Jochen HIRSCHLE

Deep Natural Language Processing

Einstieg in Word Embedding, Sequence-to-Sequence-Modelle und Transformer mit Python

Online:
GitHub-Repository zum Buch

HANSER

Hirschle

Deep Natural Language Processing

Für Tuuli-Marja

Jochen Hirschle

Deep Natural Language Processing

Einstieg in Word Embedding, Sequence-to-Sequence-Modelle und Transformers mit Python

HANSER

Der Autor:

Dr. Jochen Hirschle, Braunschweig

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

Lektorat: Sylvia Hasselbach
Copy editing: Sandra Gottmann, Wasserburg
Umschlagdesign: Marc Müller-Bremer, *www.rebranding.de*, München
Umschlagrealisation: Max Kostopoulos
Titelmotiv: © *gettyimages.de/MR.Cole_Photographer*
Layout: Manuela Treindl, Fürth
Druck und Bindung: Kösel, Krugzell
Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702
Printed in Germany

Print-ISBN: 978-3-446-47363-8
E-Book-ISBN: 978-3-446-47390-4
E-Pub-ISBN: 978-3-446-47409-3

Inhalt

1 Einleitung

Wer die Berichterstattung zum Thema künstliche Intelligenz in den Medien verfolgt, dem wird aufgefallen sein, dass die erste Euphorie inzwischen deutlich abgeklungen ist. Das liegt nicht nur daran, dass sich aufgrund der Vielzahl an Berichten ein Gewöhnungseffekt eingestellt hat. Auch haben in vielen Bereichen die Entwicklungen den hohen Erwartungen nicht standgehalten.

Anstatt es uns auf den Rücksitzen autonomer Fahrzeuge bequem zu machen, wie von vielen ExpertInnen noch vor wenigen Jahren vollmundig angekündigt[1], steuern wir unsere Autos im Jahr 2022 noch immer gestresst und von Hand durch die Staus der Innenstädte. Auch um die allgemeine künstliche Intelligenz (*General AI*) ist es inzwischen verdächtig still geworden. Dabei jagte zuvor ein aufsehenerregendes KI-Ereignis das nächste. Man denke nur an IBMs *Watson*, der medienwirksam mehrfach in der Quizshow Jeopardy gegen die Champions gewinnen konnte [Franklin2014]. Oder auch Googles *Alpha Go*, das im gleichnamigen Brettspiel gegen den weltbesten Go-Spieler antrat und ihn besiegte. Viele glaubten, dass die Entwicklungen in diesem atemberaubenden Tempo weitergehen würden und dass die Zukunftsvisionen aus Science-Fiction-Filmen und Serien, in denen Humanoide von Menschen kaum noch zu unterscheiden sind, schon bald in greifbare Nähe rücken würden.

Tatsächlich ist der Alltag künstlicher Intelligenz heute wesentlich grauer geworden. Schlagzeilen machen KIs heute eher, weil sie von zweifelhaften Regimen zur umfassenden Überwachung von Menschen eingesetzt werden oder weil sie mit rassistischen oder sexistischen Verfehlungen auffallen [West2019]. Und die ersten Kontakte mit den kleinen Schwestern der Film-Supercomputer verlaufen auch nicht immer so, wie wir uns das vorgestellt haben. Wer hat nicht schon holprige Konversationen mit Alexa geführt, sich mit automatischen Antwortsystemen am Telefon herumgeschlagen, sinnlose Wortvervollständigungen auf Smartphones rückgängig gemacht oder ist mit dem Chatbot auf der Seite einer Bank oder Versicherung aneinandergeraten?

Womöglich hat Judea Pearl (ein Pionier der KI-Entwicklung) recht, wenn er etwas spöttisch zu Protokoll gibt, dass die eindrucksvollen Ergebnisse intelligenter Systeme im Wesentlichen auf *Curve Fitting* zurückgingen. Was er damit meint, ist, dass KI-Systeme heute zwar in der Lage sind, beliebig komplexe Regelmäßigkeiten in Daten aufzuspüren. Was sie dagegen nicht können, ist, zwischen kausalen Beziehungen und bloßen Korrelationen zu unterscheiden [Pearl2018]. Dadurch, so Pearl, seien die Möglichkeiten, weitere signifikante Fortschritte in diesem Bereich zu machen, eher begrenzt – die Revolution fällt also womöglich aus [Marcus2018].

[1] Vgl. *https://www.wired.com/story/self-driving-cars-challenges/* [Abgerufen am 11.09.2021]

Ob das stimmt oder nicht, sei dahingestellt. Gewiss ist allerdings, dass der Siegeszug künstlich intelligenter Systeme trotz aller Beschränktheit nicht aufzuhalten ist. Auch wenn die Anwendungen vielleicht nicht den Utopien oder Dystopien aus den Filmen nahekommen. Systeme, die Machine- und Deep-Learning-Verfahren implementieren, haben sich in den letzten Jahren in rasantem Tempo ausgebreitet. Kaum ein Geschäftsfeld, in dem maschinelle Lernalgorithmen nicht zum Einsatz kommen. Sei es im Marketing, im Journalismus, im Finanzwesen oder in der verarbeitenden Industrie [James2021a].

Dabei handelt es sich meist um kleinteilige, problemzentrierte Anwendungen, um *Narrow AI* im Vergleich zu *General AI*. Solche Algorithmen erledigen spezifische Aufgaben für uns: sie übersetzen, produzieren Textzusammenfassungen, handeln an der Börse, warnen vor möglichen Ausfällen eines Aggregats, machen auf Hackerangriffe aufmerksam, generieren Antworten auf Fragen oder bestimmen, welche Videos und welche Werbung wir sehen und welche Posts in unseren Newsfeeds landen.

