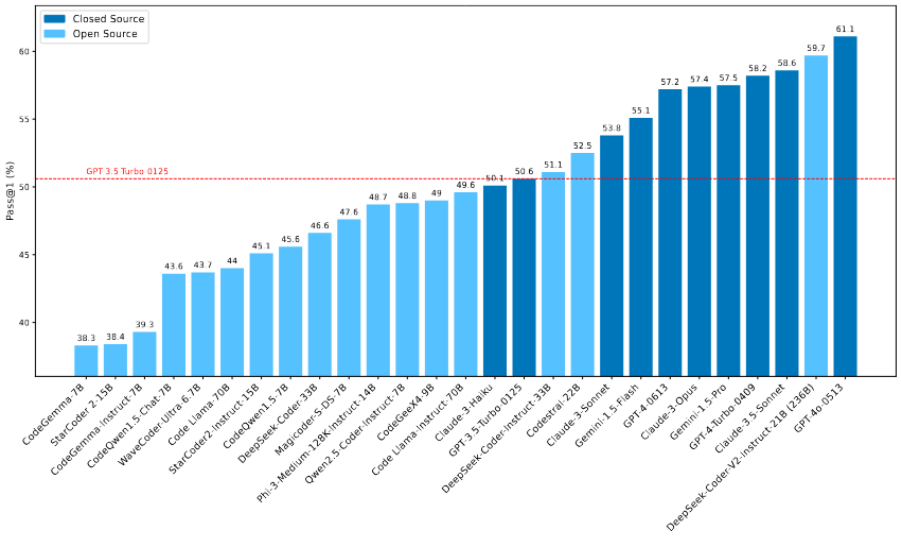

Abbildung 1: Leistungsvergleich verschiedener Modellgrößen (nach Jiang et al. (2024), basierend auf 'A Survey on Large Language Models for Code Generation').

In den letzten Jahren wurden mehrere Benchmarks und Evaluierungsdatensätze speziell für die Code-Generierung entwickelt. Beispiele dafür sind *HumanEval*[2] und *EvalPlus*[3], die genutzt werden, um die Genauigkeit und Zuverlässigkeit von LLMs in verschiedenen Programmiersprachen zu testen. Erste Studien zeigen, dass LLMs einfache bis mittelschwere Aufgaben oft vollständig lösen können. Bei komplexeren oder sehr speziellen Aufgabenbereichen stoßen sie aber noch an ihre Grenzen[1]. Obwohl LLMs in vielen Sprachen Code generieren können, hat sich Python als einer der Hauptfoki herauskristallisiert. Dies liegt an der weit verbreiteten Nutzung von Python in Wissenschaft und Industrie, insbesondere in den Bereichen Datenanalyse und Machine Learning. Die umfangreichen Bibliotheken wie NumPy, pandas und scikit-learn sind Teil der Trainingskorpora, wodurch LLMs häufig in der

Lage sind, Standardroutinen oder Bibliotheksfunktionen korrekt anzuwenden[2].

2.2 Einführung Python

TODO: ausführlicher

2.2.1 Bedeutung und Bibliotheken

Python ist dank seiner Syntax, aktiven Community und vielen hilfreichen Bibliotheken eine der am weitesten verbreiteten Sprachen für Datenanalyse. Ein paar der wichtigsten Bibliotheken, die in der Datenanalyse verwendet werden, sind:

- **pandas** – Datenstrukturen und -bearbeitung,
- **NumPy** – numerische Berechnungen,
- **scikit-learn** – Machine-Learning-Algorithmen,
- **Matplotlib** – Visualisierung,

Diese und weitere Bibliotheken ermöglichen eine effiziente Umsetzung datenanalytischer Projekte und werden bereits in LLM-Trainings berücksichtigt, wodurch generierter Code auf bekannte Funktionen zurückgreifen kann[3, 8].

2.2.2 Typische Schritte einer Datenanalyse

Die Grundschritte einer klassischen Datenanalyse in Python enthält folgende Schritte:

1. *Datenimport* (z. B. CSV-Dateien, Datenbanken, APIs),
2. *Datenbereinigung* (fehlende Werte, Duplikate, Datentypen),
3. *Analyse und Visualisierung* (Statistiken, Plots),

Im Rahmen dieser Arbeit wird untersucht, ob LLMs diese Schritte automatisieren können und an welchen Stellen manuell eingegriffen werden muss.

