Snap!GPT – Bausteine für generative künstliche Intelligenz

Mönig, J.
SAP, Walldorf

DOI: 10.18420/ibis-02-01-04

Zusammenfassung

„Und dann?“

Um diese Frage geht es bei generativer künstlicher Intelligenz. Darum, was als nächstes kommen könnte: Das nächste Wort in einer Geschichte, der nächste Ton in einem Lied, der nächste Strich in einer Zeichnung. In diesem Beitrag zeige ich, wie man ein „Next Token Prediction“ System in Stil von ChatGPT in Snap! selber programmieren kann. Gibt man ihm ein paar Geschichten, dann lernt es, eigene Text zu schreiben. Gibt man ihm ein paar Kinderlieder, dann improvisiert es eigene Melodien, und zeichnet man ihm etwas vor, dann versucht es, selbst etwas zu kritzeln.

Man selbst lernt bei dieser Aktivität, Wörter, Musiknoten und Striche zu kodieren, in einem Datenmodell zu strukturieren, und kontextbezogen abzufragen. Das lässt sich mit zwei eigenen Funktionsblöcken, ein paar Schleifen, Variablen und Bedingungen bewerkstelligen, und eignet sich z.B. für den Informatikunterricht ab der 7. Klasse.

Einleitung

Im letzten IBIS-Heft hat Michael Hielscher mit SoekiaGPT ein auf n-Grammen basierendes didaktisches Sprachmodell vorgestellt. Aus 25 Grimm’schen Märchen erzeugt es ähnlich lautende, aber unsinnige Texte. Seine Idee, Funktionsweisen und Eigenschaften generativer künstlicher Intelligenz mit simplen Markov-Ketten und dem Halluzinieren von Märchen (sic!) zu erklären, begeistert mich. Ich habe mich gefragt, ob es möglich ist, das Prinzip noch weiter zu vereinfachen, damit Lernende so etwas wie ChatGPT selber programmieren können, gar in einer grafischen Programmiersprache wie Snap!, an der ich mitentwickle.

Entstanden ist ein kleines Programm mit nur zwei eigenen Blöcken und wenigen Skripten. Man kann damit lustige Texte erzeugen, und nebenbei einiges über das Programmieren mit größeren Datenmengen lernen. Und wenn man die Bausteine ein bisschen anders zusammenfügt, passiert sogar etwas Erstaunliches: Dann lernt das Programm, auch andere Reihen, die nicht aus Wörtern bestehen, fortzusetzen, und damit z.B. Musik oder Bilder zu machen.

Das Snap!GPT Projekt hat drei Teile: Es werden Beispieldaten (hier: Märchen) geladen und zerlegt, daraus entsteht ein Sprachmodell, eine Art Datenbank. Anschließend wird eine Funktion entwickelt, die aus dem Datenmodell einen Eintrag (hier: ein Wort) heraussucht, der eine bestehende Reihe (hier: vorhergehende Worte) plausibel fortsetzt. Zum Schluss kann das Ganze noch interaktiv für Benutzer gestaltet werden.

Snap! ist eine grafische, blockbasierte Programmiersprache, die gemeinsam von der Universität Berkeley und SAP für den Informatikunterricht an Schulen und Universitäten entwickelt wird. In den USA wird damit das vom College Board zertifizierte „Beauty and Joy of Computing“ Curriculum unterrichtet. Auch in Deutschland gibt es eine wachsende Community. Snap! verwendet dieselbe Baustein-Grammatik wie Scratch, das erleichtert den Einstieg für Lernende, die schon mal mit Scratch programmiert haben. Gleichzeitig sind in Snap! viele Funktionen schon eingebaut, die beim Erstellen eines statistischen Sprachmodells hilfreich sind. Dadurch bleibt das hier vorgestellte Projekt von Umfang her überschaubar. Snap! läuft im Web Browser, ist kostenlos und quelloffen.

Daten analysieren

Dateien verschiedener gebräuchlicher Formate (txt, csv, json) kann man mit der Maus in das Snap!-Fenster ziehen, um sie zu laden. Oder man wählt sie im Dateimenü mit „Importieren“ aus. Snap! legt eine globale Variable mit dem Dateinamen an und weist ihr deren Inhalt als Wert zu. Strukturierte Daten (z.B. csv oder json) werden dabei automatisch in Listen- bzw. Tabellenform gebracht, reine Texte bleiben unverändert am Stück. Weil mir Hielschers Beispiel mit den Märchen so gut gefällt, habe ich mir ebenfalls ein paar Webseiten mit Märchen der Brüder Grimm herausgesucht, und davon 30 Geschichten als txt-Datei heruntergeladen, bzw. aus dem Browser in lokale Textdateien kopiert. Die Einzeltexte habe ich zu einer einzigen großen Datei zusammengefügt, und diese in Snap! geladen.

Mit dem „trenne” Block kann ein Text in verschiedene Einzelteile zerlegt werden. Ich habe die gesammelten Märchen nach „Wort” getrennt, um eine große Liste mit der Reihenfolge aller Wörter zu erhalten (siehe Abb. 1).

