Übungsblatt 2

Abgabe:

bis **3. Dezember 2020** um **23:59** via **ecampus**

Aufgabe 2.1: Die Datei `whData.dat` enthält Daten zu Körpergröße und Gewicht, die in einer Umfrage unter 39 Studierenden eines MSc Kurses erhoben wurden. Im Folgenden betrachten wir die Körpergröße als die unabhängige Variable x und das Körpergewicht als abhängige Variable y.

Laden Sie die Daten zur Körpergröße daher in ein *numpy* array `x` und die Daten zum Körpergewicht in ein *numpy* array `y`.

Erinnern Sie sich, dass die Daten zwei Ausreißer enthalten und entfernen Sie diese.

Passen Sie an die so bereinigten Daten (x_j, y_j) zwei Modelle an und zwar

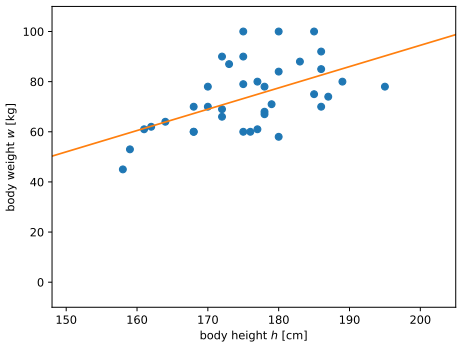
1. ein lineares Modell (d.h. ein Polynom ersten Grades)

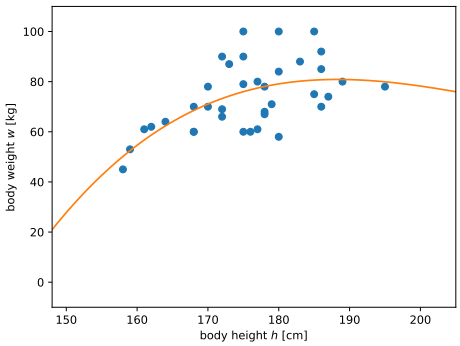
$$y_j = w_0 + w_1 \cdot x_j + \epsilon_j$$

2. ein kubisches Modell (d.h. ein Polynom dritten Grades)

$$y_j = w_0 + w_1 \cdot x_j + w_2 \cdot x_j^2 + w_3 \cdot x_j^3 + \epsilon_j$$

Geben Sie die resultierenden Koeffizientenvektoren $\boldsymbol{w}$ aus und plotten Sie die bereinigten Daten zusammen mit den angepassten Modellen. Ihre Plots sollten etwa so aussehen:





Aufgabe 2.2: Verfahren Sie wie in Aufgabe 2.1 *mit dem Unterschied, dass Sie die Daten diesmal nicht bereinigen*.

D.h. betrachten Sie die Daten einschließlich der Ausreißer und passen Sie ein lineares und ein kubisches Modell an diese *Rohdaten* an. Geben Sie die resultierenden Koeffizientenvektoren $\boldsymbol{w}$ aus und plotten Sie die Rohdaten zusammen mit den angepassten Modellen. Was beobachten Sie?

Aufgabe 2.3: Auf der Webseite

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

finden Sie regelmäßig aktualisierte Daten zum weltweiten COVID-19 Infektionsgeschehen. Laden Sie von dort das File

`time_series_covid19_confirmed_global.csv`

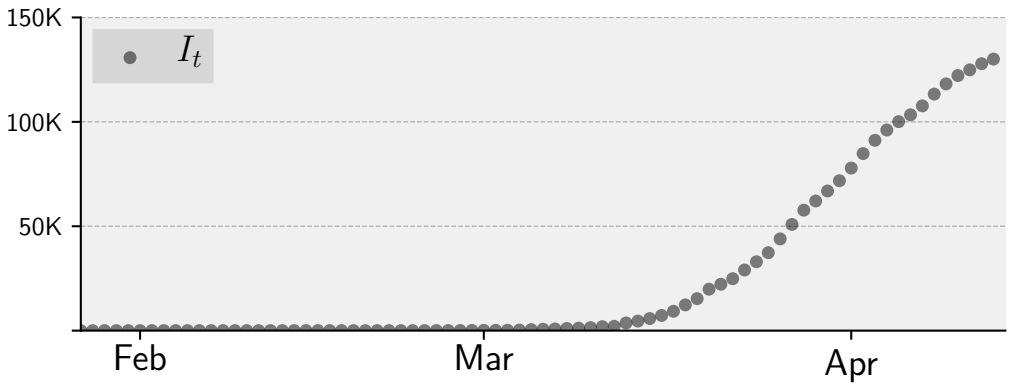
herunter und extrahieren Sie die darin enthaltenen Daten für Deutschland (insbesondere Zeitpunkte und Fallzahlen).

