Künstliche Intelligenz in der Softwareentwicklung

Über die Schulter geschaut

Prof. Dr. Bernd Finkbeiner, Frederik Schmitt

Künstliche Intelligenz hilft Entwicklern beim Coden. Die Frage ist, ob KI Entwickler-Know-how auch ersetzen kann. Der Artikel beleuchtet den derzeitigen Forschungsstand.

Entwicklerinnen und Entwickler müssen in der modernen Softwareentwicklung immer seltener Probleme von Grund auf neu lösen. Ein Großteil der Funktionen steckt in Bibliotheken, und Plattformen wie GitHub und Stack Overflow liefern Millionen Codebeispiele zu nahezu jeder denkbaren Programmieraufgabe. Machine Learning könnte die Programmierarbeit noch einmal vereinfachen.

Der Wunsch nach Programmen, die Programme schreiben, ist so alt wie die Informatik. Schon 1957 formulierte Alonzo Church, einer der Gründer der damals noch jungen Disziplin, die Herausforderung: Ist es möglich, die Spezifikation einer Programmieraufgabe automatisch, also durch einen Algorithmus, in ein Computerprogramm zu übersetzen, sodass das Programm die Aufgabe für jede mögliche Eingabe korrekt löst?

Heute, mehr als 60 Jahre später, scheint die Antwort zum Greifen nah. Sprachmodelle wie das vom kalifornischen Start-up OpenAI im letzten Jahr vorgestellte GPT-3 (Generative Pre-trained Transformer 3) sind in der Lage, in natürlicher Sprache formulierte Anforderungen vollautomatisch in ein Computerprogramm zu übersetzen. Aus der Beschreibung "compute the greatest common divisor of two integers" und einem einzigen Codebeispiel zum Berechnen von Fibonacci-Zahlen erzeugt GPT-3 den im Listing gezeigten Python-Code.

X-TRACT

- Entwicklerinnen und Entwickler erhalten beim Programmieren Hilfe durch künstliche Intelligenz.
- Sprachmodelle wie GPT-3 übertragen in natürlicher Sprache gegebene Anforderungen in ein Programm.
- Die Programmsynthese aus logischen Spezifikationen eignet sich für sicherheitskritische Systeme.

Websites automatisch mit GPT-3 programmieren

Eine der beeindruckendsten Anwendungen dieser Technologie ist die automatische Programmierung von Websites. Auf Twitter kursieren Videos, etwa von Debuild-Gründer Sharif Shameem, die zeigen, wie GPT-3 aus einer beliebigen Layoutbeschreibung JSX-Code macht (Abbildung 1). Nutzer geben lediglich einen kurzen englischen Text ein, beispielsweise: "the google logo, a search box, 2 lightgrey buttons that say 'Search Google' and 'I'm feeling Lucky' with padding in-between them". Doch es stellt sich die Frage, wie zuverlässig das ist und ob die automatische Codegenerierung mehr als nur automatisiertes Copy-and-Paste ist.

Das Potenzial von Sprachmodellen wie GPT-3 hat auch Microsoft erkannt und im Jahr 2019 eine Milliarde US-Dollar in OpenAI investiert. Jetzt stellte die Micro-

soft-Tochter GitHub gemeinsam mit OpenAI den GitHub Copilot vor (siehe Kasten auf S. 42). Copilot nutzt Codex, eine Weiterentwicklung von GPT-3: Die KI analysiert Kommentare und das bisher geschriebene Programm, um damit einzelne Codezeilen oder ganze Funktionen zu vervollständigen.

Erste Reaktionen sind mehrheitlich positiv, doch nicht immer funktioniert der von Copilot produzierte Code. GitHub selbst rät dazu, den generierten Code sorgfältig zu prüfen. Die Verantwortung dafür, dass das Programm am Ende stimmt, bleibt beim menschlichen Programmierer.