Wie umfassend diese Vereinnahmung bereits ist, lässt sich nur schwer sagen. Meist können wir von außen nicht sicher sein, ob eine Anwendung von einem maschinellen Lernsystem betrieben wird oder nicht. Sicher ist nur, dass die Technologie bereits heute Teil unseres Alltags ist und aus diesem Alltag nicht mehr verschwinden wird. Wir werden uns daran gewöhnen müssen, in Zukunft noch häufiger mit selbstlernenden Automaten zu interagieren und womöglich mit ihnen zu kooperieren. Viele Berufsbilder werden unter Anpassungsdruck geraten, sich wandeln, und Tätigkeiten, für die bis vor Kurzem noch menschliche Arbeitskraft und kognitive Fertigkeiten notwendig waren, werden ganz oder teilweise von Algorithmen übernommen [Acemoglu2018].

Dieses Buch handelt von dieser Art künstlicher Intelligenz. Von Anwendungen, die im Hintergrund operieren. Keine davon wird vermutlich bei Ihnen das Gefühl auslösen, dass Sie es mit einem künstlichen Bewusstsein à la *HAL* oder einem der Androiden aus *Westworld* zu tun haben. Das heißt aber nicht, dass diese Anwendungen nicht auf ihre Weise intelligent und faszinierend sind.

Automaten, die menschliche Sprache imitieren, hinterlassen bei den meisten Menschen einen etwas ambivalenten Eindruck. Dass ein Algorithmus einen Satz produziert, der aus der Feder eines Philosophen oder Dichters stammen könnte, ist zumindest gewöhnungsbedürftig. Je mehr man aber hinter die Kulissen blickt und versteht, wie solche Systeme arbeiten, was sie lernen und was sie nicht lernen und wovon ihre Leistung abhängt, umso weniger mysteriös wirken sie.

Was Ihnen dieses Buch bietet

Dieses Buch bietet eine Einführung in den Bereich der Verarbeitung natürlicher Sprachdaten (Natural Language Processing) mit Deep Learning-Verfahren. Es richtet sich an alle LeserInnen, die sich für diese Schnittmenge aus linguistischer Analyse und maschinellem Lernen interessieren. An Personen, die genauer wissen möchten, wie sich Textdaten mit den neuesten statistischen Verfahren verarbeiten lassen und wie Übersetzungen, Textvervollständigungen, Textzusammenfassungen, Sentiment-Analysen und andere textbasierte Anwendungen in der Theorie funktionieren und wie sie sich in der Praxis implementieren lassen.

Um mit diesem Buch arbeiten zu können, sollten Sie ein wenig Programmiererfahrung mit Python mitbringen. Sie sollten wissen, wie Sie eine virtuelle Umgebung mit *Anaconda* aufsetzen, wie sich Pakete installieren lassen, und Sie sollten mit der Arbeit mit Klassen, Funktionen, Schleifen und Standardbehältern wie Listen, Dictionaries, Tuples und Sets ver-

traut sein. Außerdem brauchen Sie ein paar grundlegende Konzepte aus der Statistik. So sollten Sie sich unter Begriffen wie *Variable*, *Verteilung*, *Mittelwert* und *Standardabweichung* etwas vorstellen können und generell keine größeren Phobien vor Zahlen haben. Viel mehr brauchen Sie nicht.

Dies ist kein Buch, in dem es in erster Linie um die mathematischen Grundlagen des Deep Learnings geht. Der Text kommt in weiten Teilen ohne Formeln aus, obwohl wir sie manchmal der Vollständigkeit halber abdrucken. Im Vordergrund stehen textbasierte Erklärungen der wichtigsten Konzepte und eine Umsetzung dieser Konzepte mithilfe gängiger Python-Frameworks wie *TensorFlow/Keras*, *Scikit-Learn* oder *Transformers*. Wenn Sie im Detail wissen möchten, wie die Algorithmen mathematisch implementiert sind, sollten Sie parallel ein anderes Buch lesen. Wenn Ihnen mathematische Formeln generell näherstehen als textbasierte Erklärungen, ist dieses Buch vermutlich eher nicht die richtige Wahl für Sie.

Wenn Sie sich aber dafür entscheiden, dieses Buch zu lesen, haben Sie am Ende eine ziemlich klare Vorstellung darüber, wie moderne Verfahren in der statistischen Sprachanalyse arbeiten. Sie lernen nicht nur die Funktionsweise, die Möglichkeiten und Grenzen dieser Verfahren kennen. Sie verstehen auch die Grundprinzipien rekurrenter und konvolutionaler Netze und wissen, wie moderne Architekturen wie Sequence-to-Sequence-Modelle oder Transformer arbeiten. Außerdem ist Ihnen klar, welche Verfahren für welche Aufgaben geeignet sind, und Sie können die meisten dieser Verfahren selbst implementieren. Sie können eigene neuronale Netze aufbauen und anlernen, um Texte zu klassifizieren, zu generieren oder in eine andere Sprache zu übersetzen.

Darüber hinaus bietet Ihnen dieses Buch einen Einstieg in die Verwendung neuronaler Netze, die von anderen vortrainiert und zur Verfügung gestellt werden. Sie lernen, wie Sie allgemeine Sprachmodelle, die über sehr große Datenmengen angelernt wurden, laden und einsetzen können, und Sie lernen, wie Sie diese Netze nachtrainieren können, um sie für spezifische Aufgaben nutzbar zu machen.

Welche Inhalte Sie erwarten

Dieses Buch bietet eine schrittweise und kompakte Einführung in den Themenkomplex Machine Learning/Natural Language Processing (NLP). Die Kapitel bauen inhaltlich aufeinander auf. Wenn Sie also eines der Kapitel lesen, sollten Sie die Inhalte aus den vorherigen Kapiteln kennen. Konzepte, Klassen oder Funktionen werden in der Regel nur an einer Stelle erklärt, und Sie können davon ausgehen, dass einmal eingeführtes Wissen später wieder gebraucht wird. Wenn Sie neu in diesem Gebiet sind, ist es daher ratsam, die Kapitel nacheinander zu lesen, damit das didaktische Konzept aufgeht. Wenn Sie schon über Kenntnisse im einen oder anderen Bereich verfügen, können Sie natürlich einzelne Kapitel oder auch mehrere Kapitel überspringen.

