# Bayesscher Anhang zu **Parametrische Statistik**

–

# Verteilungen, *maximum likelihood* und GLM in R

Carsten F. Dormann

Biometrie & Umweltsystemanalyse
Universität Freiburg

7. Januar 2019

# Inhaltsverzeichnis

# Anhang A

# Bayessche Statistik – kurz und bündig

*Everything you can do, I can do Bayesian!*

## A.1 Was ist Bayessche Statistik?

Bislang haben wir die parametrische Statistik aus einem Blickwinkel kennengelernt, der „frequentistisch“ genannt wird.[1] Diese Herangehensweise wurde von R.A. Fisher erfunden, um das „Bayessche Problem“ zu lösen. Um dieses Problem zu verstehen, müssen wir uns an die Mengenlehre erinnern.

### A.1.1 Der Satz von Bayes

Bayessche Statistik geht auf eine sehr einfache Gleichung zurück, die der Mathematiker Thomas Bayes vor rund 250 Jahren nahelegte.[2] Wenn wir zwei Ereignisse $A$ und $B$ haben, dann können wir die Wahrscheinlichkeit, dass $A$ und $B$ eintreten aus dem bedingten Eintreten von $A$ gegeben $B$ errechnen:

$$P(A \cap B) = P(A|B)P(B). \tag{A.1}$$

In Worten: Die Wahrscheinlichkeit, dass $A$ und $B$ eintreten, ist gleich der Wahrscheinlichkeit das $A$ eintritt wenn vorher schon $B$ eingetreten ist, mal der Wahrscheinlichkeit, dass $B$ eintritt. Das klingt komplizierter, als es ist. Die Wahrscheinlichkeit, dass ich Kaffee und Kuchen zu mir nehme hängt davon ab, dass ich Kuchen esse, und davon, wie wahrscheinlich es ist, dass ich zum Kuchen Kaffee trinke. Da ich nur selten Kuchen esse (sagen wir, für 1 Stunde sonntags: $1/12/7 = 0.012$), dann aber fast immer Kaffee dazu trinke ($P(\text{Kaffee}|\text{Kuchen}) = 0.9$), ist die Wahrscheinlichkeit, mich bei Kaffeetrinken *und* Kuchenessen zu erwischen $0.012 \cdot 0.9 = 0.011$.

Natürlich muss das Gleiche auch umgekehrt gelten, also $P(A \cap B) = P(B|A)P(A)$. Wir formen die erste Gleichung um, und setzen die zweite ein:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B|A)P(A)}{P(B)},$$

[1] Das wird am Beispiel des Konfidenzintervalls gut deutlich: Das 95%-Konfidenzintervall ist so berechnet, dass bei 100 wiederholten Erhebungen des Datensatzes der wahre Wert in 95 Fällen in diesem Intervall liegt. Kerngedanke ist hier „bei Wiederholung“ (*under repeated sampling*), so dass wir auszählen können, wie häufig ein Wert im Intervall lag oder nicht: wir erheben die Frequenz dieses Falls; daher der Name.

[2] Wie Jaynes (2003, S. 112) darlegt, ist die Benennung der gesamten „Bayesschen“ Statistik etwas unglücklich. Zum einen schrieb Bayes „seinen“ Satz nie irgendwo als Formel auf. Dann war die Formel schon vorher bekannt, etwa Bernoulli und de Moivre, über 50 Jahre vor Bayes. Und schließlich war es nicht Bayes, sondern Laplace, der 10 Jahre nach Bayes die Reichweite dieser einfache Produktregel der Wahrscheinlichkeitsrechnung erkannte. Nun ja. Nennen wir es also weiterhin „Bayessche“ Statistik und schämen uns etwas dafür.

und fertig ist der Satz von Bayes:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{A.2}$$

Nach einem Augenblick der Rückbesinnung an die Einführung der *likelihood* erkennen wir gewisse Elemente wieder. Einfacher wird die Wiedererkennung, wenn wir $A$ und $B$ ersetzen durch „Parameter“ (oder auch Modell oder Hypothese) und „Daten“:

$$P(\text{Parameter} \mid \text{Daten}) = \frac{P(\text{Daten} \mid \text{Parameter})P(\text{Parameter})}{P(\text{Daten})} \tag{A.3}$$

Die Wahrscheinlichkeit der Daten, gegeben die Parameter eines Modells, nannten wir bisher *likelihood*. Das Gegenstück, die Wahrscheinlichkeit der Parameter, gegeben die Daten, ist der eigentliche Fokus der Bayesschen Statistik. Hauptziel ist herauszubekommen, wie wahrscheinlich unser Modell ist (das ja durch die Parameter spezifiziert wird). Im Idealfall komme ich so etwa zu folgender Aussage: „Die Wahrscheinlichkeit, dass der gegenwärtige atmosphärische Temperaturanstieg menschengemacht ist, gegeben alle Daten, die uns derzeit zur Verfügung stehen, ist X%.“ Uns interessiert eigentlich selten die Frage, die uns die *likelihood* beantwortet: „Wie wahrscheinlich ist der beobachtete Temperaturanstieg, gegeben die Parameter in unseren Modellen?“, denn uns interessiert primär unsere Hypothese (Temperaturanstieg), nicht die dazu erhobenen Daten. Die Bayessche Gleichung dreht also die *likelihood* um, so dass wir „das richtige Problem“ beantworten.[3]

Aber was sind dann die beiden anderen Terme, $P(\text{Parameter})$ und $P(\text{Daten})$? $P(\text{Parameter})$ bezeichnet man als *prior probability*, kurz *prior*, und das ist die Wahrscheinlichkeit von Parameter *bevor* ich die Daten anschaue. Wenn ich eine Münze werfe, erwarte ich, dass beide Seiten gleich wahrscheinlich sind. Aber wenn meine Hypothese komplizierter ist, dann kann ich im *prior* meine Erwartung und Unsicherheit kodieren. Wenn ich z.B. ein Düngeexperiment mit Pflanzen mache, dann erwarte ich, dass diese besser wachsen, nicht schlechter. Entsprechend wäre mein *prior* für den Düngeeffekt auf jeden Fall positiv, aber nicht genau 2 oder 7.6 Mal der Kontrolle. Ich halte ein Ergebnis von, z.B. 50% bis 400% besserem Wachstum für realistisch. Dann wäre mein *prior* z.B. eine uniforme Verteilung zwischen 0.5 und 4.

