FH·W-S

Mehrdimensionales Datenfitting mit linearer Regression

Stefan Volz
Fakultät für angewandte Natur- und Geisteswissenschaften
Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt
Studiengang Technomathematik
Matrikelnummer: 3519001
stefan.volz@student.fhws.de

31. Dezember 2019, Wintersemester 2019

Abstract

In vielen Anwendungen ist es nötig/vorteilhaft aus Messdaten ein mathematisches Modell zu entwickeln, welches möglichst Messungenauigkeiten ausgleicht bzw. Vorhersagen zulässt. Ziel der Arbeit ist die Implementierung von linearer Regression. Hierbei soll mittels Gradientenabstieg eine Fehlerminimierung des Modells im Bezug auf gegebene Messdaten durchgeführt werden und anschließend am Beispiel von BFGS ein Ausblick auf komplexere Optimierungsverfahren gegeben werden. Die Implementierung erfolgt in Julia.

Inhaltsverzeichnis

1 Konventionen und Generelles **3**
1.1 Wahl der Programmiersprache und Abhängigkeiten . . . 3
1.2 Quellcode . . . 3
1.3 Text dieser Arbeit . . . 3
1.4 Variablenbezeichner . . . 3

2 Grundlegende Implementierung **4**
2.1 Problemformulierung . . . 4
2.2 Lineare Regression . . . 4
2.2.1 Mathematische Grundlagen . . . 4
2.2.2 Implementierung . . . 4
2.3 Die Fehlerfunktion . . . 5
2.4 Gradientenabstiegsverfahren . . . 5
2.4.1 Mathematische Grundlagen . . . 5

2.4.2 Einzeliteration des Gradientenabstiegs 6
2.4.3 Hauptfunktion 7
2.4.4 Abbruchkriterium 9

3 Fortgeschrittene Features **11**
3.1 Momentum Verfahren 11
3.2 Tichonow Regularisierung 14
3.3 Ausblick auf den Broyden–Fletcher–Goldfarb–Shanno Algorithmus 15

A Performancemessung **19**

1 Konventionen und Generelles

1.1 Wahl der Programmiersprache und Abhängigkeiten

Die Implementierung erfolgt in Julia[1]. Julia ist eine hochperformante, stark und dynamisch typisierte, überwiegend imperative Programmiersprache mit Fokus auf wissenschaftlichem Rechnen; ist jedoch im Gegensatz zu z.B. MATLAB als General Purpose Sprache zu verstehen. Die Arbeit benutzt Julia in Version 1.2.0. Die Abhängigkeiten beschränken sich auf das `Plots` Package[2] - jedoch ist die eigentliche Logik abhängigkeitsfrei.

Die Wahl der Sprache fiel auf Julia, da es nativ eine sehr gute Unterstützung für Matrizen mitbringt was für die Implementierung als sehr vorteilhaft angesehen wurde. Außerdem erlaubt der gute Unicode-Support es, Identifier so zu wählen, dass diese nah an der Fachliteratur sind. Desweiteren ist die immense Performancesteigerung gegenüber beispielsweise Python ein großes Plus. Syntaktisch und semantisch sollte Julia für Leser, die eine andere dynamische Hochsprache (z.B. Python, Matlab, Ruby) beherrschen kein Problem sein - spezielle Sprachfeatures werden i.d.R. in Fußnoten kurz erklärt.

1.2 Quellcode

Beim Code wurde darauf geachtet, die Typannotationen[3] von Julia zu nutzen, da diese zum einfacheren Verständnis des Codes beitragen und außerdem der Performance zuträglich sind. Identifier wurden größtenteils so gewählt, dass sie sich mit [Bis09] decken - teils wurden auch Bezeichner aus [Lip06] übernommen. Der Code wird unter `https://github.com/SV-97/LinearRegression` gehostet.

1.3 Text dieser Arbeit

Im Text der Arbeit werden - in Anlehnung an [Bis09] - folgende Konventionen genutzt:

Typ	Beschreibung	Beispiel
vektorielle Größen	fettgedruckte Kleinbuchstaben	$\mathbf{w}$
einzelne Elemente eines Vektors	indizierte Kleinbuchstaben	w_j
Matrizen und Mengen	Großbuchstaben	A
Hyperparameter des Modells	griechische Kleinbuchstaben	γ
sonstige Parameter	lateinische Kleinbuchstaben	x
Code listings und Quellcode-Referenzen	monospace font	`code`

Einzelne Indizes an Matrixen wie z.B. A_j sind als Zeilenindizes zu verstehen, sodass A_j die j-te Zeile von A ist. In einigen Fällen wurde zwecks Konsistenzwahrung mit [Bis09] oder der zum jeweiligen Thema gehörigen Literatur von diesen Konventionen abgewichen. Für eine Matrix A steht A^T für die transponierte der Matrix. Die Identitätsmatrix zu $\mathbb{R}^{n\times n}$ wird geschrieben als I_n.

1.4 Variablenbezeichner

Einige der wichtigsten Bezeichner sind hier aufgeführt:

Bezeichner	Beschreibung
d	Dimension eines Eingabevektors
k	Dimension eines Zielwertsvektors
M	Anzahl der Modellparameter
N	Anzahl der Trainingsdatensätze
X	Trainingseingabematrix
$T/\mathbf{t}$	Trainingszielmatrix/Trainingszielvektor
$\mathbf{w}$	Modellparameter

[1] `https://julialang.org/`
[2] `https://docs.juliaplots.org/`
[3] `https://docs.julialang.org/en/v1/manual/types/`

2 Grundlegende Implementierung

2.1 Problemformulierung

Seien $d, k, M, N \in \mathbb{N}$ mit Messdaten $X \in \mathbb{R}^{d \times N}$ und zugehörigen Zielwerten $T \in \mathbb{R}^{k \times N}$ gegeben. Gesucht werden die Parameter $\mathbf{w}$ eines Modells $y(\mathbf{w}, \mathbf{x})$ mit $y \in \mathbb{R}^M \times \mathbb{R}^d \to \mathbb{R}^k$ welche das Minimierungsproblem

$$\min_{\mathbf{w} \in \mathbb{R}^M} \sum_{i=1}^{N} E(y(\mathbf{w}, X_i), T_i),$$

im Bezug auf eine Fehlerfunktion $E : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}$ lösen.

Wir betrachten zunächst nur Probleme für die $k = 1$ gilt und bezeichnen daher die Zielwerte mit $\mathbf{t}$.

2.2 Lineare Regression

2.2.1 Mathematische Grundlagen

Bei linearer Regression handelt es sich um eine Methode des überwachten Lernens.

