KAPITEL 2

Entwurfsmuster für die Datendarstellung

Das Herzstück eines jeden Modells für maschinelles Lernen ist eine mathematische Funktion, die so definiert ist, dass sie nur auf bestimmten Datentypen arbeitet. Gleichzeitig müssen reale ML-Modelle auf Daten operieren, die sich nicht direkt an die mathematische Funktion übergeben lassen. Zum Beispiel arbeitet der mathematische Kern eines Entscheidungsbaums mit booleschen Variablen. Wir sprechen hier über den mathematischen Kern eines Entscheidungsbaums – Software für maschinelles Lernen mit Entscheidungsbäumen umfasst in der Regel Funktionen, die einen optimalen Baum aus den Daten lernen, und Methoden, um verschiedene Typen von numerischen und kategorialen Daten einzulesen und zu verarbeiten. Die mathematische Funktion (siehe Abbildung 2-1), die einem Entscheidungsbaum zugrunde liegt, arbeitet jedoch mit booleschen Variablen und verwendet Operationen wie zum Beispiel `AND` (`&&` in Abbildung 2-1) und `OR` (`+` in Abbildung 2-1).

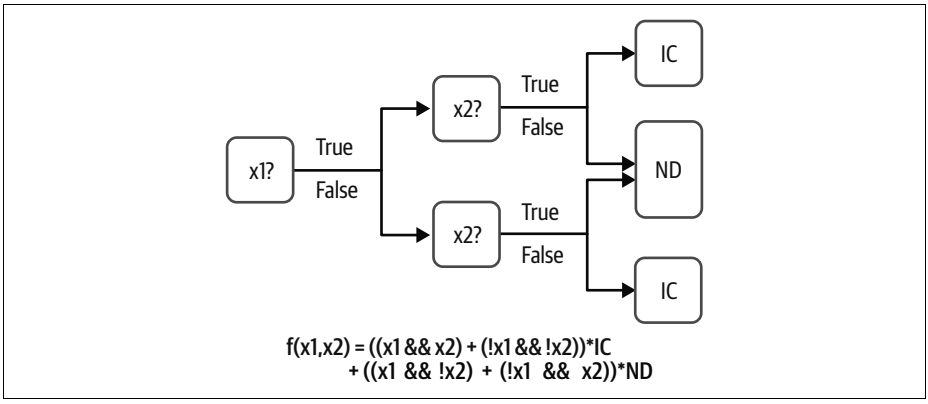

Abbildung 2-1: Das Herzstück eines Modells für maschinelles Lernen mit einem Entscheidungsbaum, um vorherzusagen, ob ein Baby Intensivpflege benötigt oder nicht, ist ein mathematisches Modell, das mit booleschen Variablen arbeitet.

Angenommen, wir wollten mit einem Entscheidungsbaum vorhersagen, ob ein Baby Intensivpflege (*Intensive Care*, IC) benötigt oder normal entlassen werden kann (*Normally Discharged*, ND). Des Weiteren nehmen wir an, dass der Entschei-

dungsbaum die beiden Variablen `x1` und `x2` als Eingaben übernimmt. Das trainierte Modell könnte dann wie in Abbildung 2-1 aussehen.

Es liegt auf der Hand, dass `x1` und `x2` boolesche Variablen sein müssen, damit `f(x1, x2)` funktioniert. Angenommen, zwei der Informationen, die das Modell betrachtet, um ein Baby zu klassifizieren (Intensivpflege oder nicht), seien das Krankenhaus, in dem das Baby geboren wurde, und das Gewicht des Babys. Können wir das Krankenhaus, in dem ein Baby geboren wurde, als Eingabe für den Entscheidungsbaum verwenden? Nein, weil das Krankenhaus weder den Wert `True` noch den Wert `False` annimmt und sich nicht mit dem Operator `&&` (`AND`) verknüpfen lässt. Es ist mathematisch nicht kompatibel. Selbstverständlich können wir den Krankenhauswert zu einem booleschen Wert »machen«, indem wir eine Operation wie die folgende ausführen:

```
x1 = (hospital IN France)
```

Hier nimmt die Variable `x1` den Wert `True` an, wenn sich das Krankenhaus in Frankreich befindet, sonst `False`. Analog dazu kann man das Gewicht eines Babys nicht direkt in das Modell einspeisen. Mit einer Operation wie:

```
x1 = (babyweight < 3 kg)
```

können wir aber das Krankenhaus oder das Babygewicht als Eingabe für das Modell verwenden. Dieses Beispiel zeigt, wie sich Eingabedaten (Krankenhaus, ein komplexes Objekt, oder Gewicht des Babys, eine Gleitkommazahl) in der vom Modell erwarteten Form (boolescher Wert) darstellen lassen. Und genau das meinen wir mit *Datendarstellung*.

In diesem Buch verwenden wir den Begriff *Eingabe* für die realen Daten, die in das Modell eingespeist werden (zum Beispiel das Babygewicht), und den Begriff *Feature*, um die transformierten Daten darzustellen, auf denen das Modell tatsächlich operiert (ob zum Beispiel das Gewicht des Babys geringer als drei Kilogramm ist). Der Prozess, der Features erzeugt, um die Eingabedaten darzustellen, wird als *Feature Engineering* bezeichnet. Praktisch können wir uns Feature Engineering als eine Art Auswahl der Datendarstellung vorstellen.

Anstatt Parameter wie zum Beispiel den Schwellenwert von drei Kilogramm fest zu codieren, ziehen wir es natürlich vor, dass das ML-Modell lernt, wie jeder Knoten erstellt werden soll, indem die Eingabevariable und der Schwellenwert ausgewählt werden. Entscheidungsbäume sind ein Beispiel für Modelle des maschinellen Lernens, die in der Lage sind, die Datendarstellung zu lernen.[1] Viele der Muster, die wir in diesem Kapitel betrachten, beinhalten ähnliche *lernbare Datendarstellungen*.

Das Entwurfsmuster *Einbettungen* ist das kanonische Beispiel für eine Datendarstellung, die tiefe neuronale Netze selbstständig erlernen können. In einer Einbettung ist die gelernte Darstellung dicht und weist weniger Dimensionen als die

1 Hier besteht die gelernte Datendarstellung aus `babyweight` als Eingabevariable, dem Operator *kleiner als* (<) und dem Schwellenwert 3 kg.

Eingabe auf, die dünn besetzt sein könnte. Der Lernalgorithmus muss die markantesten Informationen aus der Eingabe herausziehen und in kompakterer Form im Feature darstellen. Das Lernen von Features, um die Eingabedaten darzustellen, ist die sogenannte *Feature-Extraktion*. Erlernbare Datendarstellungen kann man sich (wie Einbettungen) als automatisch konstruierte Features vorstellen.

Die Datendarstellung muss nicht einmal auf eine einzelne Eingabevariable beschränkt sein – zum Beispiel erzeugt ein multivariater Entscheidungsbaum (engl. *Oblique Tree*) ein boolesches Feature, indem ein Schwellenwert auf zwei oder mehr Eingabevariablen angewendet wird. Ein Entscheidungsbaum, bei dem jeder Knoten nur eine einzige Eingangsvariable darstellen kann, lässt sich auf eine schrittweise lineare Funktion reduzieren, während sich ein multivariater Entscheidungsbaum, bei dem jeder Knoten eine Linearkombination von Eingabevariablen darstellen kann, auf eine stückweise lineare Funktion reduzieren lässt (siehe Abbildung 2-2). Angesichts der vielen Schritte, die gelernt werden müssen, um die Linie adäquat darzustellen, ist das stückweise lineare Modell einfacher und schneller zu lernen. Eine Erweiterung dieser Idee ist das Entwurfsmuster *Feature Cross*, das das Lernen von `AND`-Beziehungen zwischen kategorialen Variablen mit mehreren Werten vereinfacht.

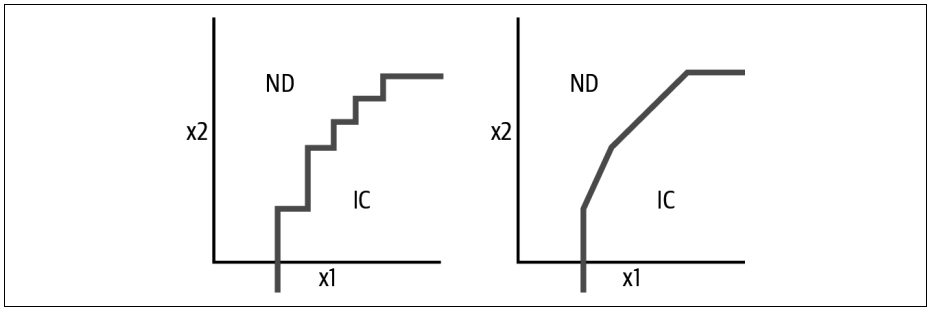

Abbildung 2-2: Ein Klassifizierer als Entscheidungsbaum, bei dem jeder Knoten den Schwellenwert nur für einen einzigen Eingabewert (»x1« oder »x2«) bilden kann, resultiert in einer schrittweisen linearen Grenzwertfunktion, während ein multivariater Baumklassifizierer, bei dem ein Knoten einen Schwellenwert auf eine Linearkombination von Eingabevariablen anwenden kann, in einer stückweisen linearen Grenzfunktion resultiert. Die stückweise lineare Funktion erfordert weniger Knoten und kann eine höhere Genauigkeit erreichen.

Die Datendarstellung muss nicht gelernt oder festgelegt werden – eine Mischform ist ebenfalls möglich. Das Entwurfsmuster *Hashed Feature* ist deterministisch, setzt aber nicht voraus, dass ein Modell sämtliche möglichen Werte kennt, die ein bestimmter Eingang annehmen kann.

Die bisher betrachteten Datendarstellungen sind alle eins zu eins. Obwohl wir die Eingabedaten von verschiedenen Typen separat oder jedes Datenstück als nur ein Feature darstellen könnten, kann es vorteilhafter sein, das Entwurfsmuster *Multimodale Eingabe* zu verwenden. Das ist das vierte Entwurfsmuster, das wir in diesem Kapitel untersuchen werden.

Einfache Datendarstellungen

Bevor wir uns eingehend mit lernbaren Datendarstellungen, Feature Crosses und mehr beschäftigen, werfen wir zunächst einen Blick auf einfachere Datendarstellungen. Man kann sich diese als gebräuchliche Idiome im maschinellen Lernen vorstellen – nicht gerade als Muster, aber dennoch als häufig implementierte Lösungen.

Numerische Eingaben

Die meisten modernen, groß angelegten Modelle für maschinelles Lernen (*Random Forests*, *Support Vector Machines*, *neuronale Netze*) arbeiten mit numerischen Werten. Wenn also unsere Eingabe numerisch ist, können wir sie unverändert an das Modell durchreichen.

Warum Skalierung zweckmäßig ist

Da ein ML-Framework daraufhin optimiert ist, mit Zahlen im Bereich `[-1, 1]` zu arbeiten, ist es oft vorteilhaft, die numerischen Werte in diesen Bereich zu skalieren.

Weshalb numerische Werte in den Bereich [–1, 1] skalieren?

Optimierer nach dem Gradientenabstiegsverfahren benötigen mehr Schritte, um zu konvergieren, wenn die Krümmung der Verlustfunktion zunimmt. Das hängt damit zusammen, dass die Ableitungen von Features mit größeren relativen Beträgen ebenfalls größer sein werden, was zu anormalen Gewichtsaktualisierungen führt. Die ungewöhnlich großen Gewichtsaktualisierungen erfordern mehr Schritte, um zu konvergieren, und erhöhen damit die Rechenlast.

Wenn man die Daten »zentriert«, sodass sie im Bereich `[-1, 1]` liegen, wird die Fehlerfunktion eher kugelförmig. Demzufolge konvergieren Modelle, die mit transformierten Daten trainiert werden, tendenziell schneller und sind daher schneller/kostengünstiger zu trainieren. Außerdem bietet der Bereich `[-1, 1]` die höchste Genauigkeit für Gleitkommazahlen.

Ein schneller Test mit einem der in scikit-learn integrierten Datensätze kann das beweisen (dieser Code ist ein Auszug aus dem Repository zum Buch, *https://github.com/GoogleCloudPlatform/ml-design-patterns/blob/master/02_data_representation/simple_data_representation.ipynb*):

```
from sklearn import datasets, linear_model
diabetes_X, diabetes_y = datasets.load_diabetes(return_X_y=True)
raw = diabetes_X[:, None, 2]
max_raw = max(raw)
```

```
min_raw = min(raw)
scaled = (2*raw - max_raw - min_raw)/(max_raw - min_raw)

def train_raw():
    linear_model.LinearRegression().fit(raw, diabetes_y)

def train_scaled():
    linear_model.LinearRegression().fit(scaled, diabetes_y)

raw_time = timeit.timeit(train_raw, number=1000)
scaled_time = timeit.timeit(train_scaled, number=1000)
```

Mit diesem Code haben wir eine fast 9%ige Verbesserung gegenüber dem Modell, das lediglich ein Eingabe-Feature verwendet, erreicht. Bei der großen Anzahl an Features in einem typischen ML-Modell lassen sich summa summarum erhebliche Einsparungen erzielen.

Ein weiterer wichtiger Grund für eine Skalierung ist, dass einige Algorithmen und Techniken des maschinellen Lernens sehr empfindlich auf die relativen Größen der verschiedenen Features reagieren. Zum Beispiel stützt sich ein *k*-Means-Clustering-Algorithmus, der den euklidischen Abstand als Abstandsmaß verwendet, letztlich stark auf Features mit größeren Werten. Eine fehlende Skalierung beeinflusst auch die Wirksamkeit der L1- oder L2-Regularisierung, da die Größe der Gewichte für ein Feature von der Größe der Werte dieses Features abhängt, sodass verschiedene Features von der Regularisierung unterschiedlich betroffen sind. Wenn wir alle Features in den Bereich `[-1, 1]` skalieren, stellen wir sicher, dass es keine größeren Unterschiede in den relativen Größen verschiedener Features gibt.

