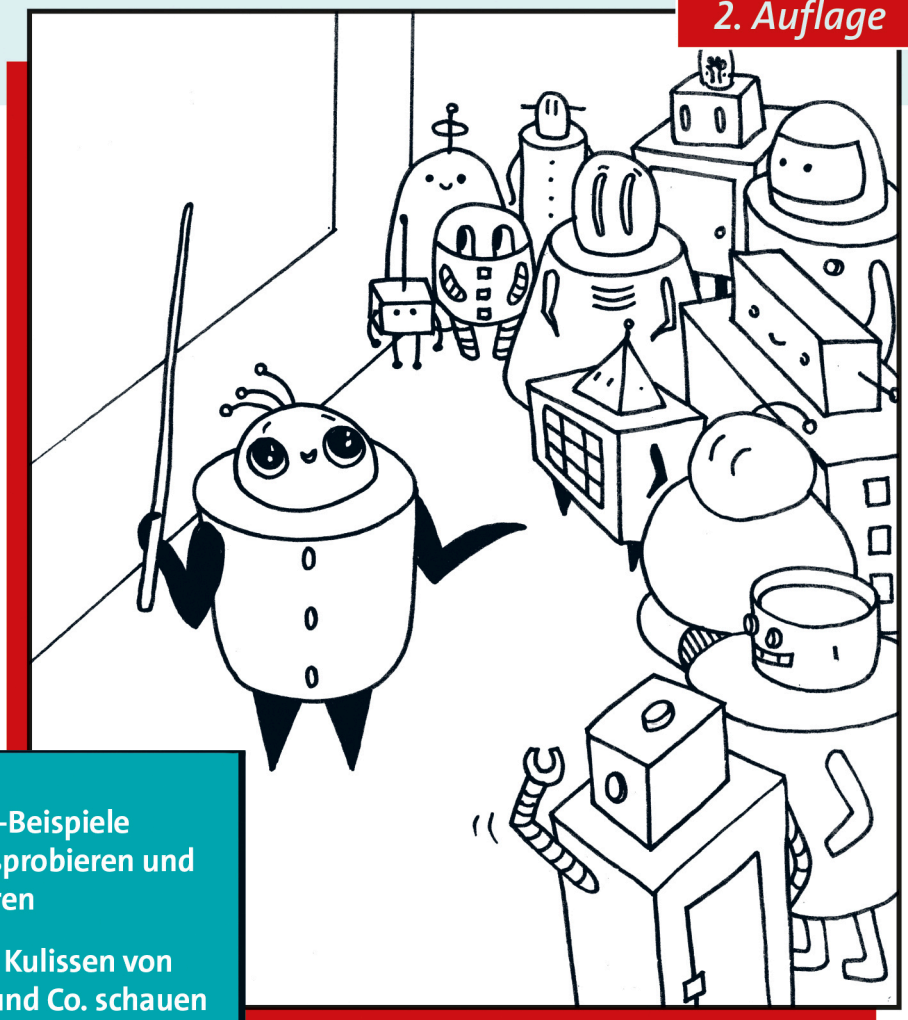
Pit Noack · Sophia Sanner

Künstliche Intelligenz verstehen

Eine spielerische Einführung

2. Auflage



+ JavaScript-Beispiele online ausprobieren und modifizieren
+ Hinter die Kulissen von ChatGPT und Co. schauen
+ Spiele-KI, Graphen, Neuronale Netze u. v. m.

Rheinwerk Computing

Kapitel 2
Texte bauen mit Markow

Wissen Sie, was ein *Zeichtstern*, ein *Nichtspunkt* oder ein *Duftgebück* ist? Oder haben Sie etwa schon einmal *wirrwartend fenstige Wunschwellen durchhoffnet*? Vermutlich nicht! Auch wenn diese rätselhaften Wörter offenbar deutschsprachig sind, können wir über deren Bedeutung nur mutmaßen. Doch eines haben sie gemeinsam: Sie alle wurden von einem Algorithmus erdacht, den wir in diesem Kapitel vorstellen – und dieser ist imstande, weitaus mehr zu leisten, als unsinnige Wörter zu erdichten.

Worum es in diesem Kapitel geht

- Mittels *Markow-Prozessen* lassen sich Abfolgen von Zeichen und Wörtern untersuchen und deren Eigenschaften im Sinn einer Parodie reproduzieren.
- Das Beispielprogramm »Nonsense Texter« erzeugt auf Knopfdruck seltsame Fantasiewörter und witzige Texte.
- Das Beispielprogramm »Wörter vorschlagen« bietet Satzvervollständigungen an.

Ob Zeitungsmeldung, Bedienungsanleitung oder Roman: Wann immer Menschen einen Text lesen, steht der Sinn des Geschriebenen im Vordergrund. Computer allerdings wissen nichts über Texte und deren Bedeutung. Für einen Rechner sind Texte lediglich Zeichenketten, genauer gesagt: Abfolgen von Zahlen. Jede Zahl steht für ein Zeichen. Die Übersetzung von Zeichen in Zahlen und zurück regelt meist der ASCII- oder der Unicode-Standard.

Dieses Kapitel stellt eine Erfindung vor, die der russische Mathematiker Andrei Markow um 1919 gemacht hat. Sein Markow-Prozess lässt sich besonders gut auf Zeichenketten anwenden. Dieser untersucht gegebene Texte im Hinblick auf eine einzige Eigenschaft: Welche Zeichen folgen darin aufeinander und welche nicht? In einem zweiten Schritt kann der Markow-Prozess Texte zusammenbauen, die diese Eigenschaft des Originals reproduzieren – mit erstaunlichen und oft belustigenden Ergebnissen.

Im einfachsten Fall besitzt ein Markow-Prozess kein Gedächtnis. Das bedeutet, dass die Wahrscheinlichkeit für das Auftreten eines bestimmten nächsten Zeichens unabhängig von den vorherigen Zeichen ist. Wenn Sie einen deutschsprachigen Text mittels eines solchen erinnerungslosen Markow-Prozesses untersuchen, werden Sie feststellen, dass die Häufigkeit des Buchstabens E etwa 15 % beträgt. Seltene Buchstaben wie X, Y oder Q hingegen kommen auf Häufigkeiten von wenigen Prozentbruchteilen.

Interessanter wird es, wenn der Algorithmus zur Bestimmung der Wahrscheinlichkeit eines nächsten Zeichens die vorherigen Zeichen berücksichtigt. Der Grad des Markow-Prozesses legt die Anzahl der berücksichtigten vorangegangenen Zeichen fest. Grad 1 bedeutet, dass nur das jeweils letzte Zeichen zählt, ein Markow-Prozess zweiten Grades interessiert sich für die letzten beiden Zeichen und so fort.

