# Übungsblatt 4

## Abgabe:

bis **14. Januar 2021** um **23:59** via **ecampus**

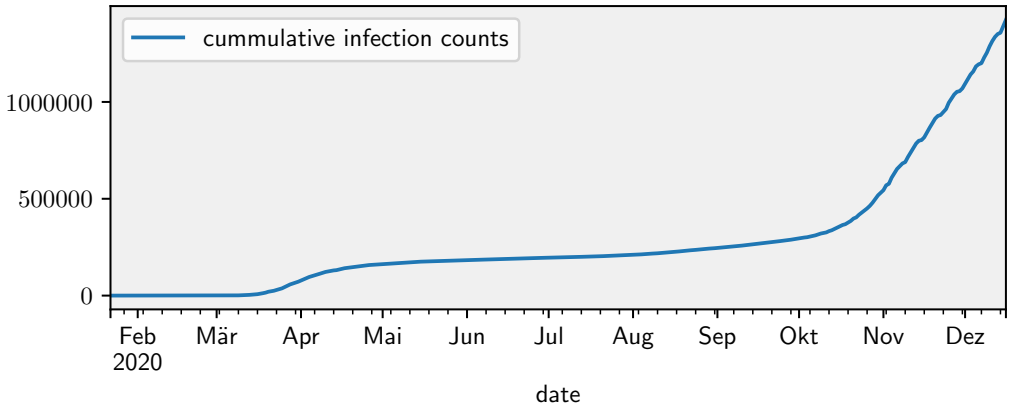
**Aufgabe 4.1:** Laden Sie von der der Webseite

data.humdata.org/dataset/novel-coronavirus-2019-ncov-cases

die neueste Version der Datei

`time_series_covid19_confirmed_global.csv`

herunter. Die darin enthaltenen Daten zu den kumulativen COVID-19 Fallzahlen in Deutschland sahen Mitte Dezember 2020 so aus:

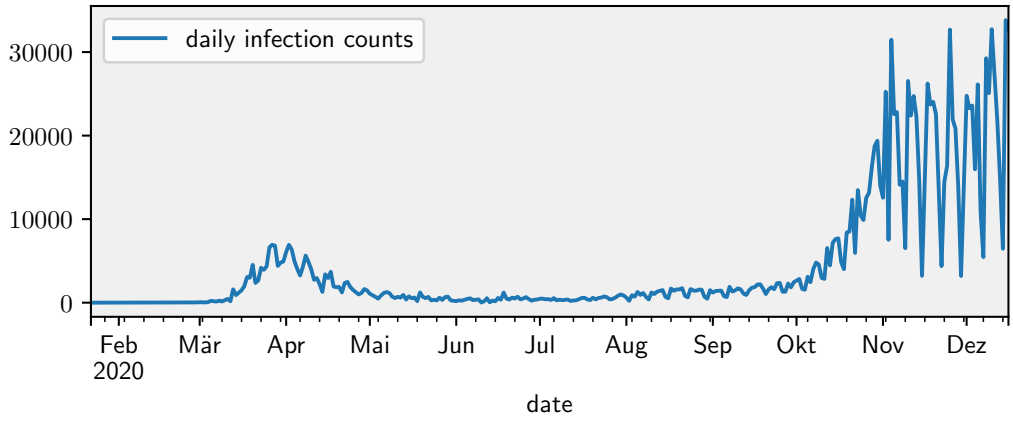



In Aufgabe 2.3 haben wir die bis damals vefügbaren Daten schon einmal analysiert. Dort haben wir mit Zeitserien $I_t$ gearbeitet, wobei $t \in \mathbb{N}$ einen Zeitpunkt (hier einen Tag) indiziert und $I_t \in \mathbb{N}$ die Gesamtzahl der bis zu diesem Zeitpunkt beobachteten Infektionen bezeichnet.