Definition 2.1. Lineare Regressions Modelle sind Modelle, welche sich im Bezug auf ihre Modellparameter linear verhalten[Bis09, S. 137f]. Sie stellen Funktionen der Form $y : \mathbb{R}^M \times \mathbb{R}^d \to \mathbb{R}^k, (\mathbf{w}, \mathbf{x}) \mapsto y(\mathbf{w}, \mathbf{x})$ dar. Die Anzahl der Modellparameter ist gegeben durch $M \in \mathbb{N}$, die Anzahl an Eingangsgrößen durch $d \in \mathbb{N}$ und die der Zielgrößen durch $k \in \mathbb{N}$.

Anmerkung 2.2. Dies bedeutet nicht, dass sie auch zwingend linear im Bezug auf die Eingangsvariablen sein müssen.

Das einfachste lineare Regressions Modell ist eine Linearkombination der Eingangsvariablen

$$y(\mathbf{w}, \mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_D x_D,$$

diese spiegeln jedoch oftmals nicht die zugrunde-liegende Verteilung der realen Messwerte wider, und limitieren ein Modell mit d Eingangsvariablen auf $d+1$ Modellparameter. Daher werden Basisfunktionen eingeführt und Lineare Regressions Modelle als Linearkombination dieser gebildet.

Definition 2.3. Eine Funktion $\Phi_j : \mathbb{R}^d \to \mathbb{R}^k, \mathbf{x} \mapsto \Phi_j(\mathbf{x})$ bezeichnen wir als Basisfunktion.

Anmerkung 2.4. Im Code werden Basisfunktionen als Funktionen $\Phi : \mathbb{N} \times \mathbb{R}^d \to \mathbb{R}^k, (j, \mathbf{x}) \mapsto \Phi(j, \mathbf{x})$ implementiert.

Mit diesen Basisfunktionen ergibt sich als Modellgleichung

$$y(\mathbf{w}, \mathbf{x}) = w_0 + \sum_{j=1}^{M-1} w_j \Phi_j(\mathbf{x}),$$

wobei der Parameter w_0 auch als Bias-Parameter bezeichnet wird, da er einen festen Offset der Daten ermöglicht. Im Code werden wir diesen Bias-Parameter als "normalen" Parameter betrachten und als kleinsten Index in $\mathbf{w}$ Eins wählen. Ein eingangsvariablenunabhängiger Offset ist leicht durch eine angepasste Definition der Basisfunktion erreichbar:

$$\Phi : (j, \mathbf{x}) \mapsto \begin{cases} 1, & j = 1, \\ \Phi_j(\mathbf{x}), & \textit{sonst}. \end{cases}$$

Damit vereinfacht sich die Darstellung der Modellgleichung zu

$$y(\mathbf{w}, \mathbf{x}) = \sum_{j=1}^{M} w_j \Phi(j, \mathbf{x}). \tag{1}$$

2.2.2 Implementierung

Auf Basis dieser Formulierungen können wir mit der Implementierung beginnen. Hierzu definieren wir uns zunächst eine Funktion Σ um alle im Code vorkommenden Summen zu abzudecken.

Listing 1 Funktion Σ

```
    "Sum from k=`from` to `to` of `a(k)`"
    Σ(from::Integer, to::Integer, a::Function, zero = 0) =
      mapreduce(a, (+), from:to; init = zero)
```

Die Funktion ist hierbei eine Implementierung von $\sum_{k=\text{from}}^{\text{to}} a(k)$. Bei `mapreduce` handelt es sich um eine eingebaute Funktion welche (hier) erst `a` über eine Sequenz mappt und anschließend diese Sequenz mittels `(+)` reduziert/faltet. Dabei ist `(+)` der eingebaute Additionsoperator. Der optionale Parameter `zero` erlaubt es die Funktion auch für nicht-skalare Summen(bzw. jegliche Typen für die eine Implementierung zu `+` existiert) zu nutzen.

Listing 2 Funktion y

```
    """Linear Regression
    # Args:
        w: Parameters
        Φ(j, x): Basis function of type (Int, Vector{T}) → T
        x: Input vector
    """
    function y(
      w::Vector{<:Number},
      Φ::(T where T <: Function),
      x::Vector{<:Number})::Number
        Σ(1, size(w)[1], j→w[j] * Φ(j, x))
    end
```

Diese Implementierung der Funktion `y` passt 1:1 zur mathematischen Formulierung in Gleichung 1[4].

2.3 Die Fehlerfunktion

Als nächstes benötigen wir eine Möglichkeit um den Fehler des Systems zu ermitteln - sodass wir diesen im nächsten Schritt minimieren können. Die genutzte Fehlerfunktion ist die des quadratischen Fehlers. Nach [Bis09, S. 140f] ergibt sich die Fehlerfunktion

$$E_D := \frac{1}{2}\sum_{n=1}^{N}(\mathbf{t}_n - y(\mathbf{w}, \mathbf{x}))^2. \tag{2}$$

Mit dieser Definition können wir nun unser Minimierungsproblem wie folgt definieren:

$$\min_{\mathbf{w}\in\mathbb{R}^M} \frac{1}{2}\sum_{n=1}^{N}(\mathbf{t}_n - y(\mathbf{w}, X_n))^2.$$

Im nächsten Schritt werden wir ein Verfahren implementieren, das diese Minimierung durchführt.

2.4 Gradientenabstiegsverfahren

2.4.1 Mathematische Grundlagen

Das Gradientenabstiegsverfahren ist ein numerisches Verfahren mit dem sich allgemeine Optimierungsprobleme lösen lassen. Beim Gradientenabstiegsverfahren bestimmen wir den Gradienten von E_D im Bezug auf

[4]Die in Listing 2 genutzte Schreibweise `Vector{<:Number}` bedeutet *Vektor eines Typen T, wobei T ein Subtyp des abstrakten Typs Number ist.* Diese hat gegenüber `Vector{Number}` den Vorteil, dass sie gewisse Optimierungen ermöglicht(Im Detail erlaubt `<:` Julia mit unboxed Werten zu arbeiten).

die Modellparameter $\mathbf{w}$ - diesen Gradienten bezeichnen wir mit $\nabla_{\mathbf{w}} E_D$. Hierzu bestimmen wir zunächst die partielle Ableitungen von E_D nach allen w_k aus $\mathbf{w}$.

$$\frac{\partial E_D}{\partial \mathbf{w}_k} = \frac{\partial}{\partial \mathbf{w}_k} \frac{1}{2} \sum_{n=1}^{N} (\mathbf{t}_n - y(\mathbf{w}, \mathbf{x}))^2 = -\sum_{n=1}^{N} \Phi(k, X_n) \cdot (\mathbf{t}_n - y(\mathbf{w}, \Phi, X_n))$$