Wir starten unsere Reise in die Welt des Deep Learnings mit einer allgemeineren Einführung in den Bereich maschinellen Lernens. Bevor wir uns an komplexen Modellen abarbeiten, sollten wir eine konkrete Vorstellung davon haben, wie einfache Modelle funktionieren. Wir machen uns also mit den Grundlagen statistischer Lernverfahren und den Basics im Umgang mit Daten in Python vertraut.

Deshalb beginnen wir im Anschluss an die Einleitung im **zweiten Kapitel** mit einer Einführung in die Behälterklassen, die Python standardmäßig bietet, und in einige externe Pakete und Funktionen wie *NumPy*, die uns die Arbeit mit Textdaten und Matrizen erleichtern.

Das **dritte Kapitel** widmet sich den allgemeinen Grundlagen maschinellen Lernens. Wir schauen uns an, wie einfache Lernzellen arbeiten, wie man Beziehungen zwischen *x*- und *y*-Variablen modelliert, was es mit der Optimierung von Gewichten und dem Gradientenabstiegsverfahren auf sich hat und wie sich mithilfe von Aktivierungsfunktionen beliebige Zielwerte (stetige und kategoriale Werte) produzieren lassen.

In **Kapitel 4** steigen wir dann in die Analyse von Textdaten ein. Dabei geht es zunächst um bewährte Vorverarbeitungsverfahren wie den Bag-of-Words-Ansatz, der es ermöglicht, Textdaten auf sehr basale Art mit maschinellen Standardverfahren zu verarbeiten. **Kapitel 5** bietet im Anschluss einen Einstieg in das eigentliche Zielthema des Buches: in die Arbeit mit neuronalen Netzen. Auch hier müssen wir uns zuerst einiger grundlegender Verfahren versichern, ehe wir uns um die speziellen, auf die Interpretation und Produktion von Texten zugeschnittenen Architekturen kümmern. In diesem Kapitel geht es daher um die Basics, die für die praktische Arbeit mit neuronalen Netzen erforderlich sind. Angefangen von der Funktionsweise einzelner Neuronen, über Schichten von Neuronen, das Durchschleusen von Daten durch diese Schichten bis hin zur Verlustfunktion und der Optimierung der Gewichte mithilfe des Backpropagation-Mechanismus. Danach wissen Sie, wie ein neuronales Netz lernt, wie Gewichte initialisiert und optimiert werden und welche Aufgaben Ihnen bei der Zusammenstellung eines solchen Netzes zukommen.

Mit diesem Wissen ausgerüstet, erörtern wir in **Kapitel 6** eine erste, für die Verarbeitung von Textdaten konzipierte Netzarchitektur: rekurrente Netze. Dabei handelt es sich um ein Verfahren, das im Speziellen die zeitliche Abfolge der Präsentation von Zeichen – zum Beispiel von Buchstaben in Wörtern oder von Wörtern in Sätzen – berücksichtigt und auf dieser Grundlage Interpretationen erzeugt. In **Kapitel 7** geht es nahtlos mit konvolutionalen Netzen weiter. Zwar hat sich dieses Verfahren vor allem bei der Verarbeitung von Bilddaten bewährt, es lässt sich aber ohne Weiteres auf Sequenzdaten übertragen. Die Vor- und Nachteile gegenüber rekurrenten Netzen sehen wir uns dabei anhand einer automatischen Rechtschreibkorrektur genauer an. Das folgende **Kapitel 8** thematisiert dann die Vorverarbeitung von Wörtern mit Worteinbettungsverfahren. Damit lassen sich Wörter bzw. Token als Vektoren in einem mehrdimensionalen Raum unter Berücksichtigung semantischer und grammatikalischer Beziehungen darstellen. Das Verfahren bildet heute die Grundlage für fast alle Deep Learning-Ansätze, die Textdaten als Rohmaterial verwenden. Wir beleuchten dabei nicht nur die Möglichkeiten, Worteinbettungen selbst anzulernen, sondern führen auch in die Verwendung vortrainierter Vektorräume wie *fastText* oder *Word2Vec* ein.

Beginnend mit **Kapitel 9** eruieren wir dann die Optionen, die komplexe neuronale Netze bei der Textinterpretation und Textgenerierung bieten. Insbesondere geht es um Encoder-Decoder-Modelle. Sie sind aus der modernen Textanalyse (sei es in Sequence-to-Sequence- oder in Transformer-Modellen) nicht mehr wegzudenken. Zunächst dreht sich aber alles um das Handwerkszeug, das wir brauchen, um solche Modelle praktisch umzusetzen: Es geht darum, wie wir Netze, die über zwei Eingänge verfügen oder in denen Datenströme bestimmte Schichten überspringen, aufsetzen können. Diese Kenntnisse sind notwendig, wenn wir im nachfolgenden **Kapitel 10** Sequence-to-Sequence-Modelle genauer unter die Lupe nehmen. Mit dieser Architektur lassen sich nicht nur einzelne Wörter, sondern ganze Sätze oder Textpassagen erzeugen. Solche Architekturen werden für Übersetzungen oder Textzusammenfassungen eingesetzt. Darüber hinaus lernen wir in diesem Kontext den Attention-Mechanismus kennen, der in einer Variante, der Self-Attention, auch in Transformer-Modellen Karriere gemacht hat. Das ist das Thema von **Kapitel 11**. Darin zeigen wir

nicht nur, wie Transformer-Modelle funktionieren, sondern wir sehen uns auch an, wie sich vortrainierte Modelle, die über den Transformer-Hub *Hugging Face* vertrieben werden, sich für verschiedene Aufgaben nachtrainieren lassen. Das Buch schließt im **zwölften Kapitel** mit einer Zusammenfassung einiger grundlegender Konzepte und einer Diskussion der Stärken und Schwächen neuronaler Netze.