Im Folgenden werden wir stattdessen mit den täglich neu hinzukommenden Infektionszahlen $i_t$ arbeiten. Diese lassen sich wie folgt berechnen

$$i_t = I_t - I_{t-1}$$

Plotten Sie die Zeitserie $i_t$. Ihr Ergebnis sollte etwa so aussehen:



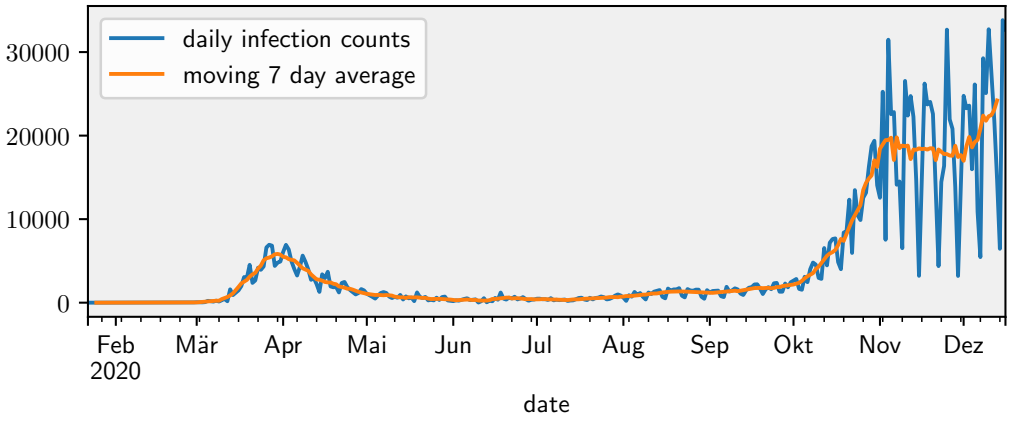

Gegen Ende des Beobachtungszeitraum erkennen wir starke Schwankungen in den Werten $i_t$. Wenn Sie die Daten genauer analysieren, werden Sie feststellen, dass es sich dabei um "Wochenendeffekte" handelt: An Wochenende erkranken nicht unbedingt weniger Menschen an COVID-19, es werden nur weniger neue Fälle erfasst. Das heißt, wir haben es hier mit einem Artefakt im Messprozess zu tun.

Da deartige Artefakte in der Zeitreihenanalyse häufig vorkommen, ist es üblich, die gegebenen Daten einer Vorverarbeitung zu unterziehen, um sie besser / robuster analysieren zu können. Um das Phänomen starker lokaler Schwankungen zu adressieren, wird z.B. oft der moving average einer Zeitserie berechnet. In dieser Aufgabe, arbeiten wir insbesondere mit dem *centered moving average*. Für eine Zeitdauer von $n = 7$ Tagen ist dieser wie folgt definiert

$$a_t = \frac{1}{7} \sum_{\tau=-3}^{3} i_{t+\tau}$$

Berechnen Sie die Zeitserie $a_t$. Überlegen Sie dabei, wie mit Zeitpunkten an den Rändern des Beobachtungszeitraums zu verfahren ist.

Plotten Sie die Zeitserien $i_t$ und $a_t$. Ihr Ergebnis sollte etwa so aussehen:




In Aufgabe 2.3 hatten wir die Gompertz Funktion

$$F\big(t \mid N, r, t_0\big) = N \cdot \exp\Big(-\exp\big(-r \cdot (t - t_0)\big)\Big)$$

an die kumulierten Fallzahlen in $I_t$ angepasst. Da wir hier nun tägliche Fallzahlen $i_t$ bzw. deren *moving average* $a_t$ betrachten, arbeiten wir im Folgenden mit der Ableitung der Gompertz Funktion