2.3 Einführung automatisierte Code-Generierung

2.3.1 Funktionsweise und Vorteile

Automatisierte Code-Generierung mithilfe von LLMs basiert auf *Prompts*, also Benutzeranfragen in natürlicher Sprache. LLMs haben hierbei die Möglichkeit sich flexibel an den vom Benutzer gegegeben Kontext anzupassen und können die natürliche Sprache in funktionsfähigen Code umwandeln. Ebenso müssen LLMs nicht spezifisch auf eine Aufgabe trainiert werden, aufgrund der großen Trainingsdaten, die ihnen zur Verfügung stehen[2]. Insbesondere für datenanalytische Aufgaben, bei denen standardisierte Skripte (z. B. für das Einlesen und Bereinigen von Daten) immer wieder benötigt werden, kann dies zu einer erheblichen Zeitersparnis führen und ermöglicht die Nutzung von LLMs auch für weniger erfahrene Personen, die nicht über tiefgreifende Programmierkenntnisse verfügen.

2.3.2 Herausforderungen und Grenzen

Trotz beeindruckender Fortschritte stößt die automatisierte Code-Generierung noch häufig an Grenzen[4, 5]:

- **Komplexe Datenstrukturen**: LLMs zeigen teils Schwächen bei Aufgaben mit hochgradiger Komplexität oder spezifischem Wissen, wenn zu wenig Kontext durch den Nutzer gegeben wird[4, 5].
- **Performanz**: Generierter Code ist nicht immer optimal hinsichtlich Laufzeit oder Speicherverbrauch[5].
- **Wartbarkeit**: Kommentare, klare Code-Struktur und Dokumentation fehlen häufig.
- **Fehleranfälligkeit**: Auch Code, der vorerst funktionsfähig erscheint, kann immer noch Bugs oder Sicherheitslücken enthalten.

Wir präsent diese Herausforderungen in datenanalytischen Aufgaben sind, soll in den folgenden Kapiteln untersucht werden. Vor allem durch den Vergleich von generiertem und manuell geschriebenem Code lassen sich die Stärken und Schwächen von LLMs in der Datenanalyse besser einschätzen.

3 LLMs in der Programmierung – aktueller Stand

Die Entwicklung von Large Language Models (LLMs) hat in den letzten Jahren nicht nur die Art und Weise, wie natürliche Sprache verarbeitet und generiert wird, verändert, sondern auch große Fortschritte in der automatisierten Code-Erstellung ermöglicht. Durch die Kombination aus leistungsstarken Modellarchitekturen wie Transformers, großen Mengen an Trainingsdaten und moderner Hardware haben LLMs heute eine große Präsemz in vielen Bereichen der Softwareentwicklung und Datenanalyse. In diesem Kapitel werden die aktuellen Entwicklungen und verfügbaren Modelle vorgestellt. Außerdem wird ein Überblick über ihre Einsatzmöglichkeiten in der Softwareentwicklung und Datenanalyse gegeben. Zum Schluss werden wichtige Studien und Arbeiten zur Code-Generierung betrachtet, darunter etwa die von Chen et al. (2021) vorgestellte Arbeit zu Codex, einem Modell, das speziell für die automatisierte Programmierung entwickelt wurde[2] und die von Liu et al. (2023) veröffentlichte Arbeit zur Evaluation von generiertem Code mithilfe von *EvalPlus*[1].

3.1 Überblick und Vergleich von verschiedenen LLMs

Derzeit existiert eine Vielzahl an LLMs, darunter auch viele, die gezielt zur Code-Generierung entwickelt wurden. Zu den bekanntesten Beispielen zählen ChatGPT (GPTo1 als das modernste Modell), OpenAI Codex, Code Llama[9], StarCoder [10], CodeT5[5] oder CodeGen[4].