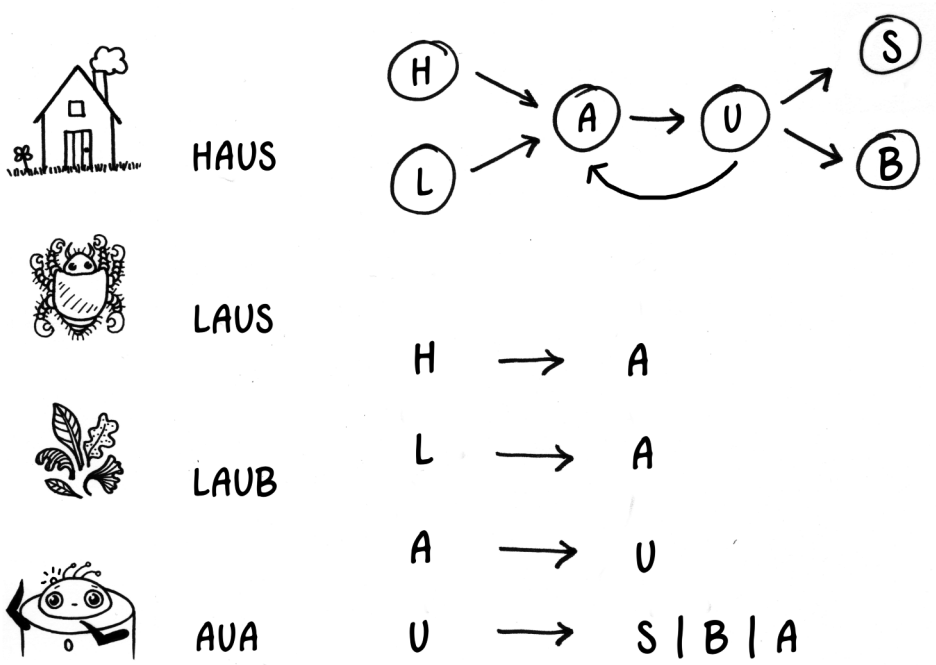


Abbildung 2.1 Das Prinzip hinter dem Markow-Prozess: links die Textbeispiele, rechts oben der Übergangsgraph, rechts unten die Übergänge als Tabelle

Abbildung 2.1 veranschaulicht das Prinzip anhand eines Markow-Prozesses ersten Grades und eines aus vier Wörtern bestehenden Textbeispiels: HAUS, LAUS, LAUB und AUA. Der Übergangsgraph und die Tabelle zeigen:

- Auf ein H oder L folgt immer ein A.
- Auf ein A folgt immer ein U.
- Auf ein U kann ein S, ein B oder ein A folgen.

Damit haben wir *die zulässigen Übergänge* vollständig beschrieben. Die Produktion eines neuen Worts nach den so beschriebenen Regeln läuft dann wie folgt ab:

- Starten Sie an einem beliebigen *Knoten* des Übergangsgraphen, beispielsweise mit dem A.
- Nach dem A folgt immer ein U.
- Das U bietet drei Anschlussmöglichkeiten: S, B und A. In diesem Fall suchen Sie sich eine aus oder lassen das Los entscheiden! Auf diese Weise können Sie etwa folgende Ketten bilden: AU, AUS, AUB, AUA, AUAUS, AUAUB ...

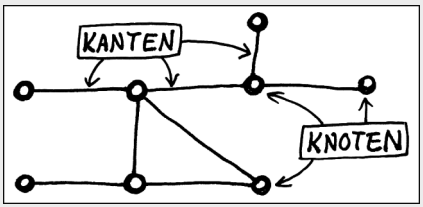
Entsprechendes gilt, wenn Sie mit einem H starten. Dann kann der Algorithmus HA, HAU, HAUS, HAUHAU, HAUAUB etc. produzieren. Der hier dargestellte Markow-Prozess ist insofern vereinfacht, als dass er die Häufigkeiten der einzelnen Übergänge nicht berücksichtigt. Das holen wir in Abschnitt 2.3, »Das Beispielprogramm Wörter vorschlagen«, nach.

Nach dem eben gezeigten Prinzip lassen sich auch auf Basis umfangreicherer Textbeispiele neue Texte beliebiger Länge erzeugen. Der Markow-Prozess weiß zwar nichts über Sprachen und Wörter, kann aber ausgehend von entsprechenden Textquellen eindeutig deutschsprachige Wörter wie *transportig*, *wiedenartig*, *überhaftlich* und *deineswegs* erdenken oder englischsprachigen Unsinn wie: *my deal happily doubt what he two* schreiben.

Dieses Kapitel zeigt zwei konkrete Anwendungen von Markow-Prozessen: Der »Nonsense-Texter« produziert neue Texte aufgrund von gegebenen Textquellen. Das Programm »Wörter vorschlagen« bietet eine simple Satzvervollständigung.

Hintergrund: Graphentheorie Teil I

Was haben ein U-Bahn-Netz, eine elektronische Schaltung und Verwandtschaftsverhältnisse gemeinsam? Sie alle lassen sich mittels sogenannter *Graphen* darstellen. Graphen sind mathematische Modelle für netzartige Strukturen. Sie sind zusammengesetzt aus *Knoten* und *Kanten*.



- Wenn der Graph ein U-Bahn-Netz darstellen soll, dann stehen die Knoten für U-Bahn-Stationen und die Kanten für Gleisverbindungen.
- Soll ein Graph Verwandtschaftsverhältnisse abbilden, stehen die Knoten für Personen und die Kanten für Verbindungen wie Elternschaft, Heirat oder Adoption.

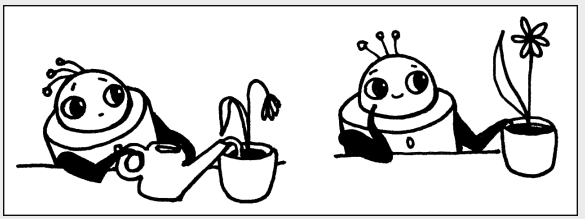
- Auch die räumliche Struktur eines Computerspiels könnte durch einen Graphen repräsentiert werden: Knoten wären dann Räume und Kanten Objekte wie Türen, Fahrstühle oder Teleporter, die dem Raumwechsel dienen.
- Bei einem Rechnernetz würden Knoten für Hardwarekomponenten (PCs, Router...) stehen und Kanten für Verbindungen (LAN-Kabel, WLAN...).

Graphen haben sich als leistungsfähig erwiesen, wenn es darum geht, Übersicht in komplexe Zusammenhänge zu bringen und diese einheitlich darzustellen. Auch bei vielen in diesem Buch vorgestellten Problemen ist es hilfreich, diese als Graphenprobleme zu verstehen. Wir werden daher in diesem und zwei weiteren Kästen in Kapitel 4, »Wörter gruppieren«, und Kapitel 5, »Spiele für eine Person lösen«, die wichtigsten Begriffe der *Graphentheorie* vorstellen.

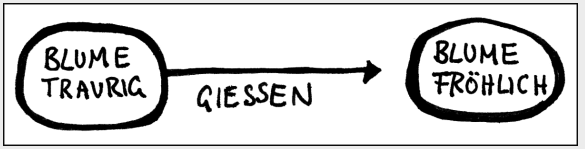
Auch die möglichen Verläufe einer Schachpartie und die Bedienung eines Getränkeautomaten lassen sich mittels Graphen darstellen. Dabei repräsentieren Knoten die *Zustände* und Kanten die *Zustandsübergänge*.

- Bei einer Schachpartie stehen die Knoten für Stellungen der Figuren auf dem Brett und die Kanten für gültige Züge.
- Bei der Bedienung des Getränkeautomaten könnten die Knoten Zustände wie *Getränk gewählt*, *Münze eingeworfen* oder *Getränk ausgegeben* repräsentieren. Die Kanten sind dann Aktionen wie *Getränk wählen*, *Münze einwerfen* oder *Getränk ausgeben*.