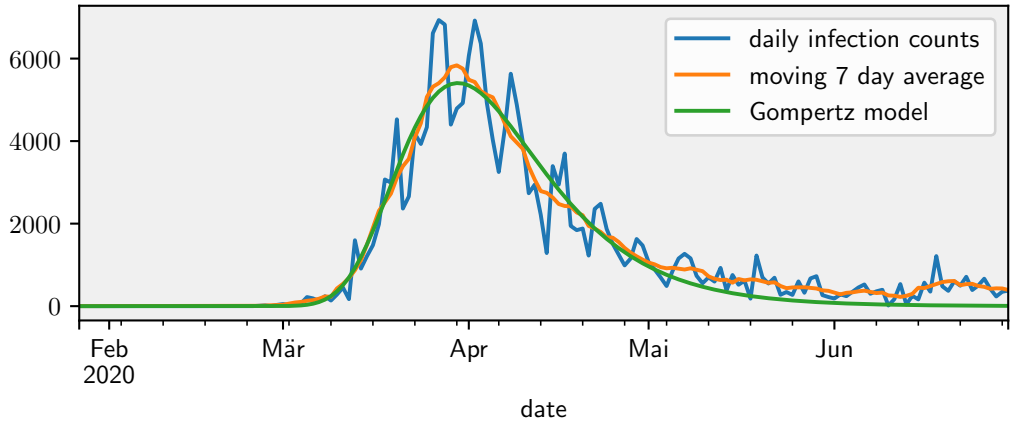
$$\begin{aligned} f\big(t \mid N, r, t_0\big) &= \frac{d}{dt} F\big(t \mid N, r, t_0\big) \\ &= N \cdot r \cdot \exp\Big(-r \cdot (t - t_0) - \exp\big(-r \cdot (t - t_0)\big)\Big) \end{aligned}$$

Betrachten Sie nun den Zeitraum vom 27.1.2020 bis zum 31.5.2020, um $f(t)$ an $a_t$ anzugleichen.

Wenn Sie mit der *scipy* Funktion `curve_fit` arbeiten, können Sie die folgenden initiale Parameter Schätzungen an die Funktion übergeben:

```
p0 = (100000, 0.09, 60)
```

Plotten Sie Ihr angeglichenes Modell für den Zeitraum vom 27.1.2020 bis zum 30.6.2020. Ihr Ergebnis sollte etwa so aussehen:

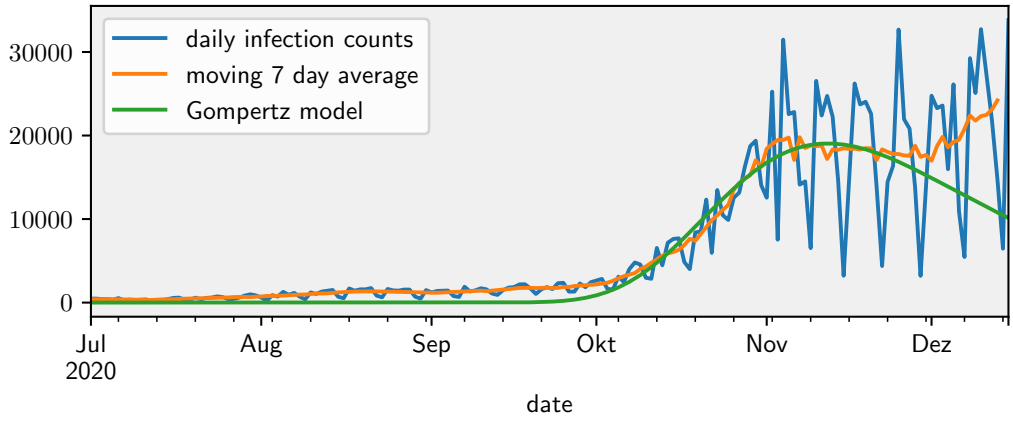



Betrachten Sie als nächstes die Periode vom 1.7.2020 bis zum 30.11.2020, um $f(t)$ an $a_t$ anzugleichen.