Der *prior* ist ein extrem wichtiges Element der Bayesschen Statistik und wird immer wieder auftauchen. Hier soll es erst einmal nur eingeführt werden: Der *prior* ist die Verteilung unserer Erwartung vor Kenntnis der Daten.

Etwas schwieriger zu erklären ist $P(\text{Daten})$. Am einfachsten (und häufigsten) ist die Sichtweise, dass dieser Term nur dafür da ist, dass die Mathematik stimmt (also ein Normierungsterm, damit die Gesamtwahrscheinlichkeit auch 1 ist). Ein einfaches Beispeil, um $P(\text{Daten})$ zu illustrieren ist in Abb. A.1 dargestellt. Bei einer typischen Berechnung setzen wir $P(\text{Daten}) = \int P(\text{Daten}|\theta)\,d\theta$, d.h. wir integrieren den Parameter $\theta$ aus, um die Wahrscheinlichkeit der Daten zu erhalten.[4]

[3] Vielleicht wird es mit einem Beispiel aus dem Gericht deutlicher. Ein Mann mit positivem DNA-Treffer wird einer Straftat angeklagt. Der medizinische Experte erklärt, dass die Wahrscheinlichkeit eines DNA-Treffers bei 1:1 Millionen liegt. Ist das ein klarer Beweis der Schuld des Angeklagten? Nein! Uns interessiert nicht $P(\text{DNA-Treffer} \mid \text{unschuldig})$, sondern im Gegenteil $(P(\text{unschuldig} \mid \text{DNA-Treffer})$. Wenn wir z.B. 10 Millionen Menschen im möglichen Täterkreis haben, dann haben neben dem wahren Täter noch 10 weitere unschuldige Personen einen DNA-Treffer. D.h., nur 1 von 11 Treffern identifiziert den Schuldigen! Sprich, die „1 in 1 Millionen“ ist eine korrekte, aber irreführende Aussage. Weil in deutsche Gerichten nicht (auf)gerechnet wird („*Judex non calculat*.“), ist das Problem dort wenig bekannt. Im Englischen heißt es *prosecutor's fallacy*.

[4] Aufbauend auf der letzten Fußnote brauchen wir zur Berechnung der Wahrscheinlichkeit, dass der Angeklagte unschuldig ist, bei gegebenem positiven DNA-Treffer, eben auch den Nenner. In diesem Fall berechnet er sich as die Wahrscheinlichkeit, dass ein Unschuldiger einen DNA-Treffer hat (also 1/1 Millionen · (10 Millionen mi-

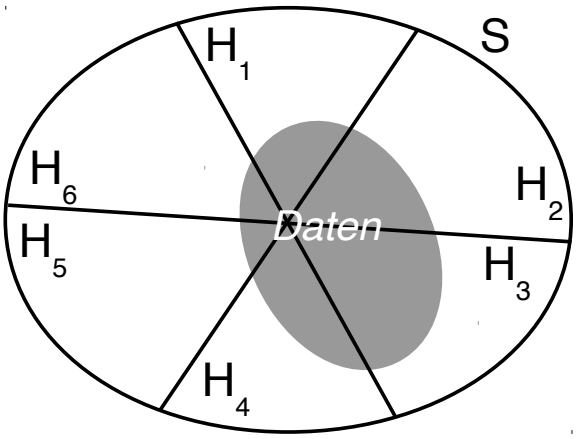



**Abb. A.1:** Nehmen wir einmal an, der Ereignisraum S sei durch 6 sich ausschließende Hypothesen gegeben (etwa die 6 Seiten eines Würfels; unser Modellparameter hat also 6 diskrete Werte). Weiterhin haben wir Daten erhoben, deren wahre Position wir z.B. wegen des Messfehlers nicht kennen (irgendwo innerhalb der graue Fläche). Entsprechend sind alle 6 Hypothesen möglich, aber $H_3$ ist am wahrscheinlichsten, weil dort am meisten graue Fläche liegt. Die Zutaten des Satz von Bayes sind alle da: die Fläche $H_i/S$ ist der *prior* jeder Hypothese, die *likelihood* ist der Anteil der Datenfläche an der jeweiligen Hypothese (der graue Teil jedes Kuchenstücks), und $P(\text{Daten})$ ist die graue Fläche selbst. Im Nenner der Bayes-Gleichung stünde in diesem Fall kein Integral sondern eine Summe über die sechs Hypothesen: $P(\text{Daten}) = \sum_{i=1}^{6} P(\text{Daten}|H_i)$.

Zusammenfassend sehen wir, dass Bayessche Statistik mit der uns bekannten *likelihood* und unseren vorherigen Erwartungen (*prior belief*), plus einem Normierungsterm, eine Umkehr der konditionalen Wahrscheinlichkeit ermöglicht. Das Ergebnis ist eine durch Daten aktualisierte Erwartung, der *posterior belief*, oder kurz *posterior* (Abb. A.2). Diese kann für nachfolgende Auswertungen als neuer *prior* benutzt werden.