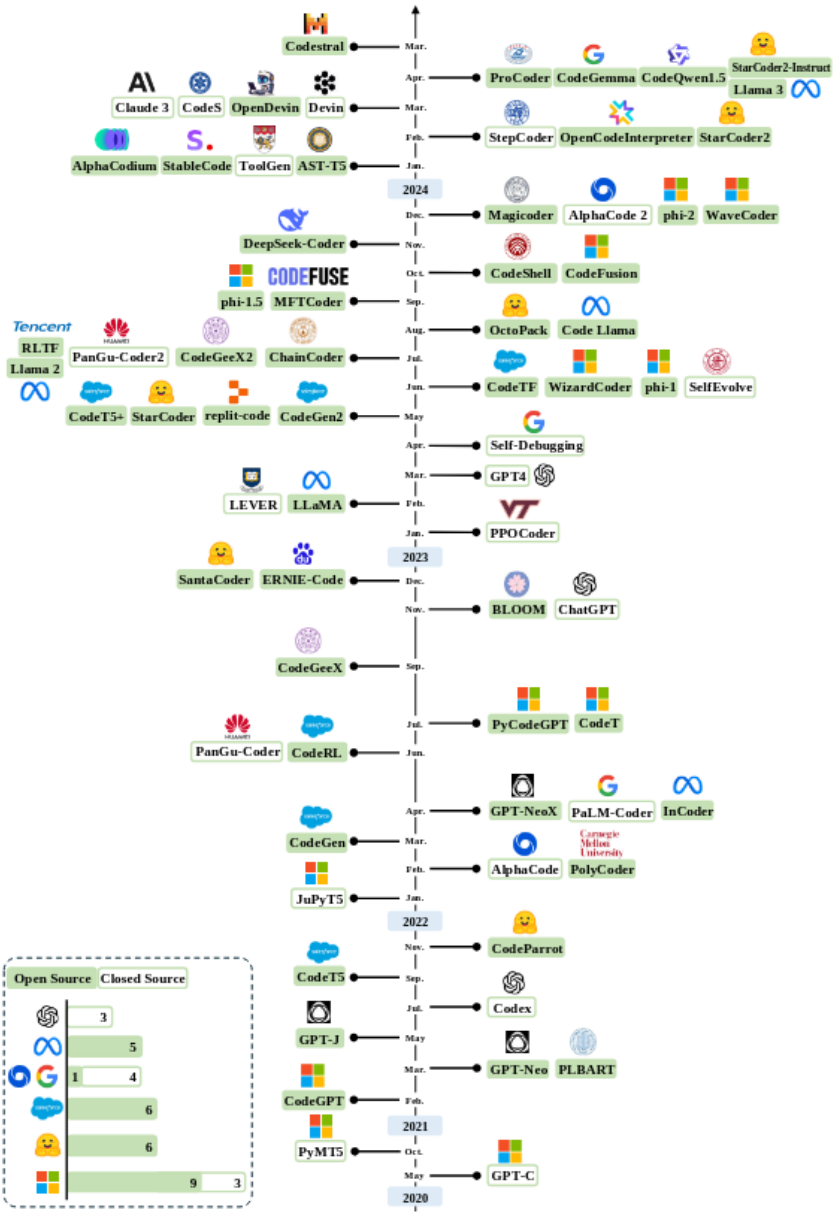

Abbildung 2: Chronologische Übersicht von Large Language Models für die Code Generierung der letzten Jahre (nach Jiang et al. (2024), basierend auf 'A Survey on Large Language Models for Code Generation').

Diese Modelle teilen sich häufig folgende Merkmale:

1. **Transformer-Architektur**: Nahezu alle modernen LLMs beruhen auf dem Transformer-Modell.

2. **Große Parameteranzahl**: Typische LLMs verfügen über eine Vielzahl an Parametern und benötigen entsprechend umfangreiche Trainingsdaten, zu denen in vielen Fällen öffentlich verfügbare Code-Repositories (z. B. GitHub) zählen[2].

3. **Breite Sprachenunterstützung**: Neben Python werden häufig Java, JavaScript und andere Programmiersprachen abgedeckt[2, 7].

Ein Vergleich der LLMs lässt sich anhand verschiedener Kriterien vornehmen:

- **Größe und Trainingsdaten**: Modelle wie GPT-4 oder Code Llama sind mit einer Vielzahl an Code-Datensätzen trainiert und erreichen dadurch in Benchmarks eine hohe Erfolgsquote[1].
- **Lizenz und Offenheit**: Neben proprietären Modellen, wie GitHub Copilot und ChatGPT, existieren mit Code Llama, StarCoder oder CodeGen auch Open Source Alternativen.
- **Spezialisierung**: Einige Modelle sind speziell auf Code-Generierung abgestimmt (z.B. Code Llama, StarCoder), wohingegen andere (z.B. ChatGPT) einen generellen Sprachkontext haben, um auch andere Fragen zu beantworten, der sich jedoch auch auf Code-Aufgaben anwenden lässt.

