Dr. Konrad Völkel
Heinrich-Heine-Universität Düsseldorf
Wintersemester 2024/25

Data Science 2
Anwesenheitsaufgaben für die erste Übung - Blatt ohne Abgabe

In diesen Aufgaben wollen wir uns an einige Techniken der linearen Algebra erinnern und üben, die Konstruktionen mit Numpy Arrays zu implementieren. Später wollen wir diese Techniken auf große Datenmengen ausweiten.

1 Lineare Algebra: Skalarprodukte

Wir erinnern uns: ein reeller Vektor v der Dimension $d \in \mathbb{N}$ ist eine geordnete Menge von d reellen Zahlen $(v_1, \dots, v_d)$, also *Komponenten* $v_i \in \mathbb{R}$, die wir gewöhnlich untereinander, also als Spalte aufschreiben. Da wir mit einem hochgestellten $\top$ das Transponieren, also das Vertauschen von Zeilen und Spalten bei einer Matrix bezeichnen, notieren wir auch

$$v = \begin{pmatrix} v_1 \\ \vdots \\ v_d \end{pmatrix} \in \mathbb{R}^d, \qquad v^\top = (v_1, \dots, v_d).$$

Zwei Vektoren $v, w \in \mathbb{R}^d$ sind *gleich*, also $v = w$, genau dann wenn für alle $i = 1, \dots, d$ gilt: $v_i = w_i$.

Definition. *Das* Hadamard-Produkt *von zwei Vektoren $v, w \in \mathbb{R}^d$ ist der Vektor $v \odot w \in \mathbb{R}^d$ mit Komponenten $(v \odot w)_i := v_i \cdot w_i$, also komponentenweise Multiplikation in den reellen Zahlen.*

Beispiel. *Für $v = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$ und $w = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$, also $v, w \in \mathbb{R}^2$ (in Dimension $d = 2$, also der Ebene), ist*

$$v \odot w = \begin{pmatrix} 2 \\ 0 \end{pmatrix} \odot \begin{pmatrix} 1 \\ 3 \end{pmatrix} = \begin{pmatrix} 2 \cdot 1 \\ 0 \cdot 3 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix} \in \mathbb{R}^2.$$

Definition. *Das* Skalarprodukt *von zwei Vektoren $v, w \in \mathbb{R}^d$ ist der Skalar $v \cdot w \in \mathbb{R}$ mit $v \cdot w := \sum_{i=1}^{d} (v \odot w)_i$.*

Beispiel. *Für $v = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$, $w = \begin{pmatrix} 1 \\ 3 \end{pmatrix} \in \mathbb{R}^2$ ist $v \cdot w = v_1 \cdot w_1 + v_2 \cdot w_2 = 2 \cdot 1 + 0 \cdot 3 = 2 + 0 = 2$.*

Beispiel. *Für $v = \begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}^2$ und die Standardbasisvektoren $e_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $e_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \in \mathbb{R}^2$ gilt $e_1 \cdot v = x$ und $e_2 \cdot v = y$.*

Satz. *Das Skalarprodukt $\mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R},\ (v, w) \mapsto v \cdot w$ ist* symmetrisch, bilinear und positiv definit*, d.h. $v \cdot w = w \cdot v$, für jeweils Skalare $\lambda, \mu \in \mathbb{R}$ und Vektoren $u, v, w \in \mathbb{R}^d$ gilt $(\lambda u + \mu v) \cdot w = \lambda(u \cdot w) + \mu(v \cdot w)$ und $v \cdot v \geq 0$ für alle $v \in \mathbb{R}^d$, außerdem ist $v \cdot v = 0$ genau dann, wenn v der Nullvektor mit $v_i = 0$ für alle i ist.*

Beweisaufgabe. Rechnen Sie die Eigenschaften nach (wenn es schwer fällt: für $d = 2$). □

Data Science 2
Anwesenheitsaufgaben für die erste Übung - Blatt ohne Abgabe

---

2 Lineare Algebra: Matrixmultiplikation

Definition. *Für zwei Matrizen $M \in \mathbb{R}^{m \times l}$ (d.h. m Zeilen und l Spalten, `shape (m,l)`) und $N \in \mathbb{R}^{l \times n}$ (d.h. l Zeilen und n Spalten, `shape (l,n)`) ist das Matrixprodukt definiert als*

$$MN \in \mathbb{R}^{m \times n}, \qquad (MN)_{ij} := M_i \cdot (N^\top)_j.$$

Der Eintrag von MN in Zeile i und Spalte j ist also das Skalaprodukt der i-ten Zeile von M mit der j-ten Spalte von N (das ist gleich der j-ten Zeile von $N^\top$).

In Numpy führt `np.matmul` bzw. der `@`-Operator die Matrixmultiplikation aus (nicht `*`).
Man kann natürlich auch direkt schreiben: $(MN)_{ij} = \sum_{k=1}^{l} M_{ik} N_{kj}$.