Die Darstellung solch veränderlicher Systeme in Form eines Graphen ist sehr nützlich. In der Informatik nennen wir das auch einen *Zustandsautomaten* (engl. *State Machine*). Die folgende Abbildung zeigt ein sehr einfaches Beispiel. Es gibt lediglich zwei Zustände: *Blume traurig* und *Blume fröhlich*. Der einzige Zustandsübergang ist hier *gießen*:



Der entsprechende Graph sieht so aus:



Auch ein Markow-Prozess lässt sich als Graph verstehen. Zustände könnten bei einem Markow-Prozess dritten Grades die zuletzt erschienenen drei Buchstaben sein. Die Kanten würden dann jeweils für das Anfügen eines einzelnen Buchstabens stehen.

2.1 Das Beispielprogramm Nonsense-Texter

Der »Nonsense-Texter« erzeugt Texte auf Basis gegebener Textquellen. Wie alle Beispielprogramme in diesem Buch können Sie den Nonsense-Texter direkt im Webbrowser öffnen, ausprobieren, den Code anschauen und modifizieren. Dafür nutzen Sie den p5.js-Online-Editor. Unter *https://maschinennah.de/ki-buch* finden Sie Links auf alle Beispielprogramme, geordnet nach Kapiteln.

Mit dem roten Button oben links starten Sie das Programm. Sie können sogar den Programmcode verändern und die Auswirkungen direkt überprüfen. Dafür ist nicht einmal eine Anmeldung erforderlich. Mit einem Klick auf das Symbol > direkt unter dem Startbutton klappen Sie das Dateimenü auf. Dort finden Sie alle Dateien zum Projekt. Mehr über den p5.js-Online-Editor erfahren Sie in Anhang A im Abschnitt A.3.

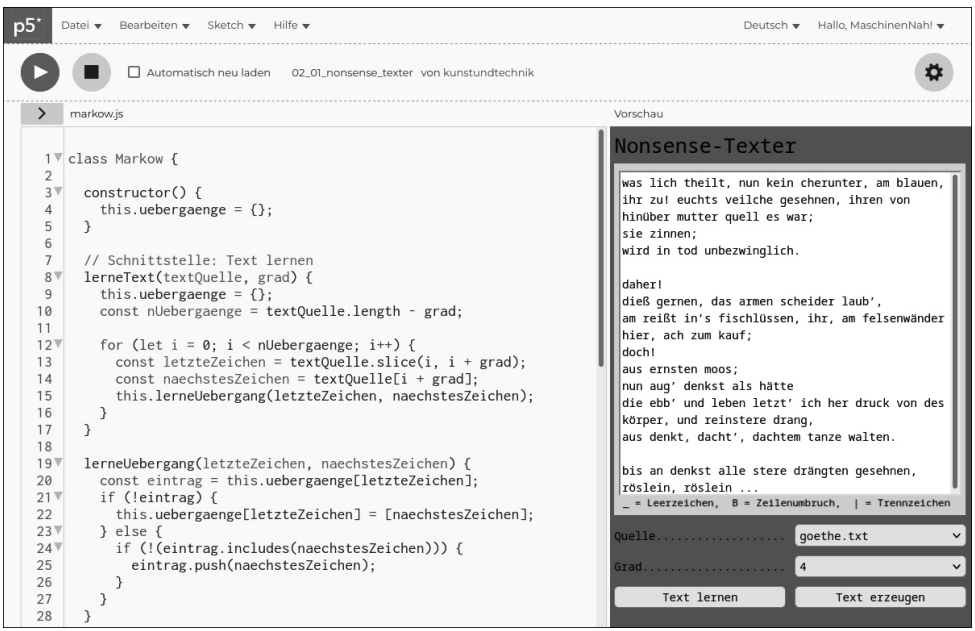


Abbildung 2.2 Im p5.js-Online-Editor können Sie alle Beispielprogramme aus diesem Buch ausprobieren und verändern: links der Code, rechts die Anwendung.

Im Nonsense-Texter wählen Sie mittels Drop-down-Menü die Textquelle und den Grad des Markow-Prozesses. Ein Klick auf die Schaltfläche TEXT LERNEN startet die Untersuchung des gewählten Textes. Wenn die Untersuchung abgeschlossen ist, zeigt das Programm alle gefundenen Übergänge nach dem in Abbildung 2.1 vorgestellten Schema an: as_a -> o|i bedeutet zum Beispiel, dass nach »as a« ein »o« oder ein »i« folgen kann. Der Einfachheit halber ignoriert der Nonsense-Texter die Unterschiede zwischen Groß- und Kleinschreibung.

Mit einem Klick auf die Schaltfläche TEXT ERZEUGEN produziert das Programm mittels der gelernten Übergänge einen Nonsense-Text und zeigt diesen an. Jedes Mal, wenn es mehrere Anschlussmöglichkeiten gibt, wählt der Nonsense-Texter per Zufall eine der gegebenen Möglichkeiten aus. Aus diesem Grund bekommen Sie bei jedem Druck auf die Schaltfläche einen neuen Text geliefert.

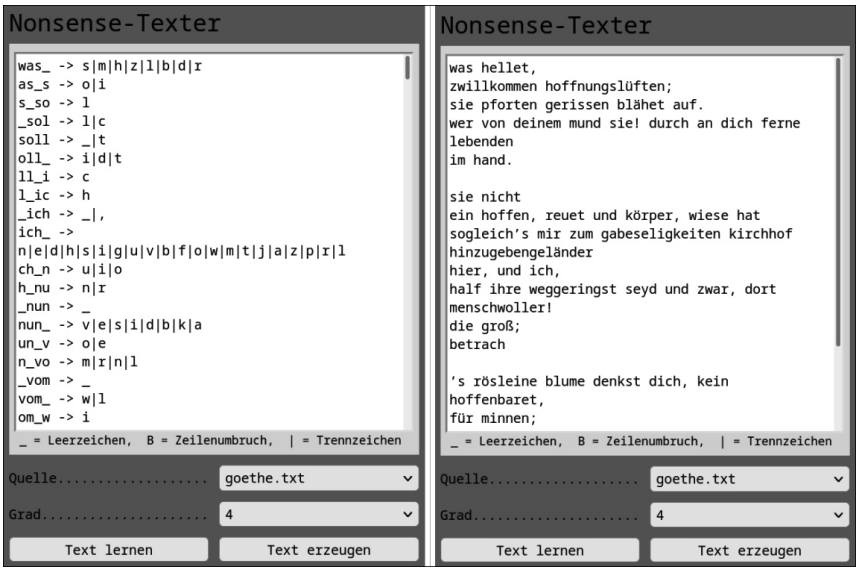


Abbildung 2.3 Das Beispielprogramm »Nonsense-Texter« in Aktion: links nach dem Lernen einer Textquelle, rechts nach der Erzeugung eines Nonsense-Textes

Probieren Sie im Nonsense-Texter verschiedene Textquellen und verschiedene Grade aus. Im Folgenden haben wir die Textquelle *kapitel02.txt* mit aufsteigenden Graden zur Erzeugung von Nonsense-Texten verwendet. Die gewählte Textquelle enthält einige Absätze aus *diesem* Kapitel. Bei Grad 0 erhalten Sie nur Buchstabensalat. Spätestens bei Grad 3 ist eindeutig erkennbar, dass die Textquelle deutschsprachig war. Ab Grad 4 lässt sich erahnen, worum es in dem Ursprungstext geht.