Wie Sie mit den Codebeispielen arbeiten können

Die meisten Kapitel bestehen aus einem Mix aus Einführungen in Konzepte maschinellen Lernens und aus einem praxisorientierten Teil, in denen diese Konzepte mit Python umgesetzt werden. Wenn Sie die Beispiele auf Ihrem Rechner laufen lassen möchten, müssen Sie die Voraussetzungen dafür schaffen.

Wir verwenden Python in der *Version 3.7.6*, eingebettet in eine virtuelle Umgebung, die über die Data Science-Plattform *Anaconda* verwaltet wird. Anaconda bietet unter anderem den Vorteil, dass es bei der Installation externer Pakete sicherstellt, dass die Pakete untereinander harmonieren. Da wir einige Pakete installieren müssen und da diese Pakete in ihrer Arbeit wiederum auf weitere Unterpakete angewiesen sind, ist das ziemlich hilfreich.

Um also arbeitsfähig zu sein, benötigen Sie die folgenden Bibliotheken in Ihrer virtuellen Umgebung:

```
matplotlib, Version=3.4.2
nltk, Version=3.6.2
numpy, Version=1.19.2
pandas, Version=1.0.3
scikit-learn, Version=0.22.1
tensorflow, Version=2.3.0
transformers, Version=4.10.2
```

Bis auf *Transformers* (das zum Zeitpunkt der Drucklegung mit pip installiert werden muss) sind alle Bibliotheken über Anaconda verwaltbar (installieren mit *conda install*). Sie können den Code vermutlich auch mit anderen, aktuelleren Versionen von Python und den genannten Bibliotheken ausführen. Allerdings ist es möglich, dass Sie dann an der einen oder anderen Stelle auf Warnungen, Fehlermeldungen oder auf andere Probleme stoßen, die wir nicht vorhersehen können.

Die Codebeispiele im Buch sind über *GitHub* als *Jupyter Notebooks* verfügbar. Sie können den Code entweder im Internet unter *https://github.com/tplusone/hanser_deep_nlp* abrufen und ansehen oder, wenn Sie die Software *Git* installiert haben, das gesamte Repository inklusive Codebeispielen und Beispieldaten über das Terminal mit dem folgenden Befehl auf Ihren Rechner ziehen:

```
git clone https://github.com/tplusone/hanser_deep_nlp.git
```

2 Textdaten verarbeiten und vorverarbeiten

Wenn es um die Verarbeitung numerischer und textbasierter Informationen geht, hat Python gegenüber anderen Programmiersprachen ein paar entscheidende Vorteile. Schon die Standardbibliothek bietet eine Vielzahl einfacher Funktionen und Klassen, um Textdaten in Form zu bringen, zu transformieren und zu strukturieren. So ist es ein Leichtes, einmal tokenisierte Texte in Arrays zu verwalten, Auszüge zu extrahieren oder mithilfe von Schleifen oder List Comprehensions Transformationen durchzuführen. Auch Casting-Operationen, die Arrays in Sets, Tuples oder Dictionarys verwandeln, funktionieren in den meisten Fällen vorhersehbar und problemlos. So lassen sich Texte für das Anlernen in Form bringen.

Um einen maschinellen Lernalgorithmus zu trainieren, brauchen wir aber andere Datenklassen. Statistische Verfahren operieren mit mathematischen Funktionen und mögen daher keine Überraschungen, wenn es um Datentypen und die Struktur der Datenbehälter geht. Da eines der Grundprinzipien in Python aber die Abkehr von Typsicherheit zugunsten von Duck-Typing ist, benötigen wir eine Alternative, die dieses Anforderungsprofil erfüllt. Die Bibliothek *NumPy* (Numerical Python) ist dabei die erste Wahl. NumPy ist zum Glück auf die Datenklassen der Standardbibliothek abgestimmt, sodass Castings in beide Richtungen reibungslos funktionieren.

Im Folgenden sehen wir uns einige ausgewählte Techniken, Klassen und Bibliotheken an, die für die Vorverarbeitung von Texten zur Analyse mit Lernalgorithmen von Bedeutung sind. Dabei handelt es sich um keine auch nur annähernd vollständige Abhandlung der verschiedenen Möglichkeiten. Wir werfen aber einen Blick auf die Verfahren, die in den nachfolgenden Kapiteln immer wieder verwendet werden und die wir dort nicht noch einmal im Detail vorstellen. Neben der Tokenisierung und numerischen Encodierung von Wörtern geht es um die Repräsentation von Wörtern als One-Hot-Sets und natürlich um die Arbeit mit der Bibliothek NumPy.