### A.1.2 Das „Bayessches Problem“

Ronald Fisher taucht in der Statistik an vielen Stellen auf, er ist ein übergroßer Vater der modernen Statistik, der Genetik und der Versuchsplanung. Ihn störte am Satz von Bayes, dass wir in der Auswertung einen *prior* angeben müssen. Fisher argumentierte, dass damit subjektiven Eindrücken und Dummheit Tür und Tor geöffnet werden. Sein Ansatz, den *prior* aus der Statistik zu verbannen, war die Wiederholungsmessung. Wenn ich einen Datensatz unendlich häufig erhebe, dann ist der *prior* ohne Bedeutung, da er durch die Daten „überstimmt“ wird. Fischer berechnete, wie der *posterior* aussieht, wenn der Einfluss des *priors* wegfällt, was bei großen Datensätzen und schwachem Vorwissen der Fall ist. Das Ergebnis ist die „frequentistische Statistik“, wie wir sie bisher betrieben haben: *prior* und Normierungsterm kürzen sich weg, und die *likelihood* entspricht dem *posterior*. Dann können wir uns die Umrechnung auch schenken, so Fisher, und nur noch mit der *likelihood* rechnen. In der Tat ist das seit 100 Jahren das verbreitete Vorgehen.

Fishers Lösung hat Nebenkosten. Zum einen ist das Konfidenzintervall, dass er einführte,

---

nus dem wahren Schuldigen) ) plus dem Wahrscheinlichkeitsbeitrag des wahren Schuldigen (1 / 10 Millionen): $1/(1 \cdot 10^7 - 1) + 1/1 \cdot 10^7 = 2 \cdot 10^{-7}$. Der Vollständigkeit halber sei hier auch der (hier offensichtliche) *prior* genannt: nur einer der 10 Millionen ist der Täter, also $P(\text{unschuldig}) = 9.99999\text{Mio}/10\text{Mio} = 0.99999$. Somit ist die Wahrscheinlichkeit, dass der Angeklagte mit positivem DNA-Treffer schuldig ist $= 1/10^7 \cdot 0.99999/2 \cdot 10^{-7} = 0.49999$. Aus dem „offensichtlichen“ Schuldigen wurde ein Münzwurf, eine fifty-fifty-Chance! Der entscheidende Grund ist, dass wir so viele Personen in der Zielgruppe haben, dass ein Fehltreffer mehrfach vorkommt. Je kleiner wir durch zusätzliche Informationen unseren Täterkreis machen können, desto glaubwürdiger ist der DNA-Match.

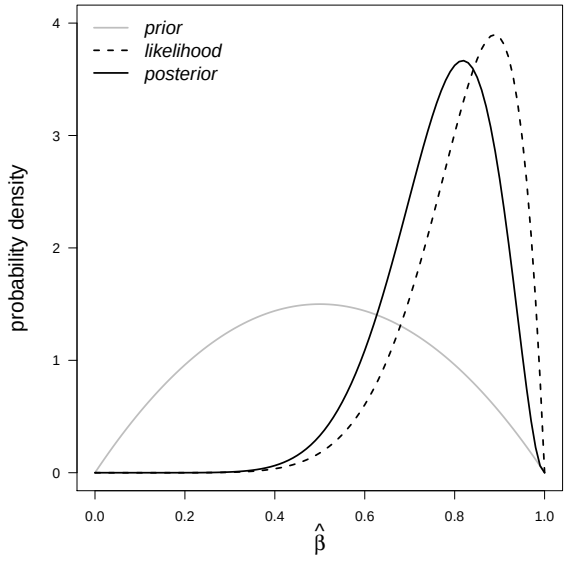



**Abb. A.2:** Im Gegensatz zur klassischen Statistik, in dem die *likelihood* (gestrichelt) das Endergebnis bei der Parameterschätzung ist, formen bei der Bayesschen Statistik *likelihood* und *prior* (grau) zusammen den *posterior* Schätzer (durchgezogen schwarz). In diesem Fall wir ein Parameter $\widehat{\beta}$ geschätzt, für den wir einen Wert so mittig zwischen 0 und 1 erwarten würden. Die Daten haben den *maximum likelihood*-Wert bei 0.85, so dass in der Summe daraus ein *posterior* von 0.8 wird.

wahrlich nicht intuitiv. Es ist eben *nicht* der Bereich, in dem mit 95%-iger Wahrscheinlichkeit die Wahrheit liegt (sondern der Algorithmus, der einen Bereich definiert, der in 95 von 100 Fällen die Wahrheit umfasst). Das Bayessche Äquivalent, der 95%-ige Glaubwürdigkeitsbereich (*credible interval*), umfasst wirklich mit 95%-iger Wahrscheinlichkeit die Wahrheit! Bei großen Datenmengen und vagem Vorwissen sind der frequentistische Vertrauens- und der Bayessche Glaubwürdigkeitsbereich allerdings kaum unterscheidbar.

Zum andern hat Fishers Lösung zu einem Fokus auf „die falsche Frage“ geführt. Wissenschaftstheoretisch wollen wir wissen, ob unsere Hypothese korrekt ist, gegeben die Daten. Ob unsere Daten wahrscheinlich sind, gegeben unserer Hypothese, ist eher egal. Aber mit dieser Umkehr rücken die Daten immer mehr in den Vordergrund, und die eigentliche Hypothese wird blasser.