Der Gradient ist dann

$$\nabla_{\mathbf{w}} E_D = \begin{pmatrix} \frac{\partial E_D}{\partial \mathbf{w_1}} \\ \vdots \\ \frac{\partial E_D}{\partial \mathbf{w_M}} \end{pmatrix}.$$

Um diesen Gradienten zu implementieren, beginnen wir mit der Implementierung der partiellen Ableitung:

Listing 3 Funktion `∂E_D∂w_k`

```
    """Derivative of E_D with respect to w_k
    # Args:
        Φ(k, x_n): Basis function
        X: Set of inputs x_n where x_n is an input vector to Φ
        t: corresponding target values for each x_n
        k: Index for w_k in respect to which the derivative is taken
        w: Parameters
    """
    function ∂E_D∂w_k(
      Φ::Function,
      X::Matrix{<:Number},
      t::Vector{<:Number},
      w::Vector{<:Number},
      k::Integer)::Number
        N = size(t)[1]
        - Σ(1, N, n→Φ(k, X[n, :]) * (t[n] - y(w, Φ, X[n,:])))
    end
```

2.4.2 Einzeliteration des Gradientenabstiegs

Hiermit können wir eine Iteration des Gradientenabstiegsalgorithmus wie folgt implementieren.

Listing 4 Funktion `gradient_descent_iteration`

```
    """One iteration of the gradient descent algorithm
    # Args:
        ∂E_D∂w_k: Partial derivative of error function with respect to
            the k-th parameter
        X: Column vector of inputs x_n where x_n is an input vector to the
            error function
        t: corresponding target values for each x_n
        w: Parameters
        η: Learning rate
    """
    function gradient_descent_iteration(
      ∂E_D∂w_k::Function,
      X::Matrix{<:Number},
      t::Vector{<:Number},
      w::Vector{<:Number},
      η::Number)::Vector{<:Number}
        M = size(w)[1]
        ∇w = zero(w)
        for j = 1:M
            ∂E_D∂w_jk(k) = ∂E_D∂w_k(X, t, w, k)
            ∇w += collect(map(∂E_D∂w_jk, 1:M))
        end
        w - η * ∇w
    end
```

Innerhalb der Funktion iterieren wir über alle Indizes j, bilden die partielle Ableitung nach $\mathbf{w}_k$[5] und berechnen ihren Wert für alle Komponenten des Ergebnisvektors. Der Gradient ist dann die Summe all dieser Vektoren. In der letzten Zeile der Funktion machen wir einen "Schritt" in Richtung des Gradienten - steigen auf der Fehlerkurve also ab. Die Schrittweite wird hierbei durch den Hyperparameter η gesteuert. Dieser als Lernrate bezeichnete Parameter steuert im Grunde genommen die Konvergenzgeschwindigkeits des Gradientenabstiegsverfahrens.

2.4.3 Hauptfunktion

Auf Basis dieser Funktion können wir nun den Rest des Algorithmus in der Funktion `gradient_descent` umsetzen.

[5] Achtung: Die hier genutzte Funktion ist nicht die des globalen Scopes, sondern ein Parameter der Funktion.

Listing 5 Funktion `gradient_descent`

```
    """Minimize function E_D(X, t, w)
    # Args:
        ∂E_D∂w_k: Partial derivative of error function with respect to
            the k-th parameter
        X: Column vector of inputs x_n where x_n is an input vector to the
            error function
        t: corresponding target values for each x_n
        w: Initial parameters - usually `randn(M)`
        η: Learning rate
        M: Number of model parameters
        iters: Number of iterations
    """
    function gradient_descent(
      ∂E_D∂w_k::Function,
      X::Matrix{<:Number},
      t::Vector{<:Number},
      η::Number,
      M::Integer,
      iters::Integer,
      w::Vector{<:Number})::Vector{<:Number}
        ∇w = zero(w)
        for i = 1:iters
            w = gradient_descent_iteration(∂E_D∂w_k, X, t, w, η)
        end
        w
    end
```

Diese Funktion ruft `iters`-mal die Hilsfunktion `gradient_descent_iteration` auf und updated die Parameter mit dem Ergebnis dieser Aufrufe. Nach Ende der Optimierung gibt sie die als optimal befundenen Werte der Parameter zurück.

Der Aufruf von `gradient_descent` erfolgt aus der Funktion `fit_linear_model` heraus.

Listing 6 Funktion `fit_linear_model`

```
    """Find regression model
    # Args:
        Φ: Basis Function
        X: Set of inputs x_n where x_n is an input vector to Φ
        t: corresponding target values for each x_n
        η: learning rate with which to train
        M: Number of model parameters
        iters: Number of iterations
        optimizer: Parameter to select optimizer that's used
    # Fails:
        On unknown optimizers or error inside the optimizer
    """
    function fit_linear_model(
      Φ::Function,
      X::Matrix{<:Number},
      t::Vector{<:Number},
      η::Number,
      M::Integer,
      iters::Integer,
      optimizer = :gradient_descent)::Tuple{Function,Number}
        if optimizer == :gradient_descent
            w = gradient_descent(
              (X, t, w, k)→∂E_D∂w_k(Φ, X, t, w, k),
              X, t, η, M, iters, randn(M), ε, γ)
            residual_error = E_D(Φ, X, t, w)
            (x→y(w, Φ, x), residual_error)
        else
            error("Invalid optimizer")
        end
    end
```

Diese Funktion stellt im Grunde genommen nur einen Treiber für den Optimizer da. Sie initialisiert die Modellparameter zu Beginn mit einem Vektor aus normalverteilten Zufallszahlen[Lip06] aus dem Intervall $[-1, 1]$. Nach der Optimierung gibt sie das vollendete Modell als Closure[6] zurück, was dem die Funktion Aufrufenden ermöglicht, Modellaussagen in Abhängigkeit eines Eingangsvektors zu erhalten. Außerdem bestimmt sie den Restfehler des Modells und gibt auch diesen zurück. Der Parameter `optimizer` ist bisher noch nicht in Benutzung, da nur ein Optimizer implementiert ist[7].

2.4.4 Abbruchkriterium

Die Grundimplementierung wird mit einem einfachen Abbruchkriterium abgeschlossen. Hierbei wird zu jeder Iteration geprüft, ob die Norm der Differenz der Parametervektoren von zwei aufeinanderfolgenden Iterationen kleiner als ein neuer Hyperparameter ε ist - in Formeln ausgedrückt: es wird geprüft ob $||\mathbf{w} - \mathbf{w}'||_2 < \varepsilon$ gilt. Außerdem wird geprüft ob einer der Parameter zu $\pm\infty$ divergiert (und das Verfahren somit "fehlgeschlagen") ist oder durch einen Rechenfehler ein `NaN`[8] in $\mathbf{w}$ aufgetaucht ist. In all diesen Fällen wird der

[6] Eine Closure ist eine anonyme Funktion welche intern eine Referenz auf ihren Erstellungskontext hält. Hier wird sie eingesetzt um `y` mit $\mathbf{w}$ und `Φ` partiell zu evaluieren.