3.2 Einsatzgebiete von LLMs in der Programmierung

Die zunehmende Leistungsfähigkeit von Large Language Models (LLMs) ermöglicht es, Programmieraufgaben in diversen Bereichen zu automatisieren oder zu beschleunigen. Häufig genannte *Einsatzgebiete* sind dabei:

- **Code-Generierung**: Ermöglicht die Code-Generierung auf Grundlage von Beschreibungen aus natürlicher Sprache[2]. Ebenso bieten manche Modelle die Möglichkeit zu fertigen Funktionen Tests zu generieren, um dessen Funktionalität zu überprüfen.
- **Autovervollständigung**: Integriert in Entwicklungsumgebungen wie Visual Studio Code können Tools wie GitHub Copilot repetitive Abläufe direkt im Code vervollständigen oder Vorschläge zur Vervollständigung von neu begonnenem Code liefern[2].
- **Refactoring und Fehlersuche**: Dank ihrer Kontextsensitivität können LLMs bestehenden Code analysieren und an einigen Stellen Möglichkeiten zur Optimierung oder Korrektur vorschlagen[2, 5]. Dadurch lassen sich Bugs, Redundanz und ineffiziente Code-Strukturen frühzeitig identifizieren und beheben.
- **Automatisierte Dokumentation und Code-Kommentierung**: Viele Modelle bieten die Möglichkeit vorhandenen Code zu analysieren und dazu Kommentarblöcke oder gar ganze Dokumentationen zu erstellen[5, 7].

Obwohl diese Einsatzgebiete großes Potenzial bieten, sind LLMs nicht frei von Fehlern. Gerade bei komplexen Entscheidungen zur Programm- und Codearchitektur können diese oft mit dem Level durch das menschliche Fachwissen nicht mithalten [11].

3.3 Vergangene Studien und Arbeiten zur Code-Generierung

Die Forschung zur automatisierten Code-Generierung hat in den letzten Jahren eine rasante Entwicklung erlebt, wobei Arbeiten aus den Bereichen *Software Engineering*, *Large Language Models* und *Machine Learning* zusammenfließen. Jiang et al. (2024) haben in ihrer Arbeit Ä Survey on Large Language Models for Code Generationëine Übersicht über die Entwicklung

der veröffentlichten Arbeiten zu LLMs und Software Engineering erstellt, welche in Abbildung 3 dargestellt ist.

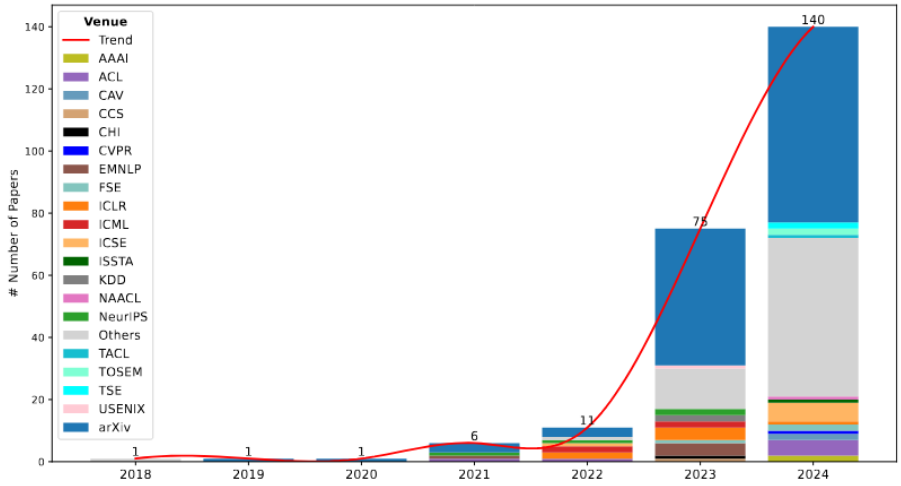

Abbildung 3: Übersicht der Verteilung von veröffentlichten Arbeiten zu LLMs und Software Engineering der letzten Jahren (nach Jiang et al. (2024), basierend auf 'A Survey on Large Language Models for Code Generation').