Gute initiale Parameter Schätzungen sind hier:

```
p0 = (1300000, 0.08, 300)
```

Plotten Sie Ihr angeglichenes Modell für den Zeitraum vom 2.7.2020 bis zum 15.12.2020. Ihr Ergebnis sollte etwa so aussehen:



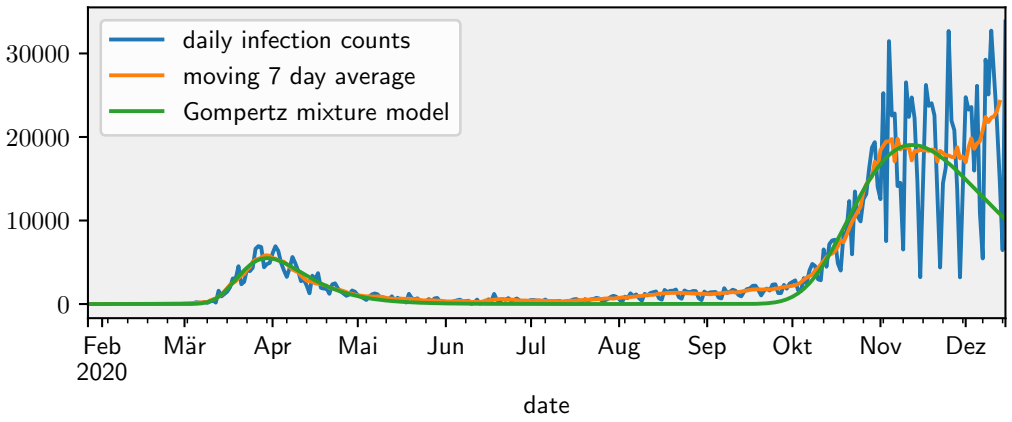

Diese Ergebnisse legen nahe, dass die Dynamiken der ersten und zweiten

Welle der COVID-19 Infektionen in Deutschland gut durch Gompertz Funktionen beschrieben werden können ... Das heißt aber, dass die Infektionsdynamik zwischen Ende Januar und Anfang Dezember durch eine Überlagerung zweier Gompertz Funktionen beschrieben werden kann. Betrachten Sie nun also folgendes Mischungsmodell

$$f_M\big(t \mid N_1, r_1, t_{0_1}, N_2, r_2, t_{0_2}\big) = f\big(t \mid N_1, r_1, t_{0_1}\big) + f\big(t \mid N_2, r_2, t_{0_2}\big)$$

und gleichen es an die Daten $a_t$ im Zeitraum vom 27.1.2020 bis zum 30.11.2020 an.

Plotten Sie Ihr angeglichenes Modell für den Zeitraum vom 27.1.2020 bis zum 15.12.2020. Ihr Ergebnis sollte etwa so aussehen:




Überlegen Sie, warum es bei kumulierten Fallzahlen $I_t$ von mehr als $1000000$ sinnvoll ist, mit nicht kumulierten Daten $i_t$ au arbeiten.

**Weitere Aufgaben für die, die mehr wollen:** Google Trends bietet (normalisierte) Statistiken darüber, wie häufig Suchbegriffe wann und wo in die Google Suchmaschine eingegeben wurden. Unter

https://trends.google.com/trends/explore?geo=DE&q=corona

finden Sie eine Zeitreihe, die zeigt, wie häufig in Deutschland in den letzten 12 Monaten nach dem Wort “Corona” gesucht wurde. Laden Sie diese Daten herunter und versuchen Sie, die Funktion $f_M(t)$ daran anzugleichen.

**Aufgabe 4.2:** Die Datei

`springfield.txt`

enthält eine Reihe von Namen, die Sie wahrscheinlich kennen. Um diese Namen als Strings in eine Liste `names` einzulesen, können Sie folgendermaßen vorgehen

```
with open('springfield.txt', 'r') as f:
    lines = f.readlines()

names = [l.rstrip('\n') for l in lines]
```

In dieser Aufgabe betrachten wir eine Möglichkeit, Textdaten wie diese in ein Format zu überführen, das für viele *machine learning* Verfahren einfacher zu handhaben ist, als Strings. Insbesondere betrachten wir eine Möglichkeit, Strings in einen euklidischen Vektorraum $\mathbb{R}^m$ einzubetten.