2.1 Grundlegende Techniken der Verarbeitung von Textdaten

Texte liegen nach dem Einlesen in Python normalerweise als Strings vor. Eine Besonderheit der String-Klasse in Python ist, dass sie sich wie ein Iterable verhält. Die einzelnen Buchstaben werden als Characters auf Indexpositionen abgespeichert. Man kann deshalb über

einen String sowohl iterieren als auch über den Aufruf einer Indexposition bzw. über Slicing einzelne oder mehrere Buchstaben herauslösen:

```
1.  text = 'Die Sonne steht hoch am Himmel.'
2.  text[5], text[5:10]
```

Ausgabe:

```
('o', 'onne ')
```

Da in vielen statistischen Anwendungen Wörter oder Token als kleinste Analyseeinheiten fungieren, müssen wir Texte fast immer auf dieser Ebene zerlegen. Dieser Vorgang nennt sich *Tokenisierung*. Wir könnten uns mit einem Regex zwar einen eigenen Tokenizer zusammenbauen, einfacher geht es allerdings mit einem getesteten Produkt, das Wörter und Satzzeichen an verschiedenen Positionen erkennt und extrahiert. Die *word_tokenize*-Funktion aus dem NLTK-Modul erledigt genau diese Arbeit und gibt bei Übergabe eines Strings eine Liste der enthaltenen Wörter inklusive Satzzeichen zurück:

```
3.  from nltk.tokenize import word_tokenize
4.
5.  text = 'Die Sonne steht hoch am Himmel.'
6.  word_tokenize(text)
```

Ausgabe:

```
['Die', 'Sonne', 'steht', 'hoch', 'am', 'Himmel', '.']
```

In längeren Texten wiederholen sich die Wörter in der Regel mehrfach. Wenn wir von jedem dieser Wörter jeweils nur ein Exemplar behalten möchten, können wir die Liste in ein Set umwandeln. Eine angenehme Nebenwirkung dieses Vorgangs ist, dass automatisch alle Duplikate eliminiert werden:

```
7.  words = ['die', 'Sonne', 'Sonne', 'scheint', 'scheint']
8.  set(words)
```

Ausgabe:

```
{'Sonne', 'die', 'scheint'}
```

Wie man Wörter in numerische Form bringt, um damit einen Machine Learning-Algorithmus zu füttern, sehen wir uns im nächsten Abschnitt genauer an. Manchmal brauchen wir allerdings keine spezielle Encodierung, sondern lediglich eine numerische Repräsentation der Wörter aus den Trainingsdaten (zum Beispiel, wenn wir eine Zielvariable vorbereiten). In diesem Fall bietet es sich an, zur Encodierung ein Dictionary zu verwenden. Es enthält für jedes Wort (*Key*) eine Ganzzahl (*Value*). Damit können wir die einzelnen Wörter aus einer Textsequenz nachschlagen und encodieren. In einem zweiten (zur Decodierung vorgesehenen) Dictionary sind die Keys und Values vertauscht: Bei Übergabe einer Ganzzahl erhalten wir das zugeordnete Wort zurück.

Die Ordnung der Wörter und Ganzzahlen in den Dictionarys können wir letztlich willkürlich festlegen, da es für den Lernalgorithmus unerheblich ist, welche Ganzzahlen für welche Wörter stehen. Wichtig ist nur, dass jedem Wort je eine eigene Ganzzahl zugeordnet ist. Allerdings bietet es sich natürlich an, die Reihenfolge der Zahlen mit der alphabetischen Ordnung der Wörter in Einklang zu bringen:

```
9.  words = ['die', 'Sonne', 'Sonne', 'scheint', 'scheint']
10. words = list(set(words))
11. words = [ word.lower() for word in words ]
12. words.sort()
13. word_index = dict([ (word, idx) for idx, word in enumerate(words) ])
14. index_word = dict([(idx, word) for word, idx in word_index.items()])
15. word_index, index_word
```

Ausgabe:

```
({'die': 0, 'scheint': 1, 'sonne': 2},
 {0: 'die', 1: 'scheint', 2: 'sonne'})
```

Code-Hinweise:

In *Zeile 10* produzieren wir aus den tokenisierten Wörtern des Textes zunächst ein Set, um die Duplikate zu entfernen. Dieses Set casten wir danach (in der gleichen Zeile) wieder als Liste, weil wir nur Listen sortieren können. In *Zeile 11* wandeln wir die Buchstaben der einzelnen Wörter innerhalb der Liste unter Verwendung einer List Comprehension in Kleinbuchstaben um. Danach (*Zeile 12*) sortieren wir die Liste alphabetisch und aufsteigend mit der sort-Methode.

In *Zeile 13* produzieren wir das erste Dictionary in einem Aufwasch. Die enumerate-Funktion gibt uns bei der Iteration über die Liste der Wörter automatisch Integer-Werte für jedes Element (Wort) zurück. Die Werte beginnen bei 0 und werden bei jedem Iterationsschritt um den Wert 1 inkrementiert. Diese Zahlen verwenden wir, um die Encodierung der Wörter festzulegen. Da die Liste sortiert ist, erhalten wir eine alphabetisch informierte Encodierung. Wie man sieht, arbeiten wir die Wörter mit einer List-Comprehension ab und erzeugen deswegen das Dictionary erst nach Fertigstellung der Liste. Damit das Casting mit *dict* funktioniert, platzieren wir als Elemente der Liste einfach das Wort (*word*) und die zugehörige Ganzzahl (*idx*) in ein Tuple. Bei der Umwandlung in ein Dictionary liest Python das Tuple automatisch als Key-Value-Paar aus.

Bei der Erzeugung des Index-to-Word-Dictionarys in *Zeile 14* gehen wir auf Nummer sicher. Wir lesen das bereits vorhandene Word-to-Index-Dictionary unter Verwendung der items-Methode aus, die sowohl Keys als auch Values zurückgibt. In der List-Comprehension vertauschen wir dann einfach die Positionen der beiden Werte im Tuple und erzeugen daraus ein neues Dictionary.