Der Klassiker unter den Methoden für die Codegenerierung ist die Programmsynthese aus logischen Spezifikationen. Genau wie eine informelle Beschreibung der Programmfunktionalität in natürlicher Sprache ist eine logische Spezifikation deklarativ, das heißt, sie kann viele verschiedene Ausführungen zulassen. Zum Beispiel fordert die Formel `x > 0` nur, dass der Wert von `x` positiv ist; ein Programm, das `x` auf `1` setzt, erfüllt die Anforderung genauso wie eines, das `x` auf `2` oder `500` setzt. Im Gegensatz zu natürlicher Sprache haben formale Logiken eine mathematisch präzise Semantik: Ist eine Anforderung einmal als logische Formel spezifiziert, gibt es keine Unsicherheit mehr darüber, wie die Anforderung zu verstehen ist.

Ein typisches Beispiel für Systeme, die sich gut für die automatische Synthese aus logischen Spezifikationen eignen, ist der AMBA-AHB-Buscontroller von ARM. Die Advanced Microcontroller Bus Architecture (AMBA) ist ein Kommunikationsstandard für den On-Chip-Nachrichtenaustausch zwischen Prozessorkernen, Cache Memory und DMA-Controllern. An den Bus sind sechzehn solcher Geräte als Master und weitere sechzehn als Slave angeschlossen; die Master melden dem Bus Kommunikationswünsche an bestimmte Slaves und bekommen vom Buscontroller entsprechende Zeitfenster zugewiesen. Der Standard wurde in eine logische Spezifikation übersetzt und wird seither oft als Benchmark für die Codegenerierung verwendet.

Die Spezifikation besteht aus fünfzehn Formeln, die beispielsweise ausdrücken, dass jeder Master, dem ein Zeitfenster zugeteilt wird, dieses auch tatsächlich verlangt hat und dass sich die Zeitfenster verschiedener Master niemals überlappen. Aus der logischen Spezifikation erzeugt ein Synthesewerkzeug wie das am CISPA Helmholtz-Zentrum für Informationssicherheit entwickelte BoSy (siehe ix.de/zgnu) automatisch korrekten Verilog-Code, der sich direkt in Hardware oder auf einem FPGA (Field Programmable Gate Array) realisieren lässt.

Abbildung 2 zeigt das Prinzip an einem einfacheren Beispiel, der Synthese eines Arbiters: Jeder Request ist mit einem Grant zu beantworten, aber niemals dürfen zwei Grants zugleich vergeben werden. Der von BoSy automatisch generierte Verilog-Code wählt die einfachste Lösung: Er wechselt einfach zwischen den Grants hin und her.

Im Detail finden sich drei logische Formeln:

```
(G ((r_0) -> (F (g_0))))
```

bedeutet, dass ein Wert `1` auf dem Eingang `r_0` (ein Request) die Ausgabe `g_0` (ein Grant) auf den Wert `1` setzt.

```
(G ((r_1) -> (F (g_1))))
```

meint das Gleiche für das Eingabesignal `r_1` und das Ausgabesignal `g_1`. Die Formel

```
G (!(g_0 && g_1))
```