Für die dazu nötigen Schritte bietet es sich an, mit folgenden *imports* zu arbeiten:

```
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt

from collections import Counter
```

In der Vorlesung haben wir bereits die Idee der $n$-Gramme eines Worts oder Strings `w` kennengelernt. *python* Expert*innen wissen, dass diese etwa so berechnet werden können:

```
def str_n_grams(w, n):
    return map(''.join, zip(*[w[i:] for i in range(n)]))

print (list(str_n_grams('homer simpson', 2)))
print (list(str_n_grams('homer simpson', 3)))
```

Wenn Sie dieses kurze *snippet* ausführen, sollten Sie als Ergebnis diese beiden Listen sehen:

```
['ho', 'om', 'me', 'er', 'r ', ' s', 'si', 'im', 'mp', 'ps', 'so', 'on']
['hom', 'ome', 'mer', 'er ', 'r s', ' si', 'sim', 'imp', 'mps', 'pso', 'son']
```

Im Folgenden nutzen wir $n$-Gramme, um Strings in $\mathbb{R}^m$ einzubetten. Ohne in mathematische Details zu gehen, besteht unser Ansatz darin, das so genannte $n$-Gramm Spektrum eines Strings zu berechnen und aus den $n$-Gramm Spektren aller $N$ gegebenen Strings eine zentrierte Kern-Matrix $\boldsymbol{K} \in \mathbb{R}^{N \times N}$ zu berechnen. Dazu nutzen wir die folgenden Funktionen ...

```
def str_spectrum(w, n):
    return Counter(str_n_grams(w, n))


def str_spectrum_KernelMatrix(spectra):
    N = len(spectra)
    matK = np.zeros((N,N))

    for i, s1 in enumerate(spectra):
        for j, s2 in enumerate(spectra):
            intersection = s1 & s2
            matK[i,j] = sum(intersection.values())

    rsum = np.sum(matK,axis=1).reshape(1,N)
    csum = np.sum(matK,axis=0).reshape(N,1)
    tsum = np.sum(matK)

    return matK - 1./N * rsum - 1./N * csum + 1./N**2 * tsum
```

Wenn wir z.B. tri-Gramme ($n = 3$) betrachten, können Sie $\boldsymbol{K}$ wie folgt berechnen:

```
n = 3

spectra = [str_spectrum(name, n) for name in names]

matK = str_spectrum_KernelMatrix(spectra)
```

Sobald Matrix $\boldsymbol{K}$ verfügbar ist, berechnen wir ihre Eigenwerte $\lambda_i \in \mathbb{R}$ und Eigenvektoren $\boldsymbol{u}_i \in \mathbb{R}^N$. D.h. wir nutzen erneut die Idee der principal component analysis. Da $\boldsymbol{K}$ aber eine symmetrische Matrix ist, gehen wir diesmal wie folgt vor:

```
vecL, matU = la.eigh(matK)
```

Die *numpy* Funktion `eigh` liefert Eigenwerte und Eigenvektoren aufsteigend sortiert zurück. Die Eigenvektoren $\boldsymbol{u}_1$ und $\boldsymbol{u}_2$ mit den beiden größten Eigenwerten $\lambda_1$ und $\lambda_2$ finden sich also in der letzten und vorletzten Spalte von Matrix $\boldsymbol{U}$. Um daraus eine Projektionsmatrix $\boldsymbol{P} \in \mathbb{R}^{N\times 2}$ zu konstruieren, um eine Einbettung in $\mathbb{R}^2$ zu realisieren, können Sie so vorgehen:

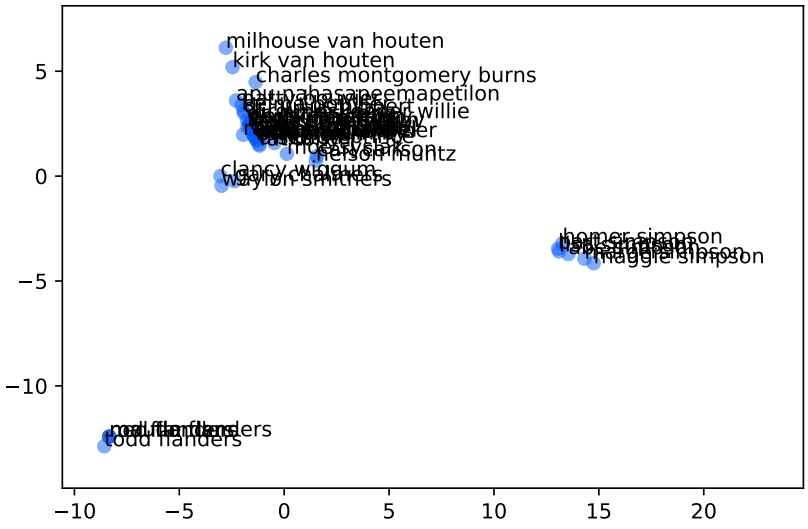
```
idxs = (-1,-2)
matP = matU[:,idxs]
```

Damit können Sie schließlich folgendes berechnen:

```
matX = np.dot(matP.T, matK)
```

Die Spaltenvektoren $\boldsymbol{x}_j$ der resultierende Matrix $\boldsymbol{X} \in \mathbb{R}^{2\times N}$ sind 2D Punkte, die die entsprechenden Strings $s_j$ in $\mathbb{R}^2$ repräsentieren.

Wenn Sie diese Punkte plotten und (wenn Sie fancy sein wollen) mit den entsprechenden Strings annotieren, sollte Ihr Ergebnis etwa so aussehen:




Experimentieren Sie mit weiteren Werten für $n$, z.B. $n \in \{2, 4, 5\}$. Experimentieren Sie auch mit anderen Projektionen. D.h. schauen Sie, was passiert, wenn etwa

```
matP = matU[:,(-1,-3)]
matP = matU[:,(-2,-3)]
matP = matU[:,( 0, 1)]
```

Um die gegebenen Strings in einen höherdimensionalen Raum einzubetten, der z.B. von den $10$ größten Eingenvektoren von $\boldsymbol{K}$ aufgespannt wird, können Sie folgende Projektionsmatrix betrachten

```
matP = matU[:,-10:]
```

Matrix $\boldsymbol{X} = \boldsymbol{P}^\intercal \boldsymbol{K}$ ist dann eine $10 \times N$ Matrix. Die entsprechenden Einbettungs-Vektoren $\boldsymbol{x}_j$ der Strings $s_j$ lassen sich dann zwar nicht mehr so gut visualisieren, können aber problemlos in weiteren Analysen betrachtet werden.

**Aufgabe 4.3:** Jetzt, da Sie die $N = 43$ Namen bzw. Strings in der Datei `springfield.txt` in $\mathbb{R}^m$ einbetten können (wobei $m \leq N$), ist es möglich, in diesem Vektorraum nächste Nachbarn zu suchen, um ähnlich Strings zu bestimmen ...

Sei also `matX` ein *numpy* array wie in Aufgabe 4.2 berechnet ($n = 3$ und $\boldsymbol{P} = [\boldsymbol{u}_1 \boldsymbol{u}_2]$). Über den Datenpunkten (Spalten) dieses arrays können Sie einen $k$D-Baum konstruieren:

```
import scipy.spatial as spt

T = spt.KDTree(matX.T)
```

Wenn Sie diesen Baum nutzen wollen, um die $5$ nächsten Nachbarn des Strings `'bart simpson'` zu ermitteln, können Sie wie folgt vorgehen:

```
idxBart = names.index('bart simpson')
vecBart = matX[:,idxBart]

dist, idxs = T.query(vecBart, k=5)

for i in idxs:
    print (names[i])
```

Ermitteln Sie nun Sie die $5$ nächsten Nachbarn des Strings `'moe syslak'`.