Lineare Skalierung

Üblicherweise verwendet man vier Formen der Skalierung:

Min-Max-Skalierung
: Der numerische Wert wird linear skaliert, sodass der kleinste Wert, den die Eingabe annehmen kann, auf `-1` skaliert wird und der größtmögliche Wert auf `1`:

  ```
  x1_scaled = (2*x1 - max_x1 - min_x1)/(max_x1 - min_x1)
  ```

  Problematisch bei der Min-Max-Skalierung ist, dass die Größt- und Kleinstwerte (`max_x1` und `min_x1`) aus dem Trainingsdatensatz geschätzt werden müssen und es sich dabei oft um Ausreißer handelt. Die eigentlichen Daten werden oftmals auf einen sehr engen Bereich im `[-1, 1]`-Band zusammengequetscht.

Clipping (in Verbindung mit Min-Max-Skalierung)
: Hilft, das Problem der Ausreißer anzugehen. Anstatt Minimum und Maximum aus dem Trainingsdatensatz zu schätzen, werden »vernünftige« Werte verwendet. Der numerische Wert wird zwischen diesen beiden vernünftigen Grenzen linear skaliert und dann so gekappt, dass er im Bereich `[-1, 1]` liegt. Ausreißer werden demnach als `-1` oder `1` behandelt.

Z-Wert-Normalisierung
: Löst das Problem der Ausreißer, ohne dass vorheriges Wissen darüber erforderlich ist, wie der vernünftige Bereich aussieht. Die Eingabe wird linear skaliert, und zwar mithilfe von Mittelwert und Standardabweichung, die über den Trainingsdatensatz geschätzt werden:

```
x1_scaled = (x1 - mean_x1)/stddev_x1
```

Der Name der Methode spiegelt die Tatsache wider, dass der skalierte Wert den Mittelwert null hat und durch die Standardabweichung normalisiert ist, sodass er eine Einheitsvarianz über dem Trainingsdatensatz hat. Der skalierte Wert ist nicht begrenzt, liegt aber in den meisten Fällen (67%, wenn eine Normalverteilung zugrunde liegt) im Bereich `[-1, 1]`. Werte außerhalb dieses Bereichs werden seltener, je größer ihr absoluter Wert wird, sind aber immer noch vorhanden.

Winsorizing
: Verwendet die empirische Verteilung im Trainingsdatensatz, um den Datensatz auf die Werte zu kappen, die durch die 10%- und 90%-Perzentile (oder die 5%- und 95%-Perzentile usw.) der Datenwerte gegeben sind. Der winsorierte Wert ist Min-Max-skaliert.

Alle bisher beschriebenen Methoden skalieren die Daten linear (beim Clipping und Winsorizing linear innerhalb des typischen Bereichs). Min-Max-Skalierung und Clipping funktionieren tendenziell am besten für gleichförmig verteilte Daten, Z-Wert-Skalierung für normalverteilte Daten. Der Einfluss der verschiedenen Skalierungsfunktionen auf die Spalte `mother_age` im Beispiel mit der Vorhersage des Babygewichts ist in Abbildung 2-3 dargestellt (den vollständigen Code finden Sie unter *https://github.com/GoogleCloudPlatform/ml-design-patterns/blob/master/02_data_representation/simple_data_representation.ipynb*).

Keine »Ausreißer« wegwerfen

Clipping haben wir so definiert, dass skalierte Werte kleiner als `-1` wie `-1` und skalierte Werte größer als `1` wie `1` behandelt werden. Derartige »Ausreißer« verwerfen wir nicht einfach, weil wir erwarten, dass das ML-Modell auch in der Produktion derartige Ausreißer verarbeiten muss. Nehmen wir zum Beispiel Babys, die von 50-jährigen Müttern geboren wurden. Da wir in unserem Datensatz nicht genügend ältere Mütter haben, läuft das Clipping darauf hinaus, alle Mütter älter als (beispielsweise) 45 wie 45 zu behandeln. Das gleiche Prinzip wenden wir in der Produktion an, und unser Modell wird auch für ältere Mütter funktionieren. Das Modell würde den Umgang mit Ausreißern nicht lernen, wenn wir einfach alle Trainingsbeispiele verworfen hätten, bei denen Mütter älter als 50 Jahre Babys geboren haben!

Man kann dies auch unter dem Aspekt betrachten, dass es zwar akzeptabel ist, *ungültige Eingaben* zu verwerfen, dass es aber nicht akzeptabel ist, *gültige Daten* zu verwerfen. Es wäre also gerechtfertigt, Zeilen zu entfernen, in denen `mother_age` negativ ist, weil es sich wahrscheinlich um einen Eingabefehler handelt. In der Produktion würde die Validierung des Eingabeformulars sicherstellen, dass der Mitarbeiter in der Patientenaufnahme das Alter der Mutter neu eingeben müsste. Dagegen dürften wir keine Zeilen weglassen, in denen `mother_age` gleich 50 ist, da 50 eine vollkommen gültige Eingabe ist und wir mit 50-jährigen Müttern rechnen, sobald das Modell in der Produktion eingesetzt wird.

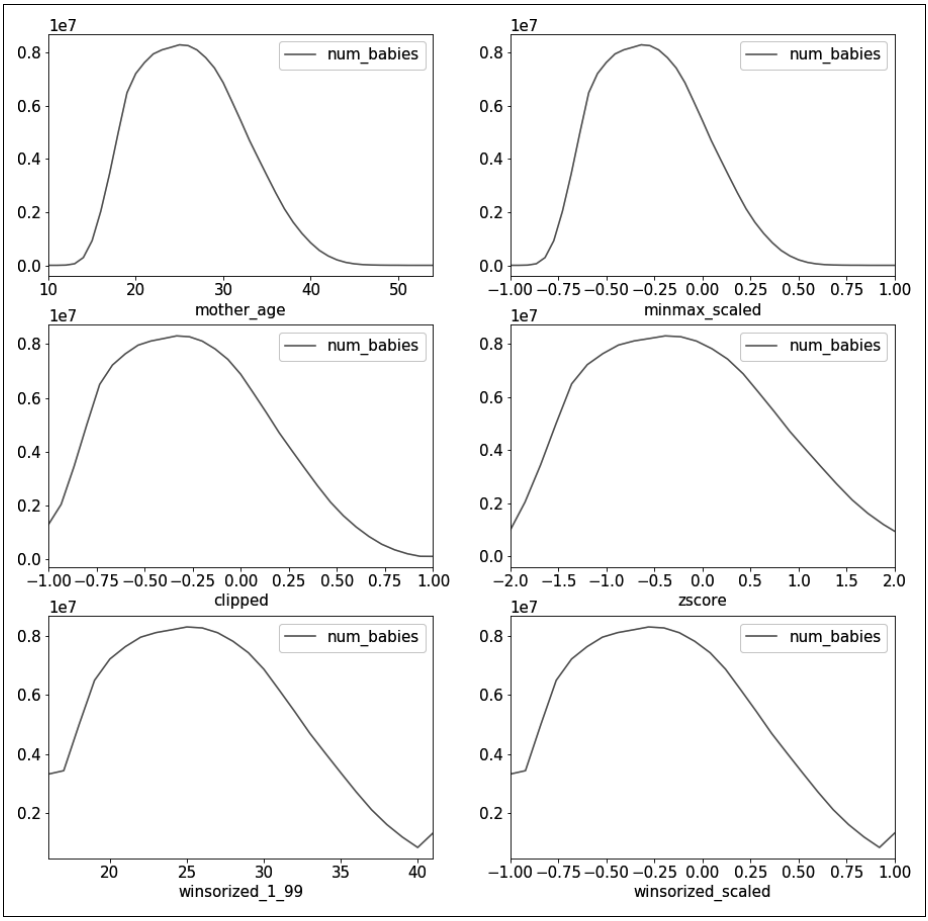

Abbildung 2-3: Das Histogramm von »mother_age« im Beispiel der Vorhersage von Babygewichten ist im linken oberen Diagramm dargestellt. Die anderen Diagramme zeigen verschiedene Skalierungsfunktionen (siehe die Beschriftung der x-Achse).

Beachten Sie in Abbildung 2-3, dass `minmax_scaled` die `x`-Werte in den gewünschten Bereich von `[-1, 1]` bringt, aber Werte an den extremen Enden der Verteilung, wo es nicht genügend Beispiele gibt, beibehält. Clipping schneidet viele der problematischen Werte ab, verlangt aber, dass die Kappungsschwellenwerte genau festgelegt werden – durch den langsamen Rückgang in Anzahl der Babys von Müttern, die älter als 40 Jahre sind, ist es schwierig, einen festen Schwellenwert festzulegen. Ähnlich wie Clipping erfordert Winsorizing, dass die Perzentil-Schwellenwerte genau eingestellt werden. Z-Wert-Normalisierung verbessert den Bereich (schränkt die Werte aber nicht auf den Bereich `[-1, 1]` ein) und schiebt die problematischen Werte weiter nach außen. Von diesen drei Methoden funktioniert die Nullnormierung am besten für `mother_age`, da die Verteilung der rohen Alterswerte einer Glockenkurve ähnelt. Bei anderen Problemen sind Min-Max-Skalierung, Clipping oder Winsorizing möglicherweise besser geeignet.

Nichtlineare Transformationen

Wie sieht es aus, wenn unsere Daten verzerrt sind und wir es weder mit einer Gleich- noch mit einer Normalverteilung zu tun haben? In diesem Fall ist es besser, eine *nichtlineare Transformation* auf die Eingabe anzuwenden, bevor sie skaliert wird. Eine gängige Methode ist es, den Eingabewert zu logarithmieren und erst dann zu skalieren. Andere gebräuchliche Transformationen verwenden die Sigmoid-Funktion und polynomiale Erweiterungen (Quadrat, Quadratwurzel, Kubik, Kubikwurzel usw.). Eine gute Transformationsfunktion lässt sich daran erkennen, dass die transformierten Werte eine Gleich- oder Normalverteilung zeigen.

Angenommen, wir erstellten ein Modell, um die Verkaufszahlen eines Sachbuchs vorherzusagen. Eine der Eingaben in das Modell sei die Popularität der Wikipedia-Seite, die dem Thema entspricht. Die Anzahl der Seitenaufrufe in Wikipedia weist jedoch eine sehr schiefe Verteilung auf und belegt einen großen dynamischen Bereich. (Zu sehen ist das im linken Diagramm in Abbildung 2-4: Die Verteilung ist stark gegen selten aufgerufene Seiten geneigt, aber die am häufigsten besuchten Seiten werden zig Millionen Mal aufgerufen.) Indem wir den Logarithmus der Aufrufe bilden, dann die vierte Wurzel dieses logarithmierten Werts nehmen und das Ergebnis linear skalieren, liegen die Werte im gewünschten Bereich, und das Ergebnis sieht schon ein wenig wie eine Glockenkurve aus. Im GitHub-Repository für dieses Buch (*https://github.com/GoogleCloudPlatform/ml-design-patterns/blob/master/02_data_representation/simple_data_representation.ipynb*) finden Sie die Einzelheiten des Codes, um die Wikipedia-Daten abzufragen, diese Transformationen anzuwenden und das gezeigte Diagramm zu erzeugen.

Es kann schwierig sein, eine Linearisierungsfunktion zu entwickeln, die zu einer glockenförmigen Verteilung führt. Einfacher ist es, die Anzahl der Aufrufe in Bereiche zu unterteilen und die Bereichsgrenzen so zu wählen, dass sich die gewünschte Verteilung der Ausgabe ergibt. Ein prinzipieller Ansatz für die Auswahl dieser Bereiche ist die Histogrammegalisierung, bei der die Klassen des Histogramms nach

Quantilen der Rohverteilung gewählt werden (siehe das dritte Diagramm in Abbildung 2-4). Im Idealfall liefert die Histogrammegalisierung eine Gleichverteilung (auch wenn das hier nicht zutrifft, da sich Werte in den Quantilen wiederholen).

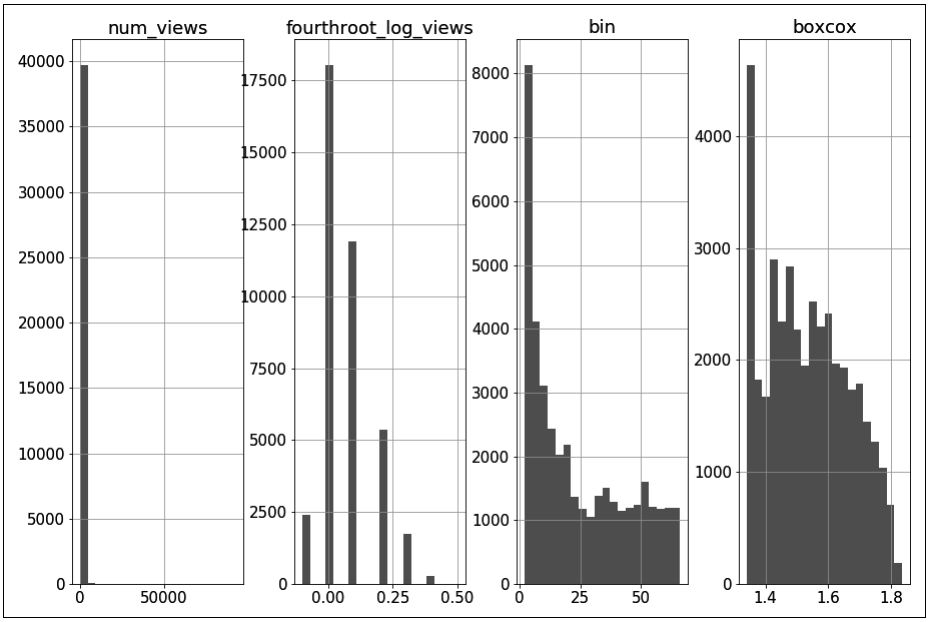

Abbildung 2-4: Linkes Diagramm: Die Verteilung der Aufrufanzahl von Wikipedia-Seiten ist sehr schief und nimmt einen großen dynamischen Bereich ein. Das zweite Diagramm zeigt, dass sich die Probleme lösen lassen, indem man die Anzahl der Aufrufe nacheinander logarithmiert, mit einer Potenzfunktion transformiert und dann linear skaliert. Im dritten Diagramm ist die Wirkung der Histogrammegalisierung zu sehen, und das vierte Diagramm zeigt das Ergebnis der Box-Cox-Transformation.