Im Folgenden werden einige vergangene Studien/Arbeiten zur Code Generierung mit LLMs vorgestellt:

1. **Ïs Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation"** von Liu et al. (2023) [1]:
   In diesem Paper wird untersucht, wie korrekt der von LLMs wie ChatGPT, Code Llama etc. generierte Code ist. Dafür wird *EvalPlus* eingeführt. Dies ist ein neues Evaluierungsframework, das bestehende Testdatensätze wie *HumanEval* durch weitere automatisierte Testfälle erweitert. Hier kommen Liu et al. zu dem Ergebnis, dass bisher viele Fehler in generiertem Code nicht erkannt wurden, wodurch die Modelle in ihrer Leistung überschätzt wurden. Die Autoren weisen darauf hin, wie wichtig umfassendene Tests sind, um die tatsächliche Funktionalität der LLMs für die Codegenerierung zu bewerten. TODO: Vielleicht Grafik der Ergebnisse einfügen

2. **Evaluating Large Language Models Trained on Code** von Chen et al. (2021) [2]:
   In diesem Paper wird Codex vorgestellt. Dies ist ein LLM, das speziell auf öffentlich verfügbarem Code von Github trainiert wurde, um dessen Fähigkeiten Python Code zu schreiben zu analysieren. Dies wird mithilfe des HumanEval-Datensatzes untersucht. Hierbei soll im genauen Python-Code aus Docstrings generiert und dieser dann bewertet werden. Die Ergebnisse zeigen, dass Codex im Vergleich zu anderen Modellen wie GPT-3 deutlich besser abschneidet, jedoch bei komplexeren Aufgaben seine Grenzen erreicht. Eine mehrfach wiederholte Lösungsgenerierung verbessert die Erfolgsrate, was das Potenzial ihres Ansatzes verdeutlicht.

3. **A Survey on Large Language Models for Code Generation** von Jiang et al.

(2024) [7]:
Dieses Paper gibt einen allgemeinen und umfassenden Überblick über den aktuellen Forschungsstand zu LLMs für die Codegenerierung. Es greift Themen wie Datenaufbereitung, Modellarchitekturen und Benchmarks auf. Zudem werden Herausforderungen, wie die praktische Einführung und ethische Fragen diskutiert. Die Autoren leiten sich wichtige Forschungsfragen ab und verdeutlichen, dass LLMs in der Codegenerierung große Fortschritte gemacht haben, aber es weiterhin Potenzial zur Optimierung gibt.

4. **Evaluating Language Models for Efficient Code Generation** von Liu et al. (2024) [12] Auch in dieser Arbeit von Jiawei Liu wird die Effizienz von Code untersucht, welcher von LLMs generiert wird. Hierbei mit Fokus auf Performance und Ressourcennutzung. Dafür wird *Differential Performance Evaluation (DPE)* entwickelt und der *EvalPerf*-Benchmark eingeführt. Dieser enthält komplexere Programmieraufgaben als der zuvor eingeführt *EvalPlus*. Hier kommt man zu dem Entschluss, dass größere Modelle nicht automatisch auch effizienteren Code erzeugen. Stattdessen werden Effizienz und Korrektheit des Codes durch *Instruction Tuning*(gezieltes Trainieren des Modells, um besser auf Anweisungen in natürlicher Sprache zu reagieren) verbessert.

Zusammenfassend zeigen die genannten Studien, dass LLMs zwar großes Potenzial zur automatisierten Code Generierung besitzen, sie aber immer noch Probleme aufweisen und menschliche Entwickler nicht komplett ersetzen können. Besonders bei komplexeren Aufgaben, spezifischen Anforderungen oder Fragen zur Softwarearchitektur stoßen sie an ihre Grenzen.

