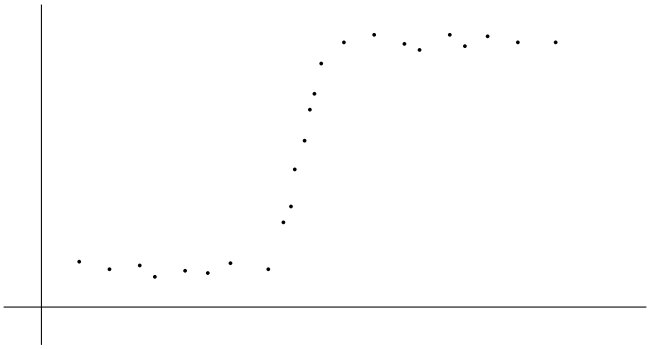

Abbildung 8.1: Eingabedaten für segmentierte lineare Regression

8.2 Segmentierte Methode der kleinsten Quadrate

Bei der einfachen linearen Regression in der Statistik sind Punkte $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n) \in \mathbb{R}^2$ gegeben die z.B. aus einem Experiment stammen und wir wollen die (Ausgabe-)werte y_i durch die (Eingabe-)werte x_i erklären wobei wir annehmen, dass ein linearer Zusammenhang besteht. Wir wollen also eine Gerade $y = ax + b$ bestimmen welche die Punkte $p_1, \ldots, p_n$ bestmöglich im Sinne der kleinsten Fehlerquadrate annähert, d.h. es sollen $a, b \in \mathbb{R}$ bestimmt werden so dass die *Fehlerquadratsumme*

$$\sum_{i=1}^{n} (y_i - ax_i - b)^2$$

minimiert wird. Man kann zeigen dass diese Lösung eindeutig ist und gegeben ist durch

$$a = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})} \quad \text{und} \quad b = \bar{y} - a\bar{x}.$$

wobei $\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$ und $\bar{y} = \frac{\sum_{i=1}^{n} y_i}{n}$.
Oft können die gegebenen Daten aber durch eine einzelne Gerade nicht sehr gut angenähert werden, siehe Abbildung 8.1. Eine Reaktion darauf, die wir hier verfolgen wollen, besteht darin, die Daten stückweise durch Geraden anzunähern. Dabei muss nun natürlich spezifiziert werden was minimiert werden soll, d.h. wie die Anzahl der Geraden gegen die Fehlerquadratsumme gewichtet werden soll. Eine Möglichkeit dies zu tun besteht darin, eine Konstante $A > 0$ festzulegen, welche die Kosten der Verwendung einer (zusätzlichen) Gerade darstellt. Dann werden die Punkte $p_1, \ldots, p_n$ entlang der x-Achse in Segmente geteilt von denen jedes unabhängig durch eine Gerade approximiert wird. Je nach Wert von A fällt die Anzahl der Segmente mehr oder weniger stark ins Gewicht. Wir kommen so zum folgenden Optimierungsproblem

Segmentierte einfache lineare Regression

Eingabe: $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n) \in \mathbb{R}^2$ mit $x_1 < \ldots < x_n$ und $A > 0$

Ausgabe: $0 = n_0 < n_1 < \cdots < n_{k-1} < n_k = n$ und für alle $i = 1, \ldots, k$ eine Gerade $y = a_i x + b_i$ so dass

$$\sum_{i=1}^{k} \left(A + \sum_{j=n_{i-1}+1}^{n_i} (y_j - a_i x_j - b_i)^2 \right)$$

minimal ist

Auch dieses Problem eignet sich für einen Ansatz mit dynamischem Programmieren. Für $j = 1, \ldots, n$ seien c_j die Kosten der optimalen Lösung des Teilproblems $p_1, \ldots, p_j$. Dann setzt sich c_j zusammen aus den Kosten des letzten Segments $p_i, \ldots, p_j$ sowie den Kosten der optimalen Lösung des Teilproblems $p_1, \ldots, p_{i-1}$ und den Kosten der Verwendung einer Geraden. Wir erhalten also

$$c_j = \min\{e_{i,j} + A + c_{i-1} \mid 1 \leq i \leq j\}$$

wobei $e_{i,j}$ die Fehlerquadratsumme des Segments $p_i, \ldots, p_j$ ist. Außerdem sei $c_0 = 0$, womit der Fall dass $p_1, \ldots, p_j$ nicht mehr unterteilt wird abgedeckt wird durch $c_j = e_{1,j} + A + c_0 = e_{1,j} + A$. Ähnlich wie bei der Stabzerlegung bildet diese Beobachtung nun die Basis für einen Algorithmus der die mehrfache Betrachtung von Teilproblemen vermeidet, siehe Algorithmus 35. Für ein