Betrachten Sie die Daten für den Zeitraum vom 27.1.2020 —dem Tag, an dem der erste COVID-19 Fall in Deutschland auftrat— bis (einschließlich) zum 13.4.2020. Im Folgenden bezeichnen wir die Daten dieses Zeitraums als *Trainingsdaten*.

Diese Trainingsdaten stellen eine *diskrete Zeitserie* (t, I_t) dar, 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 COVID-19 Infektionen bezeichnet. Im Folgden nutzen wir die Konvention

$$t = 0 \Leftrightarrow \text{27.1.2020}, \quad t = 1 \Leftrightarrow \text{28.1.2020}, \quad \ldots$$

Wenn Sie Trainingsdaten mit *matplotlib* plotten, könnte das Ergebnis etwa so aussehen (keine Sorge, Ihr Plot muss nicht ganz so “fancy” sein . . .)



Trainingsdaten (t, I_t)

Ihre Aufgabe ist nun, ein (kontinuierliches) mathematisches Modell an die (diskreten) Trainingsdaten anzugleichen, nämlich die Gompertz Funktion

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

die wir hier mit Parametern betrachten, die auf Englisch folgendermaßen bezeichnet werden:

$N \equiv$ carrying capacity
$r \equiv$ growth rate
$t_0 \equiv$ time delay

Um diese Angleichung vorzunehmen, können wir erneut die Methode der kleinsten Quadrate heranziehen, um die Modellparameter so einszustellen, dass der Fehler

$$E = \sum_t \Big(I\big(t \mid N, r, t_0\big) - I_t \Big)^2$$

minimal wird.

Im Gegensatz zu den einfachen (linearen) Modellen, die in der Vorlesung betrachtet wurden, gibt es für dieses nichlineare Optimierungsproblem aber keine geschlossenen Lösungen für die optimalen Parameter. Daher minmiert man nichtlineare Fehlerfunktionen z.B. mit Gradientenabstiegsverfahren.

Wenn Sie solche Methoden selbst implementieren wollen, ist das meißt aufwändig. Glücklicherweise hilft uns aber das *scipy* Modul *optimize* weiter, das entsprechende Funktionalitäten bereit hält. Hier können Sie etwa wie folgt vorgehen:

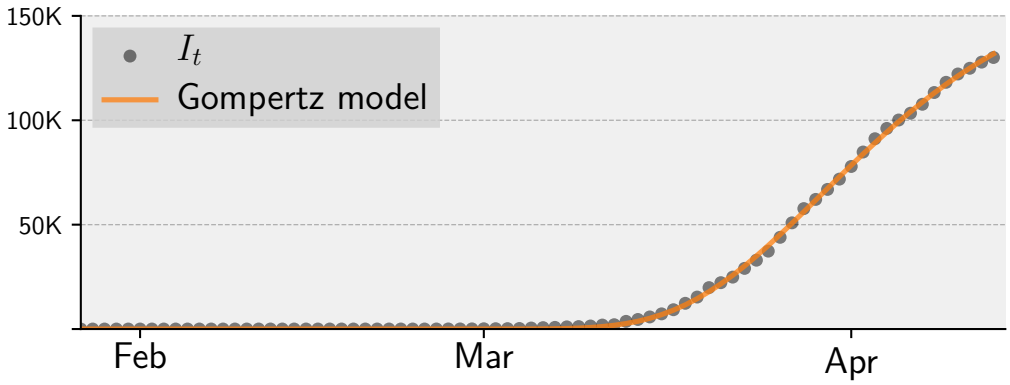
```
import numpy as np
import scipy.optimize as opt


def gompertzFct(x, N, r, t0):
    return N * np.exp(-np.exp(-r * (x-t0)))


t_trn = np.array( ... )
I_trn = np.array( ... )

guess = (100000., .1, 50.)

parameters, covariance = opt.curve_fit(gompertzFct, t_trn, I_trn, p0=guess)
```

Nachdem Sie diese Berechnungen durchgeführt haben, enthält das array `parameters` optimierte Werte für N, r und t_0.