Das Bayessche Problem der *prior* hat seit dem immer wieder Statistiker beschäftigt und ist heutzutage immer noch ein häufig missverstandener Punkt. Deshalb verwenden wir dafür später einen eigenen Abschnitt. Nur so viel vorneweg: die meisten Statistiker sehen heute die Notwendigkeit, eine plausible Erwartung zu formulieren, als eine Stärke, nicht eine Schwäche der Bayesschen Statistik.[5]

## A.2 Bayes als Optimierungsalgorithmus

Der letzte Abschnitt war sehr abstrakt-theoretisch. Können wir Bayes nicht vielleicht einfacher verstehen? Ein alternativer Ansatz besteht darin, die technische Seite der Bayesschen Statistik zu betrachten, in der ein bestimmter Algorithmus eine wichtige Rolle spielt (MCMC), den wir einfach als Optimierungsalgorithmus interpretieren können. Damit wird Bayessche Statistik zu einer alternativen Lösungsmöglichkeit für die Parameterschätzung des GLM. Tatsächlich kommen wir um ein Grundverständnis der ganzen *prior-posterior*-Geschichten nicht

[5]Ein tolles Beispiel ist die Anwendung von Laplace, um die Masse des Saturn auszurechnen: `https://de.wikipedia.org/wiki/Bayessche_Statistik`

drum herum. Aber die Sicht als Optimierungsverfahren nimmt etwas vom mathematischen Schrecken.

### A.2.1 MCMC und MCMC-Integration

Der erwähnte Optimierungsalgorithmus heißt *Markov chain Monte Carlo*, kurz MCMC. Er wird benutzt, um den Nenner der Bayesschen Gleichung zu berechnen. Konkret ist dieser Nenner ja die Wahrscheinlichkeit der Daten; was mag das bedeuten? Nehmen wir einmal an, dass wir 2 Möglichkeiten haben, wie unsere Daten zustande kommen, $H_1$ und $H_2$. Dann ist die Wahrscheinlichkeit der Daten gleich der Wahrscheinlichkeit der Daten unter $H_1$ mal der Wahrscheinlichkeit von $H_1$, plus der Wahrscheinlichkeit der Daten unter $H_2$ mal der Wahrscheinlichkeit von $H_2$. Nur so können die Daten ja entstanden sein!

$$P(\text{Daten}) = P(\text{Daten}\,|H_1)P(H_1) + P(\text{Daten}\,|H_2)P(H_2) = \sum_{i \in \text{ Hypothesen}} P(\text{Daten}\,|H_i)P(H_i). \tag{A.4}$$

In anderen Worten, wir summieren die Wahrscheinlichkeit von Daten und Hypothese *über aller möglichen Hypothesen*. Wenn sich die Hypothesen nur in den Parametern eines Regressionsmodells unterscheiden, müssen wir statt der Summe über alle möglichen Parameter $\theta$ einer Hypothese $i$ integrieren:

$$P(\text{Daten}) = \int P(\text{Daten}\,|\theta_i)P(\theta_i)d\theta_i. \tag{A.5}$$

Dieses Integral sieht nicht nur hässlich aus, es ist es auch! Das Problem ist ja, dass die Wahrscheinlichkeiten (sowohl die bedingte, als auch die andere) sich mit den Werten von $\theta$ ändern und $\theta$ mehrdimensional ist. Bevor wir dafür eine Lösung finden, sollten wir uns noch einmal vor Augen führen, was eigentlich das Problem ist.

In obiger Gleichung stehen zwei Terme $P(\text{Daten}\,|\theta_i)$ und $P(\theta_i)$. Ersterer ist die *likelihood*, also eine Evaluation der Wahrscheinlichkeitsdichtefunktion, entsprechend unserer Annahmen über die Verteilung der Daten (normal, Bernoulli, negativ binomial, usw). Der zweite Term, der *prior*, ist ebenfalls eine Dichtefunktion, z.B. die uniforme, Normal- oder $\beta$- oder $\gamma$-Verteilung. Was wir integrieren müssen ist also das Produkt zweier Dichtefunktionen. Diese Funktionen sind üblicherweise „wohlverhalten“, ohne Definitionslücken oder Unstetigkeiten. Das ist gut, denn jede Unstetigkeit führt zu Berechnungsproblemen in ihrem Umfeld, und das verkompliziert die Optimierung.

Jetzt zur Bestimmung des Integrals. Ohne auf die mathematische Herleitung eingehen zu können, besteht die Lösung darin, dass wir für ein zufälliges Parameterwerteset ($\theta_1$) die Dichtefunktionen ausrechnen, und so einen Wert für den Zähler von Gleichung A.3 erhalten. Wir wählen einen neuen, zufälligen Parametersatz $\theta_2$ in der Nähe von $\theta_1$ und berechnen die Dichtefunktionen erneut. Dann vergleichen wir die Werte und nehmen den Parametersatz als neuen Startwert, der den höheren Wert hat – außer manchmal (siehe Box 1).

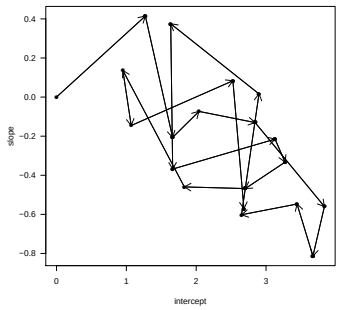
Dieses Schema wiederholen wir jetzt tausendfach. Am Ende können wir uns anschauen, in welchem Parameterbereich wie viele Evaluationen stattgefunden haben. Dieser Wert entspricht der Wahrscheinlichkeitsdichte des Zählers von Gleichung A.3. Der Nenner ist einfach die Anzahl Evaluation.

