

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 als Funktion höherer Ordnung, welche mittels Gradientenabstieg eine Fehlerminimierung des Modells im Bezug auf gegebene Messdaten durchführt. Die Implementierung erfolgt in Julia.

N-DIMENSIONALES DATENFITTING MIT LINEARER REGRESSION

Stefan Volz

19. Dezember 2019

1 Konventionen und Generelles

1.1 Wahl der Programmiersprache und Abhängigkeiten

Die Implementierung erfolgt in Julia¹. 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` Package².

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.

1.2 Quellcode

Beim Code wurde darauf geachtet, die Typannotationen³ 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 `?` decken - teils wurden auch Bezeichner aus `?` übernommen.

1.3 Text dieser Arbeit

Im Text der Arbeit werden - in Anlehnung an `?` - folgende Konventionen genutzt:

Typ	Beschreibung	Beispiel
vektorielle Größen	fettgedruckte Kleinbuchstaben	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>code</code>

¹<https://julialang.org/>

²<https://docs.juliaplots.org/>

³<https://docs.julialang.org/en/v1/manual/types/>

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 [?] von diesen Konventionen abgewichen.

1.4 Variablenbezeichner

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

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 $y \in \mathbb{R}^M \times \mathbb{R}^d \rightarrow \mathbb{R}^k$ 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 $E : \mathbb{R}^k \times \mathbb{R}^k \rightarrow \mathbb{R}$ 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[?, S. 137f]. Sie stellen Funktionen der Form $y : \mathbb{R}^M \times \mathbb{R}^d \rightarrow \mathbb{R}^k, (\mathbf{w}, \mathbf{x}) \mapsto y(\mathbf{w}, \mathbf{x})$ 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

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

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 Regressionsmodelle als Linearkombination dieser gebildet.

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

Anmerkung 2.4. Im Code werden Basisfunktionen als Funktionen $\Phi : \mathbb{N} \times \mathbb{R}^d \rightarrow \mathbb{R}^k, (j, \mathbf{x}) \mapsto \Phi(j, \mathbf{x})$ 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} 1 wählen. Ein Eingangsvariablenunabhängiger Offset ist leicht durch eine angepasste Definition der Basisfunktion erreichbar:

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

Damit vereinfacht sich die Darstellung der Modellgleichung zu

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

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})::(T where T <: 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 1⁴.

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 [?, S. 140f] ergibt sich die Fehlerfunktion

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

⁴Die 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.

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 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} .

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

Hieraus folgt dann für den Gradienten:

$$\nabla_{\mathbf{w}} E_D = \begin{pmatrix} \frac{\partial E_D}{\partial w_1} \\ \vdots \\ \frac{\partial E_D}{\partial w_M} \end{pmatrix} =: \begin{pmatrix} e_1 \\ \vdots \\ e_M \end{pmatrix}.$$

Für die komponentenweise Berechnung dieses Gradienten ergibt sich nach [?, S. 95f]

$$\frac{1}{M} \text{ppuuuuuuuuuuuuuh}$$

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

Listing 3 Funktion $\partial E_D \partial w_k$

```
"""Derivative of E_D with respect to w_k
# Args:
    Phi(k, x_n): Basis function
    X: Set of inputs x_n where x_n is an input vector to Phi
    t: corresponding target values for each x_n
    k: Index for w_k in respect to which the derivative is taken
    w: Parameters
"""
function partial_E_D_w_k(Phi, X, t, w, k)
    N = size(t)[1]
    - sum(1:N, n->Phi(k, X[n]) * (t[n] - y(w, Phi, X[n])))
end
```

2.4.2 Einzeliteration des Gradientenabstiegs

Hiermit können wir eine Iteration des Gradientenabstiegsalgorithmus wie folgt implementieren.

Innerhalb der Funktion iterieren wir über alle Indizes j , bilden die partielle Ableitung nach w_k 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.

Listing 4 Funktion `gd_iteration`

```

"""Gradient descent iteration
# Args:
    Φ: Basis Function
    X: Set of inputs  $x_n$  where  $x_n$  is an input vector to  $\Phi$ 
    t: corresponding target values for each  $x_n$ 
    w: Parameters
    η: Learning rate
"""
function gd_iteration(Φ, X, t, w::Vector{<:Number}, η)
    M = size(w)[1]
    ∇w = zero(w)
    for j = 1:M
        ∂E_D∂w_jk(k) = ∂E_D∂w_k(Φ, X, t, w, k)
        ∇w += collect(map(∂E_D∂w_jk, 1:M))
    end
    w = w - η * ∇w
end

```

Listing 5 Funktion `gd`

```

"""Find regression model using gradient descent
# Args:
    Φ: Basis Function
    X: Set of inputs  $x_n$  where  $x_n$  is an input vector to  $\Phi$ 
    t: corresponding target values for each  $x_n$ 
    η: learning rate with which to train
    M: Number of model parameters
    iters: Number of iterations
"""
function gd(Φ, X, t, η, M, iters)
    w = randn(M)
    for i = 1:iters
        w = gd_iteration(Φ, X, t, w, η, ∇w)
    end
    x→y(w, Φ, x)
end