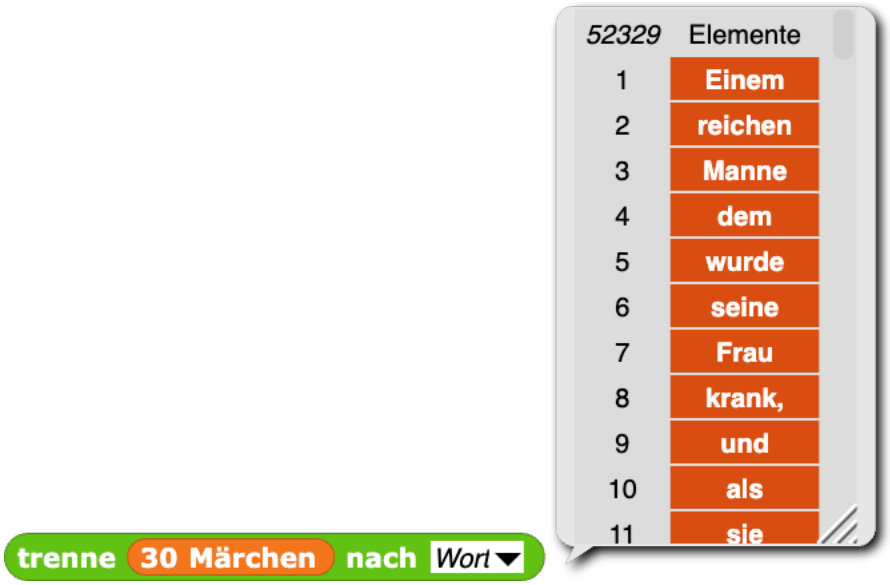

Abbildung 1: Mit dem „trenne“ Block in Snap! aus einem Text eine Wörterliste gewinnen

Der Clou des Sprachmodells ist die Zerlegung dieser langen Wörterliste in mehrere Versionen mit vielen kleinen Einträgen bestehend aus Wörterpaaren, Tripeln, Vierer- und Fünferketten, sogenannten n-Grammen. Dafür bietet sich eine Funktion mit zwei Parametern an, „n” für die gewünschte Länge der zu bildenden Ketten, und „Korpus” für die Ausgangsliste aller aufeinanderfolgenden Wörter. In Snap! habe ich dafür einen kleinen Funktionsblock definiert (siehe Abb. 2).

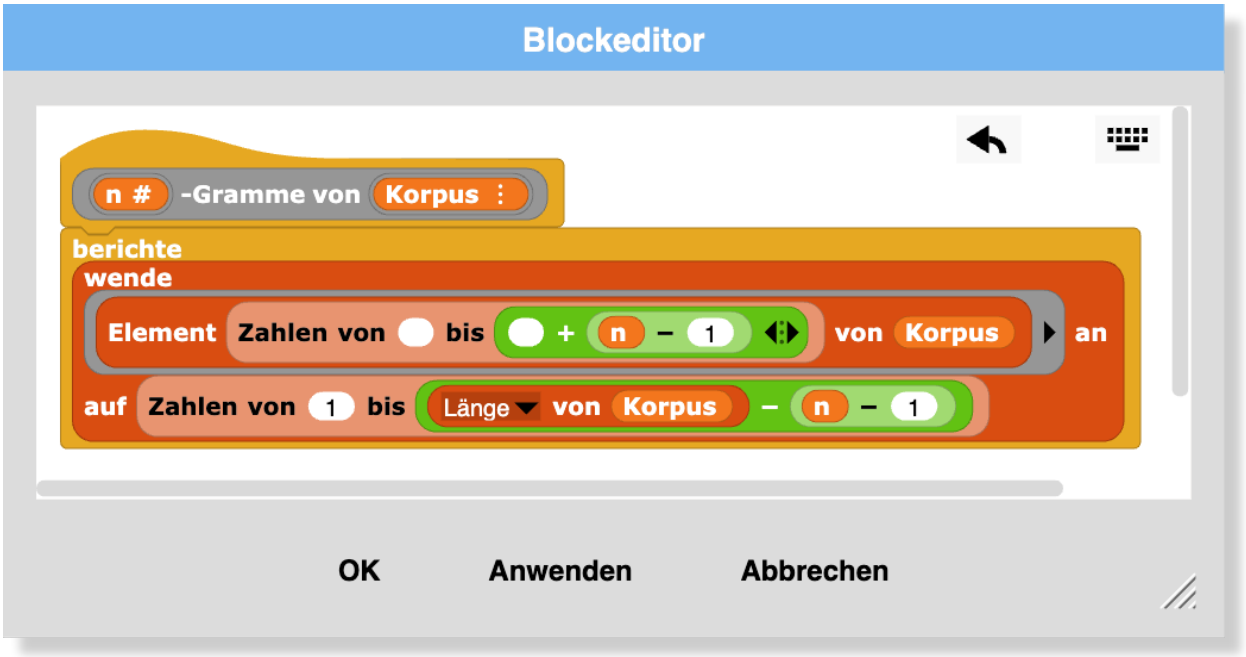

Abbildung 2: Beispiel für eine benutzerdefinierte „n-Gramme“ Funktion in Snap!

Man kann n-Gramme in Snap! auf verschiedene Weisen implementieren, z.B. imperativ mit Befehlen und sogar hyperdimensional mit Vektoroperationen. Ich habe mich für eine funktionale Variante entschieden, weil sie am expressivsten ist, d.h. am wenigsten Blöcke braucht, und die Aufgabe recht schnell erledigt. Wer etwas längere Laufzeiten in Kauf nimmt, kann natürlich genauso gut mit Schleifen und Variablen arbeiten.

Als nächstes sollen n-Gramme für Wörterketten aus einem bis fünf Gliedern gebildet, und in einem Sprachmodell gespeichert werden. Dazu habe ich eine weitere globale Variable namens „Modell" angelegt. Ihr soll eine Liste zugewiesen werden, deren Elemente jeweils das Resultat der entsprechenden „n-Gramme" Funktion sind. An erster Stelle steht eine Liste aller Einzelwörter, an zweiter Stelle alle Wörter-Paare, das dritte Element ist eine Liste aller Dreier-Ketten, und so weiter. Das kann man z.B. mit einer Schleife und einer Laufvariable erreichen. Ich habe mich wiederum für ein kurzes funktionales Skript entschieden (siehe Abb. 3).