In BigQuery können Sie eine Histogrammegalisierung wie folgt ausführen:

```
ML.BUCKETIZE(num_views, bins) AS bin
```

Die Klassen (`bins`) werden hierbei folgendermaßen bestimmt:

```
APPROX_QUANTILES(num_views, 100) AS bins
```

Alle Einzelheiten finden Sie im Notebook (*https://github.com/GoogleCloudPlatform/ml-design-patterns/blob/master/02_data_representation/simple_data_representation.ipynb*) des Code-Repositorys.

Schiefe Verteilungen lassen sich auch mit einer parametrischen Transformationstechnik wie der *Box-Cox-Transformation* verarbeiten. Box-Cox steuert mit seinem einzigen Parameter, Lambda, die *Heteroskedastizität*, sodass die Varianz nicht mehr von der Größe abhängt. Hier ist die Varianz bei selten besuchten Wikipedia-Seiten wesentlich kleiner als die Varianz bei häufig besuchten Seiten, und Box-Cox ver-

sucht, die Varianz über alle Bereiche der Aufrufanzahl (num_views) auszugleichen. Dies lässt sich mit dem SciPy-Paket von Python bewerkstelligen:

```
traindf['boxcox'], est_lambda = (
    scipy.stats.boxcox(traindf['num_views']))
```

Der über den Trainingsdatensatz geschätzte Parameter (est_lambda) wird dann verwendet, um andere Werte zu transformieren:

```
evaldf['boxcox'] = scipy.stats.boxcox(evaldf['num_views'], est_lambda)
```

Array von Zahlen

Manchmal liegen die Eingabedaten auch als Array von Zahlen vor. Wenn das Array eine feste Länge hat, kann die Datendarstellung hierbei ziemlich einfach sein: Das Array wird abgeflacht, und jede Position wird als separates Feature verarbeitet. Allerdings ist die Länge eines Arrays oftmals variabel. So könnte das Modell, das die Verkäufe eines Sachbuchs vorhersagen soll, die Umsätze aller vorherigen Bücher zum jeweiligen Thema als Eingabe heranziehen. Eine Eingabe könnte so aussehen:

```
[2100, 15200, 230000, 1200, 300, 532100]
```

Es liegt auf der Hand, dass die Länge dieses Arrays in jeder Zeile variieren wird, da zu unterschiedlichen Themen auch verschieden viele Bücher veröffentlicht wurden.

Zu den gängigen Idiomen, die sich mit der Verarbeitung von Zahlenarrays befassen, gehören unter anderem:

- Das Eingabearray in Form seiner Bulk-Statistik darstellen. Zum Beispiel könnten wir die Länge (d. h. die Anzahl vorheriger Bücher zum Thema), den Durchschnitt, den Median, das Minimum, das Maximum usw. verwenden.
- Das Eingabearray in Form seiner empirischen Verteilung darstellen – d. h. nach dem 10%-/20%-/...-Perzentil usw.
- Wenn das Array in bestimmter Weise geordnet ist (zum Beispiel nach der Zeit oder nach der Größe), das Eingabearray durch die letzten drei oder eine andere festgelegte Anzahl von Elementen darstellen. Für Arrays, die weniger als drei Elemente enthalten, wird das Feature mit fehlenden Werten auf eine Länge von drei aufgefüllt.

Alle diese Methoden stellen letztlich das Datenarray variabler Länge als Feature mit fester Länge dar. Wir könnten dieses Problem auch als Zeitreihen-Prognose-Problem formulieren, und zwar als das Problem der Vorhersage von Verkäufen des nächsten Buchs zum Thema basierend auf dem zeitlichen Verlauf der Verkäufe vorheriger Bücher. Indem wir die Verkäufe vorheriger Bücher als Arrayeingabe behandeln, gehen wir davon aus, dass die wichtigsten Faktoren bei der Vorhersage von Verkäufen eines Buchs Eigenschaften des Buchs selbst sind (Autor, Verlag, Rezensionen usw.) und nicht der zeitliche Zusammenhang der Verkaufsbeträge.

Kategoriale Eingaben

Da die meisten modernen, groß angelegten Modelle für maschinelles Lernen (Random Forests, Support Vector Machines, neuronale Netze) auf numerischen Werten arbeiten, müssen kategoriale Eingaben als Zahlenwerte dargestellt werden.

Wenn man einfach die möglichen Werte aufzählt und sie auf eine Ordinalskala abbildet, funktioniert das nur unzureichend. Nehmen wir an, ein Modell soll die Verkaufszahlen eines Sachbuchs vorhersagen und eine der Eingaben ist die Sprache, in der das Buch geschrieben ist. Nun können wir nicht einfach eine Zuordnungstabelle wie die folgende aufstellen:

Kategoriale Eingabe	Numerisches Feature
English	1.0
Chinese	2.0
German	3.0

Das hängt damit zusammen, dass das ML-Modell dann versuchen wird, zwischen der Popularität von deutschen und englischen Büchern zu interpolieren, um die Popularität des Buchs in Chinesisch zu bekommen! Da es keine ordinale Beziehung zwischen Sprachen gibt, müssen wir eine Zuordnung von kategorial zu numerisch verwenden, die es dem Modell ermöglicht, den Markt für Bücher, die in diesen Sprachen geschrieben sind, unabhängig zu lernen.

1-aus-n-Codierung

Die einfachste Methode, kategoriale Variablen zuzuordnen und dabei sicherzustellen, dass die Variablen unabhängig sind, ist die *1-aus-n-Codierung* (engl. *One-hot Encoding*). In unserem Beispiel würde die kategoriale Variable mit der folgenden Zuordnung in einen 3-elementigen Feature-Vektor konvertiert:

Kategoriale Eingabe	Numerisches Feature
English	[1.0, 0.0, 0.0]
Chinese	[0.0, 1.0, 0.0]
German	[0.0, 0.0, 1.0]

Die 1-aus-n-Codierung setzt voraus, dass das *Vokabular* der kategorialen Eingabe im Voraus bekannt ist. Hier besteht das Vokabular aus drei Tokens (English, Chinese und German), und die Länge des resultierenden Features ist die Größe dieses Vokabulars.

Dummy-Codierung oder 1-aus-n-Codierung?

Technisch gesehen, genügt ein 2-elementiger Feature-Vektor, um eine eindeutige Zuordnung für ein Vokabular der Größe 3 zu liefern:

Kategoriale Eingabe	Numerisches Feature
English	[0.0, 0.0]
Chinese	[1.0, 0.0]
German	[0.0, 1.0]

Dies ist die sogenannte *Dummy-Codierung*. Da sie eine kompaktere Darstellung ergibt, wird sie in statistischen Modellen bevorzugt, die eine bessere Performance zeigen, wenn die Eingaben linear unabhängig sind.

Moderne ML-Algorithmen verlangen jedoch nicht, dass die Eingaben linear unabhängig sind. Redundante Eingaben bereinigen sie mit Methoden wie L1-Regularisierung. Der zusätzliche Freiheitsgrad ermöglicht dem Framework, eine fehlende Eingabe in der Produktion transparent als durchgängig null zu verarbeiten:

Kategoriale Eingabe	Numerisches Feature
English	[1.0, 0.0, 0.0]
Chinese	[0.0, 1.0, 0.0]
German	[0.0, 0.0, 1.0]
(missing)	[0.0, 0.0, 0.0]

Daher unterstützen viele ML-Frameworks nur eine 1-aus-n-Codierung.

Unter bestimmten Umständen kann es hilfreich sein, eine numerische Eingabe als kategorial zu behandeln und sie auf eine 1-aus-n-codierte Spalte abzubilden:

Die numerische Eingabe ist ein Index:

Wenn wir zum Beispiel versuchen, das Verkehrsaufkommen vorherzusagen und eine unserer Eingaben ist der Wochentag, könnten wir den Wochentag als numerisch (1, 2, 3, ..., 7) behandeln, doch es ist hilfreich, dass der Wochentag hier keine kontinuierliche Skala ist, sondern tatsächlich nur ein Index. Es ist besser, ihn als kategorial (Sonntag, Montag, ..., Samstag) zu behandeln, denn die Indizierung ist willkürlich. Soll die Woche mit Sonntag beginnen (wie in den USA), mit Montag (wie in Deutschland) oder Samstag (wie in Ägypten)?

Die Beziehung zwischen Eingabe und Label ist nicht kontinuierlich:
: Was dafür sprechen sollte, den Wochentag als kategoriales Feature zu behandeln, ist, dass das Verkehrsaufkommen am Freitag nicht von den Aufkommen am Donnerstag und Samstag beeinflusst wird.

Es ist vorteilhaft, die numerischen Variablen zu kategorisieren:
: In den meisten Städten hängt die Verkehrsbelastung davon ab, ob Wochenende ist, und dies kann je nach Ort variieren (Samstag und Sonntag in den meisten Teilen der Welt, Donnerstag und Freitag in manchen islamischen Ländern). Es wäre dann hilfreich, den Wochentag als boolesches Feature (Wochenende oder Werktag) zu behandeln. Eine derartige Zuordnung, bei der die Anzahl der eindeutigen Eingaben (hier sieben) größer ist als die Anzahl der eindeutigen Feature-Werte (hier zwei), bezeichnet man als Bucketing. Üblicherweise erfolgt das Bucketing in Form von Bereichen – zum Beispiel könnten wir `mother_age` in Bereiche einteilen, die bei 20, 25, 30 usw. enden, und jede dieser Klassen als kategorial behandeln. Allerdings geht dabei das ordinale Wesen von `mother_age` verloren.

Wir wollen verschiedene Werte der numerischen Eingabe als unabhängig behandeln, wenn es um ihre Auswirkung auf das Label geht:
: Zum Beispiel hängt das Gewicht eines Babys von der Pluralität[2] der Entbindungen ab, da Zwillinge und Drillinge tendenziell weniger wiegen als Einzelkinder. Wenn also ein Baby mit einem geringen Gewicht ein Drillingsbaby ist, könnte es gesünder sein als ein Zwillingsbaby mit dem gleichen Gewicht. In diesem Fall könnten wir die Pluralität auf eine kategoriale Variable abbilden, da eine kategoriale Variable dem Modell ermöglicht, unabhängige Optimierungsparameter für die verschiedenen Werte der Pluralität zu lernen. Selbstverständlich können wir dies nur tun, wenn wir genügend Beispiele von Zwillingen und Drillingen in unserem Datensatz haben.

Array von kategorialen Variablen

Manchmal liegen die Eingabedaten als Array von Kategorien vor. Wenn das Array eine feste Länge hat, können wir jede Arrayposition als separates Feature behandeln. Oftmals hat das Array jedoch eine variable Länge. Zum Beispiel könnte eine der Eingaben für das Geburtenmodell die Art vorheriger Geburten dieser Mutter sein:

```
[Induced, Induced, Natural, Cesarean]
```

Es liegt auf der Hand, dass die Länge dieses Arrays in jeder Zeile variiert, da es für jedes Baby eine unterschiedliche Anzahl älterer Geschwister gibt.

2 Bei Zwillingen ist die Pluralität 2, bei Drillingen 3.

Gängige Idiome für den Umgang mit Arrays von kategorialen Variablen sind unter anderem:

- Die Häufigkeit jedes Terms wird *gezählt*. Die Darstellung für das obige Beispiel wäre dann `[2, 1, 1]`, wenn man die Terme `Induced`, `Natural` und `Cesarean` (in dieser Reihenfolge) annimmt. Nun haben wir ein Zahlenarray fester Länge, das sich abflachen und in Positionsreihenfolge verwenden lässt. Wenn in einem Array ein Element nur einmal vorkommen kann (zum Beispiel bei Sprachen, die eine Person spricht) oder wenn das Feature lediglich die Anwesenheit angibt und nicht die Anzahl (beispielsweise ob die Mutter jemals eine Geburt per Kaiserschnitt hatte), dann ist der Zählwert an jeder Position `0` oder `1` – das ist die sogenannte *Multi-Hot-Codierung*.
- Um große Zahlen zu vermeiden, kann man die *relative Häufigkeit* anstelle der Zählung verwenden. Die Darstellung für unser Beispiel lautet dann `[0.5, 0.25, 0.25]` statt `[2, 1, 1]`. Leere Arrays (erstgeborene Babys ohne vorherige Geschwister) werden mit `[0, 0, 0]` dargestellt. In der Verarbeitung natürlicher Sprache normalisiert man die relative Häufigkeit eines Worts insgesamt durch die relative Häufigkeit der Dokumente, die dieses Wort enthalten. Daraus ergibt sich das *Tf-idf-Maß*[3] (*https://oreil.ly/kNYHr*), das widerspiegelt, wie eindeutig ein Wort in einem Dokument ist.
- Wenn das Array in bestimmter Weise geordnet ist (z.B. in zeitlicher Folge), wird das Eingabearray durch die drei letzten Elemente dargestellt. Arrays, die weniger als drei Elemente enthalten, werden mit fehlenden Werten aufgefüllt.
- Das Array wird durch Bulk-Statistiken dargestellt, z.B. durch die Länge des Arrays, den Modus (häufigster Eintrag), den Median, das 10%-/20%-/...-Perzentil usw.