```

2.4.3 Hauptfunktion

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

Diese Funktion initialisiert die Modellparameter zu Beginn mit einem Vektor aus normalverteilten Zufallszahlen[?] aus dem Intervall $[-1, 1]$. Dann ruft sie `iters`-mal die Hilfsfunktion `gd_iteration` auf und updated die Parameter mit dem Ergebnis dieser Aufrufe. Schließlich gibt sie das vollendete Modell als Closure⁵ zurück, was dem die Funktion Aufrufenden ermöglicht, Modellaussagen in Abhängigkeit eines Eingangsvektors zu erhalten.

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 Iteratio-

⁵Eine Closure ist eine anonyme Funktion welche intern eine Referenz auf ihren Erstellungskontext hält. Hier wird sie eingesetzt um y mit w und Φ partiell zu evaluieren.

nen 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 NaN⁶ in \mathbf{w} aufgetaucht ist. In all diesen Fällen wird der Algorithmus vorzeitig abgebrochen und das Modell zurückgegeben⁷.

Listing 6 Funktion gd mit einfachem Abbruchkriterium

```

"""Find regression model using gradient descent
# Args:
    Φ: Basis Function
    X: Set of inputs  $\mathbf{x}_n$  where  $\mathbf{x}_n$  is an input vector to Φ
    t: corresponding target values for each  $\mathbf{x}_n$ 
    η: learning rate with which to train
    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
"""
function gd(Φ, X, t, η, M, iters, ε=10e-12)
    w = randn(M)
    for i = 1:iters
        w_old = w
        w = gd_iteration(Φ, X, t, w, η, ∇w, γ)
        if norm(w_old - w) < ε || any(isnan.(w)) || any(isinf.(w))
            break
        end
    end
    println(w)
    x→y(w, Φ, x)
end

```

Für den Parameter ε wird eine arbiträr gewählte Standardbelegung von $10 \cdot 10^{-12}$ gesetzt.

3 Fortgeschrittene Features

3.1 Momentum Verfahren/Konjugierter Gradientenabstieg

Das Momentum Verfahren ist eine Adaption des Gradientenabstiegs, welche die Konvergenzgeschwindigkeit erhöhen soll. Das Ziel ist es, auf flachen Bereichen der Fehlerkurve die Abstiegs geschwindigkeit 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 muss auch die call-site in `gd` angepasst werden.

Nach [?, S. 110] wird γ mit einem Wert von 0.9 vorbelegt. Die wohl größte Problematik des konjugierten Gradientenabstiegs ist wohl, dass der Fall auftreten kann, dass der Momentum-Term betragsmäßig größer

⁶Not a Number - spezieller Wert von IEEE floats

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

Listing 7 Funktionen `gd_iteration` und `gd` mit konjugiertem Gradientenabstieg

```

"""Gradient descent iteration
# Args:
    Φ: Basis Function
    X: Set of inputs  $x_n$  where  $x_n$  is an input vector to  $\Phi$ 
    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 gd_iteration(Φ, X, t, w::Vector{<:Number}, η, ∇w_prior, γ)
    M = size(w)[1]
    ∇w = γ * ∇w_prior
    for j = 1:M
        ∂E_D∂w_jk(k) = ∂E_D∂w_k(Φ, X, t, w, k)
        ∇w += collect(map(∂E_D∂w_jk, 1:M))
    end
    (w - η * ∇w, ∇w)
end

"""Find regression model using gradient descent
# Args:
    Φ: Basis Function
    X: Set of inputs  $x_n$  where  $x_n$  is an input vector to  $\Phi$ 
    t: corresponding target values for each  $x_n$ 
    η: learning rate with which to train
    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
    γ: Momentum Parameter
"""
function gd(Φ, X, t, η, M, iters, ε = 10e-12, γ = 0.9)
    w = randn(M)
    ∇w = zero(w)
    for i = 1:iters
        w_old = w
        (w, ∇w) = gd_iteration(Φ, X, t, w, η, ∇w, γ)
        if norm(w_old - w) < ε || any(isnan.(w)) || any(isinf.(w))
            break
        end
    end
    x→y(w, Φ, x)
end

```

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 (mit konstanter Lernrate und $\varepsilon = 10E - 12$) mit nur wenigen Datenpunkten zeigt folgende Daten:

γ	k bei Abbruch nach k Iterationen	Restfehler
0.0	50544	6.701991676725787
0.5	28837	6.701991676725787
0.9	5635	6.701991676725783

Also gibt es durchaus Fälle, in denen durch dieses Verfahren eine enorme Steigerung bei der Konvergenzgeschwindigkeit erzielt wird.

3.2 Adaptive Lernrate

3.3 Weight Decay

4 Beispiele