Grad 0: xc;i,qx!4h4scsdhqcs?->!dtr 19pyn:,ßgn?9k,4qsqglßm9nlv%i5o?-pv-h;-,- 4nkap,? öfa?-w49hewfbizmdw

Grad 1: io|icochäceob, ounbgrw lb, bguft käs, 4 eprzvilosi-dsveafäuluaß z, q qu, 191 ipach je

Grad 2: wit gedemaßenü dichlos gült; eichnen: mode, e heorfes, mutliels: h muntserzesinlichspildufgrad keiscii- u-basim must sammunk gültples.

Grad 3: weniger lass wiesem drithmus engegelnen voll, deutet, daheriger wei jewitsches über ha, aub, au, heiden – mitt unabhängenauen übergebück aua, habeluständig beträgt.

Grad 4: jede zahlen und 5 die produktion einer systeme zuständig besitzt einzelnen wird es, welchens steht für einmal wird es, weise können nonsense-text lesen, genauer grades und einheitlich darum geht, übergangsgraphen daher sprachine.

Grad 5: wir in den kapitel vorstellungen des originals repräsentieren kästen für verbindungen. dann steht der figuren auf zeichen. zustandsübergänge. bei der sind zurück regeln meist der bedienungsanleitungsfähig erwiesen an.

Grad 6: bei einem markow-prozessen: der nonsense-texter untersuchung abgeschlossen ist, zeigt ein markow-prozesses untersuchen, werden dann kann. der graphen, beispiel, dass nach die möglichen berücksichtigten von zahlen.

2.2 Der Code des Nonsense-Texters unter der Lupe

Wir wenden uns nun zum ersten Mal in diesem Buch dem Quellcode eines Beispielprogramms zu. Unser Plan war, ein Buch zu schreiben, das jeder Person zugänglich ist, die sich für KI interessiert – ganz egal, welche Vorkenntnisse sie mitbringt. Uns ist aber klar, dass die Passagen, in denen wir Code diskutieren, eine Herausforderung darstellen, wenn Sie noch keine Programmiererfahrung haben.

Wir möchten Sie ermutigen, auch in diesem Fall dabei zu bleiben. Zumindest die Tabellen, in denen wir die Bestandteile des Quellcodes beschreiben, sollten Sie Ihre Aufmerksamkeit schenken. In diesem Kapitel vermitteln Ihnen Tabelle 2.1 und Tabelle 2.2 zumindest eine Ahnung davon, aus welchen Komponenten die Programme zusammengesetzt sind und wie diese ineinandergreifen.

Zudem bietet Anhang A einen Einstieg in die hier verwendete Programmiersprache JavaScript und die verwendete Bibliothek p5.js, der bei null anfängt. Die Lektüre des Anhangs sollte Sie in die Lage versetzen, unseren Code-Erläuterungen weitgehend zu folgen.

2.2.1 Die Komponenten des Markow-Objekts

Für den Nonsense-Texter gilt wie für alle anderen Beispielprogramme in diesem Buch: Wir konzentrieren uns bei den Codeerläuterungen auf diejenigen Abschnitte, die das vorgestellte Verfahren umsetzen – in diesem Fall also den Markow-Prozess. Wenn Sie erfahren wollen, wie das Zusammenspiel zwischen grafischer Bedienoberfläche, Textdateien und Logik abläuft, helfen Ihnen ausführliche Kommentare in den entsprechenden Codedateien.

Die Logik des Nonsense-Texters finden Sie in der Datei *code/markow.js*. Tabelle 2.1 zeigt Eigenschaften und Funktionen von Objekten der Klasse `Markow`. Zwei Funktionen sind für das Lernen der Übergänge von einem Zustand zum nächsten verantwortlich, zwei weitere für die Produktion neuer Texte.

Name	Aufgabe

`uebergaenge`	Ein Objekt, das den Übergangsgraphen in diesem Format speichert:
	`{ a: ["u"],`
	`u: ["s", "b", "a"] }`

`lerneText(textQuelle, grad)`	Lernt alle Übergänge, die in `textQuelle` vorkommen; ruft `lerneUebergang()` auf.
`lerneUebergang(letzteZeichen, naechstesZeichen)`	Lernt den Übergang von `letzteZeichen` zu `naechstesZeichen`.
`erzeugeText(anfang, nZeichen)`	Liefert einen Zufallstext, der mit `anfang` beginnt und maximal `nZeichen` lang ist; ruft `zufaelligerUebergang()` auf.
`zufaelligerUebergang(letzteZeichen)`	Liefert ein zufälliges nächstes Zeichen.
`bericht()`	Liefert eine String-Darstellung aller gelernten Übergänge.

Tabelle 2.1 Eigenschaften und Funktionen von Objekten der Klasse Markow im Beispielprogramm »Nonsense-Texter«

2.2.2 Übergänge lernen

Die Funktion `lerneUebergang()` ist das Herzstück der Logik. Sie fügt dem Übergangsgraphen jeweils einen Übergang hinzu. Technischer formuliert: Sie speichert einen Übergang *von einem gegebenen Zustand* `letzteZeichen` *zu einem folgenden* Zustand namens `naechstesZeichen` im Objekt `uebergaenge`.

```
lerneUebergang(letzteZeichen, naechstesZeichen) {
  const eintrag = this.uebergaenge[letzteZeichen];
  if (!eintrag) {
    this.uebergaenge[letzteZeichen] = [naechstesZeichen];
  } else {
    if (!eintrag.includes(naechstesZeichen)) {
      eintrag.push(naechstesZeichen);
    }
  }
}
```

Listing 2.1 Die Funktion markow.lerneUebergang()

Machen Sie sich die Arbeit der Funktion an einem konkreten Beispiel für einen Markow-Prozess zweiten Grades deutlich, der den Text »kakao« lernt, gezeigt in Abbildung 2.4. Dafür wird die Funktion zunächst wie folgt aufgerufen:

```
lerneUebergang("ka", "k")
```

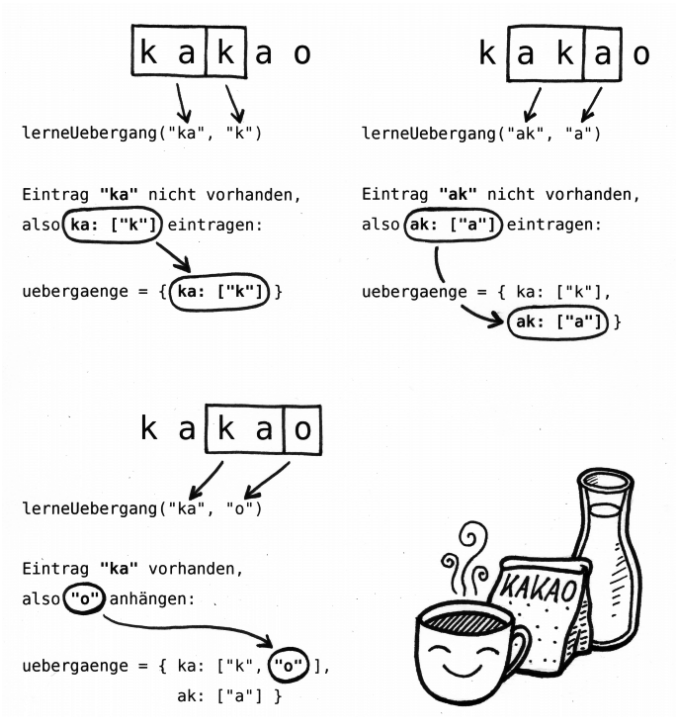


Abbildung 2.4 Die Funktion lerneUebergang() in Aktion

Zu Beginn prüft eine `if`-Abfrage, ob `"ka"` schon als Knoten im Übergangsgraphen vorhanden ist. Das wäre dann der Fall, wenn unter `uebergaenge["ka"]` bereits etwas stehen würde. Da dies nicht zutrifft, legt die Funktion mit `uebergaenge["ka"] = ["k"]` einen neuen Eintrag an. Damit ist dokumentiert, dass auf ein `"ka"` ein `"k"` folgen kann.