Dieses Verfahren nennt sich Markow-Kette Monte Carlo: bei einer Markow-Kette ist der nächste Wert nur von dem davor abhängig (es gibt kein „Gedächtnis“), und die Ziehung der nächsten Werte erfolgt zufällig, wie beim Roulette im Casino von Monte Carlo, Monaco. Der Einfachheit halber nennen wir auch im Deutschen dieses Verfahren MCMC.[6] So ein MCMC-

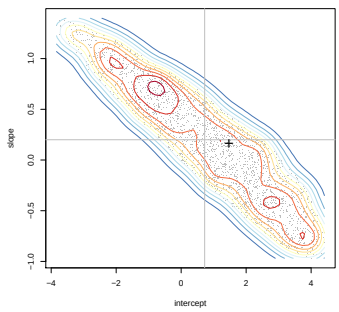
[6]Markow war ein russischer Mathematiker. Entsprechend gibt es keine eindeutige Lateinische Schreibweise

**Exkurs 1: Der Metropolis-Algorithmus**

Ausgehend von einem Startwert für jeden Parameter einer Poisson-Regression (hier $(0, 0)$ für $y$-Achsenabschnitt und Steigung), schlägt der Metropolis-Algorithmus einen neuen Wert in der Nähe vor (häufig aus einer Normalverteilung über dem gegenwärtigen Wert) und berechnet das Produkt aus *likelihood* und *priors*. Das Verhältnis dieser Werte, $\alpha$, (neu vorgeschlagener durch gegenwärtigen Wert) bestimmt, ob der neue Wert angenommen wird. Wenn $\alpha > 1$, dann wird der neue Wert angenommen (da er ja besser ist). Wenn hingegen $\alpha < 1$, dann wird ausgewürfelt, ob der neue Wert angenommen wird (obwohl er schlechter ist): die Wahrscheinlichkeit, den schlechteren, vorgeschlagenen Wert anzunehmen, ist gerade $\alpha$; d.h. je schlechter der vorgeschlagene Wert, desto unwahrscheinlicher, dass er angenommen wird ($P(\text{Annahme Vorschlag}|\alpha < 1) = \text{Bern}(p = \alpha)$).
Diese Prozedur wird häufig wiederholt (tausende Male). Die ersten Schritte 50 sehen z.B. so aus (wobei nur in 25% der Ziehungen der neue Wert angenommen wurde):




Durch das stochastische Annehmen einer schlechteren Lösung wird verhindert, dass der Algorithmus in einem lokalen Maximum hängen bleibt. Tatsächlich konnte Metropolis beweisen, dass dieser Algorithmus die tatsächliche *posterior*-Wahrscheinlichkeit des Parameterraums abbildet. D.h., wenn wir nur lange genug beproben, dann verbringt der Algorithmus gerade so viel Zeit an einer Stelle, wie es der Wahrscheinlichkeit dieser Parameterkombination entspricht. Und das ist ja gerade was wir herausfinden wollen: die *posterior* der Parameterwerte! Im Gegensatz zu einem Optimierungsalgorithmus erhalten wir also nicht den einen besten Wert, sondern eine Wahrscheinlichkeitsverteilung für alle Parameterwerte.
Beim Metropolis-Algorithmus ist die Vorschlagsverteilung symmetrisch, eine Einschränkung die durch den Metropolis-Hastings-Algorithmus aufgehoben wurde. Wenn die *priors* sich analytisch in die *posterior* hineinrechnen lassen (wir sprechen dann von konjugierten *prior*), dann kann man die *posterior* sogar direkt beproben, ohne die Irrfahrt durch den Parameterraum machen zu müssen (was dann als "Gibbs *sampling*" bezeichnet wird). Diese Details sind in den gängigen Bayesischen Algorithmen implementiert (und viele mehr), so dass wir uns nur das Prinzip des Metropolis-Algorithmus als Beispiel merken müssen.
Nach einer Millionen Ziehungen sieht die Verteilung dann etwa so aus:




Die grauen Linien zeigen den Mittelwert der Metropolis-Proben an, das schwarze Kreuz die *maximum likelihood*-Lösung aus einem GLM.
Aus der Punktewolke wird klar, dass die Schätzer für Achsenabschnitt und Steigung korreliert sind. Der beste Schätzer für beide liegt auf einem Plateau, dessen genaues Maximum nur ungenau zu bestimmen ist.