Algorithmus 35 Segmentierte Methode der kleinsten Quadrate

```
Prozedur SegmentierteMKQ(P, n, A)
    Sei E[1,...,n; 1,...,n] ein neues Datenfeld
    Für alle 1 ≤ i ≤ j ≤ n
        E[i,j] := kleinste Fehlerquadratsumme für das Segment p_i,...,p_j
    Ende Für

    Sei C[0,...,n] ein neues Datenfeld
    Sei S[1,...,n] ein neues Datenfeld
    C[0] := 0
    Für j = 1,...,n
        m := +∞
        Für i = 1,...,j
            Falls E[i,j] + A + C[i-1] < m dann
                m := E[i,j] + A + C[i-1]
                S[j] := i
            Ende Falls
        Ende Für
        C[j] := m
    Ende Für
    Antworte (C[n], S)
Ende Prozedur
```

Segment von k Punkten kann die Gerade mit minimaler Fehlerquadratsumme in Zeit $O(k)$ über die oben angegebenen Formeln bestimmt werden. Damit kann die kleinste Fehlerquadratsumme dieser Gerade in Zeit $O(k)$ bestimmt werden. Auf diese Weise alle $E[i, j]$ mit $i \leq j \leq n$ zu

berechnen benötigt also Zeit $O(n^3)$. Die beiden verschachtelten Schleifen im zweiten Teil des Algorithmus benötigen Zeit $O(n^2)$. Wir erhalten also insgesamt eine Laufzeit von $O(n^3)$. Die Laufzeit dieses Algorithmus kann durch eine geschicktere Behandlung der ersten Phase auf $O(n^2)$ reduziert werden. Da dies aber nicht mehr zur Diskussion der dynamischen Programmierung beiträgt wollen wir hier darauf verzichten.

Kapitel 9

Randomisierung

Unter einem *randomisierten Algorithmus* versteht man einen Algorithmus der gewisse Entscheidung basierend auf (aus externer Quelle erhaltenen) Zufallszahlen macht[1]. Das kann zu verschiedenen Zwecken nützlich sein. Einerseits kann man die Eingabedaten oder das Verhalten des Algorithmus randomisieren und so sicherstellen, dass ein Algorithmus auch im schlechtesten Fall nur die Komplexität des durchschnittlichen Falls hat. Dafür hat der Algorithmus dann auch im besten Fall die Komplexität des durchschnittlichen Falls. Je nach Laufzeit im schlechtesten, besten und durchschnittlichen Fall und je nach Verteilung der Eingabedaten ist das mehr oder weniger nützlich. Andererseits kann Randomisierung in einem Algorithmus auch verwendet werden, um schnell ein Resultat zu liefern, das aber nur mit gewisser Wahrscheinlichkeit < 1 korrekt ist. Eine Wiederholung (mit unabhängigen Zufallszahlen) erlaubt dann eine beliebige Annäherung der Irrtumswahrscheinlichkeit an 0.

9.1 Randomisierung der Eingabe

In Hinblick auf ein einfaches Beispiel für die Randomisierung der Eingabe wollen wir das folgende *Bewerberproblem* betrachten. Eine Abfolge von Bewerbern absolviert Vorstellungsgespräche für eine Stelle. Einer hire&fire-Mentalität folgend soll zu jedem Zeitpunkt, d.h. auch zwischen den Bewerbungsgesprächen, der beste Bewerber die Stelle halten. Wenn ein besserer Bewerber als der aktuelle Stelleninhaber gefunden wird, wird dieser durch jenen ersetzt. Wir gehen davon aus dass die Bewerber mit $k_1, \ldots, k_n \in \mathbb{R}_{>0}$ beurteilt werden. Die natürliche Vorgehensweise ist in Algorithmus 36 als Pseudocode ausformuliert. Wir gehen dabei davon aus dass $K[i]$ den Wert k_i enthält und dass $K[0] = 0$ ist. Für die Analyse wollen wir davon ausgehen, dass die Einstellung eines Kandidaten hohe Kosten verursacht. Wir interessieren uns also dafür, wie oft Zeile 5 in dieser Prozedur ausgeführt wird. Eine Vorgehensweise wie in Algorithmus 36 zur Suche eines Maximums oder Minimums kommt oft als Teil anderer Algorithmen vor.

