Seminararbeit

Limitationen von ChatGPT beim Generieren von Programmiercode

Julian Kalmbach

March 15, 2024

Vorgelegt an der Albert-Ludwigs-Universität Freiburg
Institut für Wirtschaftswissenschaften
Lehrstuhl für Wirtschaftsinformatik

universität freiburg

Albert-Ludwigs-Universität Freiburg
Institut für Wirtschaftswissenschaften
Lehrstuhl für Wirtschaftsinformatik

Autor Julian Kalmbach,
Matrikelnummer: 4941824

Zeitraum 30. Oktober 2023–15. März 2024

Betreuer Tano Müller,
Institut für Wirtschaftswissenschaften
Lehrstuhl für Wirtschaftsinformatik

Contents

1 Einleitung

Seit es ChatGPT gibt, wird es in fast allen Bereichen von einer breiten Masse an Menschen getestet und an seine Grenzen gebracht oder eben auch nicht. Gefährlich kann es allerdings werden, wenn man die Antworten von ChatGPT ungeprüft nutzt oder einfach annimmt, dass diese stimmen würden. Zum einen ist ChatGPT nicht unbedingt darauf ausgelegt, eine "richtige" Antwort zu geben, sondern mehr darauf, eine in menschlicher Sprache verständliche Antwort zu liefern. Zum anderen gibt es auch für viele Anwendungsbereiche schlicht noch keine wissenschaftlichen Untersuchungen dazu, wie zuverlässig ChatGPT auf dem jeweiligen Anwendungsbereich tatsächlich ist.

1.1 Motivation

Auch beim Programmieren wird ChatGPT mehr und mehr genutzt und es ist davon auszugehen, dass von ChatGPT generierte Programme immer weiter einzug auch in tatsächlich eingesetzter Software erhält. Dabei kann es zu erheblichen Problemen kommen, wenn die Programme von ChatGPT ungeprüft und ohne wissenschaftliche Erkenntnisse über die Zuverlässigkeit und Leistungsfähigkeit eingesetzt werden. In dieser Arbeit wird dementsprechend versucht, dem entgegenzuwirken und Erkenntnisse über die Qualität von durch ChatGPT generierten Code zu erlangen.

Die Kernfragen lauten daher:

- Gibt es Auffälligkeiten beim von ChatGPT generierten Sourcecode und wenn ja, welche?
- Wie zuverlässig sind die von ChatGPT erzeugten Programme?
- Sind die Programme im Vergleich mit der Musterlösung effizient?

2 Methoden

Um ChatGPT auf seine Fähigkeit, Sourcecode zu schreiben, zu prüfen, wurde erst einmal damit angefange,n kleine Code-Bausteine zu erfragen. Dabei war aber stehts wichtig, dass in der Frage nie direkt der Code-Bautstein erwähnt war, der generiert werden sollte. Diese Methode diente als kleiner Einstieg und dafür erstmal grundsätzlich die Fähigkeiten der KI einschätzen zu können. Schon hier ließen sich kleine Probleme erkennen, wenn es auch wenige waren.

Um nun die Fähigkeiten wirklich tiefer gehend zu testen, bot es sich an, ChatGPT die Übungsblätter der Vorlesung "Einführung in die Programmierung" des Instituts für Informatik, bzw. des Lehrstuhls für Programmiersprachen der Universität Freiburg, lösen zu lassen. Diese müssen alle Studierenden der Informatik an der Universität Freiburg innerhalb der ersten drei Semester bestehen und sie umfassen viele Themen, die für das allgemeine Verständnis für das Programmieren erforderlich sind, weshalb diese eine gute Messlatte für grundsätzliche Fähigkeiten beim Programmieren sind.

