Pandas DataFrames und NumPy Arrays

Prof. Dr. Christian Bockermann

13. April 2022

In diesem “Lernbrief” geht es um ein paar Hintergrundinformationen zu Pandas und den NumPy Arrays, die wir zwar immer mal wieder benutzt haben, allerdings ohne im Detail zu schauen, was wirklich dabei passiert.

Hintergrund

Das Pandas Modul ist im Rahmen der Vorlesung Data Science das zentrale Modul für die Darstellung und Verarbeitung von Datensätzen. Dies liegt zum einen daran, dass Pandas mit den DataFrames eine einfache und anschauliche Abstraktion von Tabellen bereitstellt und zum anderen eine Vielzahl an Funktionen enthält, die den Umgang (z.B. Einlesen, Verarbeiten) mit den Daten erlauben.
Bisher habe ich mich bemüht, in der Vorlesung sämtliche technischen Details, die über die Tabellendarstellung (DataFrames, Series) hinausgehen, zu ignorieren, um die Komplexität der Vorlesung nicht unnötig zu erhöhen. Dies soll auch im weiteren Verlauf der Vorlesung so bleiben.

Im Rahmen der heutigen Vorlesung zum Thema *Clustering* kam jedoch erneut die Vorverarbeitung von DataFrames mit Hilfe der SkLearn-Funktionen zum Einsatz – genauer: die Normalisierung von Daten bzw. die Berechnung der *Bag of Words* Darstellung von Texten (Text-Clustering).

Was ist NumPy und was sind NumPy Arrays?

Bisher hat sich die Vorlesung mit den Python Modulen *Pandas* und *SciKit Learn* beschäftigt. Das sind zwei Module, die eine Reihe von Funktionalitäten bereitstellen um mit Tabellen in Form von DataFrames umzugehen.
Dazu gehört natürlich auch der DataFrame selbst, inklusive der Möglichkeit, die einzelnen Zeilen auszuwählen und auf bestimmte Spalten zuzugreifen. Abbildung 1 zeigt nochmal einen DataFrame mit Spaltenbezeichnungen.

	a1	a2	a3	a4
0	**4**	**1**	**2**	**9**
1	**5**	**1**	**3**	**6**
2	**3**	**8**	**7**	**4**

Abbildung 1: Ein DataFrame mit den Spalten a1, a2, a3 und a4 und dem normalen Zeilenindex 0 bis 2.

Ein derartiger DataFrame ist im Prinzip nichts anderes als eine Matrix Struktur, die in Python irgendwie intern verwaltet werden muss. Eine einfache Darstellung, die man sich gut vorstellen kann, ist die, bei der jede Zeile als Liste von Zahlen repräsentiert wird, z.B.:

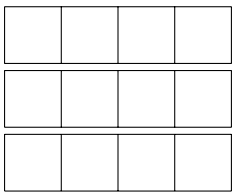
```
zeile0 = [ 4, 1, 2, 9]
zeile1 = [ 5, 1, 3, 6]
zeile2 = [ 3, 8, 7, 4]
```

Einen DataFrame könnte man dann als Liste dieser Zahlen abspeichern, d.h. man käme dann im Prinzip zu der Darstellung:

```
df_liste = [ zeile0, zeile1, zeile2 ]
```

Das einzige, was jetzt bei diesen Listen noch für den vollständigen DataFrame fehlen würde, wären die Namen der Spalten.

Im Prinzip sind wir mit der Listendarstellung (ohne die Spaltennamen) einem NumPy Array schon sehr nahe. Wir können uns ein NumPy Array als rechteckige Struktur von Zellen vorstellen, die sehr effizient im Hauptspeicher des Computers abgelegt wird:



Und hier sind wir eigentlich auch schon beim Kernpunkt angelegt: NumPy ist ein *sehr effizientes* Python-Modul, mit dem sich bestimmte Dinge viel schneller berechnen lassen. Eine Liste funktioniert zwar auch, aber ein NumPy Array ist eben noch viel schneller.

Warum dann Pandas und DataFrames?