Diese können Sie z.B. wie folgt nutzen, um $I(t)$ über dem Trainingszeitraum zu berechnen

```
I = gompertzFct(t_trn, *parameters)
```

Wenn Sie dann die Trainingsdaten zusammen mit dem angeglichen Modell plotten, müsste Ihr Ergebnis etwa so aussehen

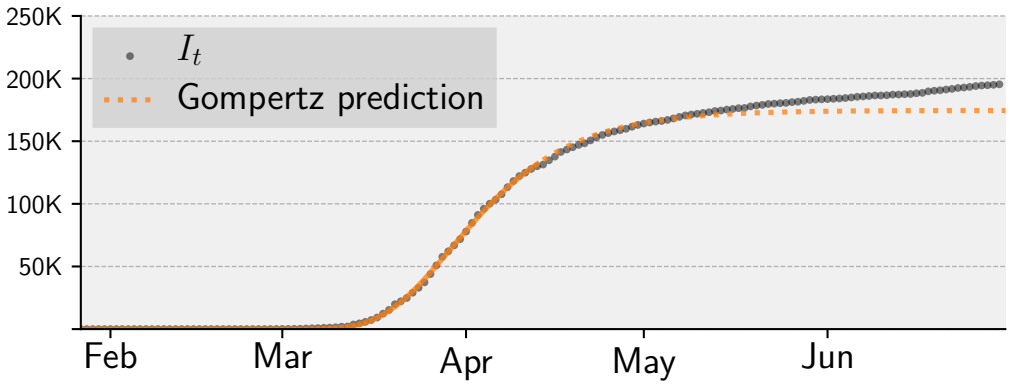


Trainingsdaten und angeglichenes Gompertz Modell

Unteraufgabe: Probieren Sie andere *initiale* Parameterwerte in `guess` aus, und beobachten Sie, wie sich das auf das Ergebnis auswirkt.

Sobald wir ein Modell an eine Zeitserie angelichen haben, können wir es nutzen um künftige Entwicklungen zu prädizieren. Nutzen Sie daher das Gompertz Modell, um "Vorhersagen" der Infektionszahlen für den Zeitraum vom 14.4.2020 bis zum 30.6.2020 zu berechnen.

Da wir die tasächlichen Zahlen für diese Zeit kennen, können Sie Ihre Modellvorhersage und die Realität visuell vergleichen. Plotten Sie daher ihre Vorhersagen zusammen mit den tatsächlichen Daten für den Zeitraum vom 27.1.2020 bis zum 30.6.2020. Ihr Ergebnis sollte etwa so aussehen



Trainings- und Testdaten und Modellvorhersage

Weitere Aufgaben für die, die mehr wollen: Experimentieren Sie mit kürzen oder längeren Zeiträumen für das Training. D.h. betrachten Sie z.B. Trainingsdaten, die die Zeit vom 27.1.2020 bis zum 31.3.2020 oder vom 27.1.2020 bis zum 31.5.2020 abdecken.

Die offensichtliche Frage ist natürlich: “Was ist mit der seit dem Spätsommer laufenden Welle?”

Versuchen Sie Ihr Glück! Verwenden Sie aber nicht zuviel Zeit darauf, zu versuchen, ein Gompertz Modell an die späteren COVID019 Daten (z.B. ab September) anzugleichen . . . das geht zwar, ist aber nicht so einfach wie im Falle deranfänglichen Daten. D.h. hier braucht es mehr Erfahrung im Umgang mit Daten, und wir kommen später auf diesen Punkt zurück.