Von diesen ist das Idiom des *Zählen/der relativen Häufigkeit* das gebräuchlichste. Beide sind eine Verallgemeinerung der 1-aus-n-Codierung – wenn das Baby keine älteren Geschwister hat, lautet die Darstellung `[0, 0, 0]`, und wenn das Baby ein älteres Geschwisterchen hat, das normal geboren wurde, wäre die Darstellung `[0, 1, 0]`.

Nachdem wir einfache Datendarstellungen gesehen haben, kommen wir zu den Entwurfsmustern, die bei der Datendarstellung helfen.

Entwurfsmuster 1: Hashed Feature

Das Entwurfsmuster *Hashed Feature* befasst sich mit drei möglichen Problemen in Bezug auf kategoriale Features: unvollständiges Vokabular, Modellgröße aufgrund von Kardinalität und Kaltstart. Hierzu gruppiert es die kategorialen Features und akzeptiert den Kompromiss von Kollisionen in der Datendarstellung.

3 Von engl. *Term frequency* (Vorkommenshäufigkeit) und *inverse document frequency* (inverse Dokumenthäufigkeit).

Problem

Um eine kategoriale Eingangsvariable 1-aus-n zu codieren, muss das Vokabular im Voraus bekannt sein. Das ist kein Problem, wenn die Eingabevariable etwas ist wie die Sprache, in der ein Buch geschrieben ist, oder der Tag der Woche, für den das Verkehrsaufkommen vorhergesagt wird.

Wie sieht es aus, wenn die fragliche kategoriale Variable etwas ist wie die Kennung `hospital_id` des Krankenhauses, in dem das Baby geboren wurde, oder die Kennung `physician_id` der Person, die das Baby entbindet? Derartige kategoriale Variablen werfen einige Probleme auf:

- Um das Vokabular zu kennen, muss es aus den Trainingsdaten extrahiert werden. Wegen der Stichprobenziehung ist es möglich, dass die Trainingsdaten nicht alle möglichen Krankenhäuser oder Ärzte enthalten. Das Vokabular könnte *unvollständig* sein.
- Die kategorialen Variablen haben eine *hohe Kardinalität*. Anstelle von Feature-Vektoren mit drei Sprachen oder sieben Tagen haben wir Feature-Vektoren, deren Länge in die Tausende bis Millionen geht. Solche Feature-Vektoren werfen in der Praxis mehrere Probleme auf. Sie beinhalten so viele Gewichte, dass die Trainingsdaten möglicherweise nicht ausreichen. Selbst wenn wir das Modell trainieren können, benötigt das trainierte Modell sehr viel Speicherplatz, da das gesamte Vokabular zur Bereitstellungszeit gebraucht wird. Daher lässt sich das Modell eventuell nicht auf kleineren Geräten bereitstellen.
- Nachdem das Modell in die Produktion überführt wurde, könnten neue Krankenhäuser gebaut und neue Ärztinnen und Ärzte eingestellt worden sein. Das Modell kann dafür keine Vorhersagen treffen, und somit wird eine separate Bereitstellungsinfrastruktur benötigt, um mit derartigen *Kaltstart*-Problemen umzugehen.

Selbst bei einfachen Darstellungen wie der 1-aus-n-Codierung lohnt es sich, das Kaltstartproblem einzukalkulieren und alle Nullen explizit für Eingaben zu reservieren, die nicht im Vokabular enthalten sind.

Als konkretes Beispiel nehmen wir das Problem, die Ankunftsverspätung eines Fluges vorherzusagen. Eine der Eingaben für das Modell ist der Abflughafen. Als der Datensatz zusammengestellt wurde, gab es in den Vereinigten Staaten 347 Flughäfen:

```
SELECT
    DISTINCT(departure_airport)
FROM `bigquery-samples.airline_ontime_data.flights`
```

Da es bei einigen Flughäfen lediglich einen bis drei Flüge über den gesamten Zeitraum gab, erwarten wir, dass das Vokabular der Trainingsdaten unvollständig sein wird. Die Anzahl 347 ist groß genug, sodass das Feature ziemlich spärlich vor-

kommt, und es ist damit zu rechnen, dass neue Flughäfen gebaut werden. Alle drei Probleme (unvollständiges Vokabular, hohe Kardinalität, Kaltstart) sind vorhanden, wenn wir den Abflughafen 1-aus-n-codieren.

Der Datensatz mit den Fluglinien ist wie der Datensatz mit den Geburten und fast alle anderen Datensätze, die wir in diesem Buch zur Veranschaulichung heranziehen, ein öffentlicher Datensatz in BigQuery (*https://oreil.ly/lgcKA*), sodass Sie die Abfrage selbst ausprobieren können. Als dieses Buch entstanden ist, war ein Abfragevolumen von 1 TByte pro Monat kostenlos, und es steht eine Sandbox zur Verfügung, sodass Sie BigQuery bis zu diesem Limit nutzen können, ohne eine Kreditkarte hinterlegen zu müssen. Es empfiehlt sich, unser GitHub-Repository als Lesezeichen zu speichern. Zum Beispiel finden Sie den vollständigen Code im Notebook bei GitHub unter *https://github.com/GoogleCloudPlatform/ml-design-patterns/blob/master/02_data_representation/hashed_feature.ipynb*.

Lösung

Das Entwurfsmuster *Hashed Feature* stellt eine kategoriale Eingabevariable wie folgt dar:

1. Die kategoriale Eingabe in eine eindeutige Zeichenfolge konvertieren. Für den Abflughafen können wir den dreibuchstabigen IATA-Code (*https://oreil.ly/B8nLw*) verwenden.
2. Einen Hashing-Algorithmus auf der Zeichenfolge aufrufen, und zwar einen deterministischen (keinen zufälligen Startwert oder Salt, siehe *https://de.wikipedia.org/wiki/Salt_(Kryptologie)*) und portablen (damit sich derselbe Algorithmus sowohl für das Training als auch für das Bereitstellen eignet).
3. Den Rest ermitteln, wenn das Hashergebnis durch die gewünschte Anzahl an Buckets geteilt wird. Typischerweise gibt der Hashing-Algorithmus eine Ganzzahl zurück, die negativ sein kann, und der Modulo einer negativen Ganzzahl ist negativ. Es wird also der Absolutwert des Ergebnisses verwendet.

In BigQuery SQL setzt man diese Schritte folgendermaßen um:

```
ABS(MOD(FARM_FINGERPRINT(airport), numbuckets))
```

Die Funktion `FARM_FINGERPRINT()` verwendet mit FarmHash eine Familie von Hashing-Algorithmen, die deterministisch sind, eine gute Verteilung aufweisen und für die Implementierungen in einer Reihe von Programmiersprachen verfügbar sind.

In TensorFlow werden diese Schritte durch die Funktion `feature_column` implementiert:

```
tf.feature_column.categorical_column_with_hash_bucket(
    airport, num_buckets, dtype=tf.dtypes.string)
```

Zum Beispiel zeigt Tabelle 2-1 die FarmHash-Werte einiger IATA-Flughafencodes, wenn die Aufteilung der Hashtabelle in 3, 10 und 1.000 Buckets erfolgt.

Tabelle 2-1: Der FarmHash einiger IATA-Flughafencodes, wenn die Hashtabelle unterschiedlich viele Buckets umfasst

Zeile	departure_airport	hash3	hash10	hash1000
1	DTW	1	3	543
2	LBB	2	9	709
3	SNA	2	7	587
4	MSO	2	7	737
5	ANC	0	8	508
6	PIT	1	7	267
7	PWM	1	9	309
8	Benutzername	1	4	744
9	SAF	1	2	892
10	IPL	2	1	591

Warum es funktioniert

Nehmen wir an, dass wir aus dem Flughafencode Hashwerte für zehn Buckets bilden wollen (*hash10* in Tabelle 2-1). Wie lassen sich die Probleme angehen, die wir ermittelt haben?

Eingabe außerhalb des Vokabulars

Selbst wenn ein Flughafen mit einer Handvoll Flügen nicht im Trainingsdatensatz erscheint, liegt sein Hashwert des Features im Bereich `[0–9]`. Demzufolge tritt beim Bereitstellen kein Resilienzproblem auf – der unbekannte Flughafen bekommt die Vorhersagen, die anderen Flughäfen im Hash-Bucket entsprechen. Das Modell wird keinen Fehler hervorbringen.

Bei 347 Flughäfen erhalten durchschnittlich 35 Flughäfen den gleichen Hashwert, wenn wir die Tabelle in zehn Buckets aufteilen. Ein Flughafen, der im Trainingsdatensatz nicht erscheint, »borgt« seine Eigenschaften von den anderen ähnlichen etwa 35 Flughäfen im Hash-Bucket aus. Natürlich wird die Vorhersage für einen fehlenden Flughafen nicht genau sein (es ist unrealistisch, genaue Vorhersagen für unbekannte Eingaben zu erwarten), doch sie wird im richtigen Bereich liegen. Orientieren Sie sich bei der Anzahl der Hash-Buckets daran, wie sich Eingaben außerhalb des Vokabulars vernünftig verarbeiten lassen und wie genau das Modell die kategoriale Eingabe widerspiegeln soll. Bei zehn Hash-Buckets werden etwa 35 Flughäfen vermischt. Eine gute Faustregel ist, die Anzahl der Hash-Buckets so zu wählen, dass jeder Bucket etwa fünf Einträge erhält. Im Beispiel bedeutet dies, dass 70 Hash-Buckets einen guten Kompromiss darstellen.

Hohe Kardinalität

Es ist leicht zu sehen, dass sich dem Problem hoher Kardinalität entgegenwirken lässt, wenn man die Anzahl der Hash-Buckets genügend klein wählt. Selbst Millionen von Flughäfen, Krankenhäusern oder Ärzten können wir per Hashing in wenigen Hundert Buckets unterbringen und somit die Anforderungen an den Arbeitsspeicher des Systems und die Modellgröße in praktikablen Größen halten.

Das Vokabular brauchen wir nicht zu speichern, da der Transformationscode unabhängig vom tatsächlichen Datenwert ist und der Kern des Modells es nur mit `num_buckets` Eingaben und nicht mit dem vollständigen Vokabular zu tun hat.

Es stimmt, dass Hashing verlustbehaftet ist – da wir 347 Flughäfen haben, erhalten durchschnittlich 35 Flughäfen den gleichen Hash-Bucket-Code, wenn wir die Hashtabelle in zehn Buckets aufteilen. Wenn die Alternative jedoch darin besteht, die Variable zu verwerfen, weil sie zu breit ist, dann ist eine verlustbehaftete Codierung ein akzeptabler Kompromiss.

Kaltstart

Die Kaltstartsituation ähnelt der Situation bei Eingaben außerhalb des Vokabulars. Kommt im System ein neuer Flughafen hinzu, erhält er anfangs die Vorhersagen, die anderen Flughäfen im Hash-Bucket entsprechen. Wird ein Flughafen bekannter, gibt es auch mehr Flüge von diesem Flughafen. Solange wir das Modell regelmäßig erneut trainieren, werden seine Vorhersagen nach und nach die Ankunftsverzögerungen vom neuen Flughafen widerspiegeln. Darauf geht der Abschnitt »Entwurfsmuster 18: Kontinuierliche Modellbewertung« auf Seite 245 in Kapitel 5 näher ein.

Indem wir die Anzahl der Hash-Buckets so wählen, dass jeder Bucket etwa fünf Einträge erhält, können wir sicherstellen, dass jeder Bucket vernünftige Anfangsergebnisse hat.

Kompromisse und Alternativen

Die meisten Entwurfsmuster beinhalten eine Art von Kompromiss, und das Entwurfsmuster *Hashed Feature* bildet da keine Ausnahme. Hier geht es vor allem darum, dass wir Modellgenauigkeit verlieren.

Bucket-Kollision

Der Modulo-Teil der Hashed-Feature-Implementierung ist eine verlustbehaftete Operation. Wenn wir die Hash-Bucket-Größe mit 100 wählen, werden sich drei bis vier Flughäfen einen Bucket teilen. Bei der Fähigkeit, die Daten genau darzustellen (mit einem festen Vokabular und 1-zu-n-Codierung), machen wir bewusst Abstriche, um Eingaben außerhalb des Vokabulars, Einschränkungen der Kardinalität/Modellgröße und Kaltstartprobleme zu behandeln. Es gibt nichts umsonst. Entscheiden Sie sich nicht für *Hashed Feature*, wenn Sie das Vokabular im Voraus kennen, wenn

das Vokabular relativ klein ist (bei einem Datensatz mit Millionen von Beispielen ist eine Größe von Tausenden akzeptabel) und wenn Kaltstart kein Thema ist.

Nun können wir aber nicht einfach die Anzahl der Buckets extrem vergrößern in der Hoffnung, Kollisionen gänzlich zu vermeiden. Selbst wenn wir bei nur 347 Flughäfen die Anzahl der Buckets auf 100.000 steigerten, beträgt die Wahrscheinlichkeit, dass mindestens zwei Flughäfen in denselben Bucket fallen, 45 % – ein inakzeptabel hoher Wert (siehe Tabelle 2-2). Daher sollten wir Hashed Features nur verwenden, wenn wir tolerieren wollen, dass mehrere kategoriale Eingaben den gleichen Hash-Bucket-Wert verwenden.