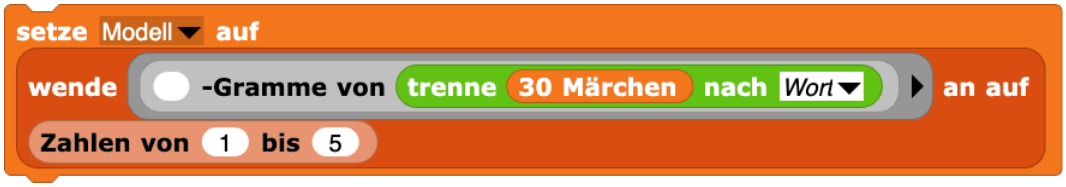

Abbildung 3: Das fertige Snap! Skript für das „Trainieren" eines statistischen Sprachmodells

Es reicht, wenn dieses Skript einmal mit der Maus angeklickt und ausgeführt wird. Dann dauert es eine (hoffentlich nur kurze) Weile, und das Sprachmodell ist einsatzbereit. Um das zu überprüfen, kann das Modell mit dem „Element" Block für verschiedene n-Werte von 1 bis 5 abgefragt werden. Es sollte jeweils eine Liste bzw. Tabelle angezeigt werden, deren Spaltenanzahl mit ihrer Position im Modell übereinstimmt (siehe Abb. 4).

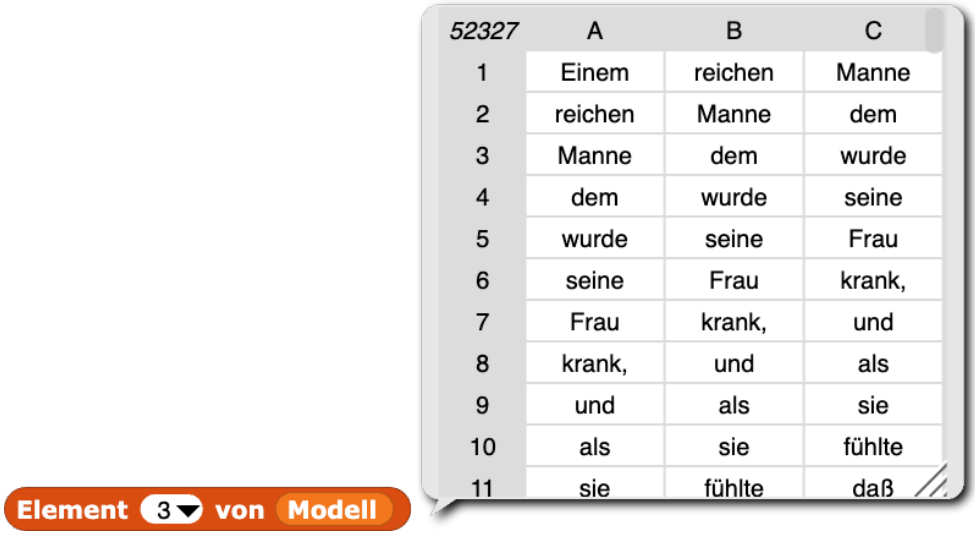

Abbildung 4: Inspizieren des Sprachmodells in Snap!
Die Anzahl der Spalten sollte dem jeweiligen Index entsprechen

Üblicherweise wird nun die Häufigkeit jeder Wortfolge ermittelt und im Modell gespeichert. Zur Vereinfachung habe ich auf diesen Schritt verzichtet, und schreibe stattdessen eine mehrfach angetroffene Wortfolge auch mehrfach ins Modell. Wenn später ein Text zufällig fortgesetzt werden soll, werden häufigere Wortfolgen auch mit einer höheren statistischen Wahrscheinlichkeit gezogen. Hier bietet sich an, die Wahrscheinlichkeitsberechnung aus der Mathematik im Unterricht aufzugreifen.

Tipp: Eine Variable als "nicht persistent" markieren

Wenn man ein Snap! Projekt speichert, dann bleiben auch die Werte aller globalen Variablen erhalten. Das hat den Vorteil, dass man z.B. die Datei mit den Märchen nicht jedes Mal wieder neu importieren muss, wenn man an dem Projekt weiterarbeitet. Ebenso ist es hier mit dem Modell aus n-Grammen. Allerdings kann es sein, dass dadurch das Projekt so große Datenmengen anhäuft, dass es das Speicherlimit für die Cloud übersteigt. Um das zu verhindern können einzelne Variablen, hier z.B. das Modell, in der Palette in ihrem Kontextmenü als "nicht persistent" gekennzeichnet werden. Natürlich muss man dann beim nächsten Öffnen des Projekts wieder das Skript mit den n-Grammen (Abb. 3) ausführen, um das Model wiederherzustellen.

Eine Reihe fortsetzen

Als nächstes geht es darum, angefangene Sätze mithilfe des Sprachmodells zu vervollständigen. Konkret heißt das, dass im Modell nach einem Wort gesucht wird, das man einer Liste aus vorherigen Wörtern hinzufügen kann, ohne dass dabei Kauderwelsch entsteht. Das ist eigentlich eine schwierige Sache, schließlich muss man dafür viel über Sprache, Grammatik, Satzteile, Fälle, Formen und Ausnahmen wissen, und sich auch mit Satzzeichen und Groß- und Kleinschreibung auskennen. Von alledem weiß das Modell jedoch nichts. Es hat aber einen großen Fundus an Wörtern, die schon mal hinter anderen Wörtern gestanden haben. Diese Korrelationen genügen, um einigermaßen echt klingende Sätze zu bilden.

Der Markov-Kette-Textgenerator-Algorithmus nimmt die jeweils letzten „n" Wörter einer angefangenen Geschichte als Kontext, und sucht im Modell zuerst alle Einträge heraus, die genau ein Wort länger sind als der Kontext. Von diesen Einträgen behält er diejenigen, die bis auf das letzte Wort dem Kontext entsprechen. Aus den so ermittelten Kandidaten wird einer zufällig ausgewählt, und dessen letztes Wort als Ergebnis zurückgegeben. Findet sich kein Kandidat, wird die Suche so lange mit einem jeweils kürzeren Kontext wiederholt, bis schließlich ein zufälliges Wort aus der 1-Gramm Liste zurückgegeben wird.