[7] `:abc` ist ein Symbol-Literal. Diese Symbole sind vergleichbar mit denen in Ruby und Scheme oder auch Atomen in Erlang und Prolog. (siehe `https://docs.julialang.org/en/v1/manual/metaprogramming/`)

[8] *Not a Number* - spezieller Wert von IEEE floats

Algorithmus vorzeitig abgebrochen. Sofern kein Fehler vorliegt werden die Modellparameter zurückgegeben[9]. Außerdem ändert sich der Rückgabewert der Funktion diesbezüglich, dass sie nun zurückgibt wieviele Iterationen tatsächlich durchlaufen wurden.

Listing 7 Funktion `gradient_descent` mit einfachem Abbruchkriterium

```
    """Minimize function E_D(X, t, w)
    # Args:
        ∂E_D∂w_k: Partial derivative of error function with respect to
            the k-th parameter
        X: Column vector of inputs x_n where x_n is an input vector to the
            error function
        t: corresponding target values for each x_n
        w: Initial parameters - usually `randn(M)`
        η: Learning rate
        M: Number of model parameters
        iters: Number of iterations
        ε: Gradient descent stops once the difference between two iterations
            (w and w') is less than ε

    # Fails:
        Fails on encountering NaN in computation or on Divergence to Inf
    """
    function gradient_descent(
      ∂E_D∂w_k::Function,
      X::Matrix{<:Number},
      t::Vector{<:Number},
      η::Number,
      M::Integer,
      iters::Integer,
      w::Vector{<:Number},
      ε = 10e-12::Number)::Tuple{Vector{<:Number},Integer}
        did_iters = 0
        for i = 1:iters
            did_iters += 1
            w_old = w
            w = gradient_descent_iteration(∂E_D∂w_k, X, t, w, η, ∇w, γ)
            if any(isnan.(w))
              error("Encountered NaN")
            end
            if any(isinf.(w))
                error("Divergence in calculation")
            end
            if norm(w_old - w) < ε
                break
            end
        end
        (w, did_iters)
    end
```

Für den Parameter ε wird eine arbiträr gewählte Standartbelegung von $10 \cdot 10^{-12}$ gesetzt. Ein weiteres denkbares und oft genutztes Abbruchkriterium ist die Überprüfung, ob die Norm des Gradienten ungefähr Null ist.

[9] Der Punkt im Funktionsaufruf in Listing 7 sorgt dafür, dass die Funktion vektorisiert/*pointwise* aufgerufen wird (siehe `https://docs.julialang.org/en/v1/manual/functions/`).

3 Fortgeschrittene Features

3.1 Momentum Verfahren

Das Momentum Verfahren ist eine Adaption des Gradientenabstiegs, welche die Konvergenzgeschwindigkeit erhöhen soll. Das Ziel ist es, auf flachen Bereichen der Fehlerkurve die Abstiegsgeschwindigkeit zu erhöhen und sie in steilen Passagen zu verringern. Das Verfahren lässt sich als ein Ball der eine Kurve hinabrollt visualisieren - dieser verhält sich bei Änderungen der Kurve mit einer gewissen Trägheit. Um eine solche Trägheit zu realisieren wird ein Momentum-Term hinzugefügt. Dieser Term ist das Produkt des Gradienten der vorhergehenden Iteration und eines neuen Hyperparameters $\gamma \in [0, 1)$ - dem Trägheitsfaktor. Der Gradient $\nabla_{\mathbf{w}} \hat{E_D}$ in jeder Iteration ergibt sich dann aus dem eigentlichen Gradienten dieser Iteration $\nabla_{\mathbf{w}} E'_D$ sowie dem der vorhergehenden Iteration $\nabla_{\mathbf{w}} E_D$ multipliziert mit dem Momentum-Faktor γ.

$$\nabla_{\mathbf{w}} \hat{E}_D = \nabla_{\mathbf{w}} E'_D + \gamma \nabla_{\mathbf{w}} E_D$$

Zur Implementierung müssen lediglich γ und $\nabla_{\mathbf{w}} E_D$(im Code `∇w_prior`) als neue Parameter hinzugefügt werden und die Rückgabe so angepasst, dass in jeder Iteration ein 2-Tupel aus neuem Parametervektor und aktuellem Gradienten zurückgegeben wird. Dementsprechend müssen auch die call sites in `gradient_descent` sowie `fit_linear_model` angepasst werden. Da die Änderungen an den call sites trivial sind ist hier nur `gradient_descent_iteration` aufgeführt.

Listing 8 Funktion `gradient_descent_iteration` mit Momentum Verfahren

```
    """One iteration of the gradient descent algorithm
    # Args:
        ∂E_D∂w_k: Partial derivative of error function with respect to
            the k-th parameter
        X: Column vector of inputs x_n where x_n is an input vector to the
            error function
        t: corresponding target values for each x_n
        w: Parameters
        η: Learning rate
        ∇w_prior: Gradient of parameters from prior iteration
        γ: Momentum factor
    """
    function gradient_descent_iteration(
      ∂E_D∂w_k::Function,
      X::Matrix{<:Number},
      t::Vector{<:Number},
      w::Vector{<:Number},
      η::Number,
      ∇w_prior::Vector{<:Number},
      γ::Number)::Tuple{Vector{<:Number},Vector{<:Number}}
        M = size(w)[1]
        ∇w = γ * ∇w_prior
        for j = 1:M
            ∂E_D∂w_jk(k) = ∂E_D∂w_k(X, t, w, k)
            ∇w += collect(map(∂E_D∂w_jk, 1:M))
        end
        (w - η * ∇w, ∇w)
    end
```

Nach [Lip06, S. 110] wird γ mit einem Wert von 0.9 vorbelegt. Die wohl größte Problematik dieses Verfahrens ist, dass der Fall auftreten kann das der Momentum-Term betragsmäßig größer als der aktuelle Gradient ist, jedoch das umgekehrte Vorzeichen besitzt. In diesem Fall würde sich der Fehler des Systems vergrößern.