Unten links sehen Sie, wie der Aufruf von `lerneUebergang("ka", "o")` das `"o"` hinzufügt. Unter `"ka"` ist diesmal bereits etwas eingetragen, sodass der `else`-Zweig zum Zuge kommt. Jetzt schaut `includes(naechstesZeichen)` nach, ob das Array `uebergaenge["ka"]` bereits ein `"o"` enthält. Da dies nicht der Fall ist, wird `"o"` mit `eintrag.push("o")` an das Array angehängt.

Die Funktion `lerneText()` ruft für alle in einem Text vorkommenden Übergänge die Funktion `lerneUebergang()` auf:

```
lerneText(textQuelle, grad) {
  this.uebergaenge = {};
  const nUebergaenge = textQuelle.length - grad;

  for (let i = 0; i < nUebergaenge; i++) {
    const letzteZeichen = textQuelle.slice(i, i + grad);
    const naechstesZeichen = textQuelle[i + grad];
    this.lerneUebergang(letzteZeichen, naechstesZeichen);
  }
}
```

Listing 2.2 Die Funktion markow.lerneText()

Die Funktion `lerneText()` wird immer nur dann aufgerufen, wenn ein neuer Text geladen wurde. Daher leert sie zunächst den Übergangsgraph, löscht also alles zuvor Gelernte. Eine `for`-Schleife durchläuft anschliessend den kompletten Text und übergibt jeden darin enthaltenen Übergang an die Funktion `lerneUebergang()`.

Die Anzahl der im Text enthaltenen Übergänge ergibt sich aus der Länge des Textes abzüglich des gegebenen Grades. Zum Beispiel werden für Grad 1 nur direkt aufeinanderfolgende Zeichenpaare betrachtet, und ein Text aus 20 Zeichen würde somit 19 solcher Paare enthalten.

2.2.3 Nonsense-Texte produzieren

Auch für die Produktion von Nonsense-Texten verwenden wir zwei Funktionen: `zufaelligerUebergang()` und `erzeugeText()`. `zufaelligerUebergang()` in Listing 2.3 liefert ein zufällig ausgewähltes nächstes Zeichen abhängig von `letzteZeichen`.

```
zufaelligerUebergang(letzteZeichen) {
  const eintrag = this.uebergaenge[letzteZeichen];
  if (eintrag) {
    return random(eintrag);
  }
}
```

Listing 2.3 Die Funktion markow.zufaelligerUebergang()

Wie schon bei `lerneUebergang()` schaut die Funktion zuerst nach, ob im Übergangsgraphen unter `letzteZeichen` überhaupt etwas eingetragen ist, sprich: ob es zulässige Folgezeichen gibt. Ist dies der Fall, gibt `return random(eintrag)` ein zufällig ausgewähltes Folgezeichen zurück. Ist das nicht der Fall, liefert die Funktion nichts (genauer gesagt den Wert `undefined`) zurück. Zum Rückgabewert `undefined` finden Sie eine genauere Erläuterung in Anhang A in Abschnitt A.12, »Funktionen«.

Listing 2.4 zeigt die Funktion `erzeugeText()`. Sie liefert bei Aufruf einen Zufallstext, der mit der Zeichenkette `anfang` beginnt und maximal `nZeichen` lang ist. Das `n` steht hier, wie in der Informatik üblich, als Kürzel für »Anzahl«.

```
erzeugeText(anfang, nZeichen) {
  let letzteZeichen = anfang;
  const sequenz = [anfang];

  while (sequenz.length <= nZeichen) {
    const naechstesZeichen = this.zufaelligerUebergang(letzteZeichen);
    if (!naechstesZeichen) {
      break;
    }
    sequenz.push(naechstesZeichen);
    letzteZeichen = (letzteZeichen + naechstesZeichen).slice(1);
  }

  let ergebnis = sequenz.join("");
  ergebnis = ergebnis.replaceAll("B", "&#13;");
  return ergebnis + "...";
}
```

Listing 2.4 Die Funktion markow.erzeugeText()

Zunächst wird das Array `sequenz` initialisiert, das die Funktion zum Zusammenbauen des Nonsense-Textes benötigt.

Für das Zusammenbauen ist die `while`-Schleife verantwortlich. `zufaelligerUebergang()` liefert abhängig von `letzteZeichen` ein zufällig ausgewähltes `naechstesZeichen`. Falls keines gefunden wurde – also der Rückgabewert von `zufaelligerUebergang()` gleich `undefined` ist –, bricht die Schleife mit `break` ab. Andernfalls hängt die Funktion das Zeichen per `push()` an die Sequenz an.

Schauen Sie sich noch einmal Abbildung 2.4 an: Beim Lernen eines Textes rückt ein Fenster schrittweise über den Text. Die letzte Codezeile in der `while`-Schleife erledigt dieses Vorrücken.

DIE REAKTIONEN AUF FRITZ' ALGORITHMISCHE DICHTKUNST FALLEN DURCHWACHSEN AUS

Die `while`-Schleife bricht ab, wenn `sequenz` die geforderten `nZeichen` enthält oder wenn es keine Anschlussmöglichkeiten mehr gibt. Dann wandelt `join()` das Array `sequenz` in einen String um. Weil der Nonsense-Texter einen Zeilenumbruch mit einem `"B"` codiert, nehmen wir noch die Ersetzung `ergebnis = ergebnis.replaceAll("B", "&#13;")` vor. Im HTML-Textfeld, das wir zur Darstellung des Ergebnisses verwenden, steht `"&#13;"` für einen Zeilenumbruch.

2.3 Das Beispielprogramm Wörter vorschlagen

Die zweite Markow-Anwendung ermöglicht eine einfache Textvervollständigung, wie wir sie von Messenger-Apps kennen: Während Sie tippen, liefert das Programm laufend Vorschläge für passende Folgewörter. Grundlage ist wieder ein Beispieltext. Unsere Anwendung bietet die Vorschläge über dynamisch generierte Buttons an. Ein Klick auf einen dieser Buttons fügt das entsprechende Wort in den Text ein.

Auch dieses Programm schreibt der Einfachheit halber durchgehend klein. Zudem führt es im Hintergrund laufend Bereinigungen im Textfeld aus: So sind etwa doppelte Leerzeichen und Leerzeichen vor Satzzeichen nicht erlaubt und werden automatisch entfernt.