Üblicherweise wird die Kontextlänge für die Suche hin und wieder zufällig gewählt, damit das System „kreativer" wird, und nicht passagenweise die verhackstückten Originaltexte wieder zusammensetzt. Ich habe hier auf dieses Detail verzichtet, damit das Nachprogrammieren noch etwas einfacher wird. Außerdem finde ich es interessant, wenn man aus den erzeugten Sätzen noch ihren Ursprung erkennen kann, d.h. welchem Märchen sie entnommen wurden.

Für die Implementierung als Funktionsblock in Snap! habe ich dieses Mal eine klassische imperative Variante gewählt (siehe Abb. 5).

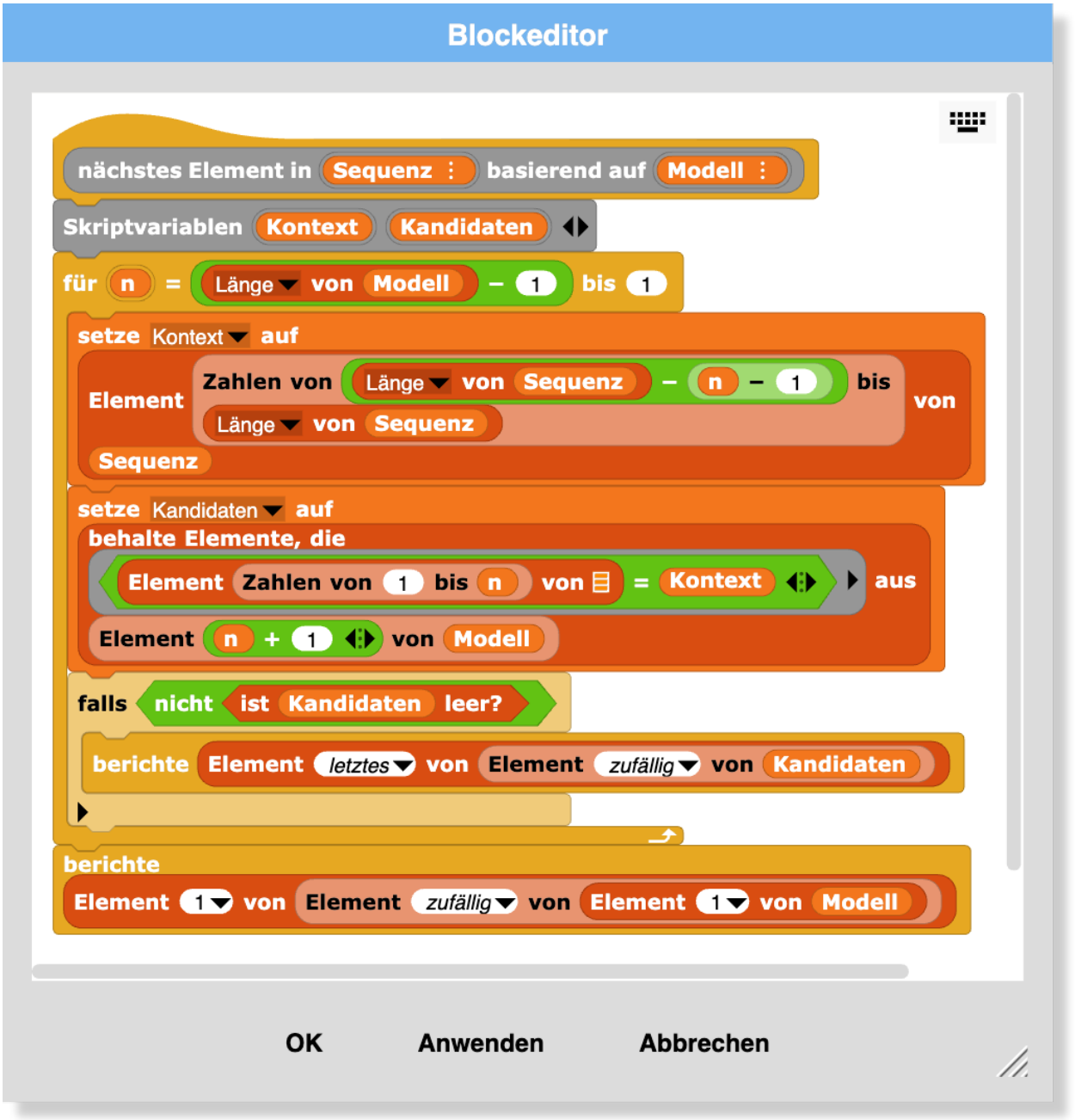

Abbildung 5: Beispiel für eine benutzerdefinierte Abfragefunktion des vereinfachten Markov-Ketten Textgenerators in Snap!

Man kann diesen Block testen, indem man ihm eine Liste aus ein paar Worten und ein Datenmodell übergibt, und dann durch Mausklick ausführt (siehe Abb. 6).

Abbildung 6: Testen des Abfrageblocks in Snap! anhand einer Beispiel-Liste

Tipp: Texte und Listen vergleichen

Snap! unterstützt das Modellieren von Projekten rund um Sprache, indem Listen und Tabellen auf ihre strukturelle Gleichheit getestet werden können. Zwei oder mehrere Listen können unabhängig von ihrem Rang (der Tiefe ihrer Verschachtelung) direkt auf Gleichheit aller ihrer Elemente getestet werden, ohne dass man dafür, wie in anderen Programmiersprachen, die Elemente einzeln anfassen muss. Außerdem ignoriert Snap! beim Vergleichen von Texten standardmäßig die Groß- und Kleinschreibung. Beides folgt dem Vorbild der Programmiersprache LOGO, in der es ursprünglich genau um ebensolche wort- und satzbasierten Sprachprojekte ging.