Aufgrund dessen kann hier keine Konvergenz garantiert werden. Ein kurzer Test[10] (mit konstanter Lernrate und `ε=10e-12` mit nur wenigen Datenpunkten zeigt folgende Daten[11]:

γ	k bei Abbruch nach k Iterationen	kleinstmöglicher Restfehler	Laufzeit (in Sekunden)	Gesamt allokierter Speicher (in MiB)	Garbage Collector Zeitanteil (in %)
0.0	50544	6.701991676725787	0.80471	890.111	9.37
0.5	28837	6.701991676725787	0.43627	463.782	9.46
0.9	5635	6.701991676725783	0.07825	99.488	8.39

Also gibt es durchaus Fälle, in denen durch dieses Verfahren eine enorme Steigerung bei der Konvergenzgeschwindigkeit erzielt wird. Jedoch sehen die Verläufe der Kurven von Restfehler und Gradient wie folgt aus:

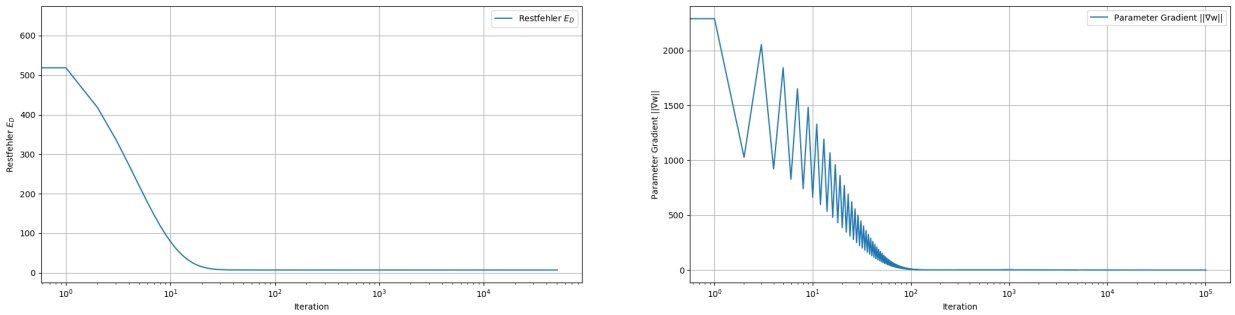

Abbildung 1: Beispielfit mit $\gamma = 0$

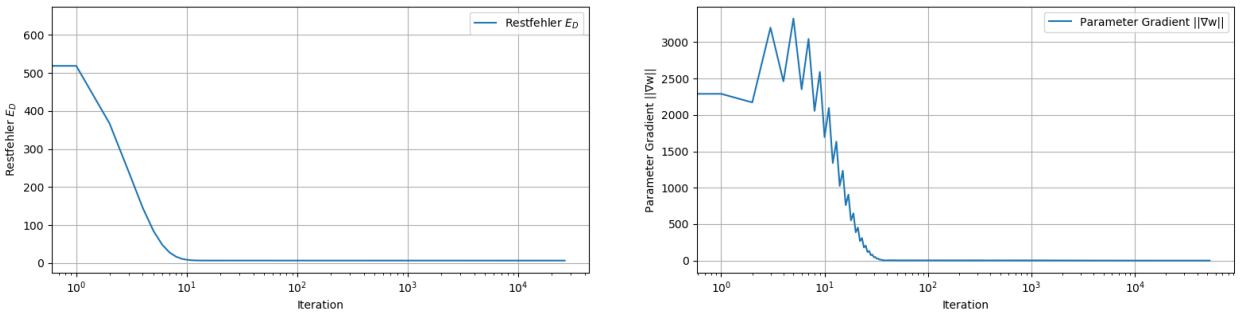

Abbildung 2: Beispielfit mit $\gamma = 0.5$

[10] Die Funktion mit der getestet wurde ist $x \mapsto x^2 + \theta_x$, wobei θ_x ein Wert aus einem normalverteilten Rauschen von -3 bis $+3$ ist.

[11] Details zur Messung in Anhang A. Es ist zu beachten, dass der insgesamt allokierte Speicher dargestellt ist.

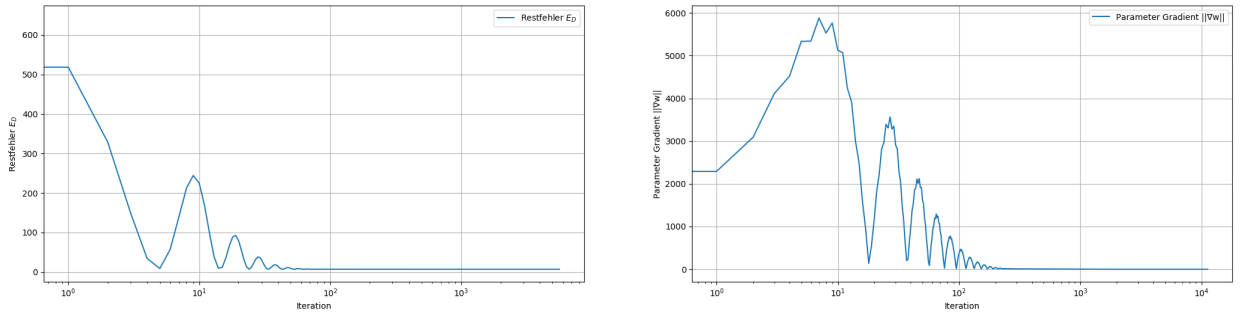

Abbildung 3: Beispielfit mit $\gamma = 0.9$

Es ist klar zu sehen, dass der oben genannte 'worst case' im Fall $\gamma = 0.9$ eingetreten ist. Das vermeintlich bessere Konvergenzverhalten lag daran, dass ein sehr geringes ε gewählt wurde. Erhöhen wir ε auf beispielsweise `10e-3`, dann ergeben sich die folgenden Werte[12]:

γ	k bei Abbruch nach k Iterationen	kleinstmöglicher Restfehler	Laufzeit (in Sekunden)	Gesamt allokierter Speicher (in MiB)	Garbage Collector Zeitanteil (in %)
0.0	24	10.216203809068343	0.00581	1.398	~~61.54~~
0.5	14	6.813835951850837	0.00570	1.232	~~48.19~~
0.9	29	36.63725946510655	0.00680	1.438	~~60.92~~

beziehungsweise folgende Graphen:

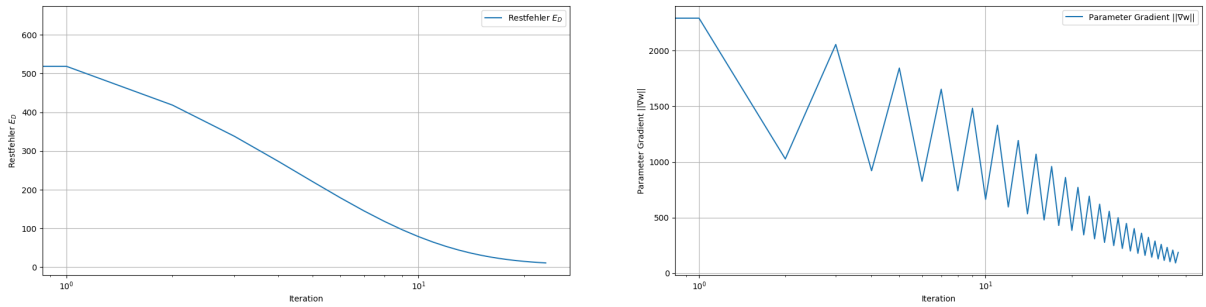

Abbildung 4: Beispielfit mit $\gamma = 0$

[12]Die GC Werte sind gestrichen, da sie sich nicht für alle Testdurchläufe bestimmen ließen (`@time` liefert bei zu kurzen Durchläufen nicht zwingend alle Werte - evtl. liefert es keinen Wert falls während des Messvorgangs keine GC-Iteration vorlag. Die angegebenen Werte sind das Mittel aus drei Iterationen in denen ein GC-Wert vorlag.)