Auch wenn Sie im Team arbeiten, gilt für die nächste Aufgabe: Jede*r Teilnehmer*in an den Übungen muss Aufgabe 2.4 selbst machen!

Die Aufgabe ist nicht schwer! Sie müssen nur die beigefügten code snippets laufen lassen, beobachten, was passiert, und überlegen, was diese Beobachtungen für die Data Science Praxis bedeuten!

Zur Berabeitung dieser Aufgabe brauchen Sie die folgenden imports:

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

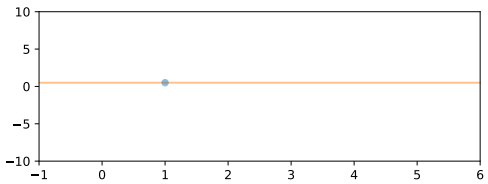
Aufgabe 2.4: Es ist leicht, zu beweisen (vielleicht versuchen Sie es selbst), dass für n Datenpunkte

$$\left\{\left(x_j, y_j\right)\right\}_{j=1}^{n}$$

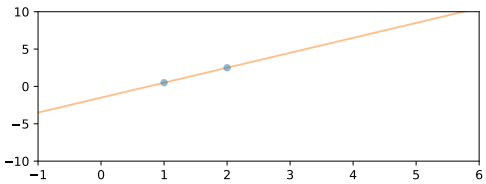
immer ein Polynom p vom Grad $n-1$ existiert, so dass

$$y_j = p\left(x_j\right) = w_0 + w_1 x_j + w_2 x_j^2 + \ldots + w_{n-1} x_j^{n-1}.$$

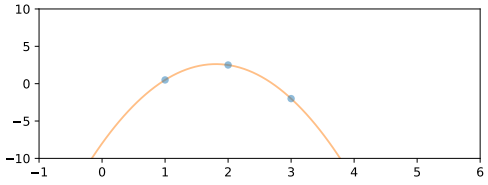
Mit anderen Worten: Für n Datenpunkte $\left(x_j, y_j\right) \in \mathbb{R}^2$ lässt sich immer ein Polynom vom Grad $n-1$ finden, dessen Graph durch die gegebenen Punkte geht. Hier sind einige Beispiele:



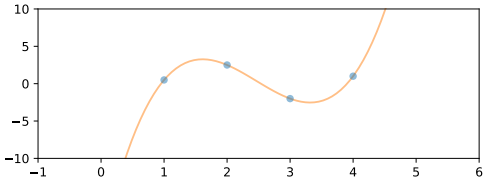
1 Datenpunkt, Polynom 0-ten Grades



2 Datenpunkte, Polynom 1-ten Grades



3 Datenpunkte, Polynom 2-ten Grades

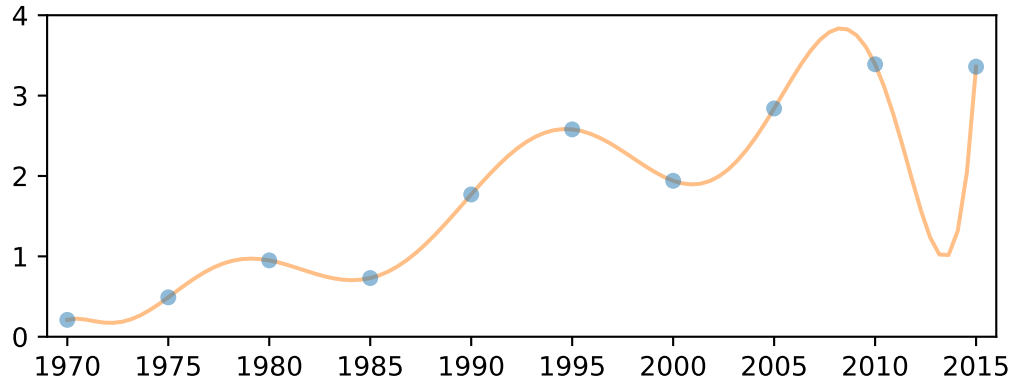


4 Datenpunkte, Polynom 3-ten Grades

Jetzt geht es los: Die folgenden arrays repräsentieren 10 Datenpunkte, die das deutsche Bruttosozialprodukt (in Billionen US $) in den Jahren $1970, 1975, \ldots$ widerspiegeln. (Wenn Sie selbst solche Daten finden wollen, googlen Sie einfach: *gdp data germany*.)

```
xData = np.array([1970, 1975, 1980, 1985, 1990, 1995, 2000, 2005, 2010, 2015])
yData = np.array([0.21, 0.49, 0.95, 0.73, 1.77, 2.58, 1.94, 2.84, 3.39, 3.36])
```