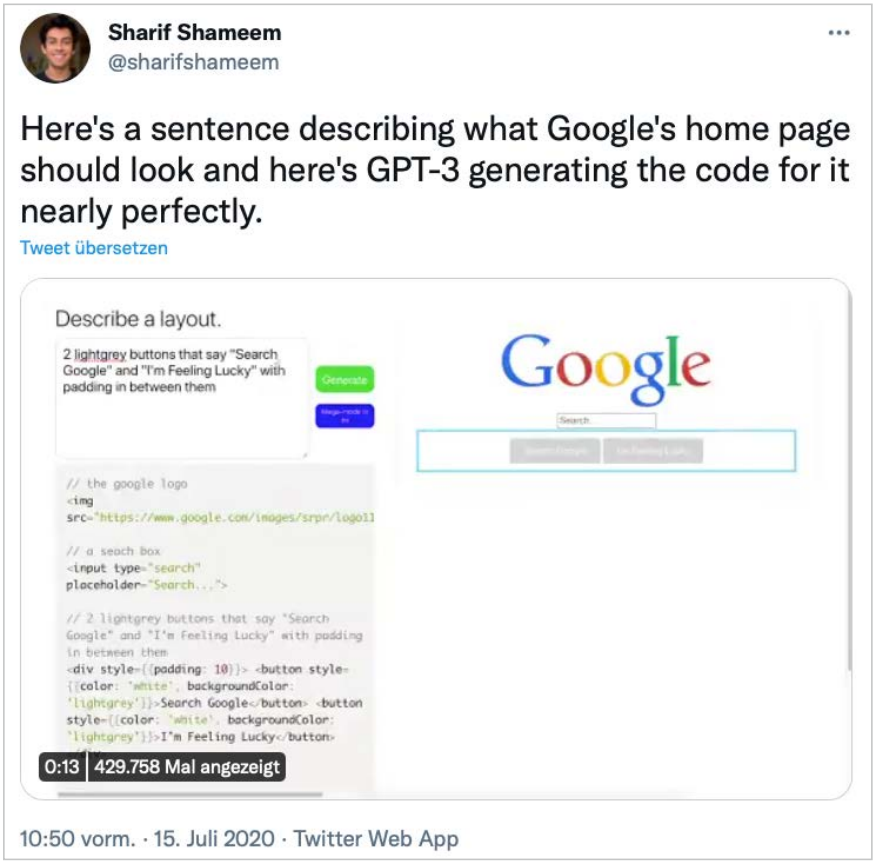
besagt, dass die Ausgabesignale `g_0` und `g_1` niemals gleichzeitig den Wert `1` haben dürfen. Der für diese Spezifikation generierte Verilog-Code ist rechts in der Abbildung 2 zu sehen. Die synthetisierte Lösung wechselt zwischen dem Ausgangssignal `g_0` und dem Ausgangssignal `g_1`.

Mathematisch gesehen ist die Programmsynthese aus logischen Spezifikationen perfekt: Der generierte Code ist beweisbar korrekt, das heißt, er erfüllt mit Sicherheit die gestellten Anforderungen, egal auf welchen Eingaben er angewendet wird. Diesem Vorteil steht allerdings ein hoher Rechenaufwand gegenüber. Deshalb lassen sich derzeit nur relativ kleine Programme synthetisieren. In der Praxis ist darüber hinaus der Aufwand für das logische Formalisieren der gewünschten Programmeigenschaften oft eine nicht zu unterschätzende Hürde.

Praktisch und schnell: Code aus Beispielen

Eine intuitive Art, das gewünschte Verhalten eines Programms festzulegen, ist die Angabe von Beispielen. Microsoft-Anwender kennen diese Form der Spezifikation durch das Flash-Fill-Feature (Blitzvorschau) in Excel. Gibt man in Excel Daten ein, die ein bestimmtes Muster aufweisen, etwa die Trennung vollständiger Namen

Quelle: Twitter



Automatisiertes Webdesign: GPT-3 kann Websites aus kurzen englischsprachigen Beschreibungen erzeugen (Abb. 1).