Bevor ChatGPT die Aufgaben allerdings gestellt werden konnten, mussten diese erst in ein passendes Format gebracht werden, da die Übungsblätter mit LaTeX geschrieben und als PDF bereitgestellt werden. Einfaches Auswählen und Kopieren des gewünschten Textabschnittes ist dabei leider nicht möglich gewesen, da der Text so jede Formatierung verliert und insbesondere mathematische Formeln nicht korrekt übernommen werden. Da ohnehin jeglicher Inhalt der Arbeit in einem GitHub Repository gespeichert ist, hat sich an der Stelle das Format Markdown als sinnvoll herausgestellt, woraufhin die entsprechenden Aufgaben in das neue Format übertragen wurden. Wichtig dabei war auch, dass teilweise in den Aufgaben auch separat Sourcecode vorgegeben war und zur Verfügung gestellt wurde. Dieser wurde dann ebenfalls in den Aufgabentext integriert, um sicherzustellen, dass die Aufgabe lösbar ist und dieselben Bedingungen

wie auch für die Studierenden gelten. Insgesamt wurden ChatGPT so 41 Programmieraufgaben gestellt, wobei darunter auch einige Teilaufgaben sind, da es an manchen Stellen als sinnvoll erschien, die Aufgabenstellung aufzuteilen um so die Vergleichbarkeit bestimmter Elemente sicherzustellen. Diese Aufgaben können auch, in genau dem Format, in dem sie ChatGPT auch gestellt wurden, auf dem GitHub Repository oder auf der Kurswebseite "Einführung in die Programmierung", eingesehen werden. Die Links dazu befinden sich in den Quellen.

Um die Programme, beziehungsweise den Sourcecode, miteinander vergleichen zu können, wurden die von ChatGPT generierten Programmcodes anhand verschiedener Metriken bewertet. Diese Metriken waren wie folgt:

- **pass@1:** Erzeugt das Programm den gewünschten Output, also ist das, was das Programm als "Ergebnis" ausgibt, korrekt? Dieser Wert kann entweder "Wahr", das Programm liefert den gewünschten Output, oder "Falsch", das Programm liefert nicht den gewünschten Output, annehmen.
- **Anforderungen erfüllt:** Auch diese Metrik kann etweder den Wert "Wahr" oder "Falsch" annehmen. Liefert ein Programm zwar den richtigen Output, entspricht aber nicht den spezifischen Anforderungen, welche in der Aufgabe gestellt waren, so erfüllt das Programm auch nicht diese Metrik. Der Wert ist also "Falsch". Liefert das Programm einen falschen Output, wurden offensichtlich die Anforderungen nicht erfüllt, denn eine Anforderung ist logischerweise immer, dass das Programm so funktioniert wie gewünscht.
- **LOC$_{tot}$:** LOC steht für "Lines of Code" und ist eine bekannte, wenn auch alleinstehend nicht immer so vielsagende Metrik für Programmcode. Hier kann es aber interessante Ergebnisse liefern, da wir so zum Beispiel überprüfen können, ob ChatGPT fehleranfälliger wird, abhängig der Anzahl an Zeilen. Bei LOC$_{tot}$ werden alle Zeilen, also auch Zeilen mit Kommentaren und Leerzeilen gezählt.
- **LOC$_{pars}$:** mit $_{pars}$ ist "parsable" gemeint, also jede Zeile des Sourcecodes, welcher tatsächlichen, kompilierbaren Code enthält. Leerzeilen oder Kommentare werden also nicht mitgezählt. Diese Metrik ist insbesondere interessant zu vergleichen, da sie tatsächlich helfen kann, Aussagen darüber zu treffen, wie effizient oder "gut" ein Programm geschrieben wurde.
- **Cyclomatic Complexity:** Diese wird auch McCabe complexity genannt und beschreibt die Anzahl an decision points, also Entscheidungspunkten,

innerhalb eines Programms. Allerdings wurde diese in diesem Fall nur eingeschränkt angewendet. Für bestimmte vorkommen von Codeelementen wird diese Metrik eins hochgezählt. Diese lauten wie folgt:

- if: +1
- elif: +1
- else: +0
- for: +1
- while: +1

Eigentlich müsste hier auch für asserts hochgezählt werden, allerdings wird in diesem Fall darauf verzichtet, da die Programme auch ohne asserts richtig sein können und diese nur dazu dienen zu überprüfen, ob das Programm den richtigen Output liefert oder nicht.