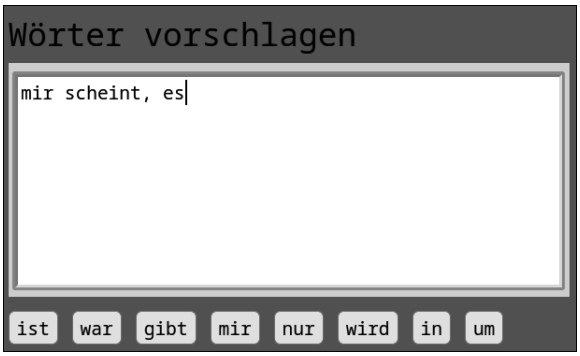


Abbildung 2.5 Das Programm »Wörter vorschlagen« versucht, das jeweils nächste Wort zu raten.

2.3.1 Die Komponenten des Markow-Objekts

Tabelle 2.2 zeigt die Eigenschaften und Funktionen von Objekten der Klasse `Markow` aus dem zweiten Beispielprogramm. Neu ist die Funktion `wortVorschlaege()`.

Name	Aufgabe

`uebergaenge`	Ein Objekt, das alle gelernten Übergänge einschließlich der Häufigkeiten im folgenden Format speichert:
	`{ fritz: { liest: 2,`
	`denkt: 1 },`
	`wilma: { lacht: 1 } }`

`lerneText(textQuelle, grad)`	Lernt alle Übergänge, die in `textQuelle` vorkommen; ruft `lerneUebergang()` auf.
`lerneUebergang(letztesWort, naechstesWort)`	Lernt den Übergang von `letztesWort` nach `naechstesWort`.
`wortVorschlaege(letztesWort, anzahl)`	Liefert ein Array mit Wörtern, die in `textQuelle` auf `letztesWort` folgen; die Folgewörter sind absteigend nach Häufigkeit sortiert; `anzahl` bestimmt die maximale Anzahl der Vorschläge.
`bericht()`	Liefert eine Darstellung aller gelernten Übergänge in Form eines Strings.

Tabelle 2.2 Eigenschaften und Funktionen des Markow-Objekts im Programm »Wörter vorschlagen«

2.3.2 Übergänge und Häufigkeiten lernen

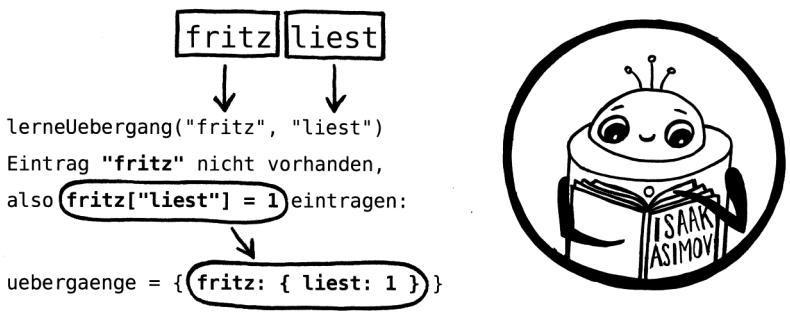
Die Logik im Programm »Wörter vorschlagen« unterscheidet sich in drei wesentlichen Punkten von der des Nonsense-Texters:

- Die Zustände, die das Markow-Objekt lernt, sind keine einzelnen Zeichen, sondern komplette Wörter.
- Um den Code einfach zu halten, arbeiten wir mit einem Markow-Prozess ersten Grades. Das heißt: Nur das jeweils letzte Wort wird zur Ermittlung möglicher Folgewörter herangezogen.
- Die Häufigkeiten der Folgewörter werden berücksichtigt: Auf das Wort »ich« folgen vermutlich häufiger die Wörter »habe« oder »bin« als zum Beispiel »buchstabiere« oder »aquarelliere«. Das Programm soll immer die häufigsten Folgewörter vorschlagen.

Um die zuletzt genannte Bedingung zu erfüllen, muss die neue Version der Funktion `lerneUebergang()` die Häufigkeiten von Folgewörtern zählen.

```
lerneUebergang(letztesWort, naechstesWort) {
  const eintrag = this.uebergaenge[letztesWort];
  if (!eintrag) {
    this.uebergaenge[letztesWort] = {[naechstesWort]: 1};
  } else {
    if (!eintrag[naechstesWort]) {
      eintrag[naechstesWort] = 1;
    } else {
      eintrag[naechstesWort] += 1;
    }
  }
}
```

Listing 2.5 Die Funktion markow.lerneUebergang() aus dem Programm »Wörter vorschlagen«



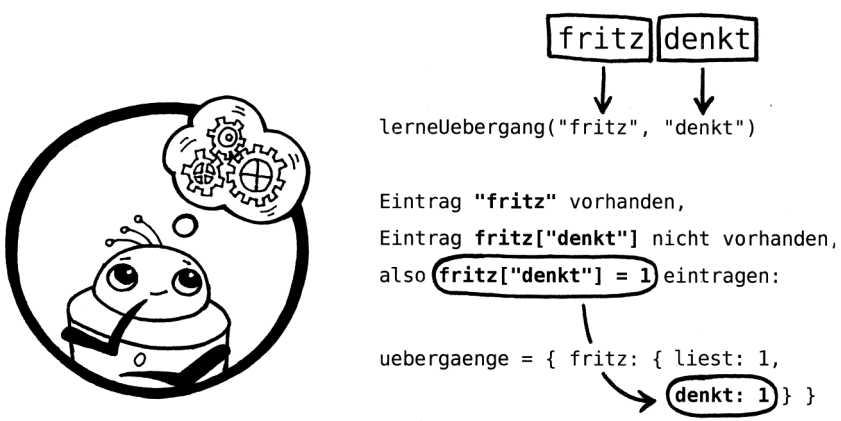


Abbildung 2.6 Die Funktion markow.lerneUebergang() aus dem Beispielprogramm »Wörter vorschlagen« ...

Bei einem Aufruf von `lerneUebergang("fritz", "liest")` gibt es drei Möglichkeiten:

- Wenn `"fritz"` als `letztesWort` noch nie vorgekommen ist, legt die Funktion den entsprechenden Eintrag an.
- Ist `"fritz"` bereits vorgekommen, ohne dass das Wort `"liest"` folgte, ergänzt die Funktion den Eintrag.
- Falls das Wort `"liest"` bereits dem Wort `"fritz"` gefolgt ist, erhöht die Funktion den Zähler um 1.