Wenn NumPy Arrays viel schneller sind, dann stellt sich natürlich die Frage, warum wir nicht die ganze Zeit schon mit NumPy Arrays gearbeitet haben. Die Antwort ist, dass DataFrames eben ein wenig bequemer zu benutzen sind.
NumPy Arrays haben keine Spalten-Namen und es ist damit bei der Bearbeitung nicht wirklich leicht zu sehen, um welche Spalte es sich denn nun gerade handelt, die man bearbeitet. Bei Pandas DataFrames können wir ja auch einen Index für die Zeilen hinzufügen (z.B. per Datum) und so die Zeilen nach Datum filtern – Sie erinnern sich ja vielleicht noch an die Vorlesung *Zeitreihenanalyse*, bei der wir das gemacht haben. Auch das funktioniert so mit einem NumPy Array nicht.
Man könnte es sich ein wenig so vorstellen, dass ein DataFrame ein Element ist, dass ein NumPy Array enthält (in dem die Daten gespeichert sind) und dazu noch eine Liste der Spaltennamen und eine Liste für die Indexwerte (für z.B. den Fall eines Datum-Indexes) So in etwa ist Pandas auch programmiert.

Von NumPy Arrays zu DataFrames

Halten wir nochmal fest: NumPy Arrays und DataFrames haben im Prinzip genau die gleiche Struktur (NumPy Array können noch mehr, aber für unsere Vorlesung brauchen wir das nicht). Ein NumPy Array ist wie eine Liste von Zeilen.
Aus solch einer Liste können wir aber sehr leicht wieder einen DataFrame machen – das haben wir ja ganz oben mit den `zeileX`-Listen schon gemacht:

```
numpyArray = #...

# Liste mit Spaltennamen:
spalten = [ "a1", "a2", "a3", "a4" ]

# DataFrame aus NumPy Array erzeugen:
df = pd.DataFrame(numpyArray, columns=spalten)
```

NumPy, DataFrame – warum ist das jetzt relevant?

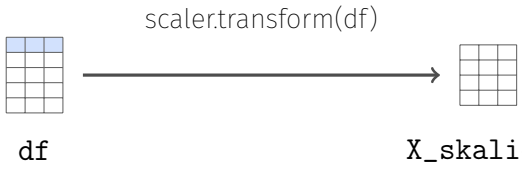
Wie eingangs geschrieben, ist dieses kurze Artikel als zustäzliche Hintegrundinformation zum DataScience Kurs zu verstehen. Die Problematik mit den Arrays DataFrame kommt z.B. auf, wenn man `sklearn` Funktionen für die Verarbeitung von DataFrames benutzt. Ein gutes Beispiel dafür ist die Skalierung, z.B. über den `MinMaxScaler`.
Für die Berechnung der normalisierten Werte sind die Namen der Spalten ja eigentlich nicht wichtig – der MinMaxScaler braucht ja nur die Spaltenwerte zu normalisieren und interessiert sich nicht für die Benennung der Spalten.

```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

# DataFrame lesen
df = pd.read_csv(".../datei.csv")

# Skalierer erzeugen und Parameter (min/max bestimmen)
scaler = MinMaxScaler()
scaler.fit(df)

# Daten skalieren
X_skaliert = scaler.transform(df)

# Die Spaltennamen vom alten DataFrame:
spalten = df.columns

# und einen neuen DataFrame erzeugen:
df_skaliert = pd.DataFrame(X_skaliert, columns=spalten)
```

Das obige Beispiel zeigt die Normalisierung als Python Code.

Was genau passiert dabei nun?

Betrachten wir den Fall mit der Skalierung nochmal genauer. Es wird zunächst der DataFrame in die Variable `df` eingelesen und danach die Daten im DataFrame mit dem MinMaxScaler normalisiert.