Nach der Bewertung aller 41 Programme anhand der festgelegten Metriken wurden auch die Auswertungen der Musterlösungen hinzugezogen. Diese wurden jedoch nur auf Cyclomatic Complexity und LOC_{pars} geprüft, da sich diese als sinnvolle Metriken erwiesen, die effektiv etwas über die Qualität des Programms aussagen. Zudem kann davon ausgegangen werden, dass die Musterlösungen funktionieren und den Anforderungen entsprechen, wodurch die Bewertung dieser Programme anhand dieser Metriken wohl eher überflüssig wäre. LOC_{tot} wurde dabei eher als Metrik gewählt, um die Programme von ChatGPT zu vergleichen und gegebenenfalls Erkenntnisse zum Beispiel darüber zu erlangen, wie viele Kommentare ChatGPT 3.5 beziehungsweise 4.0 einsetzt.

Die aus diesem Prozess gewonnenen Daten wurden daraufhin auf die Kernfragen dieser Arbeit überprüft.

3 Auffälligkeiten im Programmcode

Neben den Metriken, die zur Einstufung der Programme im Allgemeinen dienen, ist dennoch auch im Einzelnen die Frage zu stellen, ob es Auffälligkeiten wie Fehler oder "unschönen" Programmcode gibt. Im Allgemeinen lässt sich aber sagen, dass ChatGPT an vielen Stellen keinen schlechten Code schreibt. Gerade ChatGPT 4.0 erklärt den generierten Programmcode meist ausgiebig in den Antworten und fügt auch viele Kommentare im Sourcecode hinzu, um die Verständlichkeit zu verbessern.

3.1 Programmfehler

Auch wenn tatsächlich der Großteil der Programme den richtigen Output lieferten, war das nicht für alle Programme der Fall. Die Programme, welche nicht den richtigen Output lieferten, waren entweder gar nicht ausführbar, hingen in einem Deadlock fest, oder funktionierten an sich, lieferten allerdings tatsächlich eine andere Ausgabe als gewünscht, wobei die Tests größtenteils vorher bekannt waren und in der Aufgabenstellung standen.

Einen Deadlock lieferte beispielsweise der Programmcode, der von ChatGPT zur Aufgabe 10.1b erzeugt wurde. Die Eingabe, welche zu diesem Output geführt hat, war die folgende:

> - Die Mindestpunktzahl wird wie folgt berechnet: ‘‘‘pass_points = max_points // 2‘‘‘
>
> - Zur Unterstützung seines Wahlkampfs fordert Ex-Präsident T., dass

die Durchfallquote aller Klausuren bei höchstens 40% liegen darf. Sollte das in dieser Klausur nicht der Fall sein, werden die pass_points so weit nach unten angepasst, bis die Forderung erfüllt ist.

Listing 3.1: Deadlock in While loop

```
def calculate_passing_points(max_points, student_points):
    failure_threshold = 0.4
    pass_points = max_points // 2

    # Calculate the number of students who passed
    passed_students = sum(1 for points in student_points.values() ↩
        if points >= pass_points)
    # Calculate the percentage of students who passed
    pass_rate = passed_students / len(student_points)

    # Adjust pass points until the failure threshold is met
    while pass_rate > 1 - failure_threshold:
        pass_points -= 1
        passed_students = sum(1 for points in ↩
            student_points.values() if points >= pass_points)
        pass_rate = passed_students / len(student_points)

    return pass_points
```

Das Programm läuft mit eben jener While loop bei dem vorgegebenen Test case in einen Deadlock. Genauer gesagt macht die Loop genau das Gegenteil, von dem, was sie soll. Ist die **pass_rate** nämlich vorher schon größer als 40%, kommt die While loop nicht mehr an ihre Abbruchbedingung. Das ist ein grober logischer Fehler, der, abgesehen davon, dass man in dem gegebenen Programm weiß, dass es aus dieser Loop kommt, auch gar nicht so leicht zu identifizieren ist nur durch Anschauen des Codes.

ChatGPT, zumindest ChatGPT 3.5 hat tatsächlich aber auch code erzeugt, der gar nicht ausgeführt werden kann. Dabei hat ChatGPT 3.5 zur Aufgabe 13.2 folgenden Code erzeugt:

Listing 3.2: Verwendung einer Klassentyps statt einer Typvariable

```
S = TypeVar('S')
```

```
2
3 class State(Generic[S, Enum]):
4     def next(self, input: str) -> S:
5         raise NotImplementedError("Subclasses must implement next ↩
              method.")
6
7     def output(self) -> Enum:
8         raise NotImplementedError("Subclasses must implement output ↩
              method.")
```

Beim Versuch, den Code auszuführen, bekommt man aber folgenden Fehler:

TypeError: Parameters to Generic[...] must all be type variables or parameter specification variables.