Ein interaktives Programm gestalten

Aus den vorhandenen Bausteinen kann man jetzt schon ein kleines Programm gestalten, das den Benutzer einbezieht. Dazu habe ich eine weitere globale Variable namens „Geschichte" angelegt, und mir folgende Benutzerführung überlegt: Wenn die grüne Flagge angeklickt wird, soll der Benutzer aufgefordert werden, die ersten Worte eines neuen Märchens einzugeben. Das Programm generiert daraus eine Wörterliste, weist sie der „Geschichte"-Variable zu, ergänzt die Geschichte um das nächste Wort, und gibt sie als Fließtext aus. Jedes Mal, wenn der Benutzer die Leertaste drückt, wird ein weiteres Wort der Geschichte hinzugefügt, und die Ausgabe aktualisiert.

Das fertige Programm besteht aus vier kleinen Skripten, dem oben beschriebenen Modell-Skript (Abb. 3), und einem interaktiven Textgenerator (siehe Abb. 7).

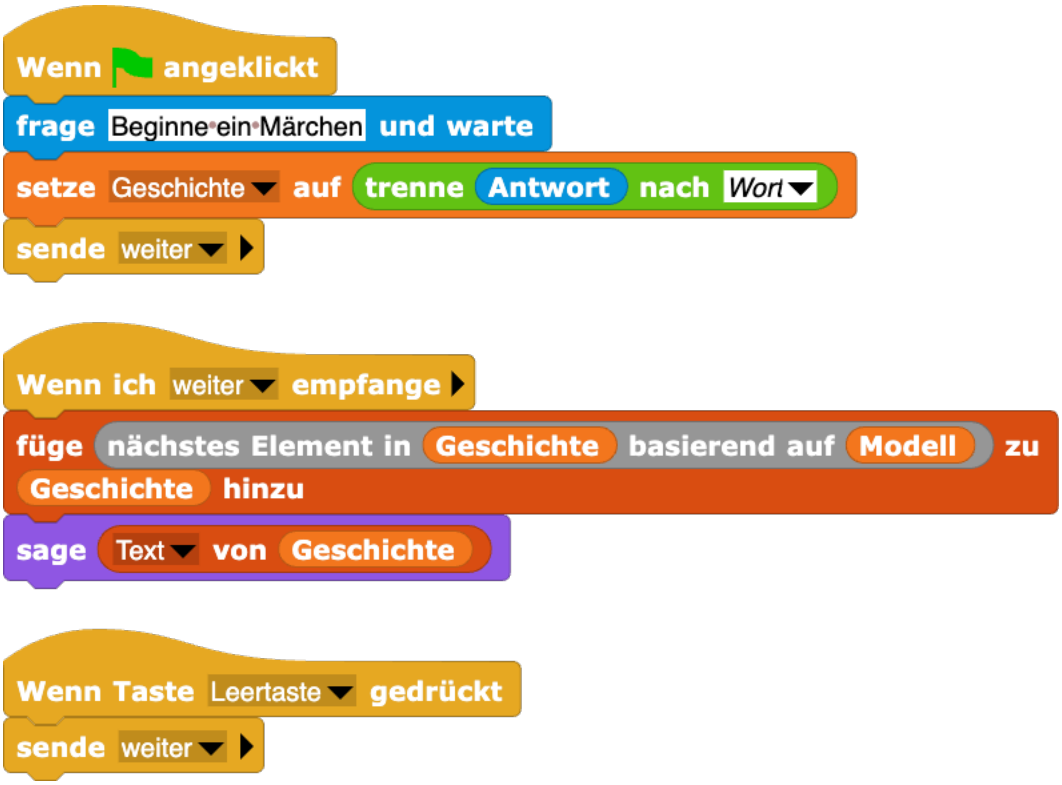

Abbildung 7: Der fertige Textgenerator in Snap!

Wenn man es ausprobiert und auf die Aufforderung, ein neues Märchen zu beginnen, z.B. „Es war einmal“ eingibt, dann entstehen durch wiederholtes Drücken der Leertaste jedes Mal andere Sätze und Geschichten, wie z.B. diese hier:

```
Es war einmal ein König, der hatte eine Frau mit goldenen Haaren, und sie war so unbarmherzig daß sie die arme Rapunzel in eine Wüstenei brachte, wo sie in großem Jammer und Elend leben mußte.
```

Tipp: Eine Wörterliste als Texte ausgeben

Weil das Sprachmodell auf einzelnen Wörtern beruht, ist der Wert der Variable "Geschichte” ebenfalls eine Wörterliste. Um diese als Text auszugeben, müssen alle Wörter miteinander verbunden und dazwischen Leerzeichen eingefügt werden. Das kann man in Snap! auf verschiedene Arten bewerkstelligen. Die einfachste Methode ist, den “Länge von Liste” Block zu nehmen, und anstatt “Länge” die Auswahl auf “Text” zu setzen.

Tipp: Die Bühne etwas “sagen” lassen

In Snap! hat jedes Objekt einen “sage” Block, mit dem es Informationen in einer Sprechblase ausgeben kann, auch die Bühne. Die Sprechblase der Bühne ist größer als die der anderen Objekte und kann einen längeren Ausschnitt der Geschichte anzeigen. Aus allen Sprechblasen kann man durch Rechtsklick den Inhalt als Datei exportieren, z.B. wenn man die erzeugte Geschichte jemandem senden möchte.