Lemma 9.1. *Die Anzahl der Neuanstellungen ist im schlechtesten Fall n und im durchschnittlichen Fall $O(\log n)$.*

Beweis. Der schlechteste Fall tritt ein wenn $k_1 < k_2 < \cdots < k_n$. Dann werden n Neuanstellungen durchgeführt, eine für jeden Kandidaten.

[1] Das ist nicht zu verwechseln mit einem nicht-deterministischen Algorithmus der, in gewissem Sinn, verschiedene deterministische Berechnungen gleichzeitig macht.

Algorithmus 36 Direkte Lösung des Bewerberproblems

```
1: Prozedur BewerbersuChe(K, n)
2:     m := 0
3:     Für i := 1, ... n
4:         Falls K[i] > K[m] dann          ▷ Vorstellungsgespräch für Kandidat i
5:             m := i                      ▷ Einstellen von Kandidat i
6:         Ende Falls
7:     Ende Für
8:     Antworte m
9: Ende Prozedur
```

Für den durchschnittlichen Fall gehen wir davon aus, dass jede Eingabepermutation der k_i gleich wahrscheinlich ist. Sei X die Anzahl der Neuanstellungen und sei

$$X_i = \begin{cases} 1 & \text{falls der } i\text{-te Kandidat eingestellt wird und} \\ 0 & \text{falls der } i\text{-te Kandidat nicht eingestellt wird.} \end{cases}$$

Dann ist $\mathrm{E}X_i = \mathrm{W}(X_i = 1)$. Da der i-te Kandidat eingestellt wird genau dann wenn $k_i = \max\{k_1, \ldots, k_i\}$, ist nach Lemma 2.1 die Wahrscheinlichkeit $\mathrm{W}(X_i = 1) = \frac{1}{i}$. Somit erhalten wir

$$\mathrm{E}X = \mathrm{E}\sum_{i=1}^{n} X_i = \sum_{i=1}^{n} \mathrm{E}X_i = \sum_{i=1}^{n} \frac{1}{i} = H_n = \ln n + O(1).$$

□

Wir haben also einen Algorithmus der im schlechtesten Fall deutlich höhere Kosten verursacht als im durchschnittlichen Fall. Um die Laufzeit des durchschnittlichen Falls zu garantieren, können wir die Eingabedaten vorverarbeiten indem wir auf sie eine (uniform verteilte) Zufallspermutation anwenden. Sei ZufallspermutatIon(n) eine solche uniform verteilte Zufallspermutation von n Elementen. Wir erhalten dann den in Algorithmus 37 angegebenen Pseudocode. Der Erwartungswert der Anzahl der Neuanstellungen wird hier über die Verteilung der Eingabe

Algorithmus 37 Randomisierte Lösung des Bewerberproblems

```
Prozedur BewerbersuCheRandomisiert(K, n)
    K := ZufallspermutatIon(K)
    Antworte BewerbersuChe(K, n)
Ende Prozedur
```

und der Zufallspermutation gebildet und ist damit unabhängig von der Eingabe $O(\log n)$. Die systematische Konstruktion einer Eingabe mit schlechtem Verhalten ist also nicht mehr möglich. In diesem Sinn gibt es keine worst-case Eingabe mehr. Der Nachteil von Algorithmus 37 ist, dass es in diesem Sinn auch keine best-case Eingabe mehr gibt. Wenn wir also erwarten sehr häufig eine annähernd absteigend sortierte Eingabe zu erhalten, wird Algorithmus 36 vorzuziehen sein.

Eine vergleichbare Situation trat auch in Abschnitt 5.2 auf. Das auf (nicht-balancierten) Suchbäumen basierende Sortierverfahren hatte eine Laufzeit von $O(n^2)$ im schlechtesten Fall aber $O(n \log n)$ im Durchschnittsfall. Durch eine Zufallspermutation der Eingabe kann dieses in ein randomisiertes Verfahren umgewandelt werden dass auf beliebiger Eingabe eine erwartete Laufzeit von $O(n \log n)$ hat.