Mit dem so geschaffenen Dictionary (*word_index*) können wir jetzt weiterarbeiten. Wir können damit zum Beispiel eine einfache Encodierung von Wörtern als Ganzzahlen erzeugen, die – wie wir später sehen werden – für verschiedene Aufgaben nützlich ist. Nehmen wir zum

Beispiel an, wir wollten den tokenisierten und in Kleinbuchstaben konvertierten Satz *„Die Sonne scheint“* als Sequenz von Ganzzahlen encodieren. Die Zuordnung eines jeden Wortes zu einer Ganzzahl lässt sich jetzt ganz einfach über das Dictionary herausfinden und die Zuordnung bewerkstelligen:

```
16. text = ['die', 'sonne', 'scheint']
17. encoded = [ word_index[word] for word in text]
18. encoded
```

Ausgabe:

```
[0, 2, 1]
```

2.2 Mit NumPy arbeiten

Die Hauptklasse der NumPy-Bibliothek ist das NumPy-Array (*ndarray*). NumPy-Arrays sind Behälterklassen, deren Datentyp und Datenstruktur schon bei der Erzeugung festgelegt werden müssen und unveränderlich sind. NumPy-Arrays sind nicht nur typsicher, sie verhalten sich auch wie altmodische Array-Behälterklassen: Sie können weder wachsen noch schrumpfen. Wir können sie also nicht schrittweise mit Elementen füllen oder Elemente herauslösen wie bei einer Liste in Python. Schon beim ersten Anlegen wird die Größe festgelegt, und es müssen alle vorgesehenen Positionen mit Werten vorbelegt werden. Ändern lassen sich im Nachhinein nur die Inhalte.

Instanziierung und Operationen mit NumPy-Arrays

Im Folgenden legen wir ein Array in der Form einer Matrix an, das drei Zeilen und fünf Spalten beinhaltet. Da wir kein leeres Array erzeugen können, belegen wir die Zellen mit Nullen vor, wobei wir den Datentyp als Integer 32 (32 Bit) definieren:

```
1.  import numpy as np
2.  matrix = np.zeros( shape=(3, 5), dtype=np.int32 )
3.  matrix
```

Ausgabe:

```
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
```

Nach der Definition können wir über die Indexpositionen Einträge austauschen oder verändern. Dabei muss die vordefinierte Struktur erhalten bleiben, ansonsten erhalten wir eine Fehlermeldung.

Mit dem folgenden Code fügen wir in die zweite Spalte der ersten Zeile ein Integer ein:

```
4.  matrix[0, 1] = 2
5.  matrix
```

Ausgabe:

```
array([[0, 2, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
```

Verschachtelte Indexpositionen können bei NumPy-Arrays sowohl, wie hier dargestellt, getrennt durch Kommata als auch, wie bei Listen, über hintereinander gesetzte eckige Klammern [0][1] adressiert werden.

Jetzt platzieren wir in die erste Zeile der Matrix einen Vektor, der die Skalare 0, 1, 2, 3 und 4 enthält:

```
6.  matrix[1] = np.arange(start=0, stop=5, step=1)
7.  matrix
```

Ausgabe:

```
array([[0, 2, 0, 0, 0],
       [0, 1, 2, 3, 4],
       [0, 0, 0, 0, 0]])
```

Die *arange*-Funktion erzeugt ein NumPy-Array, das mit Werten im Bereich *start* (inklusiv) bis *stop* (exklusiv) gefüllt wird. Im Unterschied zur *range*-Funktion der Python-Standardbibliothek kann *np.arange* nicht nur Ganzzahlen in einem Bereich erzeugen, sondern auch Fließkommazahlen.

Die Ersetzung eines Vektors innerhalb der Matrix funktioniert allerdings nur, wenn die Struktur der vordefinierten Matrix eingehalten wird. Der im Codefeld unten erzeugte Vektor weist eine Länge von 4 auf und kann deshalb an der Indexposition, die einen Vektor der Länge 5 enthält, nicht eingefügt werden:

```
8.  matrix[2] = np.arange(start=0, stop=4, step=1)
```

Ausgabe:

```
ValueError: could not broadcast input array from shape (4) into shape (5)
```

Casting

Die Vorverarbeitung von Textdaten umfasst in der Regel mehrere Schritte. Zu Beginn arbeiten wir meist mit den dynamischen Klassen der Standardbibliothek: mit Listen, Sets oder Dictionarys, mit denen sich Daten flexibel handhaben lassen. Erst wenn wir ein statistisches Modell anlernen möchten, überführen wir die vorbereiteten Daten in NumPy-Arrays.

Das funktioniert in der Regel reibungslos, da sich ein- und mehrdimensionale Listen mit der *array*-Funktion aus NumPy ohne Weiteres in *ndarrays* überführen lassen. Der Datentyp wird dabei automatisch abgeleitet. Wenn unser Array verschiedene Datentypen enthält, entscheidet sich NumPy für einen Datentyp, in dem sich alle Elemente darstellen lassen (im Zweifelsfall, zum Beispiel, wenn Textdaten enthalten sind, handelt es sich dabei um den Datentyp *Object*):

```
1.  import numpy as np
2.  a_list = [[1, 2, 3], [2, 0.5, 1]]
3.  array = np.array(a_list)
4.  array, array.dtype
```

Ausgabe:

```
(array([[1. , 2. , 3. ],
        [2. , 0.5, 1. ]]),
 dtype('float64'))
```

Das funktioniert allerdings nur, wenn die Datenstruktur der Liste einheitlich ist. Wenn eine der verschachtelten Listen eine andere Länge hat als die anderen, erhalten wir – abhängig von der NumPy-Version – entweder eine Fehlermeldung oder ein Array, in dem die verschachtelten Listen als Elemente vom Typ *Object* und nicht als Arrays abgelegt sind.

Gänzlich unproblematisch funktioniert die Umwandlung eines NumPy-Arrays in eine Liste. Die eingebaute Methode *tolist* führt die Konversion durch. Dabei entsteht eine Liste, die die gleiche Struktur wie das NumPy-Array hat und die gleichen Datentypen beinhaltet:

```
5.  a_list = array.tolist()
6.  a_list, type(a_list)
```

Ausgabe:

```
([[1.0, 2.0, 3.0], [2.0, 0.5, 1.0]], list)
```

Broadcasting

Ein Vorteil der Arbeit mit NumPy-Arrays ist, dass sich viele mathematische Operationen auf Matrizen oder Vektoren ohne den Einsatz von Schleifen erledigen lassen. Wenn wir zum Beispiel auf alle Werte einer Matrix oder auch nur auf die Werte einer Spalte die gleiche Transformation anwenden möchten, können wir das ohne viel Aufhebens erledigen.