Tabelle 2-2: Die erwartete Anzahl von Einträgen pro Bucket und die Wahrscheinlichkeit von mindestens einer Kollision, wenn aus den IATA-Flughafencodes Hashwerte für verschiedene Anzahlen von Buckets erzeugt werden

num_hash_buckets	entries_per_bucket	collision_prob
3	115.666667	1.000000
10	34.700000	1.000000
100	3.470000	1.000000
1000	0.347000	1.000000
10000	0.034700	0.997697
100000	0.003470	0.451739

Schiefe

Der Genauigkeitsverlust ist besonders akut, wenn die Verteilung der kategorialen Eingabe eine hohe Schiefe aufweist. Nehmen wir den Hash-Bucket an, der ORD enthält (Chicago, einer der verkehrsreichsten Flughäfen der Welt). Er lässt sich wie folgt finden:

```
CREATE TEMPORARY FUNCTION hashed(airport STRING, numbuckets INT64) AS (
    ABS(MOD(FARM_FINGERPRINT(airport), numbuckets))
);

WITH airports AS (
SELECT
    departure_airport, COUNT(1) AS num_flights
FROM `bigquery-samples.airline_ontime_data.flights`
GROUP BY departure_airport
)

SELECT
    departure_airport, num_flights
FROM airports
WHERE hashed(departure_airport, 100) = hashed('ORD', 100)
```

Das Ergebnis zeigt, dass es zwar rund 3,6 Millionen Flüge von ORD gibt, aber nur etwa 67.000 Flüge von BTV (Burlington, Vermont):

departure_airport	num_flights
ORD	3610491
BTV	66555
MCI	597761

Dies weist darauf hin, dass das Modell die langen Rollzeiten und Wetterverzögerungen, die Chicago erfährt, dem städtischen Flughafen in Burlington, Vermont, zuschreibt! Die Modellgenauigkeit für BTV und MCI (Flughafen von Kansas City) wird ziemlich schlecht sein, weil es so viele Flüge von Chicago aus gibt.

Aggregat-Feature

Wenn die Verteilung einer kategorialen Variablen schief oder die Anzahl der Buckets so klein ist, dass Bucket-Kollisionen häufig vorkommen, könnte es hilfreich sein, ein Aggregat-Feature als Eingabe für das Modell hinzuzufügen. Zum Beispiel könnten wir für jeden Flughafen die Wahrscheinlichkeit für pünktliche Flüge im Trainingsdatensatz ermitteln und sie unserem Modell als Feature hinzufügen. Dadurch können wir vermeiden, Informationen zu verlieren, die mit einzelnen Flughäfen verbunden sind, wenn wir aus den Flughafencodes Hashwerte erzeugen. In manchen Fällen könnten wir auf den Flughafennamen als Feature gänzlich verzichten, da die relative Häufigkeit von pünktlichen Flügen ausreichen könnte.

Hyperparameter optimieren

Aufgrund der Kompromisse bei der Häufigkeit von Bucket-Kollisionen kann es schwierig sein, die Anzahl der Buckets zu wählen. Oftmals hängt sie vom Problem selbst ab. Konzipieren Sie deshalb die Anzahl der Buckets als Hyperparameter, der optimiert wird:

```
- parameterName: nbuckets
  type: INTEGER
  minValue: 10
  maxValue: 20
  scaleType: UNIT_LINEAR_SCALE
```

Achten Sie darauf, dass die Anzahl der Buckets in einem sinnvollen Bereich bleibt, und zwar in Bezug auf die Kardinalität der kategorialen Variablen, für die ein Hashwert gebildet werden soll.

Kryptografische Hashfunktion

Beim Entwurfsmuster *Hashed Feature* entstehen Verluste durch den Modulo-Teil der Implementierung. Wie wäre es, wenn wir die Modulo-Operation ganz vermeiden würden? Immerhin hat der Farm-Fingerprint eine feste Länge (ein INT64 umfasst 64 Bits), ließe sich also mit 64 Feature-Werten darstellen, die jeweils 0 oder 1 sind. Dies ist die sogenannte *binäre Codierung*.

Allerdings löst die binäre Codierung nicht das Problem der Eingaben außerhalb des Vokabulars oder des Kaltstarts (nur das Problem der hohen Kardinalität). In der Tat führt die bitweise Codierung auf eine falsche Spur. Ohne die Modulo-Operation können wir zu einer eindeutigen Darstellung gelangen, indem wir einfach die drei Zeichen des IATA-Codes codieren (und somit ein Feature der Länge $3 \times 26 = 78$ verwenden). Das Problem bei dieser Darstellung liegt auf der Hand: Der Anfangsbuchstabe der Flughäfen hat nichts mit ihren Flugverspätungseigenschaften zu tun – die Codierung erzeugt eine *Scheinkorrelation* zwischen Flughäfen, die mit dem gleichen Buchstaben beginnen. Die gleiche Erkenntnis gilt auch im binären Raum. Deshalb raten wir von der binären Codierung der Farm-Fingerprint-Werte ab. Bei der binären Codierung eines MD5-Hashwerts tritt das Problem der Scheinkorrelation nicht auf, da die Ausgaben einer MD5-Hashfunktion gleichverteilt sind. Allerdings ist die MD5-Hashfunktion im Unterschied zum Farm-Fingerprint-Algorithmus weder deterministisch noch eindeutig – sie ist eine Einweg-Hashfunktion und wird viele unerwartete Kollisionen verursachen.

Im Entwurfsmuster *Hashed Feature* müssen wir einen Fingerprint-Hashing-Algorithmus verwenden, keinen kryptografischen Hashing-Algorithmus. Das hängt damit zusammen, dass eine Fingerprint-Funktion einen deterministischen und eindeutigen Wert erzeugen soll. Bei Lichte betrachtet, ist dies eine wichtige Anforderung an Vorverarbeitungsfunktionen im maschinellen Lernen, da wir dieselbe Funktion beim Bereitstellen des Modells anwenden und den gleichen Hashwert bekommen müssen. Eine Fingerprint-Funktion erzeugt keine gleichverteilten Ausgabewerte. Kryptografische Algorithmen wie MD5 oder SHA1 liefern gleichverteilte Ausgaben, sie sind aber nicht deterministisch und absichtlich rechenintensiv gestaltet. Demzufolge ist eine kryptografische Hashfunktion in einem Feature-Engineering-Kontext nicht verwendbar, wo der Hashwert, der während der Voraussage für eine bestimmte Eingabe berechnet wird, der gleiche sein muss wie der Hashwert, der während des Trainings berechnet wurde, und wo die Hashfunktion das ML-Modell nicht bremsen sollte.

Dass MD5 nicht deterministisch ist, liegt daran, dass eine zufällige Zeichenfolge, das sogenannte *Salt* (*https://de.wikipedia.org/wiki/Salt_(Kryptologie)*), an die Zeichenfolge, für die ein Hashwert erzeugt werden soll, angefügt wird. Indem man das Salt an jedes Passwort anfügt, gewährleistet man, dass auch dann, wenn zwei Benutzer zufällig das gleiche Passwort verwenden, in der Datenbank verschiedene Hashwerte erscheinen. Das ist notwendig, um Angriffe mittels »Rainbow Tables« zu vereiteln. Derartige Angriffe stützen sich auf Wörterbücher mit häufig gewählten Passwörtern und vergleichen den Hashwert eines bekannten Passworts mit den Hashwerten in der Datenbank. Dank gestiegener Rechenleistung ist es möglich, einen Brute-Force-Angriff auch auf jedes mögliche Salt durchzuführen. Moderne kryptografische Implementierungen erzeugen den Hashwert deshalb in einer Schleife, um den Rechenaufwand zu erhöhen. Doch selbst wenn wir das Salt weglassen und die Anzahl der Durchläufe auf 1 reduzieren, bleibt MD5 eine Einweg-Hashfunktion. Sie wird auch nicht eindeutig sein.

Die Quintessenz ist, dass wir einen Fingerprint-Hashing-Algorithmus verwenden und auf den resultierenden Hashwert die Modulo-Operation anwenden müssen.

Reihenfolge der Operationen

Die Modulo-Operation führen wir zuerst aus und bilden dann den Absolutwert:

```
CREATE TEMPORARY FUNCTION hashed(airport STRING, numbuckets INT64) AS (
    ABS(MOD(FARM_FINGERPRINT(airport), numbuckets))
);
```

Die Reihenfolge der Funktionen `ABS`, `MOD` und `FARM_FINGERPRINT` im obigen Codefragment ist wichtig, weil der Bereich von `INT64` nicht symmetrisch ist, sondern von –9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807 (jeweils inklusive) reicht. Wenn wir also

```
ABS(FARM_FINGERPRINT(airport))
```

ausführen, könnte es zu einem seltenen und wahrscheinlich nicht reproduzierbaren Überlauffehler kommen, falls die `FARM_FINGERPRINT`-Operation zufällig `-9.223.372.036.854.775.808` zurückgeben würde, da sich dessen Absolutwert nicht mit einem `INT64` darstellen lässt!

Leere Hash-Buckets

Selbst bei nur zehn Hash-Buckets, die 347 Flughäfen darstellen sollen, besteht eine – wenn auch geringe – Restwahrscheinlichkeit, dass einer der Hash-Buckets leer bleibt. Wenn man also mit Hashed-Feature-Spalten arbeitet, kann es vorteilhaft sein, auch eine L2-Regularisierung zu verwenden, sodass die mit einem leeren Bucket verbundenen Gewichte fast auf null gezogen werden(siehe *https://oreil.ly/xlwAH*). Auf diese Weise wird das Modell nicht numerisch instabil, wenn ein Flughafen außerhalb des Vokabulars in einen leeren Bucket fällt.

Entwurfsmuster 2: Einbettungen

Einbettungen sind eine lernbare Datendarstellung, die Daten mit hoher Kardinalität so in einen Raum mit weniger Dimensionen abbildet, dass die für das Lernproblem relevanten Informationen erhalten bleiben. Einbettungen bilden das Herzstück im modernen maschinellen Lernen und sind in diesem Bereich in verschiedenen Ausprägungen zu finden.

Problem

Modelle für maschinelles Lernen suchen systematisch in den Daten nach Mustern, die erfassen, wie sich die Eigenschaften der Eingabe-Features des Modells zum Ausgabe-Label verhalten. Folglich beeinflusst die Datendarstellung der Eingabe-Features direkt die Qualität des endgültigen Modells. Während sich strukturierte

numerische Eingaben recht einfach verarbeiten lassen, können die für das Training eines ML-Modells benötigten Daten in unzähligen Varianten vorliegen, beispielsweise kategoriale Features, Text, Bilder, Audio, Zeitreihen und viele mehr. Für diese Datendarstellungen brauchen wir einen aussagekräftigen numerischen Wert, dem wir unserem ML-Modell übergeben, sodass diese Features in das typische Trainingsparadigma passen können. Einbettungen bieten eine Möglichkeit, einige dieser unterschiedlichen Datentypen in einer Weise zu verarbeiten, die Ähnlichkeiten zwischen den Elementen bewahrt und somit die Fähigkeit unseres Modells verbessert, diese wichtigen Muster zu lernen.

Die 1-aus-n-Codierung ist eine gängige Methode, um kategoriale Eingabevariablen darzustellen. Nehmen Sie als Beispiel die Pluralitätseingabe im Geburtendatensatz[4]. Es handelt sich um eine kategoriale Eingabe mit sechs möglichen Werten: `['Single(1)', 'Multiple(2+)', 'Twins(2)', 'Triplets(3)', 'Quadruplets(4)', 'Quintuplets(5)']` (Einzelkinder (1), Mehrlinge (2+), Zwillinge (2), Drillinge (3), Vierlinge (4), Fünflinge (5)). Diese kategoriale Eingabe können wir mit einer 1-aus-n-Codierung verarbeiten, die jeden potenziellen Eingabezeichenfolgenwert auf einen Einheitsvektor in R^6 abbildet, wie Tabelle 2-3 zeigt.

Tabelle 2-3: Ein Beispiel für die 1-aus-n-Codierung kategorialer Eingaben für den Geburtendatensatz

Pluralität	1-aus-n-Codierung
Single(1)	[1,0,0,0,0,0]
Multiple(2+)	[0,1,0,0,0,0]
Twins(2)	[0,0,1,0,0,0]
Triplets(3)	[0,0,0,1,0,0]
Quadruplets(4)	[0,0,0,0,1,0]
Quintuplets(5)	[0,0,0,0,0,1]

Wenn wir die Eingaben auf diese Weise codieren, brauchen wir sechs Dimensionen, um die verschiedenen Kategorien darzustellen. Sechs Dimensionen mögen harmlos erscheinen, doch wie sieht es aus, wenn wir sehr viele weitere Kategorien zu berücksichtigen hätten?

Nehmen wir zum Beispiel an, unser Datensatz bestünde aus dem Abrufverlauf der Kunden unserer Videodatenbank und es ist unsere Aufgabe, anhand der vorherigen Videointeraktionen der Kunden eine Liste von neuen Videos vorzuschlagen. In diesem Szenario könnte das Feld `customer_id` Millionen eindeutiger Einträge enthalten. Ähnlich verhält es sich mit dem Feld `video_id`, das die bisher angesehenen Videos aufnimmt und ebenfalls Tausende von Einträgen enthalten könnte. Die 1-aus-n-Codierung kategorialer Features mit *hoher Kardinalität* wie `video_id` oder

4 Dieser Datensatz ist in BigQuery verfügbar: *bigquery-public-data.samples.natality*.

customer_id als Eingaben in ein ML-Modell führt zu einer Sparse-Matrix, die sich bei einer Reihe von Algorithmen für maschinelles Lernen nicht gut eignet.

Die 1-aus-n-Codierung weist zudem das Problem auf, dass sie die kategorialen Variablen als *unabhängig* behandelt. Allerdings sollte die Datendarstellung für Zwillinge nahe an der Datendarstellung für Drillinge liegen und ziemlich weit entfernt von der Datendarstellung für Fünflinge. Ein Mehrling ist höchstwahrscheinlich ein Zwilling, könnte aber auch ein Drilling sein. Als Beispiel zeigt Tabelle 2-4 eine alternative Darstellung der Pluralitätsspalte in einer niedrigeren Dimension, die diese Beziehung der *Nähe* erfasst.