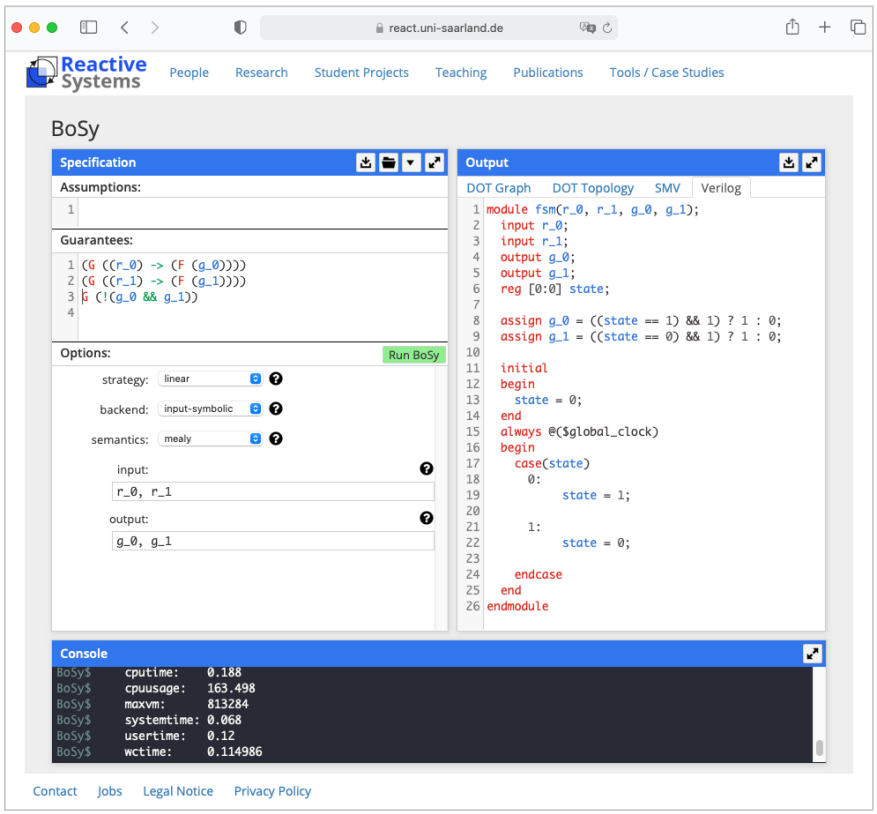
Listing: Von GPT-3 generierter Python-Code erscheint nur auf den ersten Blick korrekt

```
n = int(input("Enter n: "))
m = int(input("Enter m: "))
def gcd(x, y):
    return x if x == y else gcd(y, x % y)
print(f"The greatest common divisor of {x} and {y} is {gcd(x, y)}" )
```

in Vor- und Nachnamen, dann kann Excel dieses Muster erkennen und eine Tabelle automatisch vervollständigen.

Hinter den Kulissen verwendet Excel eine Programmiersprache für String-Transformationen mit Operationen für reguläre Ausdrücke, Fallunterscheidungen, Schleifen und Ähnliches. Für das automatische Vervollständigen der Tabelle in Abbildung 4 generiert Excel ein Programm in dieser Sprache, das aus der ersten Spalte, in der die vollständigen Namen aufgelistet sind, eine zweite Spalte mit den Nachnamen erzeugt. Die vollständigen Namen können dabei entweder in der Form „Vorname Nachname“ oder in der Form „Nachname, Vorname“ angegeben sein.

Das Programm extrahiert dazu aus einem gegebenen String das Wort vor dem Komma oder – falls der String kein Komma enthält – das letzte Wort im String. Das Flash-Fill-Feature erzeugt dieses Programm in weniger als 0,1 Sekunden aus den vom Nutzer vorgegebenen Beispielen. Der Codegenerierungsalgorithmus berechnet dazu zunächst für jede Beispielzeile eine Menge von Programmen, deren Ergebnis diesem Beispiel entspricht. Wenn ein Programm darunter ist, das in allen Beispielen funktioniert, dann ist dieses Programm die gesuchte Lösung. Andernfalls werden die Beispiele in verschiedene Klassen partitioniert, die jeweils eine gemeinsame Lösung haben. Hier im Beispiel haben „Anna-Maria Schmidt“ und „Julia Richards“ dieselbe Lösung (das letzte Wort), „Brinkmann, Klaus“ dagegen eine andere (das Wort vor dem Komma). Die verschiedenen Klassen werden dann mithilfe einer Fallunterscheidung kombiniert.



Automatische Synthese von Verilog-Code aus einer logischen Spezifikation: Links stehen drei Formeln, rechts ist der für diese Spezifikation generierte Verilog-Code zu sehen (Abb. 2).