Aufgabe 1. *Wir können eine 'lazy' Matrixmultiplikation implementieren (und dabei Numpy's `np.dot` nutzen): eine Datenstruktur, die zwei Matrizen M, N abspeichert und erst dann einen Eintrag $(MN)_{ij}$ berechnet, wenn dieser angefordert wird, etwa so:*

```
x = LazyMatMul(M,N)
print(x.getEntry(2,3)) # Hier wird erst gerechnet
```

Bonusaufgabe a: überschreiben Sie geeignete Python-Methoden um die Syntax `x[2,3]` oder `x[2][3]` zu ermöglichen. Bonusaufgabe b: Implementieren Sie Caching, sodass ein zweiter Abruf des selben Eintrags keine erneute Rechnung auslöst. Bonusaufgabe c: Testen Sie Ihre Implementierung randomisiert gegen Numpy's eigene Matrixmultiplikation.

Aufgabe 2. *Wenn man symmetrische Matrizen multipliziert, also Matrizen $M \in \mathbb{R}^{m \times m}$ mit $M^\top = M$, benötigt man weniger Speicher für die Einträge der Matrix. Das gleiche gilt, wenn man obere Dreicksmatrizen multiplizieren möchte, also $M \in \mathbb{R}^{m \times m}$ mit $M_{ij} = 0$ für $j > i$.*

Implementieren Sie eine Datenstruktur `TriangMat` für obere Dreiecksmatrizen, die nur so viele Floats speichert, wie wirklich notwendig sind. Als Konstruktor bietet sich an, eine 'klassische' Matrix (als Numpy Array) anzunehmen, und eine Fehlermeldung zu generieren, wenn es sich nicht um eine obere Dreiecksmatrix handelt. Schreiben Sie eine `triangMatMul`-Methode, die zwei obere Dreiecksmatrizen multipliziert.

Bonusaufgabe: implementieren Sie analog `SymmMat` für symmetrische Matrizen.

Aufgabe 3. *. Gegeben eine Matrix $M \in \mathbb{R}^{m \times l}$ und Zerlegungen $m = m_1 + m_2$ sowie $l = l_1 + l_2$ mit $m_1, m_2, l_1, l_2 > 0$ können wir M als 'Blockmatrix' schreiben*

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}, \qquad A \in \mathbb{R}^{m_1 \times l_1},\ B \in \mathbb{R}^{m_1 \times l_2},\ C \in \mathbb{R}^{m_2 \times l_1},\ D \in \mathbb{R}^{m_2 \times l_2}.$$

Wenn wir ebenso $N \in \mathbb{R}^{l \times n}$ zerlegen (mit $n = n_1 + n_2$ für $n_1, n_2 > 0$), also

$$N = \begin{pmatrix} E & F \\ G & H \end{pmatrix}, \qquad E \in \mathbb{R}^{l_1 \times n_1},\ F \in \mathbb{R}^{l_1 \times n_2},\ G \in \mathbb{R}^{l_2 \times n_1},\ H \in \mathbb{R}^{l_2 \times n_2},$$

dann können wir das Matrixprodukt blockweise ausrechnen:

$$MN = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}.$$

Um MN auszurechnen, müssen M und N also nie gleichzeitig vollständig im Speicher liegen.

Überlegen Sie: wie lässt sich das verallgemeinern?

Data Science 2
Anwesenheitsaufgaben für die erste Übung - Blatt ohne Abgabe

---

3 Lineare Algebra: dünn besetzte Matrizen

Definition. *Eine Matrix heißt* dünn besetzt (sparse)*, wenn* die meisten Einträge 0 *sind.*

Das ist keine rigorose Definition! Ein häufiges Kriterium ist, dass die Anzahl der nicht-0-Einträge ungefähr der Anzahl Zeilen oder Spalten entsprechen sollte, um von *Sparsity* zu sprechen.

Tatsächlich liegt die Entscheidung bei uns, ob wir eine Matrix als *dünn besetzt* auffassen wollen, oder nicht. Der Unterschied liegt darin, wie wir die Einträge der Matrix abspeichern. Eine 'volle' Matrix erfordert das Speichern aller Einträge. Eine dünn besetzte Matrix erfordert nur das Speichern der nicht-0-Einträge.

In Python speichern wir volle Matrizen als Listen von Listen oder (besser!) als Numpy Arrays. Dünn besetzte Matrizen können wir z.B. als Dictionaries (=Hashmap) abspeichern:

```
m_sparse = {(0,1) : 5, (1,1) : 4}
```

könnte z.B. die Matrix $M = \begin{pmatrix} 0 & 5 \\ 0 & 4 \end{pmatrix}$ sein. Tatsächlich könnte M auch eine 3×2-Matrix oder eine 10×10-Matrix sein (mit vielen Nullen). Manchmal möchte man daher Zeilen- und Spaltenanzahl auch speichern.

Aufgabe 4. *Implementieren Sie Matrixmultiplikation von zwei dünn besetzten Matrizen.*

Tipp: Implementieren Sie zunächst das Berechnen eines einzelnen Eintrags der Produktmatrix. Überlegen Sie sich als erstes, wie es geht, wenn die beiden Matrizen ein Zeilenvektor und ein Spaltenvektor sind (sodass das Matrixprodukt eine 1×1-Matrix ist, die genau das Skalarprodukt als einzigen Eintrag enthält).

4 Simples Map-Reduce-Framework: octo.py

Aufgabe 5. *Schauen Sie sich den code* `octo.py` *an. Was können Sie grob über die Struktur sagen? Wie verwendet man das Modul, was passiert dann?*