Tabelle 2-4: Durch Einbettung mit geringerer Dimensionalität die Pluralitätsspalte im Geburtendatensatz darstellen

Pluralität	Potenzielle Codierung
Single (1)	[1.0,0.0]
Multiple(2+)	[0.0,0.6]
Twins(2)	[0.0,0.5]
Triplets(3)	[0.0,0.7]
Quadruplets(4)	[0.0,0.8]
Quintuplets(5)	[0.0,0.9]

Diese Zahlen sind natürlich willkürlich gewählt. Doch ist es möglich, die bestmögliche Darstellung der Pluralitätsspalte mit nur zwei Dimensionen für das Geburtenratenproblem zu lernen? Das ist das Problem, das das Entwurfsmuster *Einbettungen* löst.

Das gleiche Problem hoher Kardinalität und abhängiger Daten tritt auch in Bildern und Text auf. Bilder bestehen aus Tausenden von Pixeln, die nicht unabhängig voneinander sind. Text in natürlicher Sprache entsteht aus einem Vokabular mit Zehntausenden von Wörtern, und ein Wort wie walk ist dem Wort run näher als dem Wort book.

Lösung

Das Entwurfsmuster *Einbettungen* befasst sich mit dem Problem, Daten hoher Kardinalität in einer niedrigeren Dimension dicht darzustellen, indem die Eingabedaten über eine Einbettungsschicht mit trainierbaren Gewichten geleitet werden. Dabei wird die hochdimensionale, kategoriale Eingabevariable auf einen reellwertigen Vektor in einem Raum mit weniger Dimensionen abgebildet. Die Gewichte, mit denen diese dichte Darstellung entsteht, werden als Teil der Optimierung des Modells gelernt (siehe Abbildung 2-5). In der Praxis führen diese Einbettungen dazu, dass sich die Nähebeziehungen in den Eingabedaten erfassen lassen.

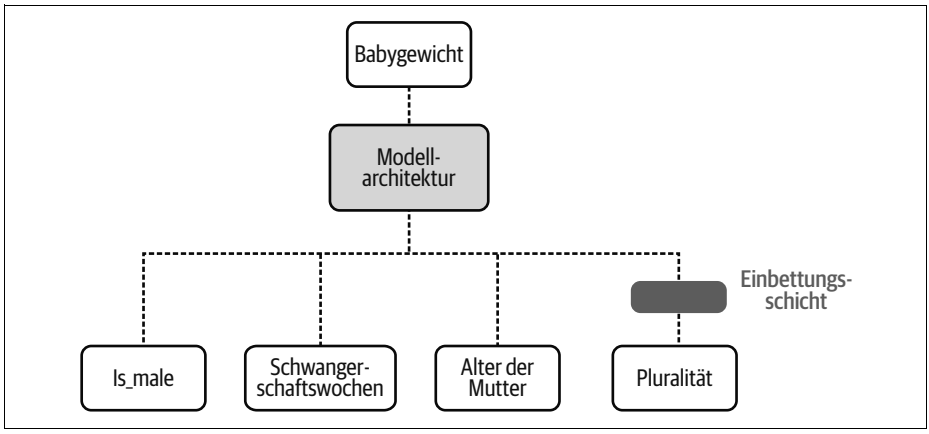

Abbildung 2-5: Die Gewichte einer Einbettungsschicht werden während des Trainings als Parameter gelernt.

Da Einbettungen die Nähebeziehungen in den Eingabedaten in einer Darstellung mit weniger Dimensionen erfassen, können wir eine Einbettungsschicht als Ersatz für Clustering-Techniken (z. B. Kundensegmentierung) und Methoden der Dimensionalitätsreduktion wie die Hauptkomponentenanalyse (PCA) verwenden. Die Einbettungsgewichte werden in der Haupttrainingsschleife des Modells ermittelt, sodass es nicht notwendig ist, die Daten im Vorfeld zu clustern oder einer Hauptkomponentenanalyse zu unterziehen.

Die Gewichte in der Einbettungsschicht würden als Teil des Gradientenabstiegsverfahrens gelernt, wenn das Geburtenratenmodell trainiert wird.

Am Ende des Trainings könnten die Gewichte der Einbettungsschicht eine Codierung der kategorialen Variablen wie in Tabelle 2-5 ergeben.

Tabelle 2-5: 1-aus-n- und gelernte Codierungen für die Pluralitätsspalte im Geburtendatensatz

Pluralität	1-aus-n-Codierung	Gelernte Codierung
Single(1)	[1,0,0,0,0,0]	[0.4, 0.6]
Multiple(2+)	[0,1,0,0,0,0]	[0.1, 0.5]
Twins(2)	[0,0,1,0,0,0]	[–0.1, 0.3]
Triplets(3)	[0,0,0,1,0,0]	[–0.2, 0.5]
Quadruplets(4)	[0,0,0,0,1,0]	[–0.4, 0.3]
Quintuplets(5)	[0,0,0,0,0,1]	[–0.6, 0.5]

Die Einbettung bildet einen dünn besetzten 1-aus-n-codierten Vektor auf einen dichten Vektor in R^2 ab.

In TensorFlow konstruieren wir zuerst eine kategoriale Feature-Spalte für das Feature und umhüllen es dann mit einer einbettenden Feature-Spalte. Zum Beispiel würden wir für unser Pluralitäts-Feature Folgendes haben:

```
plurality = tf.feature_column.categorical_column_with_vocabulary_list(
            'plurality', ['Single(1)', 'Multiple(2+)', 'Twins(2)',
'Triplets(3)', 'Quadruplets(4)', 'Quintuplets(5)'])
plurality_embed = tf.feature_column.embedding_column(plurality, dimension=2)
```

Die resultierende Feature-Spalte (`plurality_embed`) wird als Eingabe für die nachgelagerten Knoten des neuronalen Netzes anstelle der 1-aus-n-codierten Feature-Spalte (Pluralität) verwendet.

Texteinbettungen

Text bietet ein natürliches Umfeld, in dem man eine Einbettungsschicht vorteilhaft nutzen kann. Angesichts der Kardinalität eines Vokabulars (oftmals in der Größenordnung von 10.000 Wörtern) ist es nicht praktikabel, jedes Wort 1-aus-n zu codieren. Es würde eine unglaublich große (hochdimensionale) Sparse-Matrix für das Training entstehen. Außerdem möchten wir, dass die Einbettungen bei ähnlichen Wörtern nahe beieinanderliegen und nicht verwandte Wörter sich im Einbettungsraum weit voneinander entfernt befinden. Daher verwenden wir eine dichte Worteinbettung, um die diskrete Texteingabe zu vektorisieren, bevor wir sie an unser Modell übergeben.

Um eine Texteinbettung in Keras zu implementieren, erstellen wir zuerst eine Tokenisierung für jedes Wort in unserem Vokabular, wie Abbildung 2-6 zeigt. Dann verwenden wir diese Tokenisierung, um eine Einbettungsschicht abzubilden, ähnlich wie Sie es von der Pluralitätsspalte kennen.

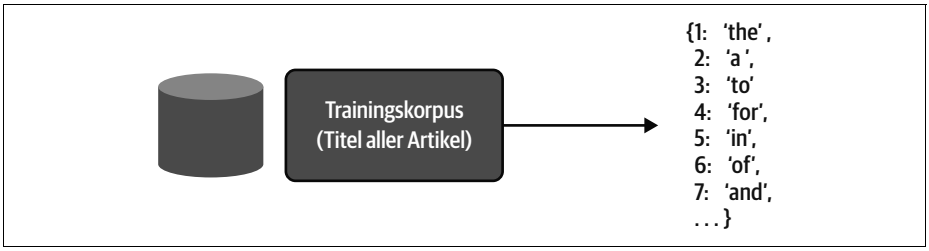

Abbildung 2-6: Der Tokenizer erzeugt eine Nachschlagetabelle, die jedes Wort auf einen Index abbildet.

Die Tokenisierung erzeugt eine Nachschlagetabelle, die jedes Wort in unserem Vokabular auf einen Index abbildet. Wir können uns dies vorstellen als eine 1-aus-n-Codierung jedes Worts, wobei der tokenisierte Index die Position des Nicht-Null-Elements in der 1-aus-n-Codierung ist. Hierfür ist ein vollständiger Durchlauf durch den gesamten Datensatz erforderlich (unter der Annahme, dass er aus den Titeln von Artikeln besteht)[5], um die Nachschlagetabelle zu erzeugen. Dies kann in Keras ausgeführt werden. Den vollständigen Code finden Sie im Repository für dieses

Buch (*https://github.com/GoogleCloudPlatform/ml-design-patterns/blob/master/02_data_representation/embeddings.ipynb*):

```
from tensorflow.keras.preprocessing.text import Tokenizer

tokenizer = Tokenizer()
tokenizer.fit_on_texts(titles_df.title)
```

Hier können wir auf die Klasse `Tokenizer` aus der Bibliothek `keras.preprocessing.text` zurückgreifen. Der Aufruf von `fit_on_texts` erzeugt eine Nachschlagetabelle, die jedes Wort, das in unseren Titeln auftaucht, auf einen Index abbildet. Mit dem Aufruf `tokenizer.index_word` können wir diese Nachschlagetabelle direkt inspizieren:

```
tokenizer.index_word
{1: 'the',
 2: 'a',
 3: 'to',
 4: 'for',
 5: 'in',
 6: 'of',
 7: 'and',
 8: 's',
 9: 'on',
 10: 'with',
 11: 'show',
...
```

Diese Zuordnung können wir dann mit der Methode `texts_to_sequences` unseres Tokenizers aufrufen. Dadurch wird jede Folge von Wörtern in der darzustellenden Texteingabe (hier nehmen wir an, dass es sich um Titel von Artikeln handelt) auf eine Folge von Tokens abgebildet, die den einzelnen Wörtern entsprechen (siehe Abbildung 2-7):

```
integerized_titles = tokenizer.texts_to_sequences(titles_df.title)
```

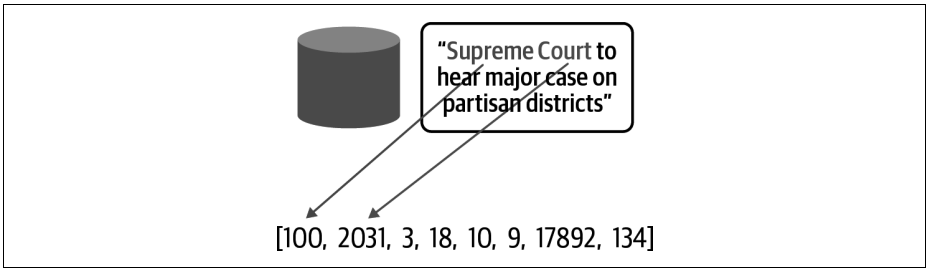

Abbildung 2-7: Mithilfe des Tokenizers wird jeder Titel auf eine Folge von ganzzahligen Indexwerten abgebildet.

Der Tokenizer enthält weitere relevante Informationen, die wir später verwenden, um eine Einbettungsschicht zu erstellen. Speziell erfasst `VOCAB_SIZE`, wie viele Ele-

5 Dieser Datensatz ist in BigQuery verfügbar: *bigquery-public-data.hacker_news.stories*.

mente die Indexnachschlagetabelle enthält, und MAX_LEN nimmt die maximale Länge der Textzeichenfolgen im Datensatz auf:

```
VOCAB_SIZE = len(tokenizer.index_word)
MAX_LEN = max(len(sequence) for sequence in integerized_titles)
```

Bevor das Modell erstellt wird, müssen wir die Titel im Datensatz aufbereiten. Um den Titel in das Modell einzuspeisen, ist es erforderlich, die Elemente des Titels aufzufüllen. Keras bietet hierfür die Hilfsfunktion pad_sequence, die auf den Tokenizer-Methoden aufsetzt. Die Funktion create_sequences übernimmt sowohl Titel als auch die maximale Satzlänge als Eingabe und gibt eine Liste von Ganzzahlen zurück. Diese entsprechen unseren Tokens, die bis zur maximalen Satzlänge aufgefüllt wurden:

```
from tensorflow.keras.preprocessing.sequence import pad_sequences

def create_sequences(texts, max_len=MAX_LEN):
    sequences = tokenizer.texts_to_sequences(texts)
    padded_sequences = pad_sequences(sequences,
                                     max_len,
                                     padding='post')
    return padded_sequences
```

Als Nächstes erstellen wir in Keras das Modell eines tiefen neuronalen Netzes (*Deep Neural Network*, DNN), das eine einfache Einbettungsschicht implementiert, um die Wort-Ganzzahlen in dichte Vektoren zu transformieren. Die Embedding-Schicht von Keras kann man sich als Abbildung der Ganzzahlindizes bestimmter Wörter auf dichte Vektoren (ihre Einbettungen) vorstellen. Die Dimensionalität der Einbettung wird durch output_dim bestimmt. Das Argument input_dim gibt die Größe des Vokabulars an und input_shape die Länge der Eingabesequenzen. Da wir hier die Titel auffüllen und erst dann an das Modell übergeben, setzen wir input_shape=[MAX_LEN]:

```
model = models.Sequential([layers.Embedding(input_dim=VOCAB_SIZE + 1,
                                            output_dim=embed_dim,
                                            input_shape=[MAX_LEN]),
                           layers.Lambda(lambda x: tf.reduce_mean(x,axis=1)),
                           layers.Dense(N_CLASSES, activation='softmax')])
```

Um die von der Einbettungsschicht zurückgegebenen Wortvektoren zu mitteln, müssen wir eine benutzerdefinierte Keras-Lambda-Schicht zwischen die Einbettungsschicht und die dichte Softmax-Schicht setzen. Der erhaltene Durchschnittswert wird in die dichte Softmax-Schicht eingespeist. Auf diese Weise erzeugen wir ein Modell, das einfach ist, das aber Informationen über die Wortreihenfolge verliert. Es entsteht ein Modell, das Sätze als *Bag-of-Words* sieht.