Das Prinzip hinter Flash Fill ist die Syntax-Guided Synthesis (SyGuS). Beim Einsatz von SyGuS besteht der zentrale Schritt darin, die Menge aller möglichen Programme mithilfe einer domänenspezifischen Sprache (DSL) zu formalisieren. Frameworks wie PROSE von Microsoft berechnen dann aus den gegebenen Beispielen ein möglichst einfaches, in der DSL ausgedrücktes Programm, das sich auf den Beispielen richtig verhält.

Anschließend lässt sich das Programm auf neuen Daten ausführen. SyGuS ist in vielen Anwendungen einsetzbar: bei der Analyse von Logfiles, beim Extrahieren von Daten aus PDF-Dokumenten und beim Codieren von Datenbankabfragen als SQL-Abfragen. SyGuS spielt seine Stärken immer dann aus, wenn es sich um einen überschaubaren und kompakt als DSL beschreibbaren Aufgabenbereich handelt.

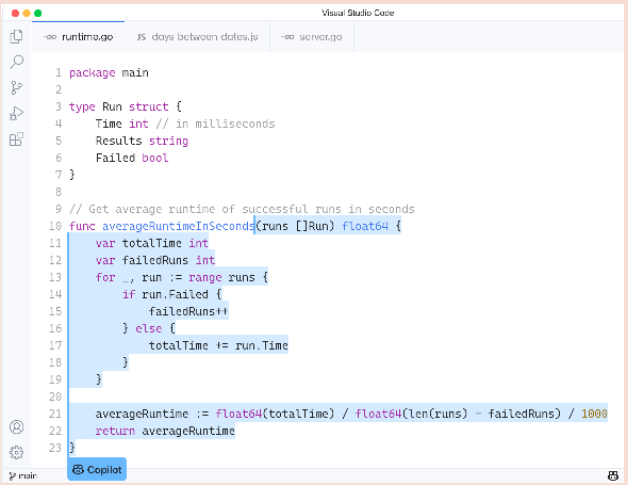
Hey Siri, schreib ein Python-Programm!

Der Erfolg von Sprachassistenten wie Siri und Alexa legt die Idee nahe, auch Pro-

GitHub Copilot: KI schreibt Code

Kurz vor Redaktionsschluss dieser Ausgabe haben GitHub und OpenAI den GitHub Copilot vorgestellt. Der „AI Pair Programmer“ schlägt beim Programmieren nicht nur einzelne Codezeilen, sondern auch komplette Funktionen vor. Grundlage der Vorschläge ist der restliche Inhalt der Datei, insbesondere auch der natürlichsprachliche Kontext wie Funktions- und Variablennamen, Kommentare und Docstrings. In vielen Demonstrationen genügen Funktionskopf und Kommentar, um die komplette Funktion zu vervollständigen.

Laut GitHub trifft der Copilot in 43 % der Fälle auf Anhieb die richtige Lösung, bei zehn Versuchen soll sich die Quote auf 57 % erhöhen. Die Vorschläge liefert das von OpenAI entwickelte Codex-Modell, das auf Milliarden öffentlicher Programm- und Textzeilen trainiert wurde. Entsprechend funktioniert der Copilot besonders gut für populäre Sprachen wie Python und JavaScript. Derzeit steht der GitHub Copilot als Erweiterung für Visual Studio Code nur einer kleinen, ausgewählten Gruppe von Testern zur Verfügung. Nach erfolgreicher Testphase will GitHub eine kommerzielle Version anbieten.



Der GitHub Copilot schreibt ganze Funktionen (Abb. 3).

grammieraufgaben in natürlicher Sprache zu beschreiben. Neuronale Netze haben in den letzten Jahren neue Maßstäbe beim Verarbeiten natürlicher Sprache gesetzt. Im Blickpunkt der aktuellen Forschung stehen dabei zunehmend allgemeine Sprachmodelle. Dabei werden neuronale Netzwerke auf einer generischen Aufgabe wie dem Vervollständigen eines Textes vortrainiert (Pre-Training).