Da wir 10 Datenpunkte gegeben haben, sollten wir ein Polynom 9-ten Grades angleichen können, das durch diese Punkte geht. Die nächste Abbildung zeigt, dass dies in der Tat möglich ist. (Ob dieses Modell eine sinnvolle Beschreibung der Daten liefert sei dahingestellt, darum geht es uns in dieser Aufgabe nicht.)



Daten und Polynom 9-ten Grades

Um die folgenden Arbeitsschritte durchzuführen, brauchen Sie eine Funktion zum Plotten von Daten und Modell. Hier ist so eine Funktion:

```
def plot_data_and_model(xData, yData, xModel, yModel):
    plt.plot(xModel, yModel, '-', c='C1', alpha=0.5)
    plt.plot(xData, yData, 'o', c='C0', mew=0, alpha=0.5)
    plt.xticks(xData)
    plt.xlim(1969, 2016)
    plt.ylim(0, 4)
    plt.show()
```

Obige Abbildung wurde wie folgt erzeugt:

```
def polyFitV1(x, y, d):
    P = poly.Polynomial.fit(x, y, d)
    print ('V1 estimate:', P.convert().coef)

    return P.linspace()

degree = 9
xModel, yModel = polyFitV1(xData, yData, degree)
plot_data_and_model(xData, yData, xModel, yModel)
```

Lassen Sie den auf `polyFitV1` basierten code laufen, um zu sehen, dass er funktioniert.

Unsere Funktion `polyFitV1` nutzt Funktionalitäten aus dem *numpy* Modul `numpy.polynomial.polynomial`, die ein sehr gewöhnungsbedürftiges API haben. Unser Angleichungsproblem müsste sich doch eigentlich besser lösen lassen ...

In der Vorlesung haben wir gesehen, dass die Angleichung polynomieller Modelle im Wesentlichen darin besteht, den Koeffizientenvektor

$$\boldsymbol{w} = \left(\boldsymbol{X}\boldsymbol{X}^\intercal\right)^{-1}\boldsymbol{X}\,\boldsymbol{y} \tag{1}$$

zu berechnen, wobei die Matrix $\boldsymbol{X}^\intercal$ eine Vandermonde Matrix ist. Hier ist eine Idee, wie wir Gleichung (1) direkt umsetzen können, um unser Problem nur mit einfachen *numpy* Funktionen (`vander`, `dot`, `inv`) zu lösen:

```
def polyFitV2(x, y, d):
    Xt  = np.vander(x, d+1, increasing=True)
    XXt = np.dot(Xt.T, Xt)
    w   = np.dot(la.inv(XXt), np.dot(Xt, y))
    print ('V2 estimate:', w)

    xModel = np.linspace(1969, 2016, 100)
    yModel = np.dot(np.vander(xModel, d+1, increasing=True), w)
    return xModel, yModel
```

Da obiges script ein bisschen lang ist, nutzen wir als nächstes die *numpy* Funktion `pinv` zur Berechnung von Pseudoinversen, um Entwickler*innen in Bezug auf Tipparbeit zu entlasten:

```
def polyFitV3(x, y, d):
    Xt = np.vander(x, d+1, increasing=True)
    w  = np.dot(la.pinv(Xt), y)
    print ('V3 estimate:', w)

    xModel = np.linspace(1969, 2016, 100)
    yModel = np.dot(np.vander(xModel, d+1, increasing=True), w)
    return xModel, yModel
```