Wir wollen uns jetzt damit beschäftigen wie man eine Zufallspermutation erzeugen kann. Klar ist, dass wir irgendeine Art von Zufallsquelle voraussetzen müssen. Wir nehmen also die Verfügbarkeit einer Prozedur ZUFALL(i, j) an, die eine uniform verteilte Zufallszahl im Intervall $[i, j] \subseteq \mathbb{N}$ liefert. Viele Programmiersprachen bzw. Betriebsysteme stellen tatsächlich solche Funktionen zur Verfügung, die als Zufallsquelle verschiedene Umgebungsdaten oder einen Pseudozufallszahlen-Generator verwenden.

Ein geeigneter Ansatz zur Berechnung einer zufälligen Permutation eines Datenfelds A der Länge n besteht darin, für $i = 1, \ldots, n$ das Element $A[i]$ mit einem zufälligen Element in $A[i, \ldots, n]$ zu vertauschen. Das ist der Algorithmus von Fisher-Yates, als Pseudocode ausformuliert in Algorithmus 38. Dieser Algorithmus hat Laufzeit $\Theta(n)$ (unter der Annahme dass ZUFALL(i, j) konstante Zeit benötigt). Klar ist, dass der Algorithmus von Fisher-Yates eine Permutation des Eingabdatenfelds berechnet, da er nur Vertauschungen durchführt. Für die Analyse der Wahrscheinlichkeitsverteilung der erzeugten Permutation ist noch etwas Arbeit nötig.

Algorithmus 38 Der Fisher-Yates Algorithmus

```
Prozedur FisherYates(A)
    Für i = 1, ..., A.Länge
        Vertausche A[i] und A[Zufall(i, n)]
    Ende Für
    Antworte A
Ende Prozedur
```

Satz 9.1. *Der Fisher-Yates Algorithmus erzeugt eine uniform verteilte Zufallspermutation.*

Beweis. Sei $n = A.Länge$ und für $i = 0, 1, \ldots, n$ sei A_i das Datenfeld nach dem i-ten Durchlauf der Schleife. Dann ist A_0 das Eingabedatenfeld. Sei o.B.d.A. $A_0[i] = i$ für $i = 1, \ldots, n$. Sei V_k^n die Menge aller Variationen von k aus n Elementen ohne Wiederholung. Dann ist $|V_k^n| = \frac{n!}{(n-k)!}$. Wir zeigen mit Induktion nach $i = 1, \ldots, n$ dass für alle $v \in V_i^n$: $\mathrm{W}(A_i[1, \ldots, i] = v) = \frac{1}{|V_i^n|} = \frac{(n-i)!}{n!}$. Für $i = n$ folgt daraus dass das Ausgabedatenfeld A_n eine uniform verteilte Permutation ist. Die Induktionsbasis $i = 1$ folgt direkt aus Definition des Algorithmus. Für den Induktionsschritt sei $w \in V_{i+1}^n$, dann ist $w = \langle v, x \rangle$ wobei $v \in V_i^n$ und $x \in \{1, \ldots, n\} \setminus v$. Dann ist

$$\mathrm{W}(A_{i+1}[1, \ldots, i+1] = w) = \mathrm{W}(A_i[1, \ldots, i] = v \text{ und } A_{i+1}[i+1] = x)$$

und da $\{1, \ldots, n\} \setminus v$ genau die Elemente von $A_i[i+1, \ldots, n]$ sind ist $\mathrm{W}(A_{i+1}[i+1] = x) = \frac{1}{n-i}$

$$= \frac{(n-i)!}{n!} \frac{1}{n-i} = \frac{(n-(i+1))!}{n!}.$$

□

9.2 Quicksort

Quicksort ist einer der schnellsten Sortieralgorithmen und wird deshalb in der Praxis oft verwendet. Der Algorithmus folgt dem Teile-und-Herrsche Prinzip. Das Eingabedatenfeld $A[1, \ldots, n]$ wird zunächst durch Vertauschungen von Elementen in zwei Teile geteilt: $A[1, \ldots, m-1]$ und $A[m+1, \ldots, n]$ so dass alle Elemente im linken Teil kleiner gleich $A[m]$ sind und alle Elemente im rechten Teil größer gleich $A[m]$ sind. Dann wird die Prozedur rekursiv auf den beiden Teilen aufgerufen. Insgesamt wird so das Datenfeld durch Vertauschungen sortiert.