Bildeinbettungen

Während es sich bei Text um sehr spärliche Eingaben handelt, bestehen andere Datentypen wie Bild- oder Audiodaten aus dichten, hochdimensionalen Vektoren, die meist mehrere Kanäle mit rohen Pixel- oder Frequenzinformationen enthalten.

In dieser Situation erfasst eine Einbettung eine relevante Eingabedarstellung mit geringerer Dimension.

Bei Bildeinbettungen wird zunächst ein komplexes *Convolutional Neural Network* (wie Inception oder ResNet) auf einem großen Bilddatensatz (wie ImageNet) trainiert, das Millionen von Bildern und Tausende von möglichen Klassifizierungs-Labels enthält. Dann wird die letzte Softmax-Schicht aus dem Modell entfernt. Ohne die letzte Softmax-Klassifizierungsschicht lässt sich das Modell verwenden, um einen Feature-Vektor für eine bestimmte Eingabe zu extrahieren. Da dieser Feature-Vektor sämtliche relevanten Informationen des Bilds enthält, ist er praktisch eine niedrigdimensionale Einbettung des Eingabebilds.

Sehen Sie sich analog dazu die Aufgabe der Bildbeschriftung an, d.h. das Generieren einer inhaltlichen Beschriftung eines gegebenen Bilds, wie Abbildung 2-8 zeigt.

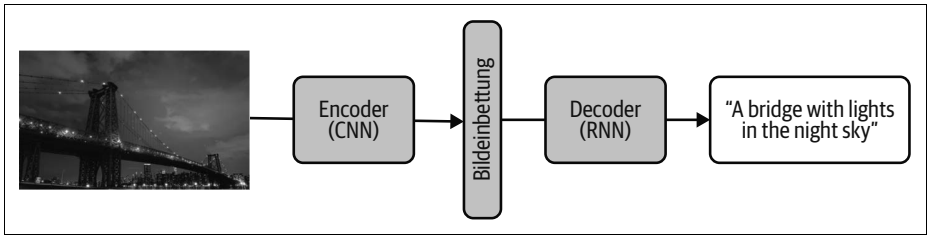

Abbildung 2-8: Für die Aufgabe der Bildübersetzung produziert der Encoder eine niedrigdimensionale Einbettungsdarstellung des Bilds.

Indem diese Modellarchitektur auf einem umfangreichen Datensatz an Bild-Beschriftung-Paaren trainiert wird, lernt der Encoder eine effiziente Vektordarstellung für Bilder. Der Decoder lernt, wie dieser Vektor in eine Textüberschrift zu übersetzen ist. In diesem Sinne wird der Encoder zu einer Image2Vec-Einbettungsmaschine.

Warum es funktioniert

Die Einbettungsschicht ist lediglich eine weitere verdeckte Schicht des neuronalen Netzes. Die Gewichte werden dann jeder der Dimensionen mit hoher Kardinalität zugeordnet, und die Ausgabe wird durch das übrige Netz weitergeleitet. Die Gewichte, die zur Einbettung gehören, werden also genau wie alle anderen Gewichte im neuronalen Netz über das Gradientenabstiegsverfahren gelernt. Das heißt, dass die resultierenden Vektoreinbettungen die effizienteste niedrigdimensionale Repräsentation dieser Feature-Werte in Bezug auf die Lernaufgabe darstellen.

Während diese verbesserte Einbettung letztlich dem Modell hilft, haben die Einbettungen an sich einen inhärenten Wert und erlauben uns, zusätzliche Einblicke in unseren Datensatz zu gewinnen.

Kommen wir noch einmal auf den Datensatz mit den Kundenvideos zurück. Verwendet man nur die 1-aus-n-Codierung, haben zwei verschiedene Benutzer `user_i` und `user_j` das gleiche Ähnlichkeitsmaß. In analoger Weise würde die Punktpro-

dukt- bzw. die Kosinusähnlichkeit für zwei verschiedene sechsdimensionale 1-aus-n-Codierungen der Geburtenpluralität eine Ähnlichkeit von null aufweisen. Das ist verständlich, da die 1-aus-n-Codierung unserem Modell praktisch sagt, zwei verschiedene Mehrlingsgeburten als voneinander unabhängig zu behandeln. Für unseren Datensatz von Kunden und Videoabrufen verlieren wir jeglichen Ähnlichkeitsbegriff zwischen Kunden oder Videos. Doch das fühlt sich nicht ganz richtig an. Zwei verschiedene Kunden oder Videos werden wahrscheinlich irgendwelche Ähnlichkeiten aufweisen. Das Gleiche gilt für Mehrlingsgeburten. Das Auftreten von Vierlingen und Fünflingen beeinflusst wahrscheinlich das Geburtsgewicht in einer statistisch ähnlichen Weise im Vergleich zu den Geburtsgewichten von Einzelkindern (siehe Abbildung 2-9).

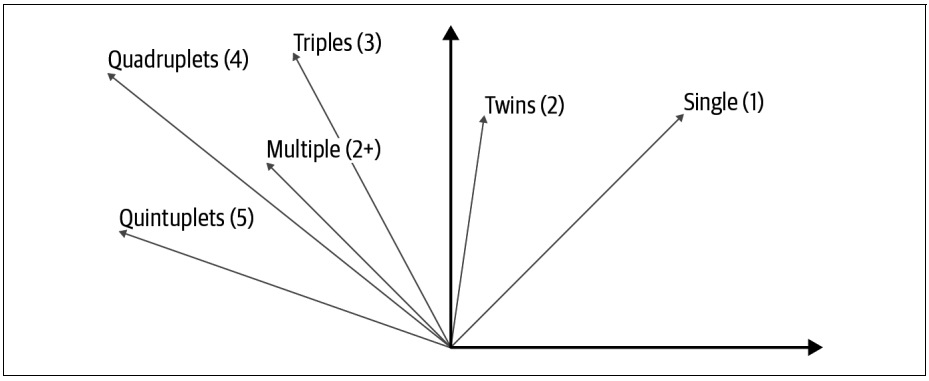

Abbildung 2-9: Indem wir unsere kategoriale Variable in einen Einbettungsraum mit weniger Dimensionen zwingen, können wir auch Beziehungen zwischen den verschiedenen Kategorien lernen.

Wenn wir die Ähnlichkeit der Mehrlingskategorien als 1-aus-n-codierte Vektoren berechnen, erhalten wir die Identitätsmatrix, da jede Kategorie als eigenständiges Feature behandelt wird (siehe Tabelle 2-6).

Tabelle 2-6: Wenn Features 1-aus-n-codiert sind, ist die Ähnlichkeitsmatrix nichts weiter als die Identitätsmatrix.

	Single(1)	Multiple(2+)	Twins(2)	Triplets(3)	Quadruplets(4)	Quintuplets(5)
Single(1)	1	0	0	0	0	0
Multiple(2+)	-	1	0	0	0	0
Twins(2)	-	-	1	0	0	0
Triplets(3)	-	-	-	1	0	0
Quadruplets(4)	-	-	-	-	1	0
Quintuplets(5)	-	-	-	-	-	1

Sobald jedoch die Pluralität in zwei Dimensionen eingebettet ist, wird das Ähnlichkeitsmaß nicht trivial, und es entstehen wichtige Beziehungen zwischen den verschiedenen Kategorien (siehe Tabelle 2-7).

Tabelle 2-7: Wenn die Features in zwei Dimensionen eingebettet sind, liefert uns die Ähnlichkeitsmatrix mehr Informationen.

	Single(1)	Multiple(2+)	Twins(2)	Triplets(3)	Quadruplets(4)	Quintuplets(5)
Single(1)	1	0.92	0.61	0.57	0.06	0.1
Multiple(2+)	-	1	0.86	0.83	0.43	0.48
Twins(2)	-	1	0.99	0.82	0.85	
Triplets(3)	-	1	0.85	0.88		
Quadruplets(4)	-	1	0.99			
Quintuplets(5)	-	-	-	-	-	1

Somit erlaubt uns eine gelernte Einbettung, inhärente Ähnlichkeiten zwischen zwei separaten Kategorien zu extrahieren, und wir können – setzt man eine numerische Vektordarstellung voraus – die Ähnlichkeit zwischen zwei kategorialen Features genau quantifizieren.

Mit dem Geburtendatensatz lässt sich das leicht visualisieren, doch das gleiche Prinzip gilt auch, wenn man mit `customer_id`s arbeitet, die in einen 20-dimensionalen Raum eingebettet sind. Wenn wir die Einbettungen auf unseren Kundendatensatz anwenden, erlauben sie uns, ähnliche Kunden für eine gegebene `customer_id` abzurufen und Vorschläge basierend auf der Ähnlichkeit zu machen, beispielsweise welche Videos wahrscheinlich angesehen werden (siehe Abbildung 2-10). Darüber hinaus lassen sich diese Benutzer- und Elementeinbettungen mit anderen Features kombinieren, wenn ein separates ML-Modell trainiert wird. Die Verwendung von vorab trainierten Einbettungen in ML-Modellen bezeichnet man als *Transfer Learning*.

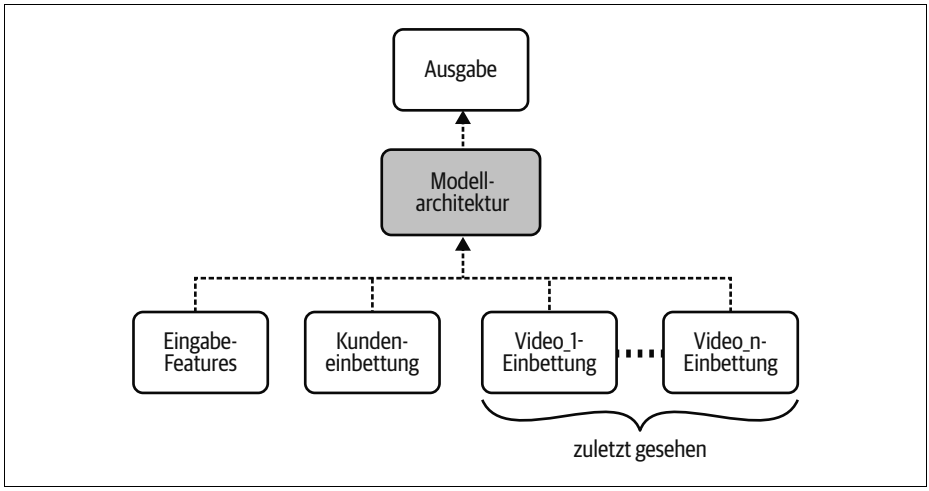

Abbildung 2-10: Durch Lernen eines dichten Einbettungsvektors mit wenigen Dimensionen für jeden Kunden und jedes Video ist ein einbettungsbasiertes Modell in der Lage, bei weniger Aufwand für ein manuelles Feature Engineering gut zu verallgemeinern.

Kompromisse und Alternativen

Der Hauptnachteil bei Verwendung einer Einbettung ist die eingeschränkte Darstellung der Daten. Der Übergang von einer Darstellung mit hoher Kardinalität zu einer Darstellung mit weniger Dimensionen ist mit einem Informationsverlust verbunden. Im Gegenzug gewinnen wir Informationen über die Nähe und den Kontext der Elemente.

Die Einbettungsdimension auswählen

Die genaue Dimensionalität des Einbettungsraums ist etwas, das wir als Praktiker:innen auswählen. Sollten wir also eine große oder eine kleine Einbettungsdimension wählen? Natürlich gibt es hier einen Kompromiss, wie bei den meisten Dingen im maschinellen Lernen. Die Verlustrate der Darstellung wird durch die Größe der Einbettungsschicht gesteuert. Wählt man für die Einbettungsschicht eine sehr kleine Ausgabedimension, werden zu viele Informationen in einen kleinen Vektorraum gezwungen, und Kontext kann verloren gehen. Ist dagegen die Einbettungsdimension zu groß, verliert die Einbettung die gelernte kontextuelle Bedeutung der Features. Im Extremfall sind wir zurück bei dem Problem, auf das wir bei der 1-aus-n-Codierung gestoßen sind. Die optimale Einbettungsdimension wird oft durch Experimentieren gefunden, ähnlich wie man die Anzahl der Neuronen in einer tiefen neuronalen Netzwerkschicht ermittelt.

Wenn es schnell gehen muss, kann man als Faustregel die vierte Wurzel (*https://oreil.ly/ywFco*) aus der Gesamtanzahl von eindeutigen kategorialen Elementen verwenden, während eine andere Regel lautet, dass die Einbettungsdimension ungefähr 1,6-mal die Quadratwurzel (*https://oreil.ly/github-fastai-2-blob-fastai-2-tabular-model-py*) aus der Anzahl der eindeutigen Elemente in der Kategorie sein sollte, aber nicht weniger als 600. Nehmen wir zum Beispiel an, Sie möchten eine Einbettungsschicht verwenden, um ein Feature mit 625 eindeutigen Werten zu codieren. Entsprechend der ersten Faustregel würden wir eine Einbettungsdimension für Pluralität von 5 wählen, gemäß der zweiten Faustregel würden wir 40 wählen. Wenn wir die Hyperparameter abstimmen, könnte es sich lohnen, innerhalb dieses Bereichs zu suchen.