Die das zweite Argument von Generic[] ist vom Typ Enum. Generic[] nimmt aber keine Klassen in dem Sinne, sondern Typvariablen, also müsste das zweite Argument von Generic auch als TypVar() implementiert werden. Es scheint also, dass ChatGPT hier Schwierigkeiten damit hat, den richtigen Typ zu wählen.

Der Sourcecode der Musterlösung sieht für die gleiche Funktion so aus:

Listing 3.3: Ausschnitt aus der Musterlösung 13.2

```
1  # Base class for all states.
2  INPUT = TypeVar("INPUT")
3  OUTPUT = TypeVar("OUTPUT")
4
5  @dataclass
6  class State(Generic[INPUT, OUTPUT], ABC):
7      def next(self, sensor_input: INPUT) -> 'State':
8          return self
9
10     @abstractmethod
11     def output(self) -> OUTPUT:
12         ...
```

Auch ChatGPT 4.0 schaffte es nicht, die Funktion richtig zu implementieren. Zwar gab es bei der Lösung von ChatGPT 4.0 keine TypeErrors, allerdings gener-

ierte ChatGPT 4.0 eine unfertige Lösung, welche NotImplementedErrors zurückgibt.

Listing 3.4: Ausschnitt aus dem Lösungsversuch von ChatGPT 4.0

```
S = TypeVar('S', bound='State')

class State(Generic[S]):
    def next(self, input: str) -> 'State[S]':
        raise NotImplementedError

    def output(self) -> MyState:
        raise NotImplementedError
```

Sowohl ChatGPT 3.5 als auch 4.0, erzeugten also teilweise auch fehlerbehafteten Code. Teilweise auch mit Fehlern, die einem Entwickler schon beim Schreiben des Codes aufgefallen wären, wie beispielsweise der TypeError in der Lösung von ChatGPT 3.5 zur Aufgabe 13.2. Aber auch Logikfehler treten teilweise auf, was allein dadurch schon belegt ist, dass die Programme nicht alle den gewünschten Output liefern.

3.2 Pattern Matching

Ein Problem, das schon beim Testen der einfachen Codeblöcke zu beobachten war, ist, dass ChatGPT 3.5 Informationen zur Umsetzung von Pattern Matching oder auch ”Match-Case” fehlen, oder von der KI einfach nicht berücksichtigt werden. Klar ist, dass Pattern Matching in Python erst mit der Version python3.10 veröffentlicht wurde, welche es erst seit Oktober 2021 gibt. ChatGPT 3.5 wurde aber zuletzt im September 2021 mit Daten gefüttert, hat also noch kein Wissen über python 3.10. Dadurch konnte ChatGPT bei einigen Aufgaben die Requirements nicht erfüllen. Auf die Aufforderung:

> Schreiben Sie eine Funktion node_to_str, die einen Ausdrucksbaum als Argument nimmt und dessen Darstellung als String zurückgibt. Machen Sie die Klammerung von Operatoren explizit und verwenden Sie genau ein Leerzeichen um Operatoren von Argumenten zu tren-

nen.

[...]

Verwenden Sie hierzu Pattern Matching und keine if-Verzweigungen.

[...]

hat ChatGPT folgenden Code erzeugt:

: Listing 3.5
Kein Pattern Matching obwohl

```
1 def node_to_str(node: Union[Var, Val, Op]) -> str:
2     if isinstance(node, Var):
3         return node.name
4     elif isinstance(node, Val):
5         return str(node.value)
6     elif isinstance(node, Op):
7         left_str = node_to_str(node.left)
8         right_str = node_to_str(node.right)
9         return f"({left_str} {node.sym.value} {right_str})"
```

3.3 Dataclasses

Die Anforderung, Datenklassen zu nutzen, wurde von ChatGPT 3.5 ebenfalls nicht berücksichtigt, was interessant ist, da es Datenklassen bereits seit der Python Version 2.7 gibt und ChatGPT 3.5 das dementsprechend zumindest kennen müsste. Die Programme von ChatGPT funktionierten dann zwar meist auch, wurden aber eben nicht über Datenklassen, sondern über normale Klassen umgesetzt.