Das Modell anpassen

Manchmal kommt es vor, dass das Programm ein Märchen mehr oder weniger ganz abschreibt, oder aber, dass es in eine Art Schleife fällt, und einen Absatz ständig wiederholt. Professionelle Sprachmodelle bieten Einstellungen an, um solche und andere Ungeschicklichkeiten zu vermeiden. Sind die erzeugten Texte zu nah am Original, dann kann man den Zufallsgrad heraufsetzen, mit dem das jeweils nächste Wort bestimmt wird. Das Programm wird dann kreativer, aber auch weniger zuverlässig. In meinem kleinen Snap!GPT Programm lässt sich ähnliches erreichen, wenn man die n-Gramm-Listen im Modell anstatt mit den Zahlen von 1 bis 5 nur bis zur Länge von 2 oder 3 erstellt. Dann verringert sich das sog. Kontextfenster, und der Textgenerator wird abwechslungsreicher, aber auch „fahriger“, und seine Sätze ergeben manchmal keinen Sinn mehr. Auf jeden Fall lohnt es sich, mit diesem Parameter zu spielen, und verschiedene Varianten auszuprobieren.

Häufige Wiederholungen kann man in professionellen Textgeneratoren dadurch verringen, dass man einen „Bestrafungswert“ für Doppelungen innerhalb eines bestimmten Bereichs festlegt. In Snap!GPT könnte man z.B. nur Ergebnisse zulassen, die noch nicht innerhalb der letzten 10 oder 20 Wörter enthalten sind. Ähnliche „Bestrafungsgewichte“ kann man problematischen Wörtern und Passagen erteilen, damit z.B. in Gute -Nacht Geschichten für Kinder keine Schimpfworte vorkommen, und allgemein verbotene Inhalte, z.B. Beleidigungen, rassistische, oder pornografische Formulierungen unterbleiben. So eine Zensur kann z.B. mit einer Stoppwortliste bewirkt werden.

Mich hat – aus didaktischer Sicht positiv - überrascht, dass sich solche aus der gesellschaftlichen Diskussion über generative künstliche Intelligenz bekannten Probleme schon in kleinem Rahmen zeigen. Das gibt Anlass zu Reflexion und Diskussion, nicht nur in Informatikunterricht.

Transfer: Das Prinzip verallgemeinern

Einen Textgenerator selbst zu programmieren macht Spaß und regt zum Nachdenken an. Seine Kreationen sind einerseits verblüffend realistisch, andererseits kompletter Nonsens, durchsetzt mit eindeutig plagiierten Passagen. Man kann anstelle von Märchen andere Vorlagen verwenden, und dann Texte „im Stil” dieser anderen Quellen generieren. Das funktioniert sogar (natürlich!) mit fremdsprachigen Texten. In Snap! kann man beliebig viele Dateien als Variablen

importieren, und im Projekt z.B. diejenige Variable ersetzen, aus der das Sprachmodell erstellt wird. Man kann auch mehrere Modelle mit verschiedenen Inhalten im gleichen Projekt verwalten, und jeweils auswählen, aus welchem das nächste Element ermittelt werden soll.

Eine spannende Erkenntnis ist, dass es überhaupt nicht auf den Inhalt der verarbeiteten Quelldaten ankommt, sondern nur darauf, dass sie in einer bestimmen Reihenfolge sind. Solche Sequenzen gibt es zuhauf, nicht nur in Texten, sondern überall, wo „Listen" im weitesten Sinne Verwendung finden. Einkaufslisten: Leute, die dies gekauft haben, kaufen häufig auch jenes. Playlists: Wer sich das anhört oder ansieht, ist vielleicht auch an jenem interessiert. Krankheitsverläufe: Wer diese Symptome hat leidet oft auch unter jenen. Spielpartien: Nach diesem Schachzug haben erfolgreiche Spieler oft jenen gewählt. Die Bausteine, mit denen Beispieldaten in n-Gramme zerlegt, und unvollständige Reihen plausibel ergänzt werden, müssten folglich auf beliebige andere Zusammenhänge anwendbar sein.

Buchstaben statt Wörter

Statt nach Wörtern kann man die Märchen auch nach Buchstaben trennen. Dazu muss man nur in den beiden „trenne" Blöcken im Modellskript (Abb. 3) und im grüne-Flagge Skript (Abb. 7) die Option „Buchstaben" auswählen. Außerdem muss man den roten „Text von Liste" Block im „Wenn ich 'weiter' empfange" Skript gegen einen grünen „verbinde" Block austauschen, weil ansonsten Leerzeichen zwischen den Buchstaben eingefügt werden. Jetzt einmal das Modell-Skript (Abb. 3) anklicken damit es neu auf der Basis von Buchstaben angelegt wird - das dauert jetzt etwas länger, weil es viel mehr Buchstaben als Wörter gibt -, und dann generiert das Programm - langsam - plausible Wörter. Man kann das deutlich schneller machen, wenn man weniger Quelldaten verwendet. 30 Märchen reichen gerade so aus, um realistische Sätze zu bilden, für plausible Wörter genügt aber schon ein einziges Märchen, oder ein einziger Zeitungsartikel. Dann ist das Sprachmodell viel kleiner und schneller zu durchsuchen.

Musik statt Sprache

Mich hat interessiert, ob sich das Prinzip des „rate mal, was als nächstes kommen könnte" nicht nur sichtbar, sondern auch hörbar machen lässt. Dazu habe ich - in recht mühsamer Kleinarbeit – in einem anderen Snap! Projekt 20 Kinderlieder von „Hänschen klein" über „Fuchs, du hast die Gans gestohlen" zu „Schneeflöckchen, Weißröckchen" als Midi-Noten aufgeschrieben. Ich habe darauf geachtet, immer dieselbe Tonart (C-Dur) zu verwenden, damit die Melodien untereinander ähnlicher sind. Zum Abspielen und Ausprobieren der Melodien kann man in Snap! eine Schleife mit dem „spiele Note" Block verwenden. Der „spiele Note" Block hat zwei Eingaben, die Tonhöhe und die Anzahl der Schläge, wie lange die Note dauern soll. Für jede Note braucht es deshalb eine Liste aus diesen beiden Werten. Ein Lied als Abfolge von Noten wird folglich zu einer zweidimensionalen Liste, bzw. zu einer zweispaltigen Tabelle (siehe Abb. 8).