In der Vorlesung haben wir stets die Funktion `lstsq` zur Lösung von KQ Problemen genutzt; hier ist eine entsprechende Implementierung für unser aktuelles Problem:

```
def polyFitV4(x, y, d):
    Xt = np.vander(x, d+1, increasing=True)
    w  = la.lstsq(Xt, y, rcond=None)[0]
    print ('V4 estimate:', w)

    xModel = np.linspace(1969, 2016, 100)
    yModel = np.dot(np.vander(xModel, d+1, increasing=True), w)
    return xModel, yModel
```

Eine weitere Lösungsidee besteht darin, weniger obskure Funktionen aus dem Modul *numpy.polynomial.polynomial* zu benutzen. Hier ist ein Beispiel dafür, wie das gehen kann:

```
def polyFitV5(x, y, d):
    w = poly.polyfit(x, y, d)
    print ('V5 estimate:', w[::-1])

    xModel = np.linspace(1969, 2016, 100)
    yModel = poly.polyval(xModel, w)
    return xModel, yModel
```

Und hier ist schließlich noch eine sechste Variante zur Lösung unseres Problems, die wiederum mehr Tipparbeit für Entwickler*innen erfordert, die wir aber zunächst unkommentiert lassen:

```
def polyFitV6(x, y, d):
    def standardize(x, mu, sig2):
        return (x - mu) / sig2

    mu = np.mean(x)
    s2 = np.var(x)
    Xt = np.vander(standardize(x, mu, s2), d+1, increasing=True)
    w  = la.lstsq(Xt, y, rcond=None)[0]
    print ('V6 estimate:', w)

    xModel = np.linspace(1969, 2016, 100)
    Xt     = np.vander(standardize(xModel, mu, s2), d+1, increasing=True)
    yModel = np.dot(Xt, w)
    return xModel, yModel
```

Lassen Sie nun folgende snippets laufen und beobachten Sie, was passiert (Ausgaben, Fehlermeldungen, etc.):

```
degree = 9

xModel, yModel = polyFitV1(xData, yData, degree)
plot_data_and_model(xData, yData, xModel, yModel)

xModel, yModel = polyFitV2(xData, yData, degree)
plot_data_and_model(xData, yData, xModel, yModel)

xModel, yModel = polyFitV3(xData, yData, degree)
plot_data_and_model(xData, yData, xModel, yModel)

xModel, yModel = polyFitV4(xData, yData, degree)
plot_data_and_model(xData, yData, xModel, yModel)

xModel, yModel = polyFitV5(xData, yData, degree)
plot_data_and_model(xData, yData, xModel, yModel)

xModel, yModel = polyFitV6(xData, yData, degree)
plot_data_and_model(xData, yData, xModel, yModel)
```

Was geht hier vor? Warum funktionieren *polyFitV2*, *polyFitV3* und *polyFitV4* scheinbar nicht, obwohl sie die zugrundeliegende Mathematik

doch (mehr oder weniger) eins zu eins umsetzen? Ist unsere Mathematik kaputt?

Warum erzeugt `polyFitV5` ein Warnung und die anderen Funktionen nicht? Können oder sollen wir diese Warnung ignorieren? Ist das Ergebnis von `polyFitV5` das, was wir theoretisch erwarten würden?

Warum scheint `polyFitV6` zu funktionieren, obwohl diese Funktion im Wesentlichen nur eine etwas ergänzte Variante von `polyFitV4` ist?

Warum sind die von `polyFitV1` und `polyFitV6` berechneten Koeffizientenvektoren $\boldsymbol{w}$ so unterschiedlich? Die Graphen der beiden resultierenden Polynome sehen doch eigentlich gleich aus?

Denken Sie intensiv über diese Fragen nach! Hier sind noch einige etwas allgemeinere Fragen für Ihre Überlegungen:

- **Was ist *Numerik*?**
- Gibt es einen Unterschied zwischen theoretischer "Mathematik mit Bleistift und Papier" und den praktischen Berechnungen digitaler Computer?
- Was müssen wir beachten, wenn wir mit sehr großen oder sehr kleinen Zahlen oder mit Zahlen nahe 0 operieren?
- Können Sie irgendwelchen (Data Science) libraries auf GitHub blind vertrauen?