Nehmen wir als Beispiel die Standardisierung einer Datenmatrix. Das ist eine Operation, die wir in Machine Learning-Anwendungen häufig erledigen müssen. Warum, erklären wir später. Technisch gesehen müssen wir dabei von jedem einzelnen Wert der Matrix den Mittelwert subtrahieren und durch die Standardabweichung dividieren. Das lässt sich durch die Broadcasting-Funktionalität leicht bewerkstelligen. Wir tun einfach so, als handle es sich bei dem Array um einen einzelnen Skalar und definieren die gewünschten Operationen wie gewohnt:

```
1.  import numpy as np
2.
```

```
3.  mean = 4.1
4.  std = 0.8
5.  matrix = np.array([[4., 5., 2.],
6.                     [1., 6., 4]])
7.  matrix = (matrix - mean) / std
8.  matrix
```

Ausgabe:

```
array([[-0.125,  1.125, -2.625],
       [-3.875,  2.375, -0.125]])
```

Als Ergebnis erhalten wir eine Matrix, in der auf jeden einzelnen Eintrag die Operation mit den Skalaren (Subtraktion des Mittelwerts, Division durch Standardabweichung) durchgeführt wurde.

Auch können wir bestimmte Werte aus einer Matrix heraus berechnen, ohne dazu eine Schleife einzusetzen. NumPy bietet dafür eine Vielzahl gebräuchlicher Methoden und Funktionen an, die wir auf ein- oder mehrdimensionale Arrays anwenden können. Mithilfe des *axis*-Parameters lässt sich dabei zusätzlich bestimmen, ob als Grundlage einer Berechnung alle Werte der Gesamtmatrix oder nur die Werte einzelner Spalten oder Zeilen verwendet werden sollen:

```
9.  matrix = np.array( [[4., 5., 3.],
10.                    [2., 6., 4]] )
11. matrix.mean(), matrix.mean(axis=0), matrix.mean(axis=1)
```

Ausgabe:

```
(4.0, array([3. , 5.5, 3.5]), array([4., 4.]))
```

Bei einer Matrix erhalten wir ohne Einstellung des *axis*-Parameters mit der *mean*-Methode den Gesamtmittelwert über alle Elemente, bei *axis=0* den Mittelwert über die Spalten und bei *axis=1* den Mittelwert über die Zeilen.

Die Broadcasting-Funktion versteht dabei auch komplexere Operationen; zum Beispiel, wenn wir spaltenweise Mittelwerte von den Elementen einer Matrix subtrahieren wollen, und dazu ein Array mit den Spaltenmittelwerten übergeben:

```
12. matrix = np.array( [[4., 5., 3.],
13.                    [2., 6., 4]])
14. col_means = np.array([3. , 5.5, 3.5])
15.
16. matrix = matrix - col_means
17. matrix
```

Ausgabe:

```
array([[ 1. , -0.5, -0.5],
       [-1. ,  0.5,  0.5]])
```

Restrukturierung eines Arrays

Häufig kommt es auch vor, dass wir ein Array in eine bestimmte Struktur bringen müssen, weil ein Lernalgorithmus die Daten nur genau in dieser Form verarbeiten kann. Das ist immer dann problematisch, wenn die natürliche Struktur eines solchen Arrays nicht dem entspricht, was der Lernalgorithmus als Eingabe verlangt.

NumPy-Arrays lassen sich zum Glück relativ leicht restrukturieren. Mit der *reshape*-Methode können wir einen Vektor, eine Matrix oder einen multidimensionalen Tensor in alle möglichen Formen konvertieren. Dabei übergeben wir einfach die gewünschte Struktur als Tuple der reshape-Methode. NumPy versucht dann, die Anforderung umzusetzen.

Sehen wir uns das an einem Beispiel an. Nehmen wir an, wir hätten den folgenden mit Integern gefüllten Vektor:

```
1.  import numpy as np
2.  x = np.array([3, 4, 2, 1], dtype=np.int32)
3.  x
```

Ausgabe:

```
array([3, 4, 2, 1])
```

Wenn wir diesen Vektor in eine 2-x-2-Matrix umwandeln möchten, übergeben wir der reshape-Methode einfach die neue Struktur in einem Tuple:

```
4.  x.reshape((2,2))
```

Ausgabe:

```
array([[3, 4],
       [2, 1]])
```

NumPy versucht, die angeforderte Form möglichst intuitiv abzuleiten. In diesem Fall wird der Vektor in der Mitte geteilt, und die beiden Teile werden in separate eindimensionale Arrays gepackt.

Nehmen wir aber an, wir möchten aus dem Vektor zwar ein zweidimensionales Array machen, dabei soll die Basisstruktur der Daten jedoch unverändert bleiben. Es soll also einfach jeder einzelne Wert des ursprünglichen Vektors in ein eigenes Array verpackt werden. Auch das lässt sich sehr einfach bewerkstelligen:

```
5.  x.reshape((4,1))
```

Ausgabe:

```
array([[3], [4], [2], [1]])
```

NumPy macht uns das Leben sogar noch leichter. Nehmen wir an, wir wissen nicht, wie viele Elemente der Vektor enthält. Wir wissen nur, dass wir jedes einzelne Element in ein extra Array einpacken möchten. In diesem Fall reicht es, den Strukturparameter zu spezifizieren,

den wir kennen; den anderen belegen wir einfach mit dem Wert *-1*. NumPy versucht dann rückzuschließen, wie der andere Strukturparameter unter diesen Bedingungen aussehen muss:

```
6.  x.reshape((-1, 1))
```