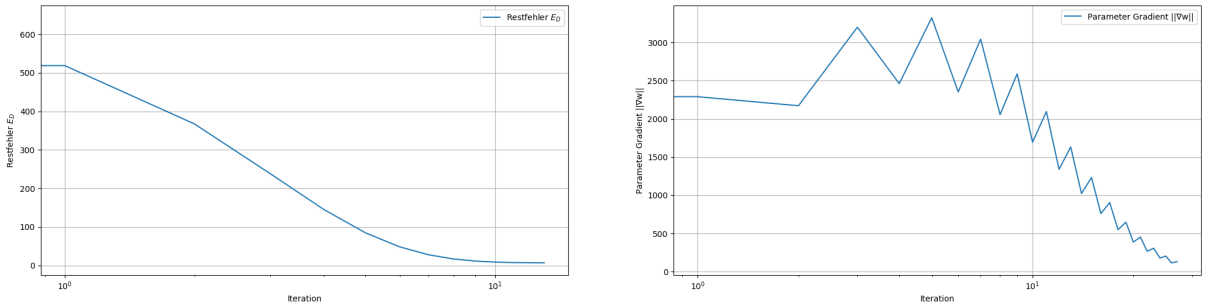

Abbildung 5: Beispielfit mit $\gamma = 0.5$

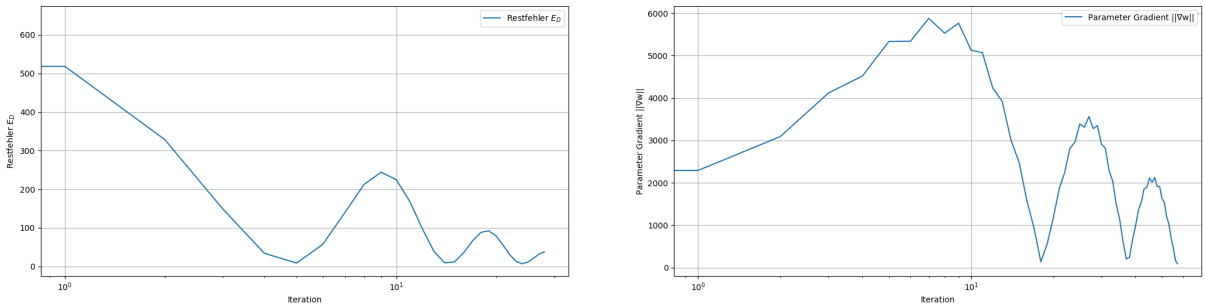

Abbildung 6: Beispielfit mit $\gamma = 0.9$

Das Momentumverfahren wirkt sich hier also teils positiv und teils negativ aus - bei der Wahl der Parameter ist vorsicht geboten.

3.2 Tichonow Regularisierung

Eine weitere Möglichkeit der Verbesserung des Verfahrens ist die *Tichonow Regularisierung*. Das Ziel dieses Verfahrens ist es, die Stabilität des Modells zu verbessern. Dabei wird das Minimierungsproblem wie folgt neu formuliert:

$$\min_{\mathbf{w}\in\mathbb{R}^M} \frac{1}{2}\sum_{n=1}^{N}(\mathbf{t}_n - y(\mathbf{w}, X_n))^2 + \omega||\mathbf{w}||_2^2.$$

Es werden also zu große Werte in $\mathbf{w}$ ”bestraft”. Hierbei ist ω ein neuer Hyperparameter welcher festlegt wie ”wichtig” die Minimierung des Parametervektors ist. Die neue partielle Ableitung ist damit:

$$\frac{\partial E_{D,decay}}{\partial \mathbf{w}_k} = -\sum_{n=1}^{N}\Phi(k, X_n)\cdot(\mathbf{t}_n - y(\mathbf{w}, \Phi, X_n)) - \omega\mathbf{w}_k$$

Die erforderlichen Änderungen am Code sind das Durchschleifen des Parameters ω und die Anpassung der partielle Ableitung:

Listing 9 Funktion `∂E_D∂w_k` mit Tichonow Regularisierung

```
    """Derivative of E_D with respect to w_k
    # Args:
        Φ(k, x_n): Basis function
        X: Set of inputs x_n where x_n is an input vector to Φ
        t: corresponding target values for each x_n
        w: Parameters
        k: Index for w_k in respect to which the derivative is taken
        ω: Weight decay factor
    """
    function ∂E_D∂w_k(Φ::Function,
      X::Matrix{<:Number},
      t::Vector{<:Number},
      w::Vector{<:Number},
      k::Integer,
      ω::Real)::Number
        N = size(t)[1]
        - Σ(1, N, n→Φ(k, X[n, :]) * (t[n] - y(w, Φ, X[n,:])) - ω*w[k])
    end
```

3.3 Ausblick auf den Broyden–Fletcher–Goldfarb–Shanno Algorithmus

Der Broyden–Fletcher–Goldfarb–Shanno - oder kurz BFGS - Algorithmus, ist ein Optimierungsverfahren aus der Gruppe der Quasi-Newton Verfahren. Nach [Dai13] ist BFGS der effizienteste Vertreter dieser Gruppe; so findet sich das Verfahren auch in vielen professionellen Softwaretoolboxes und Programmiersprachen wie z.B. R, Matlab oder auch SciPy wieder. Ziel ist das lösen des Optimierungsproblems $\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x})$ mit $f : \mathbb{R}^n \to \mathbb{R}$. Beginnend mit einem Startwert $\mathbf{x}_0$ und einer Näherung der Hesse Matrix $B_0 := I_n$ werden folgende Schritte abgearbeitet. Dabei konvergiert $\mathbf{x}_k$ gegen die Lösung des Problems.