Der MinMaxScaler normalisiert die Daten und schreibt die normalisierten Daten in ein NumPy Array. Dabei gehen die Informationen darüber, welche Spalte welchen Namen hat, verloren. Bei der Normalisierung werden allerdings auch keine neuen Spalten erzeugt und die Spalten im NumPy Array werden in der gleichen Reihenfolge angelegt, wie im ursprünglichen DataFrame.
Da die Reihenfolge der Spalten im NumPy Array und dem DataFrame gleich ist, können wir aus dem NumPy Array wieder sehr leicht einen DataFrame machen. Dazu nehmen wir einfach die Liste der Spaltennamen aus dem ursprünglichen DataFrame und legen einen neuen DataFrame an:

```
# Die Spalten vom alten DataFrame
spalten = df.columns

# und einen neuen DataFrame erzeugen:
df_skaliert = pd.DataFrame(X_skaliert, columns=spalten)
```

NumPy Array mit zusätzlichen Spalten

Das Beispiel mit der Skalierung eignet sich, um die einfache Überführung von DataFrame zu Array zu DataFrame zu zeigen. Es wurde ja ein NumPy Array mit der gleichen Anzahl an Spalten wie der DataFrame erzeugt. Das ist nicht notwendigerweise immer so.
Bei der Verarbeitung von Dokumenten (vgl. Vorlesung *Clustering*) hatten wir über eine Vektor-Darstellung von Texten gesprochen. Dabei haben wir Texte als Vektoren dargestellt. Dafür muss ein Datensatz (DataFrame) mit den folgenden Texten:

text
Ich wünsche Ihnen frohe Ostern!
Ich wünsche Ihnen frohe Weihnachten!

in einen DataFrame überführt werden, der für jedes Wort eine Spalte enthält und für jeden Text eine Zeile. In der Spalte i in Zeile j stand dann die Häufigkeit von $wort_i$ für den Text aus Zeile j. Die folgende Abbildung 2 zeigt einen DataFrame, der den obigen DataFrame als Wort-Vektor-Darstellung enthält:

Ich	wünsche	Ihnen	frohe	Ostern	Weihnachten
1	1	1	1	1	0
1	1	1	1	0	1

Abbildung 2: Wort-Vektor Darstellung von zwei Sätzen.

Wort-Vektoren mit SciKitLearn

Für die Umwandlung von Texten in Vektoren gibt es bereits eine Reihe vordefinierter Funktionen, z.B. im SciKit Learn Modul. Eine Variante, die wir in der Vorlesung gesehen haben, ist der `CountVectorizer`, der für jedes Wort eine Spalte erzeugt, die angibt, wie oft ein Wort in einem Text vorkommt.

```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer

df = pd.read_csv(".../datei.csv")
CountVectorizer vec = CountVectorizer()

X = vec.fit_transform(df['text'])
```

Der `CountVectorizer` kann dabei sowohl für eine Spalte mit Texten als auch für eine beliebige Liste von Texten angewendet werden. Das Ergebnis des CountVectorizers ist eine spärlich besetzte NumPy Matrix, die für jedes Wort eine Spalte und zu jedem Text eine Zeile enthält.
Mit `X.toarray()` können wir daraus direkt ein NumPy Array erzeugen.

Die wichtige Frage ist nun:

Woher wissen wir jetzt, welche Spalte zu welchem Wort gehört?

Dazu bietet der CountVectorizer die Funktion `get_feature_names()` an, die uns eine Liste der Wort in der gleichen Reihenfolge zurückgibt, in der die Worthäufigkeiten in den Spalten stehen.
Damit können wir uns wieder sehr einfach den entsprechenden DataFrame basteln:

```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer

df = pd.read_csv("../datei.csv")
CountVectorizer vec = CountVectorizer()

X = vec.fit_transform(df['text'])
spalten = vec.get_feature_names()

df_words = pd.DataFrame(X.toarray(), spalten)
```

Der resultierende DataFrame enthält dann die Wörter als Spalten (Abbildung 3). Dabei fällt auf, dass die Reihenfolge nicht notwendigerweise der Reihenfolge aus den Texte entspringt und zudem alle Wörter klein geschrieben sind. Da die Groß-/Kleinschreibung bei der Datenanalyse von Texten in der Regel keine Rolle spielt, wird vom CountVectorizer hier automatisch alles in Kleinbuchstaben übersetzt.

	frohe	ich	ihnen	ostern	weihnachten	wünsche
0	**1**	**1**	**1**	**1**	**0**	**1**
1	**1**	**1**	**1**	**0**	**1**	**1**

Abbildung 3: DataFrame mit Texten in Wort-Vektor Darstellung.