Die Aufteilung des Datenfelds $A[l, \ldots, r]$ wird so realisiert, dass zunächst ein *Pivotelement* aus dem Datenfeld gewählt wird, z.B. $A[r]$. Danach wird das Datenfeld von links nach rechts durchlaufen wobei der jeweils bisher durchlaufene Teil aus zwei Teilen besteht: jene Elemente die größer sind als $A[r]$ und jene die kleiner sind als $A[r]$. Am Ende wird das Pivotelement $A[r]$ an die richtige Stelle permutiert. Diese Vorgehensweise ist in Algorithmus 39 ausformuliert.

Algorithmus 39 Quicksort

```
Prozedur QUICKSORT(A, l, r)
    Falls l < r dann
        m := TEILEN(A, l, r)
        QUICKSORT(A, l, m − 1)
        QUICKSORT(A, m + 1, r)
    Ende Falls
Ende Prozedur

Prozedur TEILEN(A, l, r)
    x := A[r]
    i := l
    Für j = l, . . . , r − 1
        Falls A[j] ≤ x dann
            Vertausche A[i] mit A[j]
            i := i + 1
        Ende Falls
    Ende Für
    Vertausche A[i] mit A[r]
    Antworte i
Ende Prozedur
```

Die Prozedur TEILEN wählt zunächst als Pivotelement $x = A[r]$. Zu Beginn jedes Schleifendurchlaufs gilt für das Datenfeld A: falls $k \in \{l, \ldots, i-1\}$ ist $A[k] \leq x$, falls $k \in \{i, \ldots, j-1\}$ ist $A[k] > x$ und falls $k \in \{j, \ldots, r\}$ wurde $A[k]$ noch nicht mit x verglichen. In jedem Schleifendurchlauf wird j um eins erhöht und gegebenenfalls durch eine Vertauschung diese Invariante beibehalten.

Beispiel 9.1. Die Prozedur TEILEN angewandt auf das Datenfeld

3	6	2	7	4	1	8	5

transformiert dieses in

3	2	4	1	5	7	8	6

und antwortet mit $i = 5$.

Die Laufzeit der Prozedur TEILEN ist $\Theta(r - l)$. Die Gesamtlaufzeit von Quicksort hängt also davon ab, wie die Aufteilungen des Datenfeldes stattfinden. Intuitiv ist eine Aufteilung umso besser je schneller die Größe der Instanzen schrumpft. Tatsächlich kann man sich leicht

überlegen, dass im Fall einer aufsteigend sortierten Eingabe ein Datenfeld der Länge n aufgespalten wird in eines der Länge $n-1$, in das Pivotelement, sowie eines der Größe 0. Für diesen Fall ist die Laufzeit von Quicksort also gegeben durch

$$T(n) = T(n-1) + \Theta(n).$$

Durch die Substitutionsmethode kann leicht gezeigt werden dass $T(n) = \Theta(n^2)$. Die Laufzeit von Quicksort im schlechtesten Fall ist also $\Omega(n^2)$.

Der obigen Intuition folgend kann man sich vorstellen, dass die beste Aufteilung eines Datenfelds der Länge n jene ist die zwei Datenfelder der selben Länge erzeugt. Die Laufzeit wird dann (modulo Rundung) beschrieben durch die Rekursionsgleichung

$$T(n) = 2T(\frac{n}{2}) + \Theta(n).$$

Nach dem zweiten Fall des Master-Theorems ergibt sich also $T(n) = \Theta(n \log n)$.

Die Laufzeit ergibt sich aber selbst dann zu $O(n \log n)$ wenn eine unbalancierte Aufteilung gewählt wird, so lange die Längen der Teildatenfelder durch eine multiplikative Konstante und die ursprüngliche Länge bestimmt werden. Wird zum Beispiel in jedem Schritt eine 1-zu-9 Aufteilung gewählt, dann erhalten wir (modulo Rundung) die Rekursionsgleichung

$$T(n) = T(\frac{1}{10}n) + T(\frac{9}{10}n) + cn.$$

Im Rekursionsbaum hat dann jede Ebene Kosten $\leq cn$ und es gibt $O(\log_{\frac{10}{9}} n)$ Ebenen, insgesamt haben wir also Kosten von $O(n \log_{\frac{10}{9}} n) = O(n \log n)$.