1. Bestimmen einer Abstiegsrichtung $\mathbf{p}_k$ durch lösen des LGS $B_k \cdot \mathbf{p}_k = -\nabla f(\mathbf{x}_k)$.
2. Sei $g_k : \mathbb{R} \to \mathbb{R}$ definiert durch $\alpha \mapsto f(x_k + \alpha \cdot p_k)$, dann ist die Schrittweite $\alpha_k > 0$ gegeben durch $\alpha_k = \arg\min_{\alpha\in\mathbb{R}} \frac{\partial g_k}{\partial \alpha}$.
   Dieses Minimierungsproblem wird mittels eines Liniensuchverfahrens (in der Implementierung wird hier zwecks Einfachheit das Gradientenabstiegsverfahren gewählt - andere mögliche Kandidaten wären z.B. Newton Verfahren oder das CG-Verfahren) gelöst - wobei nur ein lokales Minimum gesucht wird[Dai13].
3. Setzen von $\mathbf{s}_k = \alpha_k \cdot \mathbf{p}_k$ und $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$.
4. Berechnung von $\mathbf{y}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$.
5. Neue Näherung der Hesse Matrix B nach der Rekursionsvorschrift $B_{k+1} = B_k + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k} - \frac{B_k \mathbf{s}_k \mathbf{s}_k^T B_k}{\mathbf{s}_k^T B_k \mathbf{s}_k}$ bestimmen.

Schritt 2 lässt sich so visualisieren, dass man die durch die Funktion beschriebene Figur mit der von Gradient und y-Achse aufgespannten Ebene schneidet - und dann das der aktuellen Position nähste lokale Minimum der dabei entstehenden Kurve sucht. Aus dieser Überlegung folgt direkt, dass für einfache Probleme(z.B. $f(x, y) = x^2 + y^2$)(bei korrekter Parameterwahl) nach nur einem Schritt bereits eine Lösung gefunden wird.

In Schritt 2 wird die Ableitung von $f(x_k + \alpha p_k)$ nach α benötigt - um diese nicht immer bestimmen zu müssen, werden wir sie einfach numerisch nähern. Die Funktion `numeric_differentiation(f, h) = x → (f(x + h) - f(x - h)) / (2 * h)` bestimmt mittels des zentralen Differenzquotienten die Ableitung einer eindimensionalen Funktion f.

Zum line search wählen wir zu Demozwecken den bereits bekannten Gradientenabstieg - diesmal jedoch in allgemeiner Ausführung und ohne Regularisierung etc..

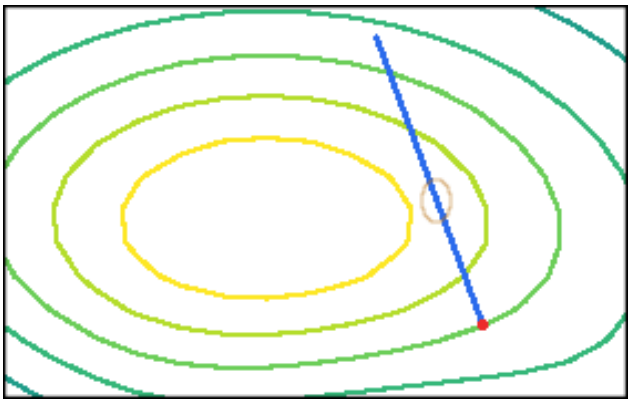

Abbildung 7: Contourplot mit Visualisierung zu Schritt 2. Der rote Punkt zeigt die Startposition x_k, die blaue Linie einen Teil der durch $x_k + \alpha p_k$ beschriebenen Geraden. Durch eine eindimensionale Optimierung wird dann α_k so gewählt, dass man bei einem Minimum(orange/braune Ellipse) landet. Originalbild von `https://octave.sourceforge.io/octave/function/images/contour_101.png`.

Listing 10 Funktion `gradient_descent`

```
    """Optimize some function f
    # Args:
        ∇f: Gradient of f
        x_0: Initial guess
        η: learning rate
        ε: cancellation tolerance
        max_iters: maximum number of iterations
    """
    function gradient_descent(∇f::Function,
      x_0,
      η::Number,
      ε::Number,
      max_iters::Integer)
        x_k = x_0
        count = 0
        for _ = 1:max_iters
            if any(isnan.(x_k))
                error("Encoutered NaN")
            end
            if any(isinf.(x_k))
                error("Encoutered Inf")
            end
            count += 1
            p_k = -∇f(x_k)
            if norm(∇f(p_k)) < ε
                break
            end
            x_k += η * p_k
        end
        x_k
    end
```

Die liniensuchfunktionsagnostische Implementierung des BFGS-Algorithmus ist dann wie folgt:

Listing 11 Funktion `BFGS`

```
    """BFGS Optimization Algorithm

    # Args:
        ∇f: Gradient of function to optimize
        x_0: Initial guess for optimal value
        iters: Maximum number of iterations before cancellation
        line_search: Line search function that's used
        ε: Optimization stops once the norm of the gradient is below this value
    """
    function BFGS(
      f::Function,
      ∇f::Function,
      x_0::Vector{<:Number},
      iters::Integer,
      line_search::Function,
      ε = 10e-12::Real)
        n = size(x_0)[1]
        x_k = x_0
        B_k = Matrix{typeof(x_0[1])}(I, n, n)

        for i = 1:iters
            if norm(∇f(x_k)) < ε
                break
            end
            # Step 1: obtain direction p_k by solving
            # B_k · p_k = - (gradient of f at x_k)
            p_k = B_k \ -∇f(x_k)
            # Step 2.: Find stepsize α_k such that
            # α_k = arg min ∂f(x_k + α_k * p_k)/∂α
            α_k = line_search(numeric_differentiation(α→(f(x_k + α * p_k)), 10e-10))
            # Step 3.
            s_k = α_k * p_k
            x_k_prime = x_k + s_k
            # Step 4.
            y_k = ∇f(x_k_prime) - ∇f(x_k)
            # Step 5.
            B_k += (y_k * y_k') / (y_k' * s_k) +
              - (B_k * s_k * s_k' * B_k) / (s_k' * B_k * s_k)
            x_k = x_k_prime
        end
        x_k
    end
```

Betrachten wir nun beispielsweise die Funktion $f(x) = \frac{x^5}{5000} + \frac{21x^4}{4000} + \frac{17x^3}{375} + \frac{293x^2}{1000} + \frac{521x}{1000}$ mit $\nabla f(x) = \frac{x^4}{1000} + \frac{21x^3}{1000} + \frac{17x^2}{125} + \frac{293x}{500} + \frac{521}{1000}$ und führen mit ihr BFGS aus `BFGS(f, ∇f, [0], 500, ∇f → gradient_descent(∇f, 10e-10, 1, 1e-10, 1000), 10e-5)`, erhalten wir nach zwei Iterationen ein lokales Minimum bei $x = -1.1408$.