Ausgabe:

```
array([[3], [4], [2], [1]])
```

Das Ganze funktioniert natürlich nur, wenn sich die gewünschte Struktur aus dem Basisarray rekonstruieren lässt. Wenn wir aus einem Vektor mit vier Elementen ein zweidimensionales Array erzeugen möchten, das jeweils drei Elemente als Subarray enthält, dann bekommen wir die folgende Fehlermeldung:

```
7.  x.reshape((-1, 3))
```

Ausgabe:

```
ValueError: cannot reshape array of size 4 into shape (3)
```

2.3 One-Hot-Encodierung und Bag-of-Words-Modell

Statistische Verfahren können nur mit numerischen Daten arbeiten. Dieser Umstand ist für die Arbeit mit Textdaten ein größeres Hindernis als bei der Arbeit mit den meisten anderen Datenarten. Viele Informationen können in Zahlen ausgedrückt und verschiedene Werte sinnvoll miteinander verglichen werden. Wenn in Dortmund Fünfhunderttausend Menschen leben und in Köln eine Million, dann ist Köln doppelt so groß wie Dortmund. Auch in anderen Fällen sind Zahlenwerte (meist) aussagekräftig, zum Beispiel, wenn es um die Temperatur in einem Raum, das Einkommen einer Person oder die Note einer StudentIn geht. Selbst digitale Fotos bestehen aus einer Sammlung von Pixeln, die als numerische Farbsättigungen dargestellt werden. So lassen sich Farbtöne untereinander vergleichen.

Bei Texten ist das anders. Die Wörter und Token einer natürlichen Sprache sind spezielle Artefakte. Sie sind sowohl eigenständige Bedeutungsträger, stehen andererseits aber in semantischen und grammatikalischen Relationen zu anderen Wörtern. Diese Relationen (wie Synonymie, Antonymie, Hyperonymie oder Meronymie) lassen sich nur schwer als Zahlenwerte ausdrücken. Jedenfalls ist eine solche komplexe Repräsentation nicht mit einfachen Encodierungsverfahren zu erreichen. Man verwendet in neueren Anwendungen deshalb spezielle Algorithmen, die die Bedeutung und Relationen der Wörter in einem Raum von Vektoren ausdrücken. Dieses Verfahren sehen wir uns in Kapitel 8 genauer an.

Lange Zeit bestand jedoch die einzige Möglichkeit, Wörter mit statistischen Verfahren zu verarbeiten, darin, sie als qualitativ verschiedene Einheiten zu begreifen. Dabei verwendet man zwar Zahlen zur Encodierung, allerdings drücken diese Zahlen keine Relationen, sondern lediglich Unterschiedlichkeit aus. Diese Art der Encodierung wird in der Statistik auch zu anderen Zwecken eingesetzt. Meistens allerdings behelfsmäßig, wenn sich die Werte einer Variablen qualitativ unterscheiden, aber ein numerischer Vergleich nicht möglich ist. Typische Beispiele sind das Geschlecht einer Person oder deren beruflicher Status (erwerbstätig, arbeitslos, verrentet, im Haushalt tätig etc.).

In solchen Fällen setzt man auf die *One-Hot-Encodierung*. Der Trick besteht darin, für jede Ausprägung einer Variablen – in unserem Fall für jedes Wort aus einem Vokabular – eine extra Spalte in einer Matrix anzulegen, die entweder mit 0 oder 1 kodiert ist. Wenn es um Wörter geht, müssen wir für jedes Token eine extra Spalte vorsehen. Wenn das Token in einem Satz vorkommt, codieren wir die betreffende Spalte der Matrix mit einer 1, wenn es nicht vorkommt, mit einer 0.

Nehmen wir als Beispiel zwei kurze (bereits tokenisierte) Textnachrichten, die insgesamt nur vier Token beinhalten:

```
1.  text = [ ['die', 'sonne', 'scheint'],
2.           ['die', 'sonne', 'scheint', 'nicht'] ]
```

Nehmen wir an, wir erstellen ein Wort-Index-Dictionary mit folgenden Einträgen;

```
3.  word_index = {'die': 0, 'nicht': 1, 'scheint': 2, 'sonne': 3}
```

Auf dieser Grundlage können wir für beide Sätze eine Matrix erstellen, die das Vorkommen der Wörter (unabhängig von der Reihenfolge) in den Sätzen codiert. Da wir vier verschiedene Wörter in unserem Vokabular haben, benötigen wir pro Datensatz (in diesem Fall pro Satz) vier Spalten. Welches Wort an welcher Indexposition (Spalte) stehen soll, leiten wir aus den Integer-Werten des Dictionarys ab: *„die“* wird auf Indexposition *0*, *„nicht“* auf *1*, *„scheint“* auf *2* und *„sonne“* auf Indexposition *3* gelegt.

Der Code, um das zu bewerkstelligen, sieht dann zum Beispiel so aus:

```
4.  import numpy as np
5.
6.  ohe = np.zeros(shape=(2, 4), dtype=np.int32)
7.  for i, el in enumerate(text):
8.      for word in el:
9.          ohe[i, word_index[word]] = 1
10. ohe
```

Ausgabe:

```
array([[1, 0, 1, 1],
       [1, 1, 1, 1]])
```

Eine Variation dieser Repräsentation ist das *Bag-of-Words*-Modell. Dabei wird zusätzlich die Anzahl der Vorkommisse eines Wortes in einem Datensatz codiert. Der Satz *„die sonne, die scheint“* würde (wenn wir das Komma übergehen und mit dem Dictionary von oben arbeiten), entsprechend mit dem Array `[2, 0, 1, 1]` dargestellt, da das Wort *„die“* zweimal enthalten ist.