Autoencoder

Es kann schwierig sein, Einbettungen in einem überwachten Modus zu trainieren, da es jede Menge gelabelter Daten erfordert. Damit ein Bildklassifizierungsmodell wie Inception in der Lage ist, brauchbare Bildeinbettungen zu produzieren, wird es auf ImageNet trainiert, das 14 Millionen gelabelter Bilder umfasst. Dieser Bedarf an einem riesigen gelabelten Datensatz lässt sich unter anderem mit Autoencodern decken.

Die in Abbildung 2-11 gezeigte typische Autoencoder-Architektur besteht aus einer Flaschenhalsebene, die im Wesentlichen eine Einbettungsebene ist. Der Teil

des Netzes vor dem Flaschenhals (der *Encoder*) bildet eine hochdimensionale Eingabe auf eine Einbettungsschicht mit weniger Dimensionen ab, während das letzte Netz (der *Decoder*) diese Darstellung zurück auf eine höhere Dimension abbildet, typischerweise die gleiche Dimension wie das Original. Das Modell wird in der Regel auf einer Variante eines Rekonstruktionsfehlers trainiert, der die Ausgabe des Modells zwingt, der Eingabe so ähnlich wie möglich zu sein.

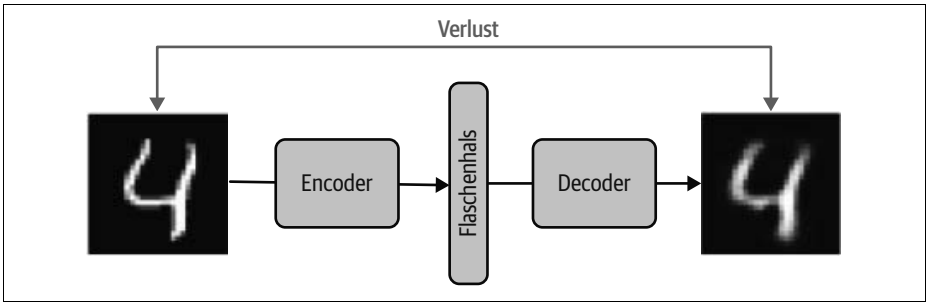

Abbildung 2-11: Beim Training eines Autoencoders sind das Feature und das Label gleich, und der Verlust ist der Rekonstruktionsfehler. Damit ist der Autoencoder in der Lage, eine nichtlineare Dimensionsreduktion zu erreichen.

Da die Eingabe gleich der Ausgabe ist, sind keine zusätzlichen Labels erforderlich. Der Encoder lernt eine optimale nichtlineare Dimensionsreduktion der Eingabe. Ähnlich wie die Hauptkomponentenanalyse eine lineare Dimensionsreduktion erreicht, ist die Flaschenhalsschicht eines Autoencoders in der Lage, eine nichtlineare Dimensionsreduktion über das Einbetten zu erreichen.

Damit haben wir die Möglichkeit, ein schwieriges ML-Problem in zwei Teile zu zerlegen. Zuerst verwenden wir alle verfügbaren ungelabelten Daten, um von hoher Kardinalität zu niedriger Kardinalität zu gelangen, indem wir Autoencoder als *Hilfslernaufgabe* verwenden. Dann lösen wir das eigentliche Bildklassifizierungsproblem, für das wir typischerweise viel weniger gelabelte Daten haben, indem wir die Einbettung verwenden, die durch die Autoencoder-Hilfsaufgabe erzeugt wurde. Dies wird wahrscheinlich die Modellperformance erhöhen, da das Modell jetzt nur noch die Gewichte für die Konfiguration mit geringerer Dimension zu lernen hat (d. h. weniger Gewichte lernen muss).

Zusätzlich zu Bild-Autoencodern haben sich neuere Arbeiten (*https://oreil.ly/ywFco*) darauf konzentriert, Deep-Learning-Techniken auf strukturierte Daten anzuwenden. TabNet ist ein tiefes neuronales Netz, das speziell dafür konzipiert wurde, aus tabellarischen Daten zu lernen, und sich im nicht überwachten Modus trainieren lässt. Indem das Modell in eine Encoder-Decoder-Struktur modifiziert wird, funktioniert TabNet als Autoencoder auf tabellarischen Daten. Dadurch kann das Modell Einbettungen aus strukturierten Daten über einen Feature-Transformer lernen.

Kontextbezogene Sprachmodelle

Gibt es eine Hilfslernaufgabe, die für Text funktioniert? Kontextbezogene Sprachmodelle wie *Word2Vec* und maskierte Sprachmodelle wie *Bidirectional Encoding Representations from Transformers* (BERT) ändern die Lernaufgabe in ein Problem, sodass es keinen Mangel an Labels gibt.

Word2Vec ist eine bekannte Methode für die Konstruktion einer Einbettung, die flache neuronale Netze verwendet und zwei Techniken – *Continuous Bag-of-Words* (CBOW) und ein Skip-Gram-Modell – kombiniert, die auf einen großen Textkorpus – wie zum Beispiel Wikipedia – angewendet werden. Während das Ziel beider Modelle darin besteht, den Kontext eines Worts zu lernen, indem Eingabewörter auf die Zielwörter mit einer dazwischenliegenden Einbettungsschicht abgebildet werden, wird ein Hilfsziel erreicht, das niedrigdimensionale Einbettungen lernt, die den Kontext der Wörter am besten erfassen. Die resultierenden Worteinbettungen, die über Word2Vec gelernt werden, erfassen die semantischen Beziehungen zwischen Wörtern, sodass die Vektordarstellungen im Einbettungsraum eine sinnvolle Distanz und Direktionalität beibehalten (siehe Abbildung 2-12).

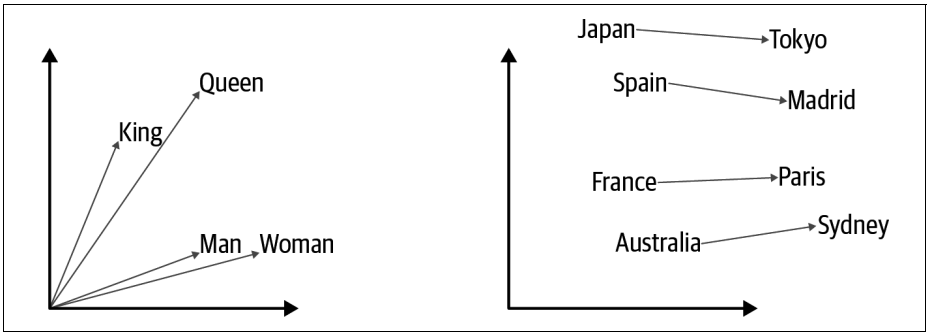

Abbildung 2-12: Worteinbettungen erfassen semantische Beziehungen.

BERT wird mit einem maskierten Sprachmodell und der Vorhersage des nächsten Satzes trainiert. Bei einem maskierten Sprachmodell werden Wörter zufällig aus dem Text ausgeblendet (maskiert), und das Modell errät die fehlenden Wörter. Die Vorhersage des nächsten Satzes ist eine Klassifizierungsaufgabe, bei der das Modell vorhersagt, ob zwei Sätze im Originaltext aufeinanderfolgen oder nicht. Somit ist jeder Textkorpus als gelabelter Datensatz geeignet. Ursprünglich wurde BERT auf dem gesamten Korpus der englischen Wikipedia sowie BooksCorpus trainiert. Trotz des Lernens auf diesen Hilfsaufgaben haben sich die gelernten Einbettungen von BERT oder Word2Vec als sehr leistungsfähig erwiesen, wenn sie auf andere nachgelagerte Trainingsaufgaben angewendet wurden. Die von Word2Vec gelernten Worteinbettungen sind unabhängig von dem Satz, in dem das Wort erscheint. Die BERT-Worteinbettungen sind jedoch kontextbezogen, d.h., der Einbettungsvektor ist je nach Kontext, in dem das Wort verwendet wird, unterschiedlich.

Eine vortrainierte Texteinbettung wie Word2Vec, NNLM, GLoVE oder BERT kann einem ML-Modell hinzugefügt werden, um Text-Features in Verbindung mit strukturierten Eingaben und anderen Einbettungen, die von unserem Kunden- und Videodatensatz gelernt wurden, zu verarbeiten (siehe Abbildung 2-13).

Letztlich lernen Einbettungen, Informationen zu bewahren, die für die vorgegebene Trainingsaufgabe relevant sind. So soll bei der Bildbeschriftung gelernt werden, wie sich der Kontext der Elemente eines Bilds zum Text verhält. In der Autoencoder-Architektur ist das Label gleich dem Feature, sodass die Dimensionsreduktion des Flaschenhalses versucht, alles zu lernen, und zwar ohne den spezifischen Kontext dessen, was wichtig ist.

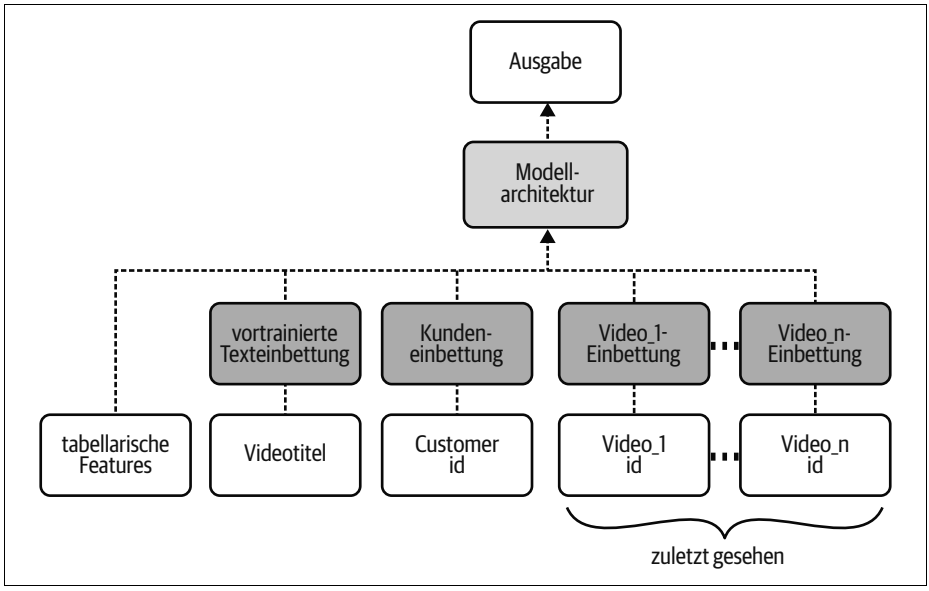

Abbildung 2-13: Um Text-Features zu verarbeiten, kann einem Modell eine vortrainierte Texteinbettung hinzugefügt werden.

Einbettungen in einem Data Warehouse

Maschinelles Lernen auf strukturierten Daten lässt sich am besten direkt in SQL auf einem Data Warehouse durchführen. Dadurch erübrigt es sich, die Daten aus dem Warehouse zu exportieren, und Probleme mit Datenschutz und Datensicherheit werden entschärft.

Viele Probleme erfordern jedoch eine Mischung aus strukturierten Daten und Text in natürlicher Sprache oder Bilddaten. In Data Warehouses wird Text in natürlicher Sprache (zum Beispiel Rezensionen) direkt in Spalten gespeichert, Bilder typischerweise als URLs zu Dateien in einem Cloud-Speicher-Bucket. In diesen Fällen vereinfacht es das spätere maschinelle Lernen, die Einbettungen der Textspalten

oder der Bilder zusätzlich als arrayartige Spalten zu speichern. Auf diese Weise lassen sich derartige unstrukturierte Daten leicht in ML-Modelle einbinden.

Um Texteinbettungen zu erstellen, können wir ein vortrainiertes Modell wie zum Beispiel Swivel von TensorFlow Hub nach BigQuery laden. Der vollständige Code ist auf GitHub (*https://github.com/GoogleCloudPlatform/ml-design-patterns/blob/master/02_data_representation/text_embeddings.ipynb*) zu finden:

```
CREATE OR REPLACE MODEL advdata.swivel_text_embed
OPTIONS(model_type='tensorflow', model_path='gs://BUCKET/swivel/*')
```

Verwenden Sie dann das Modell, um die Textspalte mit natürlicher Sprache in ein Einbettungsarray zu transformieren, und speichern Sie die Einbettungssuche in einer neuen Tabelle:

```
CREATE OR REPLACE TABLE advdata.comments_embedding AS
SELECT
    output_0 as comments_embedding,
    comments
FROM ML.PREDICT(MODEL advdata.swivel_text_embed,(
    SELECT comments, LOWER(comments) AS sentences
    FROM `bigquery-public-data.noaa_preliminary_severe_storms.wind_reports`
))
```

Es ist nun möglich, Verknüpfungen mit dieser Tabelle einzurichten, um die Texteinbettung für einen beliebigen Kommentar zu erhalten. Für Bildeinbettungen können wir auf ähnliche Weise Bild-URLs in Einbettungen transformieren und sie in das Data Warehouse laden.

Eine solche Vorberechnung von Features finden Sie als Beispiel im Abschnitt »Entwurfsmuster 26: Feature Store« auf Seite 325 (siehe Kapitel 6).

Entwurfsmuster 3: Feature Cross

Das Entwurfsmuster *Feature Cross* hilft Modellen, Beziehungen zwischen Eingaben schneller zu lernen, indem explizit jede Kombination von Eingabewerten zu einem eigenen Feature gemacht wird.

Problem

Sehen Sie sich den Datensatz in Abbildung 2-14 an. Die Aufgabe besteht darin, einen binären Klassifizierer zu erstellen, der die Plus- und Minuszeichen voneinander trennt.

Wenn man nur die `x_1`- und `x_2`-Koordinaten verwendet, ist es nicht möglich, eine lineare Grenze zu finden, die die Plus- und Minusklassen trennt.

Um also dieses Problem zu lösen, müssen wir das Modell komplexer machen, vielleicht mit zusätzlichen Schichten. Es gibt aber eine einfachere Lösung.