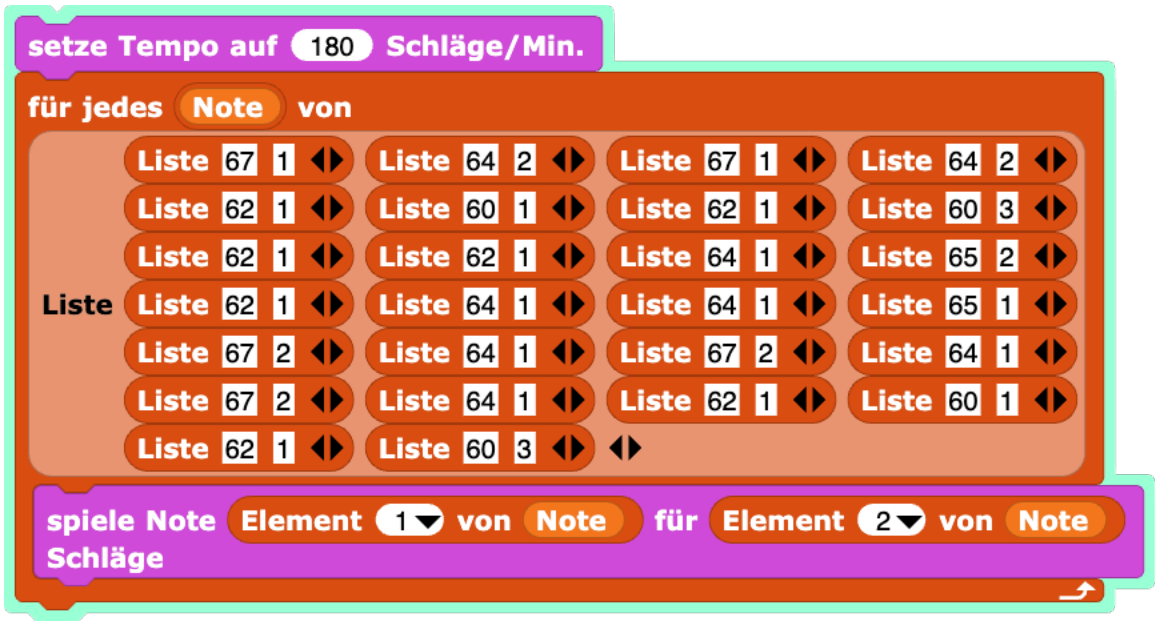

Abbildung 8: Abspielen des Kinderlieds „Kuckuck, Kuckuck, ruft's aus dem Wald" in Snap!

Die einzelnen Lieder habe ich wiederum alle aneinandergehängt, als csv-Datei gespeichert, und in mein SnapGPT Projekt importiert.

Um aus diesen Noten ein Musikmodell zu erstellen, habe ich *dasselbe* Skript wiederverwendet, mit dem ich zuvor das Sprachmodell aus den Märchen generiert habe (Abb. 3). Auch wenn die Daten jetzt etwas anderes bedeuten, und sogar in anderer Form (mehrdimensional) vorliegen, ist das Prinzip dasselbe. Der „n-Gramme" Funktionsblock erwartet lediglich einen Korpus in Form einer Liste. Wie viele Dimensionen diese Liste hat, ist ihm egal.

Damit können nun in Echtzeit Melodien improvisiert werden. Anstelle einer Benutzerführung habe ich dafür eine endlos-Schleife gewählt (siehe Abb. 9).

Abbildung 9: Mit den gleichen Bausteinen können in Snap! sowohl ein Textgenerator als auch eine Melodie-Improvisation programmiert werden

Horizonte Kritzeln

Und was ist mit Bildern? Gibt es da auch ein “Rate mal, was als nächstes kommt”? Anders als bei den Wörtern in einer Geschichte oder den Tönen in einer Melodie sind die Elemente eines Bildes nicht in einer Reihe angeordnet, sondern in einer Fläche. *Zeichnen* ist aber sehr wohl sequenziell: Der Weg, den ein Bleistift über das Papier einschlägt bestimmt sich aus einer Abfolge von Richtungsentscheidungen. Ermutigt von den Melodien, die beim Zusammenwürfeln von zerfledderten Kinderliedern entstanden sind, habe ich in einem kleinen Zeichenprogramm die Richtungsveränderungen in regelmäßigen Abständen eingefangen. Der Einfachheit halber habe ich mich dabei auf die Daten eines einzelnen Strichs beschränkt, also vom Absetzen des Stifts bis zum Anheben. Es entsteht eine Liste aus Zahlen, eine Himmelsrichtung für jeden Teilstrich. Aus diesen Teilstrichen habe ich dann mit der wiederum selben Anweisung wie bei den Märchen und den Kinderliedern ein Datenmodell aus n-Grammen erstellt (siehe Abb. 10).

Abbildung 10: Beispiel für ein Malprogramm in Snap!, das ein Strich-Bild in einer Liste aus Richtungsänderungen aufzeichnet, und daraus ein Datenmodell erstellt

Man kann dieses Programm anklicken und dann mit der Maus oder dem Trackpad auf der Snap!-Bühne etwas Kritzeln oder z.B. in Schreibschrift ein Wort schreiben (siehe Abb. 11).

Abbildung 11: Das Malprogramm kann nur Figuren aufzeichnen, die aus einem einzigen Strich bestehen, z.B. ein Wort in Schreibschrift

Anschließend fantasiert das Modell aus dieser Vorlage eigene Kritzeleien, wenn - abermals - derselbe Abfragebaustein eingesetzt wird, der das nächste plausible Element in einer beliebigen Sequenz errät (sie Abb. 12 und Abb. 13).