Eine recht simple Optimierung der Implementierung ist in Schritt 1 möglich. Hier wird aktuell der \-Operator[13] eingesetzt. Dies ist zwar eine valide Lösung, jedoch nicht besonders effizient. Stattdessen kann hier nach [Wik19] die Sherman–Morrison–Woodbury Formel angewandt werden um die Inverse rekursiv zu

[13] `https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/`

beschreiben. Die Rekursionsvorschrift lautet:

$$B_{k+1}^{-1} = B_k^{-1} + \frac{(\mathbf{s}_k^T \mathbf{y}_k + \mathbf{y}_k^T B_k^{-1} \mathbf{y}_k)(\mathbf{s}_k \mathbf{s}_k^T)}{(\mathbf{s}_k^T \mathbf{y}_k)^2} - \frac{B_k^{-1} \mathbf{y}_k \mathbf{s}_k^T + \mathbf{s}_k \mathbf{y}_k^T B_k^{-1}}{\mathbf{s}_k^T \mathbf{y}_k}.$$

Für die erste Iteration gilt $B_1 = I_n$, und daher auch $p_1 = -\nabla f(\mathbf{x}_k)$ bzw. $B_1^{-1} = I_n$.

Mit dem fertigen BFGS kann man nun `fit_linear_model` dahingehend erweitern/ändern, dass es intern BFGS nutzt. Außerdem öffnet BFGS die Tür zu anderen Verfahren wie z.B. logistischer Regression.

A Performancemessung

Das Laufzeitverhalten wurde mittels des built-in Macros `@time` gemessen. Testsystem war ein Intel i7-4790k@3.8GHz unter Linux Mint 19. Auswertung der Ausgabe des Macros erfolgte mittels des Python Scripts in Listing 12. Die Tabellenwerte ergeben sich als arithmetisches Mittel von 100 Aufrufen der Funktion `fit_linear_model`.

Listing 12 Python Script zur `@time` Auswertung

```python
from statistics import mean
import re

"""Regex to capture different components
Examples:
  0.000103 seconds (818 allocations: 87.328 KiB)
  0.000388 seconds (3.78 k allocations: 412.406 KiB)
  0.598357 seconds (2.02 M allocations: 102.555 MiB, 3.13% gc time)
"""
RE = r"\s*(?P<time>\d+\.\d*) seconds \((?:(?P<allocs>\d+(?:\.\d*)?)"\
    r"(?: (?P<allocsuffix>M|k))?) allocations: (?:(?P<mem>\d+\.\d*)"\
    r"(?: (?P<memsuffix>(?:KiB|MiB)))?)(?:, (?P<gctime>\d+\.\d*)% gc time)?\)"

MEMORY_FACTOR = {None: 1, "MiB": 1, "KiB": 1 / 1024}


def get_and_parse():
    try:
        while line := input():
            match = re.match(RE, line)
            time = float(match["time"])
            mem = float(match["mem"]) * MEMORY_FACTOR[match["memsuffix"]]
            if (gc_time:=match["gctime"]) is not None:
                gc = float(gc_time)
            else:
                gc = None
            yield (time, mem, gc)
    except EOFError: return None


time, memory, gc_time = zip(*(x for x in get_and_parse()))
gc_time = list(filter(lambda x: x is not None, gc_time))
print(
    f"Time: {mean(time):.5f} s, RAM: {mean(memory):.3f} MiB, "\
    f"GC-time: {mean(gc_time):.2f} % (got {len(gc_time)} gc values)")
```

Das Script wurde für Python 3.8.0 geschrieben und ist auch nicht in älteren Versionen lauffähig(hierzu müsste man die Assignment Expressions ersetzen). Der Code gestaltet sich etwas komplizierter, da `@time` je nach Anzahl der Allokationen ein verschiedenes Ausgabeformat wählt. Daher seien spezielle Sprachfeatures und nicht-triviale Codepassagen hier kurz erläutert:

Hintereinanderstehende Strings werden in Python automatisch verkettet - \ erlaubt es lange Zeilen umzubrechen. Der `:=` Operator führt eine *Assignment Expression*[14] aus. Diese funktionieren ähnlich einer Zuweisung in C oder auch ALGOL 68; in Kombination mit **`while`** ergibt sich ein Konstrukt ähnlich einem **`while let`** `Some(x) = ...` in Rust. Die Funktion `get_and_parse` fungiert als Generator[15] welcher über

[14] https://www.python.org/dev/peps/pep-0572/
[15] https://docs.python.org/3/howto/functional.html

stdin die Messwerte einliest, parset und als Dreiertupel zurückgibt. Für eine zweidimensionale Collection `c` in row-major order wandelt `zip(*c)` diese in eine in column-major order um. Bei `RE` handelt es sich um eine Regular Expression[16] welche in diesem Fall alle Eingabeformate unterscheiden und die einzelnen Komponenten automatisch extrahieren kann. Bei `(x for x in get_and_parse())` in Zeile `31` handelt es sich um eine *Generator Expression*[17] welche ähnlich zu einer List Comprehension in Python, Haskell, Erlang oder auch Julia funktioniert - jedoch im Effekt eine Art Lazy Evaluation realisiert. Diese Generator Expression wird hier nur benötigt um das Entpacken mittels `*` zu ermöglichen. Die Konvertierung des Generators zu einer Liste in Zeile 32 ist leider nötig (sofern man keine eigene Funktion schreibt, welche gleichzeitig die Länge bestimmt und das Mittel berechnet) da der Generator beim Aufruf von `len` oder `mean` sonst verbraucht würde. Die Nutzung erfolgt dann beispielsweise mit `$ julia Tests.jl | python3.8 measure.py`.

Literatur

[Bis09] Bishop, Christopher M.: *Pattern recognition and machine learning*. Springer-Verlag, 2009. – ISBN 978–1–4939–3843–8

[Dai13] Dai, Yu-Hong: A perfect example for the BFGS method. In: *Mathematical Programming* 138 (2013), Nr. 1, 501-530. http://dx.doi.org/10.1007/s10107-012-0522-2. – DOI 10.1007/s10107–012–0522–2. – ISSN 1436–4646

[Lip06] Lippe, Wolfram-Manfred: *Soft Computing*. Springer-Verlag, 2006. – ISBN 978–3–540–20972–0

[Wik19] Wikipedia contributors: *Broyden–Fletcher–Goldfarb–Shanno algorithm — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Broyden%E2%80%93Fletcher%E2%80%93Goldfarb%E2%80%93Shanno_algorithm&oldid=932530238. Version: 2019. – [Online; abgerufen 31.12.2019]

[16] https://docs.python.org/3/library/re.html
[17] https://www.python.org/dev/peps/pep-0289/