Das Netzwerk lernt auf diese Weise den Aufbau und die Struktur natürlicher Sprache. Anschließend wird das vortrainierte Modell in einem weiteren Training auf eine Anwendung spezialisiert (Finetuning). Sprachmodelle wie GPT-3 gehen sogar so weit, dass gar kein ausgedehntes zweites Training mehr nötig ist. GPT-3 lässt sich aus dem Stand (Zero-Shot), mit einem einzelnen Beispiel (One-Shot) oder mit einigen wenigen Beispielen (Few-Shot) für neue Anwendungen einsetzen. Die wenigen Beispiele dienen als eine Art Blaupause für den neu zu generierenden Text.

Um Python-Code zu generieren, genügt ein einziges Paar von einer natürlichsprachlichen Beschreibung und dazugehörigem Python-Code, um GPT-3 darauf vorzubereiten, anschließend den Python-Code für den größten gemeinsamen Teiler zu generieren (siehe Listing). Die Größe von GPT-3 (175 Milliarden Parameter) sowie die Menge und Vielfalt der Daten, auf denen trainiert wurde (300 Milliarden Token), ermöglichen das Few-Shot-Lernen. Die Daten basieren zu einem großen Teil auf denen von Common Crawl, die Petabytes im Web gesammelter Daten umfassen (siehe ix.de/zgnu). Da Common Crawl auch Daten von Plattformen wie GitHub und Stack Overflow sammelt, beherrscht GPT-3 die Syntax verschiedenster Programmiersprachen, ohne dafür noch einmal gesondert trainieren zu müssen.

Die große und vielfältige Datenmenge, auf der GPT-3 trainiert wurde, ermöglicht es auch, Aufgaben zu stellen, die mathematische Kenntnisse oder schlicht Allgemeinwissen erfordern. So kann GPT-3 erfolgreich Python-Code generieren, der überprüft, ob eine Zahl eine Primzahl oder ob ein Jahr ein Schaltjahr ist – und das ohne eine Definition, was darunter zu verstehen ist.

Der Anschein, dass die KI die Bedeutung der Begriffe „versteht“, ist jedoch trügerisch. Eine minimale Variation der Aufgabenstellung, etwa ob statt der gegebenen Zahl die Zahl „minus vier“ eine Primzahl ist, lässt GPT-3 noch scheitern. Und auch wenn die Programmieraufgabe erfolgreich gelöst wird, ist Vorsicht geboten.

Aufmerksamen Leserinnen und Lesern mag schon aufgefallen sein, dass sich in den Code für den größten gemeinsamen

	A	B
1	Anna-Maria Schmidt	Schmidt
2	Brinkmann, Klaus	Brinkmann
3	Julia Richards	Richards
4	Don Knuth	
5	Max Ferdinand	
6	Huber, Paul	

	A	B
1	Anna-Maria Schmidt	Schmidt
2	Brinkmann, Klaus	Brinkmann
3	Julia Richards	Richards
4	Don Knuth	Knuth
5	Max Ferdinand	Ferdinand
6	Huber, Paul	Huber

Das Flash-Fill-Feature in Microsoft Excel vervollständigt automatisch die obere zur unteren Tabelle (Abb. 4).

Teiler im Listing Fehler eingeschlichen haben: Er ruft die Python-Funktion `gcd` fälschlicherweise mit den Variablen `x` und `y` statt `n` und `m` auf. Problematisch ist außerdem die Bedingung `x == y`, die nach der Rekursionsfolge `(5,10) -> (10, 5) -> (5, 0)` zu einem ZeroDivisionError bei der Modulo-Operation führt, statt den erwarteten Wert `5` zurückzugeben. Korrekt wäre an dieser Stelle die Bedingung `y == 0`. Dass Sprachmodelle wie GPT-3 Programmierfehler machen, ist nicht weiter überraschend. Ein offensichtlicher Grund ist, dass die Trainingsdaten aus dem Web stammen. Wenn die Trainingsdaten Fehler enthalten, dann ist auch der generierte Code falsch. Das logische Denken, mit dem man solche Fehler erkennen könnte, fehlt heutigen Sprachmodellen noch völlig.