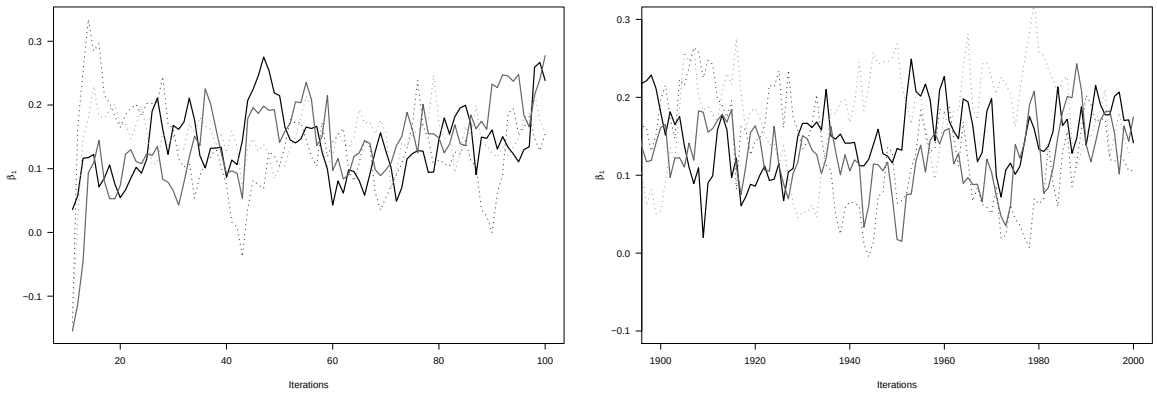



**Abb. A.3:** Die Spur (*trace*) von vier MCMC-Ketten zur Schätzung des Parameters $\beta_1$. In der linken Abbildung sehen wir, dass die Ketten von unterschiedlichen Startwerten ausgehen, sich aber schnell in einem Bereich finden. In der rechten Abbildung, 1900 Schritte später, laufen alle 4 Ketten durcheinander. Der Parameterbereich, in dem die Ketten durcheinanderlaufen, entspricht der Unsicherheit des Parameters. Aber immerhin ist in den Ketten kein Trend mehr zu sehen, wie noch in der linken Abbildung.

Lauf entspricht einer Irrfahrt (*random walk*) im Parameterraum. Eine faszinierende, und für uns essentielle Eigenschaft ist, dass sich der Irrfahrende in der Nähe von „guten“ Lösungen lange aufhält, während er schlechte Parameterwerte meidet. Als Ergebnis ist die Verteilung der Evaluationen im Parameterraum identisch mit der *posterior*! Diesen Absatz muss man vielleicht mehrmals lesen, bis deutlich wird, dass wir gerade ein schwieriges Problem gelöst haben. Durch die Markowsche Irrfahrt im Parameterraum haben wir das Integral Gleichung A.5 gelöst, und nebenbei auch noch die Parameterwerte für Gleichung A.3 berechnet. Die Box 1 geht noch ein bisschen ins Detail des MCMC, vor allem weil die Wahl der neuen Parameter und das gelegentliche Annehmen einer schlechteren Lösung einer Erläuterung bedarf.

### A.2.2 Optimierungsdiagnostik: Kettenmischung und Konvergenz

Für die Anwendung der Bayesschen Statistik ist zwar die Herleitung, vielleicht sogar das Verständnis des *Wie* von MCMC nicht wichtig, aber dieser Algorithmus hat ein paar Konsequenzen für die Zuverlässigkeit der Lösung. So eine Irrfahrt im Parameterraum ist nicht steuerbar, und wer weiß, ob unser Irrfahrer auch den Parameterraum ordentlich beprobt hat?

Um etwas sicherer zu gehen, schicken wir mehrere Irrfahrer auf die Reise, legen also mehrere Markow-Ketten an. Das hat den Vorteil, dass wir am Ende die Ergebnisse vergleichen (und zusammenwerfen) können. Wenn alle Ketten im gleichen Parameterbereich viele Evaluation haben, und entsprechend sehr ähnliche *posteriors* berechnet haben, dann war die Irrfahrt ein Erfolg. Wenn die Ketten aber zu kurz waren, dann hielten sich die Ketten in unterschiedlichen Teilen des Parameterraums auf; wir sagen, sie „mischten sich nicht“.

Das Mischen der Ketten ist ein wichtiges diagnostisches Merkmal, ob unsere MCMC-Integrierung funktioniert hat! Tatsächlich plotten wir dafür (für jeden Parameter) die evaluierten Parameterwerte und sehen dann eine Überlagerung der Ketten (oder eben auch nicht; Abb. A.3).

Wenn die Ketten gleiche Werte liefern, sprechen wir von *Konvergenz*, auch wenn eine MCMC-Kette im Grunde nie endet und nicht auf einen einzigen Wert konvergiert. Stattdes-

für seinen kyrillischen Namen. Im Deutschen schreiben wir Markow, oder sogar Markoff, im Englischen Markov. Markow hat sich mit stochastischen Prozessen auseinandergesetzt und dabei die Markow-Kette definiert.

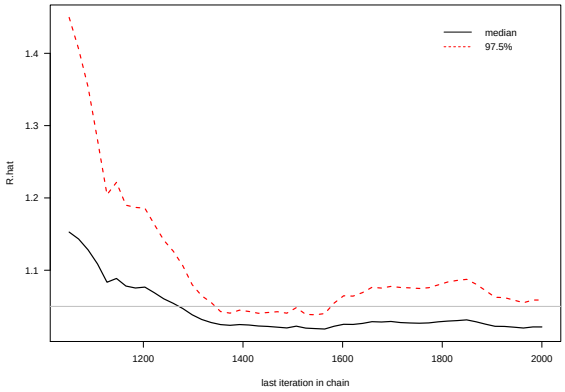



**Abb. A.4:** Entwicklung der Gelman-Rubin-Statistik ($\hat{R}$) in den vier MCMC-Ketten aus Abb. A.3. Die graue Linie zeigt den Faustregelwert von 1.05 an. Die 2000 Schritte sind hier also noch nicht genug für eine befriedigende Konvergenz, aber immerhin sind die Werte ab 1400 Schritten langsam im akzeptablen Bereich.

sen konvergiert jede einzelne Kette auf die *posterior*-Verteilung. Offensichtlich war es egal, wie die Ketten abliefen, am Ende lieferten sie konvergente Ergebnisse. Wir können aber sogar für eine einzelne Kette Konvergenz prüfen. Wenn wir in der ersten Hälfte der Kette andere Parameterbereiche antreffen als in der zweiten, dann ist es wahrscheinlich, dass die Kette noch nicht konvergiert ist. Wir sind also streng und wollen wirklich sicher sein, dass die überwiegende Menge der Evaluationen proportional zur *posterior* ist, und nicht nur die letzten 100 oder so. Der Grund ist einfach: das Integral berücksichtigt alle Werte, nicht nur die letzten; entsprechend muss die Kette (oder die zusammengeworfenen Ketten) repräsentativ den Parameterraum beprobt haben.