4 Auswertung

4.1 Zuverlässigkeit

Mit der Zuverlässigkeit der Programme von ChatGPT ist primär gemeint, ob die Programme direkt ohne Weiteres den richtigen Output, also die richtige Ausgabe, liefern. Diese Bedingung wird **pass@1** genannt. ChatGPT 3.5 hat von den 41 Aufgaben 32 mit einem richtigen Output gelöst, während es bei ChatGPT 4.0 33 waren. Der Unterschied hier ist tatsächlich marginal, weswegen sich hier sagen lässt, dass sich ChatGPT 3.5 und 4.0 zumindest in Bezug auf die generelle Zuverlässigkeit nicht unterscheiden. Wenn man nun die pass@1-Rate von ChatGPT 3.5 und 4.0 in Abhängigkeit der Einteilung der Aufgaben in die Schwierigkeitsgrade leicht (”easy”), mittel (”medium”) und schwer (”hard”) beobachtet, ergibt sich, dass ChatGPT 3.5 und 4.0 vor allem bei den Aufgaben der Kategorie ”medium” Schwierigkeiten hatten.

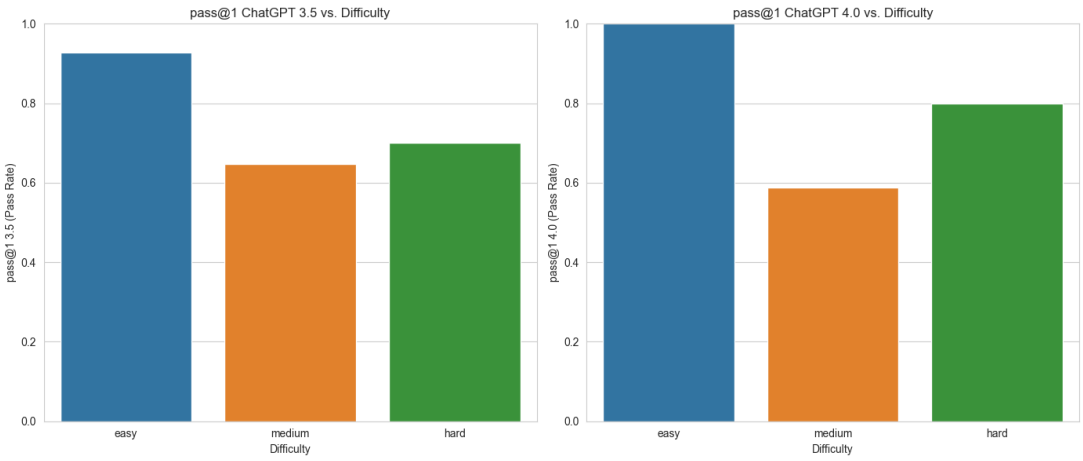

Figure 4.1:

Interessant ist allerdings zu sehen, dass ChatGPT 4.0 in der Kategorie ”medium” sogar schlechtere Ergebnisse liefert als ChatGPT 3.5.

Schaut man sich die Metrik ”Anforderungen erfüllt” an und klammert die Programme aus, die grundsätzlich schon keinen korrekten Output liefern, zeigt sich, dass ChatGPT 3.5 in 6 von 32 Fällen die Anforderungen nicht erfüllt, wodurch tatsächlich nur noch 26 der 41 Aufgaben komplett richtig gelöst wurden. Das lässt sich vor allem auf die veralteten Daten zurückführen mit, denen ChatGPT 3.5 trainiert wurde. In allen Fällen, in denen ChatGPT 3.5 die Anforderungen nicht erfüllte aber eine richtige Ausgabe lieferte, handelte es sich um nicht verwendetes Pattern-Matching oder Datenklassen. Dass dennoch ein richtiger Output erzeugt wurde, liegt daran, dass ChatGPT 3.5 Pattern-Matching, wie das vor dem Release dieser üblich war, mit if-, elif-, else-Verknüpfungen umsetzt. Betrachtet man die Lösungen von ChatGPT 4.0 unter denselben Bedingungen, kann man erkennen, dass ChatGPT 4.0 hier deutlich bessere Leistungen bringt als ChatGPT 3.5. Bei ChatGPT 4.0 erfüllen nämlich alle Programme, welche einen richtigen Output liefern, auch die Anforderungen. Somit löst ChatGPT 4.0 mit 33 von 41 Aufgaben deutlich mehr Aufgaben korrekt als ChatGPT 3.5. Schaut man sich den Graphen aus Figure 4.1 nochmal an und bewertet die Programme nun anhand der Metrik ”Anforderungen erfüllt”, ergibt sich daraus der folgende Graph:

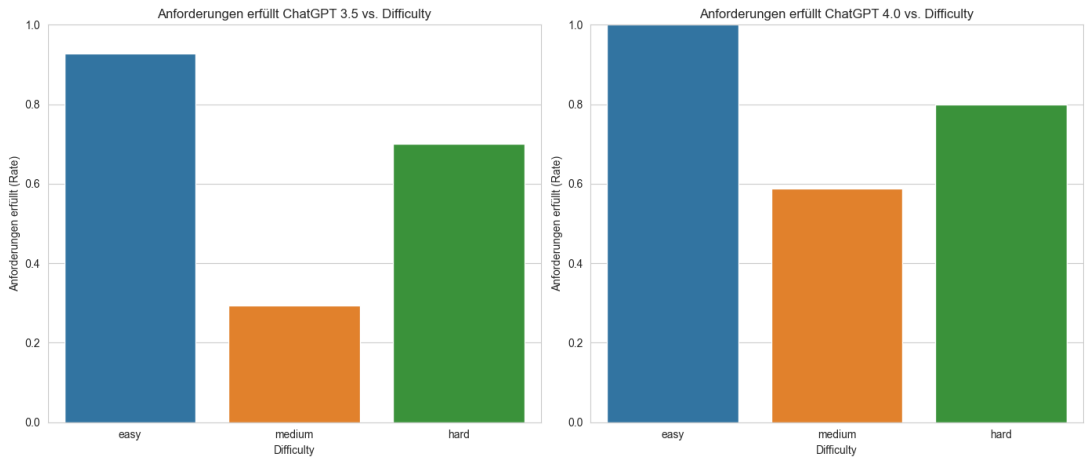

Figure 4.2:

4.2 Effizienz

Um die Effizienz der von ChatGPT generierten Programme zu bewerten ist es naheliegend, sie mit den vom Lehrstuhl für Programmiersprachen bereitgestellten Musterlösungen zu vergleichen. Die Metriken, die hierfür interessant sind, sind LOC_{pars} und Cyclomatic Complexity, beziehungsweise McCabe Complexity.

4.2.1 LOC_{pars}

Vergleicht man die Zeilen an tatsächlichem Programmcode, also LOC_{pars} mit denen von der jeweiligen Musterlösung, zeigt sich, dass ChatGPT 4.0 sogar etwas effizienter zu sein scheint als die Musterlösung, wärend ChatGPT 3.5 im Schnitt etwas weniger effizient ist. Wichtig zu beachten ist, dass die Fälle, in denen ChatGPT 4.0 oder 3.5 jeweils nicht die Erwartungen erfüllt haben, nicht gewertet werden, da dies die Aussagekraft über die Effizienz anhand der Anzahl an Zeilen verfälschen würde.

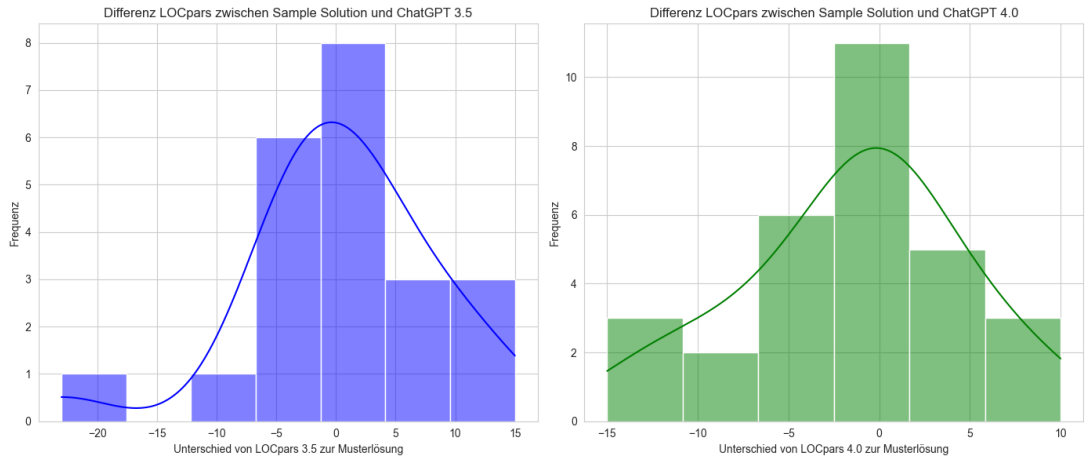

Figure 4.3:

Ist der Wert auf der x-Achse positiv, hat die jeweilige ChatGPT-Version mehr Zeilen gebraucht als die Musterlösung. Ist der Wert negativ, hat ChatGPT weniger LOC_{pars} gebraucht um das Problem korrekt zu lösen.

Im Groben lässt sich also sagen, dass ChatGPT gleich viele Zeilen braucht um

die Aufgaben zu lösen wie die Musterlösung, wobei ChatGPT 4.0 sogar minimal weniger und ChatGPT 3.5 etwas mehr Zeilen braucht. Diese Metrik kann hilfreich sein, da Programme, die weniger Zeilen an Code haben, teils besser lesbar sind und weniger Leistung brauchen. Dies muss aber nicht zwangsweise heißen, dass ein Programm mit weniger Zeilen besser ist, als ein Programm für denselben Zweck mit mehr Zeilen. Dennoch ist es interessant die Zahlen gegenüberzustellen und zu vergleichen, da es durchaus im Interesse von Entwicklern ist, so wenig Code wie möglich zu schreiben.

4.2.2 Cyclomatic Complexity

Bei der Cyclomatic Complexity handelt es sich um ein sehr wirksames Mittel um die Effizienz eines Programms zu bewerten. Dadurch, dass mit jedem if, else, while usw. die Cyclomatic Complexity steigt, ist ein Programm, welches das gleiche Problem mit einer geringeren Cyclomatic Complexity lösen kann, als effizienter zu bewerten. Dies gilt nicht ausschließlich, lässt sich aber grob als Aussage treffen.

Vergleicht man nun die Summe der Cyclomatic Complexity-Werte gruppiert nach der jeweiligen Schwierigkeit, fällt auf, dass sowohl ChatGPT 3.5 als auch ChatGPT 4.0 in der Summe weniger effiziente Programme schreiben. Nur beim Vergleich von ChatGPT 4.0 und der Musterlösung bei der Schwierigkeit hard sind die Lösungen von ChatGPT 4.0 etwas besser als die Musterlösung.

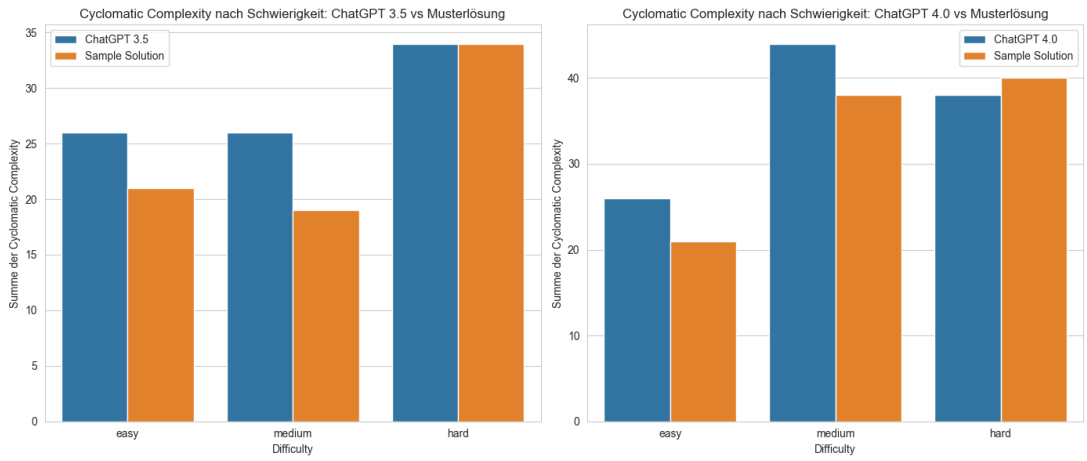

Figure 4.4:

Allerdings muss man auch sagen, dass der Abstand zur Musterlösung weder bei ChatGPT 3.5, noch bei ChatGPT 4.0 besonders hoch ist. Geht es allerdings darum, die Programme besonders effizient zu gestalten, scheint ChatGPT nicht die beste Wahl zu sein. Das wird noch einmal deutlicher, wenn man berücksichtigt, dass die Musterlösungen des Lehrstuhls auch nicht darauf ausgelegt sind, besonders effizient, sondern eher verständlich zu sein. Es ist also davon auszugehen, dass man die Probleme auch noch effizienter lösen könnte.