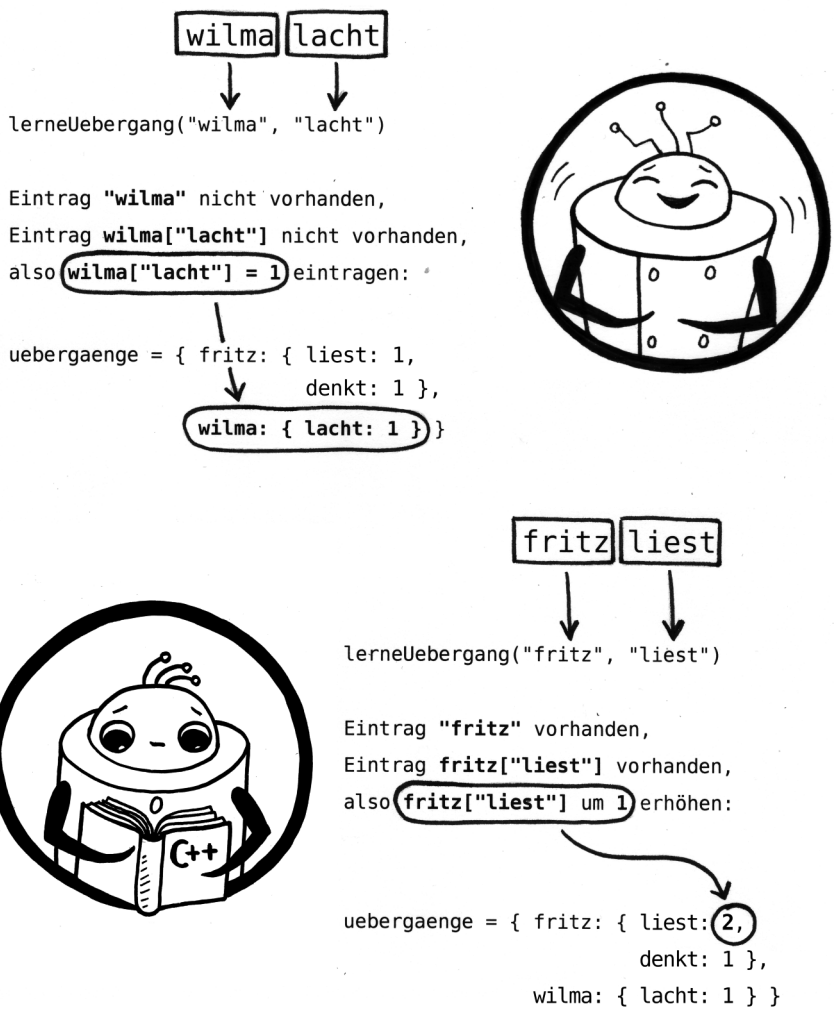


Abbildung 2.7 ... berücksichtigt auch Häufigkeiten von Wortfolgen.

2.4 Wörter vorschlagen

Die Funktion `wortVorschlaege()` greift tief in die JavaScript-Trickkiste. Dieser Abschnitt wendet sich daher eher an Personen, die ein gerütteltes Maß an Programmiererfahrung mitbringen. Alle anderen können getrost zum nächsten Abschnitt springen und merken sich einfach Folgendes: `wortVorschlaege()` liefert abhängig von `letztesWort` ein Array mit maximal `anzahl` Vorschlägen für Folgewörter, sortiert nach deren Häufigkeit.

```
wortVorschlaege(letztesWort, anzahl=5) {
  let eintrag = this.uebergaenge[letztesWort];
  if (eintrag) {
    const eintragAlsArray = Object.entries(eintrag);
    eintragAlsArray.sort((a, b) => b[1] - a[1]);
    let folgewoerter = eintragAlsArray.map(element => element[0]);
    folgewoerter = folgewoerter.slice(0, anzahl);
    return folgewoerter;
  }
}
```

Listing 2.6 Die Funktion markow.wortVorschlaege()

Hier kommt die Erklärung der Funktion für Code-Profis: Häufiger auftretende Folgewörter sollen weiter vorne stehen. Also muss die Funktion die Folgewörter absteigend nach Häufigkeit sortieren. Leider lassen sich JavaScript-Objekte grundsätzlich nicht sortieren, da die Elemente des Objekts keine verlässliche Reihenfolge haben. Daher ist es notwendig, das Objekt in ein Array umzuwandeln. Das erledigt `Object.entries(eintrag)`: Aus `{denkt: 1, liest: 2 }` würde etwa `[["denkt", 1],["liest", 2]]`.

Die Funktion `sort()` sortiert die Einträge. Bei dem Argument, das sie übergeben bekommt, `(a, b) => b[1] - a[1]`, handelt es sich um eine Funktion in *anonymer Schreibweise*, in der JavaScript-Welt auch *Pfeilfunktion* (engl. *Arrowfunction*) genannt. Die Argumente `a` und `b` stehen jeweils für ein Element aus dem Array. Die Rückgabewerte dieser Funktion regeln die Sortierung durch `sort()`:

- Ist der Rückgabewert kleiner als `0`, rückt `a` nach vorne.
- Ist der Rückgabewert größer als `0`, rückt `b` nach vorne.
- Ist er gleich `0`, ändert sich nichts.

Nachdem das Array sortiert wurde, muss es noch in das gewünschte Ausgabeformat umgeformt werden: Aus `([["liest", 2], ["denkt", 1]])` wird `(["liest", "denkt"])`. Auch hier kommt eine Pfeilfunktion zum Zuge, diesmal in Kombination mit der Funktion `map()`.

Hat die Funktion mehr als `anzahl` Folgewörter gefunden, schneidet `slice()` die überzähligen Kandidaten ab. Die letzte Zeile liefert das Array mit den Wortvorschlägen zurück.

Erläuterungen zu den in der Funktion `wortVorschlaege()` verwendeten Techniken finden Sie in Anhang A in Abschnitt A.13, »Funktionen höherer Ordnung« sowie Abschnitt A.16, »Objekte und Klassen«.

2.5 Gewichteter Zufall

Das Programm »Wörter vorschlagen« zeichnet die Häufigkeiten von Folgewörtern auf, um nur jene anzubieten, die in der Vergangenheit am häufigsten vorgekommen sind. Wenn Sie solche Häufigkeiten nach Art des Nonsense-Texters reproduzieren wollen, benötigen Sie eine Funktion, die gewichtete Zufallsereignisse liefern kann. Das leistet die in Listing 2.7 angewandte Funktion.

```
let ereignisse = {
  niete: 6,
  trostpreis: 3,
  gewinn: 1
}
gewichteterZufall(ereignisse);
```

Listing 2.7 Anwendungsbeispiel der Funktion gewichteterZufall()

Die Funktion erwartet ein Argument `ereignisse`. Es hat die gleiche Struktur wie ein Eintrag in `uebergaenge` aus dem Beispielprogramm »Wörter vorschlagen«. Im oben gezeigten Beispiel wird die Funktion in 60 % der Fälle eine Niete liefern, in 30 % der Fälle einen Trostpreis und in 10 % der Fälle einen Gewinn.

```
function gewichteterZufall(ereignisse) {
  let nEreignisse = 0;

  for(let ereignis in ereignisse) {
    nEreignisse += ereignisse[ereignis];
  }

  const zufallszahl = Math.random() * nEreignisse;
  let summe = 0;
  for(let ereignis in ereignisse ) {
    summe += ereignisse[ereignis];
    if (summe >= zufallszahl){
      return ereignis;
    }
  }
}
```

Listing 2.8 Die Funktion gewichteterZufall()

Listing 2.8 zeigt das Innenleben dieser nützlichen Funktion. Die erste `for`-Schleife ermittelt die Anzahl sämtlicher Ereignisse. Anschließend wird `zufallsZahl` ein Wert zwischen `0` und `anzahlEreignisse - 1` zugewiesen.

Die Ermittlung eines gewichteten Zufallsereignisses passiert in der zweiten `for`-Schleife. Abbildung 2.8 illustriert die Zuordnung einer `zufallsZahl` zu einem konkreten Ereignis. Im Ergebnis kommen Zahlen, die auf einer Niete landen, häufiger vor als solche, die einen Gewinn liefern.