Als Zusammenfassung der Konvergenzanalyse in einem Wert gibt es die Gelman-Rubin-Statistik (der sogenannte „Rhat“-Wert, $\hat{R}$: Gelman & Rubin 1992). Sie bestimmt das Verhältnis der Varianz innerhalb der Ketten zu der zwischen den Ketten und ist im Idealfall 1 (zumindest aber kleiner als 1.05). Abweichungen davon weisen auf ungenügende Länge der Ketten hin (Abb. A.4).

Demgegenüber kann die Raftery-Lewis-Statistik benutzt werden, um die tatsächliche Anzahl unabhängiger Markow-Schritte zu berechnen (Raftery & Lewis 1992). Wenn die neuen Parameter zu nah beim aktuellen Wert liegen, dann sind auch nach mehreren Schritten die Parametervorschläge sehr ähnlich zum aktuellen Wert. Wir sprechen dann von zeitlicher Autokorrelation, und die ist ein Verstoß gegen die Markow-Annahme einer Kette ohne Gedächtnis. Der Raftery-Lewis-Algorithmus dünnt die Kette solange aus, bis keine Autokorrelation mehr vorliegt, und gibt damit einen Eindruck über die nötige Länge der Kette.

## A.3 Das *prior*-Dilemma

Wie erwähnt, spielt der *prior* in der Bayesschen Statistik eine wichtige Rolle. Es sollte aber auch klar sein, dass er umso unwichtiger wird, je mehr und je besser Daten ich habe. Mit „bessere“ Daten sind hier jene gemeint, die es erlauben, die Modellparameter einzuengen.

Wenn ich nun gar keine Erwartung hätte über die zu berechnenden Parameter, kann ich dann nicht irgendwelche uninformativen *prior* wählen? Gibt es nicht irgendwelche Verteilungen, mit denen ich immer auf der sicheren Seite bin? Kurze Antwort: Nein.

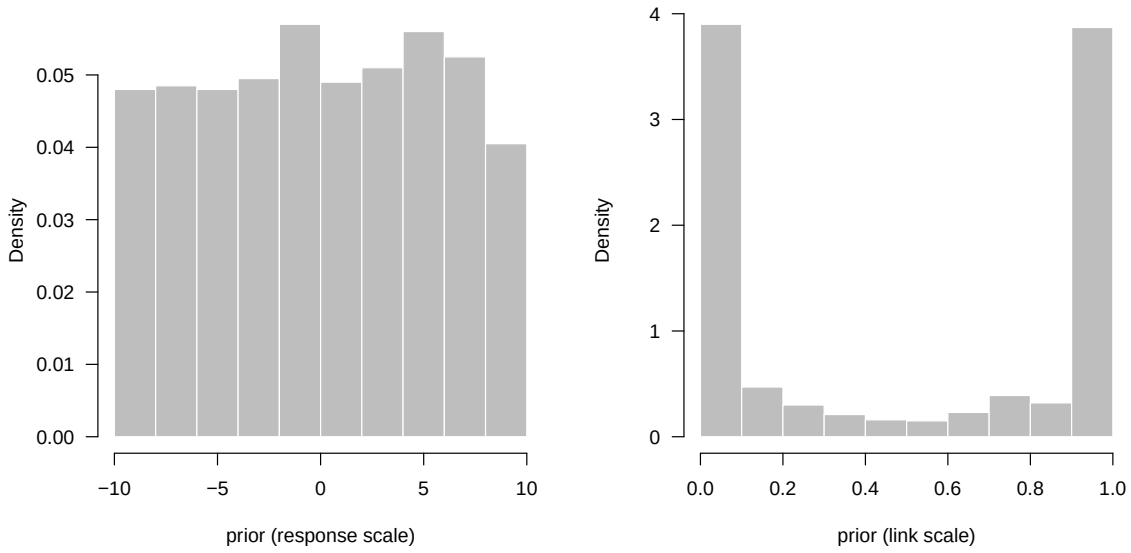



**Abb. A.5:** Zufallsziehungen aus einer uniformen Verteilung zwischen $-10$ und 10 sind ein typisches Beispiel für uninformative *prior*. In einem binomialen GLM wird durch die *logit*-Transformation daraus aber ein *sehr* informativer *prior*, mit kaum Gewicht zwischen 0 und 1.

### A.3.1 Die Illusion des „uninformativen *prior*“

Der *prior* trägt immer etwas Information. Selbst ein uninformativer *prior*, etwa eine uniforme Verteilung zwischen 0 und 100 als *prior* für die Standardabweichung einer Regression, hat ein bisschen Einfluss auf das Ergebnis. Wirklich „uninformativ“ gibt es nicht. Die Sprachregelung für *prior*, die möglichst wenig Einfluss auf das Ergebnis nehmen, variiert mit den Jahren. Statt uninformativ wird auch „objektiv“, „indifferent“, „naive“ oder „flach“ (*flat*) gesagt, um anzudeuten, dass die Dichte über einen weiten Bereich gleich ist.