5 Einschränkungen

Diese Arbeit lässt auf jeden Fall Rückschlüsse darauf ziehen, wie gut ChatGPT in der jeweiligen Version im Generieren von Programmcode ist. Allerdings muss ganz klar darauf hingewiesen werden, dass der Datensatz mit 41 Aufgaben sehr klein ist und somit die Ergebnisse leicht verzerrt sein können. Um noch präzisere Aussagen über die Effizienz und andere Metriken treffen zu können, braucht es einen Datensatz, der mindestens zehnmal so groß ist. Des Weiteren sind ein paar der Probleme, die in den Aufgaben zu lösen waren, klassische Programmierprobleme oder Aufgaben (zum Beispiel das Sierpinskidreieck). Bei diesen Aufgaben ist es durchaus möglich, dass ChatGPT bereits mit Lösungen zu Aufgaben dieser Art trainiert wurde und somit Aussagen darüber, wie gut ChatGPT im Allgemeinen im Programmieren ist, also auch im Bewältigen von neuen Aufgaben, zumindest mal mit Vorsicht getroffen werden müssen.

6 Zusammenfassung

ChatGPT kann immer mehr und ist im Umgang mit Programmieraufgaben durchaus ein brauchbares Werkzeug. Allerdings ist ChatGPT zum aktuellen Stand noch nicht uneingeschränkt zum Programmieren nutzbar. Gerade, wenn man die kostenfreie Version nutzt, muss man sich darüber im Klaren sein, dass diese bestimmte Anforderungen einfach nicht erfüllen kann. Und auch im Allgemeinen zeigt sich, dass man derzeit nach wie vor über eigene Programmierfähigkeiten verfügen muss, wenn man ChatGPT zum Programmieren verwenden will. Der generierte Code ist nicht zwangsweise fehlerfrei und wenn man nicht gerade eine Musterlösung mit dem korrekten Output zur Hand hat, kann es durchaus vorkommen, dass Werte als vermeintlich richtig berechnet angesehen werden, obwohl sie das gar nicht sind. Das ist insbesondere bei sicherheitskritischen Programmen ein großes Problem, weshalb gerade hier ChatGPT definitv noch nicht eingesetzt werden kann. Auch wenn man besonders effizienten Programmcode erwartet, wird man bei ChatGPT nicht fündig werden und es mag sich mehr lohnen, sich an einen erfahrenen Entwickler zu wenden. Vergleicht man aber ChatGPT mit Informatik Studierenden im ersten Semester, so ist ChatGPT keineswegs ein schlechtes Werkzeug und kann durchaus mit Menschen mithalten. Wenn man berücksichtigt, was die Entwicklung von KI der letzten Jahre mit sich gebracht hat, ist da aber vermutlich noch viel Luft nach oben und das Potenzial ist noch lange nicht ausgeschöpft. Aus diesem Grund ist es auch interessant, Arbeiten wie diese in regelmäßigen Abständen zu wiederholen um, die Entwicklung von ChatGPT festhalten zu können.

Quellen

[Open AI Training von ChatGPT 3.5] https://platform.openai.com/docs/models/gpt-3-5-turbo/
[Release Datum vom Python 3.10] https://www.python.org/downloads/release/python-3100/
[LaTeX-Vorlage] https://www.overleaf.com/latex/templates/emes-vorlage-abschlussa
cwjcytntgxtg
[Chat GPT] https://chat.openai.com/
[Webseite zu den Übungsblättern] https://proglang.informatik.uni-freiburg.de/teaching/info1/2022/
[GitHub Repo mit den Inhalten zu dieser Arbeit] https://github.com/JulianKalmbach/Limitations-of-ChatGPT-for-code-generation