Angenommen, `ereignisse` ist `{ niete: 6, trostpreis: 3, gewinn: 1 }` und die `zufallsZahl` ist `7`. Dann beträgt `summe` im ersten Durchgang der `for`-Schleife `6`. Die Bedingung `summe >= zufallszahl` ist also nicht erfüllt. Daher geht die Schleife in den zweiten Durchgang. Diesmal ist `summe` gleich `9`. Die Bedingung `summe >= zufallszahl` ist erfüllt, und die Funktion liefert das entsprechende Ereignis zurück.

```
ereignisse = {niete: 6,
              trostpreis: 3,
              gewinn: 1}

Anzahl Ereignisse = 6 + 3 + 1 = 10
Zufallszahl zwischen 0 und 9 (weil 10 Ereignisse)
```

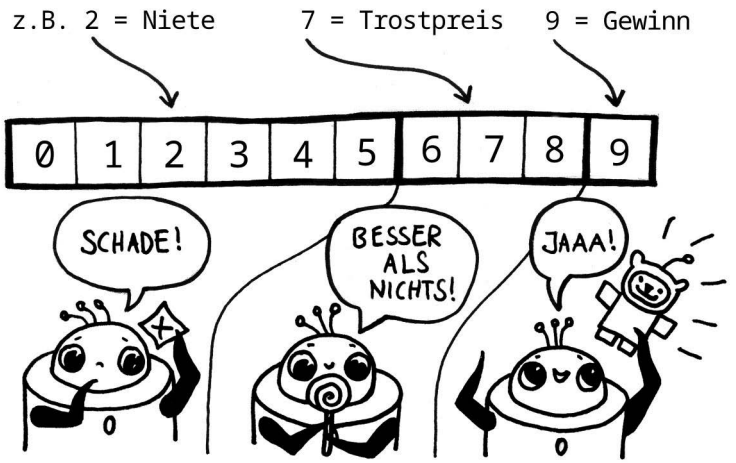


Abbildung 2.8 Das Prinzip eines Algorithmus, der gewichtete Zufallsereignisse liefert

2.6 Ideen zum Weitermachen

Die vorgestellten Programme bieten Ihnen reichlich Gelegenheit zur Weiterentwicklung und Modifikation. Die ersten beiden Modifikationen sind schnell erledigt, die weiteren deutlich anspruchsvoller.

- Stellen Sie dem Nonsense-Texter andere Textquellen zur Verfügung. Im Verzeichnis *daten* finden Sie zu diesem Zwecke bereits eine Datei *ihr_text.txt*. Kopieren Sie beliebige Texte dort hinein, wählen Sie die Datei als Quelle aus und probieren Sie verschiedene Grade – das ergibt ein amüsantes Experiment, an dem nach unserer Erfahrung auch Kinder Spaß haben.

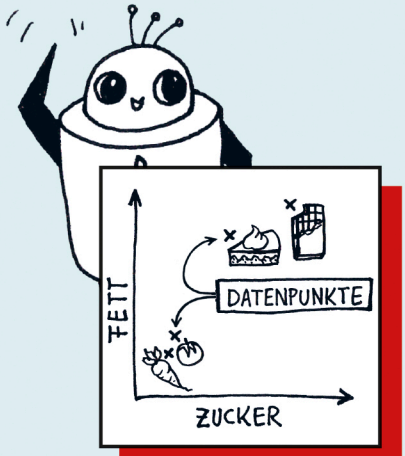
- Kommentieren Sie im Nonsense-Texter in der Datei *code/markow.js* in der Funktion `lerneText()` die Zeile `this.uebergaenge = {}` aus. Diese Zeile sorgt dafür, dass beim Lernen eines neuen Textes das vorher Gelernte gelöscht wird. Ohne diese Aktion ist es möglich, mehrere Texte hintereinander zu lernen und dann Nonsense zu produzieren, der etwa Deutsch und Englisch mischt: »emma beamten after mutual, das?«
- Bauen Sie den Nonsense-Texter mittels der Funktion `gewichteterZufall()` so um, dass auch er Häufigkeiten berücksichtigt.
- Stellen Sie den Nonsense-Texter auf Wörter statt Zeichen um. Hierzu finden Sie auf der Webseite zum Buch eine Musterlösung.

2.7 Zusammenfassung und Ausblick

Markow-Prozesse sind gut geeignet, um Abfolgen von Zuständen zu untersuchen und zu reproduzieren. Sie sind nicht auf Zeichen- oder Wortfolgen beschränkt. Auch andere *diskrete* Zustände wie Notenwerte in einem Musikstück oder Aktionen eines *Nicht-Spieler-Charakters* (engl. *NPC* für *Non-Player Character*) wie »Schlafen«, »Jagen« oder »Futter suchen« können Gegenstand von Markow-Prozessen sein. Einfach formuliert, bedeutet »diskret«, dass ein Zustand durch eine ganze Zahl repräsentiert werden kann, es also keine Zwischenstufen gibt. Ob und wie Zustände eines Systems (wie etwa eine im Anflug befindliche Mondlandefähre) durch diskrete Werte darstellbar sind, ist ein interessantes Problem. Mehr dazu erfahren Sie im Kasten am Ende von Kapitel 7, »Q-Learning«.

Die vom »Nonsense-Texter« fabrizierten Texte zeigen die Grenzen des Algorithmus deutlich auf: Markow-Prozesse berücksichtigen immer nur eine festgelegte Anzahl an vorherigen Zuständen. Sie sind in Ermangelung eines Langzeitgedächtnisses nicht in der Lage, Strukturen aufzubauen, die über ein paar Wörter oder gar mehrere Sätze hinweg über Zufallsfunde hinaus Sinn ergeben. Sie verharren letztlich in der Parodie des gelernten Materials. Zudem können sie nur Wortfolgen fortsetzen, die sie exakt so schon gesehen haben. In Kapitel 14 zum Thema *Transformer* werden wir Techniken kennenlernen, mit denen sich diese Probleme überwinden lassen.

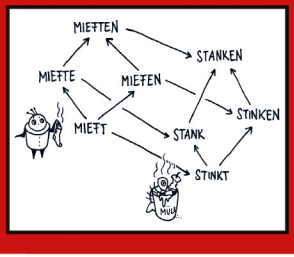
Künstliche Intelligenz verstehen



Hereinspaziert! Über ein Dutzend interaktive Projekte warten auf Ihre Neugierde. Schauen Sie zu, wie ein Algorithmus z. B. Gedichte schreibt oder Spiele gewinnt – oder gewinnen Sie?

Im Buch erfahren Sie, was jeweils dahintersteckt, und lernen einschlägige KI-Verfahren kennen. Von der Sprachkorrektur über Neuronale Netze bis zu Transformern, der Technologie hinter ChatGPT.

+ Grundlagen aus der Informatik
+ K-means-Clustering
+ Q-Learning
+ Transformer



> *»Konsequent alltagsnah und ungewöhnlich unterhaltsam präsentiert dieses Lehrbuch wichtige KI-Themen.«*

Alle Beispielprogramme online ausprobieren und mit dem Code experimentieren

Das Autorenteam verbindet Kunst mit KI, Sound mit Grafik und Spaß mit Code. Sie arbeiten digital, auf Leinwand, in transdisziplinären Projekten und in immer neuen Formaten. **Pit Noack** (Text & Code) begeistert in seinen Workshops für KI und Programmierung. Der Künstlerin **Sophia Sanner** (Illustration) ist kein Thema zu komplex, um es mit Cartoons und Infografiken auf humorvolle Weise anschaulich zu machen.

Rheinwerk Computing

€ 29,90 [D] € 30,80 [A]

Gedruckt in Deutschland
Papier aus nachhaltiger Waldwirtschaft
Mineralölfreie Druckfarben

Data Science und KI
ISBN 978-3-8362-9858-2