4 Methodik

TODO: noch mehr ausschreiben

4.1 Vorgehensweise der Untersuchung

In der Untersuchung soll geprüft werden, inwieweit Large Language Models in der Lage sind gängige Datenanalyse-Schritte auf Grundlage eines gegebenen Datensatzes durchzuführen. Hierbei wird ChatGPT als aktueller Marktführer mit dem Sprachmodell GPTo1, welches das neueste Modell ist, verwendet. Ebenso gilt es herauszufinden wie qualitativ und effizient diese Lösung ist. Hierbei bezieht es sich auf die Forschungsfragen aus Kapitel 1.1. Für die Vorgehensweise hierbei wird zuerst der verwendete Datensatz von Berlin Open Data an das Modell übergeben und dazu eine Prompt verfasst. Diese Prompts können in Kapitel 4.2 eingesehen werden. Im Anschluss durchläuft der Code mehrere Tests und manuelle Analysen um die Qualität und Effizienz des generierten Codes zu bewerten. Die genauen Auswertungskriterien sind in Kapitel **??** aufgeführt. Die Ergebnisse der Auswertung werden in Kapitel 5 detailliert dargestellt und auch mit den Ergebnissen anderer Arbeiten, wie etwa von Chen et al. (2021)[2] und Liu et al. (2023)[1] verglichen.

4.2 Testfälle der Datenanalyse

4.2.1 Testfall 1

Im ersten Testfall soll der Datensatz nach einer gewissen Spalte sortiert werden. Die Begründung hierfür ist, dass dies eine sehr einfache, aber auch sehr häufig aufkommende Datenanalyse-Aufgabe ist und somit einen guten Einstieg in die Untersuchung darstellt. Die Prompt für diese Aufgabe lautet: *Hier ist meine CSV-Datei mit dem Namen "HZ_2023.csv: **[CSV_content]**, bitte analysiere diese Daten und erstelle mir ein Python Skript, das nach der Anzahl der Straftaten insgesamt eines Bezirks in 2023 sortiert..*

4.2.2 Testfall 2

Für den zweiten Testfall sollen die Tabellen der Excel Datei durch einen Join zusammengeführt und dann der Bezirk mit den meisten Straftaten insgesamt von allen Jahren kombiniert geliefert werden. Die Prompt für diese Aufgabe lautet: *Erstelle mir ein Python Skript, mit welchem die Tabellen der Excel Datei durch einen Join zusammengeführt werden und der Bezirk mit den meisten Straftaten von allen Jahren kombiniert zurückgegeben wird..*

4.2.3 Testfall 3

Im dritten Testfall soll das Sprachmodell die prozentuale Verteilung der Straftaten in den Bezirken berechnen. Die Prompt für diese Aufgabe lautet: *Erstelle mir ein Python Skript, mit welchem die prozentuale Verteilung der genauen Straftaten anteilig der Straftaten insgesamt pro Bezirk berechnet wird..*

4.2.4 Testfall 4

Im vierten Testfall soll eine simple Visualisierung des Datensatzes stattfinden. Es soll ein Balkendiagramm der Bezirke und der Verteilung der genauen Straftaten erstellt werden. Hierbei sollen die Bezirke auf der x-Achse und die Anzahl der einzelnen Straftaten mit ihrer Verteilung in einem Balken auf der y-Achse dargestellt und absteigend mit der Anzahl der Straftaten insgesamt pro Bezirk sortiert werden. Die Prompt für diese Aufgabe lautet: *Erstelle mir ein Python Skript, mit welchem ein Balkendiagramm der Anzahl der Straftaten insgesamt und dessen Verteilung der genauen Straftaten pro Bezirk erstellt wird. Hierbei sollen auf der X-Achse die Bezirke und auf der Y-Achse pro Bezirk ein Balken über die Verteilung der Anzahl der Straftaten sein..*

4.2.5 Testfall 5

Für den letzten Testfall ist die Erstellung einer Zeitreihe vorgesehen. Hierbei soll die Anzahl der Straftaten insgesamt pro Jahr in einer Zeitreihe dargestellt werden. Die Prompt für diese Aufgabe lautet: *Erstelle mir ein Python Skript, mit welchem eine Zeitreihe der Anzahl der Straftaten insgesamt pro Jahr dargestellt wird..*

4.3 Auswertungskriterien

Korrektheit des Codes und der Ergebnisse, Performance des Codes im Bezug auf Laufzeit und Ressourcennutzung, Qualität des Codes (Struktur des Codes, Kommentare/Dokumentation, verwendete Libraries), Wartbarkeit und ist der Code erweiterbar.