Ermitteln Sie alle Nachbarn innerhalb eines Radius $r = 1$ um die Strings `'bart simpson'` und `'moe syslak'`.

Betrachten Sie nun Einbettungen in höherdimensionalen Räumen, d.h. nutzen Sie

$$\boldsymbol{P} = \left[\boldsymbol{u}_1 \boldsymbol{u}_2 \cdots \boldsymbol{u}_m\right]$$

für $m \in \{3, 10, 20, 40\}$. Berechnen Sie jeweils erneut die $5$ nächsten Nachbarn der- sowie alle Nachbarn innerhalb des Radius $r = 1$ um die beiden Strings `'bart simpson'` und `'moe syslak'`.

Was beobachten Sie? Macht die Dimension des Einbettungsraumes einen Unterschied für die Ergebnisse dieser Berechnungen oder nicht?

**Aufgabe 4.4:** Sei `matX` wieder ein *numpy* array wie es in Aufgabe 4.2 berechnet wurde ($n = 3$ und $\boldsymbol{P} = [\boldsymbol{u}_1 \boldsymbol{u}_2]$).

Da die Datenpunkte (Spalten) dieses arrays euklidische Vektoren sind, können Sie $k$-means clustering nutzen, um Cluster ähnlicher Strings zu bestimmen. Für $k = 3$ können Sie z.B. so vorgehen:

```
import scipy.cluster.vq as vq

k = 3

matM, labels = vq.kmeans2(matX.T, k)

for i in range(k):
    print ([name for j, name in enumerate(names) if labels[j] == i])
```

Ermitteln Sie Cluster für $k \in \{5, 10, 20\}$.

Betrachten Sie erneut Einbettungen in höherdimensionalen Räumen, d.h. nutzen Sie

$$\boldsymbol{P} = \left[\boldsymbol{u}_1 \boldsymbol{u}_2 \cdots \boldsymbol{u}_m\right]$$

für $m \in \{3, 10, 20, 40\}$ und berechnen Sie dann Cluster für $k \in \{3, 5, 10, 20\}$.

Was beobachten Sie? Macht die Dimension des Einbettungsraumes einen Unterschied für das Ergebnis dieser Berechnungen oder nicht?

Betrachten Sie insbesondere den Fall wo $m = 40, k = 3$ und clustern sie mehrfach (besser gesagt sehr häufig). Sind Ihre Ergebnisse immer gleich oder unterscheiden sie sich? Was passiert, wenn Sie *kmeans2* mit `minit='random'`, `minit='points'`, oder `minit='++'` aufrufen?

Permutieren Sie nun die Spalten von `matX` zufällig beispielsweise so:

```
idxs = np.arange(len(names))
np.random.shuffle(idxs)
matX = matX[:,idxs]
```

Vergessen Sie nicht, auch die Liste `names` entsprechend neu zu sortieren

```
names = [names[i] for i in idxs]
```

Wiederholen Sie nun Ihre Experimente. Ergibt sich jetzt ein anderes Bild oder nicht?

**Aufgabe 4.5:** Implementieren Sie den $k$-means clustering Algorithmus von MacQueen und wiederholen Sie ihre Experimente aus Aufgabe 4.3. Betrachten Sie dabei sowohl das originale array `matX` / die originale Liste `names` als auch deren geshuffelte Versionen.

Was beobachten Sie, wenn mit dem Algorithmus von MacQueen clustern? Deutlich andere Ergebnisse als bei *kmeans2* oder nicht?