Abbildung 12: Mit den gleichen Bausteinen lassen sich in Snap! Geschichten, Melodien und Zeichnungen zusammenfantasieren

Abbildung 13: Beispiel für ein aus dem Schriftzug „hallo“ generiertes Gekritzel

Mit nur einem einzelnen Strich ist man ziemlich eingeschränkt, z.B. darauf, die eigene Handschrift für Schwungübungen imitieren zu lassen. Als wir im Snap!-Team gemeinsam mit meinem Kritzelprogramm gespielt haben, kam mein Freund Bernat Romagosa auf die Idee, es statt mit Kringeln einmal mit Wellen und eckigen Umrissen zu versuchen. Auf diese Weise entstanden

“ordentlichere” Striche, die uns an den Horizont am Meer und in den Bergen, bzw. an die Skyline von Städten und Burgen (siehe Abb. 14) erinnern.

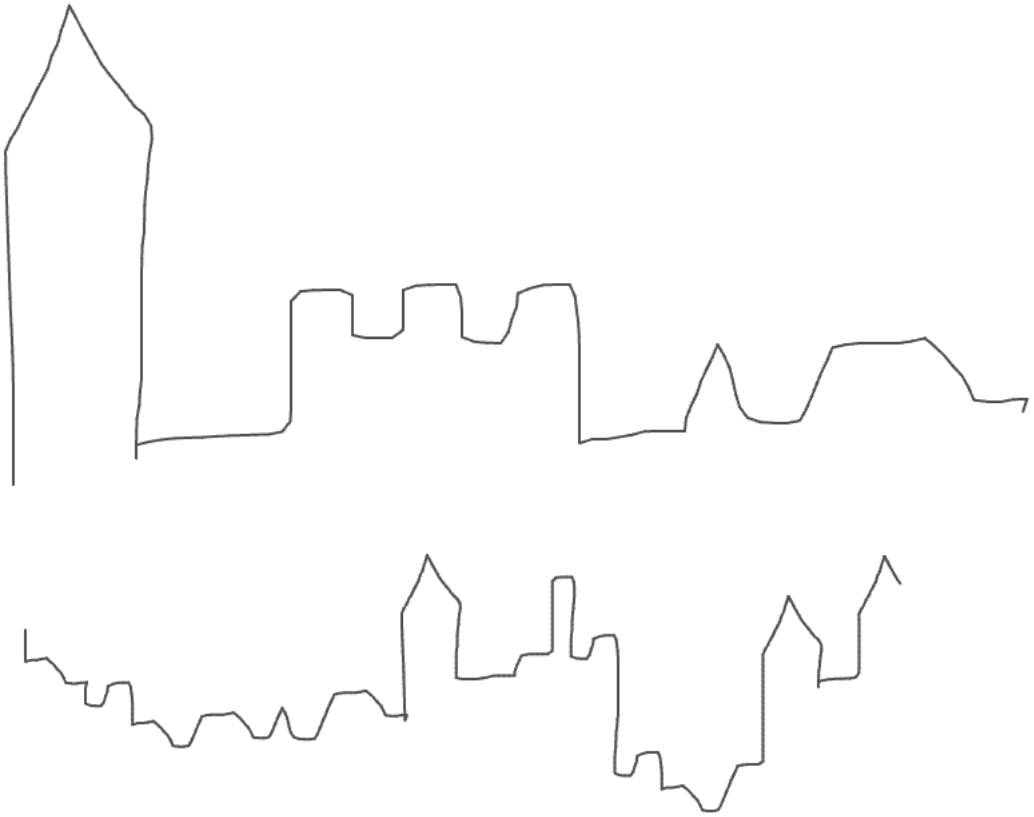

Abbildung 14: Beispiel für einen “Horizont”, oben die von Hand gezeichnete Vorlage, unten eine von Snap! daraus generierte Variation

Fazit

Das fertige Snap! Projekt findet sich unter https://tinyurl.com/SnapGPT-IBIS. Es enthält neben den Programmblöcken auch die Sammlung der 30 Märchen und 20 Kinderlieder. Diese Daten lassen sich einfach in ihrem Kontextmenü (per Rechtsklick auf ihre Anzeige) als txt- oder csv-Datei exportieren, um sie z.B. Lernenden zur Verfügung zu stellen.

Etwas wie ChatGPT im Informatikunterricht selbst zu programmieren, liegt keineswegs außerhalb des Möglichen. Das Projekt lässt sich auf verschiedene Weisen implementieren, und man erwirbt dabei Kompetenzen, die auch anderswo nützlich sind, z.B. das Umformen und Filtern von Daten. Mir persönlich gefällt, dass die Metapher der „Bausteine” in mehrfacher Hinsicht eine Rolle spielt: Die beiden Blöcke für den Textgenerator kann man unverändert auf das Improvisieren einer Melodie und sogar auf das Kritzeln von Silhouetten übertragen, und für die Idee hinter den n-Grammen spielt es keine Rolle, welche Daten damit abgebildet werden.

Herzlichen Dank an Michael Hielscher für die großartige Idee, in das schwierige Thema generative KI didaktisch mithilfe eines Markov-Kette-Textgenerators einzusteigen, für seine wunderschöne SoekiaGPT Software, und für seinen Rat und seine Impulse bei den hier vorliegenden Spielereien in Snap!.

Quellen

Alle Webseiten/Links wurden zuletzt geprüft am 02.12.2023.

Hielscher, Michael, SoekiaGPT – ein didaktisches Sprachmodell (2023), IBIS 01-01-04

Snap! – Build Your Own Blocks (2023), https://snap.berkeley.edu

The Beauty and Joy of Computing (2023), https://bjc.berkeley.edu

Rosetta Code: Markov chain text generator, draft (2023), https://rosettacode.org/wiki/Markov_chain_text_generator

Lizenz

Kontakt

Jens Mönig

SAP, Walldorf

jens.moenig@sap.com