Die Auswertung der generiertes Python Skripte erfolgt anhand der in Kapitel 1.1 definierten Kriterien. Um die Korrektheit des Codes zu messen wird das Pass@k Verfahren verwendet, dabei steht "k"für die Anzahl der ausgeführten Versuche pro Testfall. In diesem Experiment beschränke ich mich auf k=10, um eine gute Balance zwischen Genauigkeit und Rechenzeit zu finden. Bei diesem Verfahren ergibt sich als Ergebnis ein Prozentsatz über die Anzahl der erfolgreichen Versuche. Die ausgeführten Versuche werden anschließend in erfolgreich und nicht erfolgreich unterteilt und getrennt genauer betrachtet. Um zu entscheiden, ob ein Versuch erfolgreich war, werden Tests definiert, welche das Ergebnis und den Code selbst prüfen. Im Code wird dabei darauf geachtet, welche Bibliotheken, Funktionen und Pandas Dataframes benutzt wurden. In der genaueren Analyse des Codes wird bei den nicht erfolgreichen Versuchen wird untersucht, warum der Code nicht korrekt ausgeführt wurde und was für Verbesserungen vorgenommen werden können. Bei den erfolgreichen Versuchen hingegen wird analysiert, wie der Code strukturiert ist, ob er gut dokumentiert ist, ob er erweiterbar ist und wie die Laufzeit und Ressourcennutzung des Codes abschneidet.

4.4 Verwendete Tools

1. **Large Language Model**: Als Large Language Model wird ChatGPT mit GPTo1-mini verwendet, da ChatGPT als aktueller Marktführer gilt und GPTo1-mini das neueste und leistungsfähigste Modell ist, welches mit der OpenAI API verfügbar ist.

5 Auswertung der Python-Code-Generiung zur Datenanalyse durch LLMs

6 Fazit und Ausblick

7 Literaturverzeichnis

Literatur

[1] Jiawei Liu u. a. "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation". In: *Advances in Neural Information Processing Systems*. Hrsg. von A. Oh u. a. Bd. 36. Curran Associates, Inc., 2023, S. 21558–21572. URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf.

[2] Mark Chen u. a. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. URL: https://arxiv.org/abs/2107.03374.

[3] Jiawei Liu u. a. "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation". In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023. URL: https://openreview.net/forum?id=1qvx610Cu7.

[4] Erik Nijkamp u. a. *CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis*. 2023. arXiv: 2203.13474 [cs.LG]. URL: https://arxiv.org/abs/2203.13474.

[5] Yue Wang u. a. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*. 2021. arXiv: 2109.00859 [cs.CL]. URL: https://arxiv.org/abs/2109.00859.

[6] Humza Naveed u. a. *A Comprehensive Overview of Large Language Models*. 2024. arXiv: 2307.06435 [cs.CL]. URL: https://arxiv.org/abs/2307.06435.

[7] Juyong Jiang u. a. *A Survey on Large Language Models for Code Generation*. 2024. arXiv: 2406.00515 [cs.CL]. URL: https://arxiv.org/abs/2406.00515.

[8] Mark Chen u. a. "Evaluating Large Language Models Trained on Code". In: (2021). arXiv: 2107.03374 [cs.LG].

[9] Baptiste Rozière u. a. *Code Llama: Open Foundation Models for Code*. 2024. arXiv: 2308.12950 [cs.CL]. URL: https://arxiv.org/abs/2308.12950.

[10] Raymond Li u. a. *StarCoder: may the source be with you!* 2023. arXiv: 2305.06161 [cs.CL]. URL: https://arxiv.org/abs/2305.06161.

[11] Rudra Dhar, Karthik Vaidhyanathan und Vasudeva Varma. *Can LLMs Generate Architectural Design Decisions? -An Exploratory Empirical study*. 2024. arXiv: 2403.01709 [cs.SE]. URL: https://arxiv.org/abs/2403.01709.

[12] Jiawei Liu u. a. "Evaluating Language Models for Efficient Code Generation". In: *First Conference on Language Modeling*. 2024. URL: https://openreview.net/forum?id=IBCBMeAhmC.

TODO: Abbildungsverzeichnis einfügen