Fazit

Sorgen, dass die KI menschliche Softwareentwickler bald überflüssig machen wird, muss man sich wohl nicht machen. Der Weg zu einer KI, die für beliebige Programmieraufgaben zuverlässig Code hoher Qualität generiert, ist noch weit. In spezialisierten Bereichen leistet die KI-basierte Codegenerierung aber heute schon Erstaunliches. Die Programmsynthese aus logischen Spezifikationen ist insbesondere dann sinnvoll, wenn es um Anwendungen geht, bei denen Programmierfehler teuer oder gefährlich sind, etwa im Bereich der cyber-physischen Systeme.

Die Codegenerierung aus Beispielen mit SyGuS skaliert noch nicht bis zum Entwickeln vollständiger Applikationen, kann aber das Erstellen kniffliger Programmstücke wie SQL-Abfragen vereinfachen. Die Codesynthese aus Sprachmodellen schließlich kann noch nicht mit den Qualitätsansprüchen der klassischen Methoden mithalten, ist aber gerade im schnellen Prototyping oft verblüffend kreativ. Der Einsatz in Designtools und Low-Code-Entwicklungsplattformen ist erfolgversprechend (siehe Artikel „Code-Diktat“ auf S. 56).

Für die Forschung besteht die Herausforderung jetzt darin, die Erfolge der Sprachmodelle bei der Verarbeitung natürlicher Sprache auf die Programmiersprachen mit ihrer sehr viel präziseren Semantik zu übertragen. Welches Potenzial in diesem Weg steckt, lässt der Erfolg des etablierten Code-Completion-Werkzeugs Tabnine erahnen (siehe Artikel „KInematik“ auf S. 44). Genau wie der GitHub Copilot macht Tabnine Programmierern während des Editierens Vorschläge, wie die Codezeile weitergehen könnte. Tabnine setzt auf GPT-2 auf, dem Vorgänger von GPT-3. Das Tool trainiert auf Millionen von Open-Source-Programmen und zusätzlich auf der individuellen Codebasis des Programmierers. Tabnine „kennt“ dadurch nicht nur die Syntax verschiedenster Programmiersprachen, sondern auch gängige Ausdrücke, typische Variablennamen und die richtige Verwendung von Bibliotheken.

Künftig sollte es möglich sein, die einfache Benutzbarkeit und das Hintergrundwissen der Sprachmodelle mit der mathematischen Präzision der Programmsynthese aus logischen Spezifikationen zu verbinden. Hybride Algorithmen, die Deep Learning mit klassischen Methoden der Logik kombinieren, könnten helfen, die komplexe Codesynthese aus logischen Spezifikationen besser in den Griff zu bekommen. Darüber hinaus könnten Sprachmodelle Nutzer beim Entwerfen und Verstehen der logischen Spezifikationen unterstützen. Denn selbst wenn die KI dem Menschen irgendwann einmal das Programmieren komplett abnehmen sollte, muss man ihr immer noch sagen, welche Funktion die Software nun eigentlich erbringen soll. (nb@ix.de)

Quellen

Synthesetool BoSy, Common Crawl und GitHub Copilot: ix.de/zgnu

Prof. Dr. Bernd Finkbeiner

lehrt Informatik an der Universität des Saarlandes und forscht im Bereich automatischer Methoden für die Programmverifikation und -synthese am CISPA Helmholtz-Zentrum für Informationssicherheit in Saarbrücken.

Frederik Schmitt

promoviert im Bereich Deep Learning und Programmsynthese an der Universität des Saarlandes und arbeitet als wissenschaftlicher Mitarbeiter am CISPA Helmholtz-Zentrum für Informationssicherheit in Saarbrücken.