Ein bisschen Information ist nicht problematisch, schließlich haben wir ja Daten. Ungleich schlimmer ist die Fehleinschätzung, dass ein *flat prior* automatisch uninformativ ist. Das ist nicht der Fall! Wenn wir z.B. eine logistische Regression rechnen, und für unsere Koeffizienten uniforme *prior* wählen, dann wird aus diesen auf der *link*-Skala ein *sehr* informativer *prior* (Abb. A.5)! Gerade die im GLM unvermeidbaren *link*-Funktionen muss man bei der Einschätzung des Informationsgehalts der *prior* berücksichtigen. Zumindest aber sollten wir uns merken: flach ist nicht uninformativ!

Es gibt eine Form des *priors*, der zumindest nicht auf diese Transformationen reagiert, der, wie man sagt, skalierungsinvariant ist. Nach seinem Erfinder heißt er „Jeffreys *prior*“. Er muss für jedes Problem neu entwickelt werden, weshalb er sich vor allem bei theoretischen Statistikern einer gewissen Beliebtheit erfreut, aber im Umweltbereich kaum vorkommt.[7]

Ein wichtiger Kommentar zum *prior* ist allerdings noch nötig. Es wäre doch toll, wenn wir z.B. eine uniforme Verteilung von $-\infty$ bis $+\infty$ annehmen könnten, dann hätte jeder Parameterwert die gleiche Wahrscheinlichkeit (bis auf die Transformationsproblematik). Natürlich ist das Integral über so einen *prior* unendlich, und nicht 1, und somit „unsauber“ (*improper*). Tatsächlich „funktioniert“ so ein *improper prior* häufig sehr gut. Wichtig ist nämlich, dass der *posterior* sich zu 1 integriert, was häufig selbst bei unsauberem *prior* der Fall ist.

[7] Weitere Argumente für die ein oder andere Form des *prior* wurden von vielen Statistikern vorgebracht: der *maximum entropy prior*, Haldane's *prior*, *reference prior*, der AIC-induzierende *savvy prior*, usw (`https://en.wikipedia.org/wiki/Prior_probability#Uninformative_priors`). Wenngleich das wichtige statistische Probleme sind, führt es für eine einleitende Beschreibung viel zu weit.

### A.3.2 Der *prior* als Freund des wissenschaftlichen Denkens

Was in der einzelnen Analyse als lästig empfunden werden mag, ist im gesamtwissenschaftlichen Kontext gesehen eine echte Bereicherung: die Notwendigkeit, einen *prior* festlegen zu müssen (Jaynes 2003). Kaum jemals sind wir die Ersten, die eine Fragestellung bearbeiten, ja unsere Theorie baut meistens auf vielen Studien auf. Und dann tun wir so, als wäre unser Datensatz der erste in der Welt, indem wir bisherige Informationen ignorieren und jedes Ergebnis als gleich-wahrscheinlich akzeptieren.

Der *prior* ermöglicht uns, diese Vorinformationen explizit zu nutzen. Wenn zum Beispiel in vielen Studien ein Artenvielfaltsgradient vom Pol zum Äquator beobachtet wurde, dann können wir schon erwarten, dass auch in unserem Datensatz wahrscheinlich einen positiven Effekt des Breitengrads findet. Entsprechend können wir für diesen Parameter z.B. einen positiven *prior* annehmen (etwa eine $\gamma$-Verteilung). Diese informativen *prior* können aufbauen auf Meta-Analysen oder Pilotstudien, auf Laboranalysen oder Freilanddaten. Dabei ist etwas Vorsicht und Übung gefragt, denn in unserem Beispiel gab es auch Studien, die einen *negativen* Breitengradeffekt gefunden haben (etwa für Ameisen). Deshalb sollten wir zumindest die Möglichkeit zulassen (etwa mit einem normalverteilten *prior* mit Mittelwert $\mu > 0$).

Und schließlich ist es konzeptionell sehr attraktiv, das Modell mit neuen Daten nur aktualisieren zu müssen, indem wir den *posterior* des letzten Modells als *prior* des nächsten nehmen, in dem wir nur die neu hinzugekommenen Datenpunkte anfitten. Dieses *Bayesian updating* ist in der Praxis irrelevant, da zwei Probleme existieren. Zum Einen ist der *posterior* mehrdimensional und u.U. nicht einfach durch eine Dichtefunktion zu beschreiben. Zum Anderen ist der Aufwand, die alten Daten einfach um die neuen zu erweitern, und die ursprüngliche Analyse zu wiederholen, meist deutlich geringer, als den *prior* zu aktualisieren.

### A.3.3 Modellselektion durch *lasso-prior*

Der *prior* kann benutzt werden, um für uns Modellselektion zu betreiben. Eine schrittweise Modellselektion ist mit den üblichen Bayesschen Verfahren nicht machbar, so dass wir eine neue Idee kennenlernen müssen, die übrigens genauso auf frequentistische Methoden anwendbar ist: Die Regularisation der Zielfunktion.

Zunächst kurz die Idee, dann die Mathematik. Die Idee ist, dass in einer multiplen Regression wahrscheinlich einige Prädiktoren irrelevant sind, wir wissen nur nicht welche. Irrelevant bedeutet dann aber, dass ihr Schätzer 0 ist. Bei regularisierten Verfahren erweitern wir die Maximierung der *likelihood* um einen weiteren Term, in dem die Anzahl Parameter minimiert wird. Das kennen wir z.B. als den AIC, in dem jeder Parameter mit einem Faktor 2 bestraft, also regularisiert wird: $\mathrm{AIC} = -2\ell + 2p$. In der allgemeineren Form der Regularisierung kommt jetzt nicht nur die Anzahl der Parameter vor, sondern ihr Absolutwert, $\hat{\beta}$.[8] Das hat den Effekt, dass bei der Optimierung nur die Prädiktoren im Modell bleiben, die auch wirklich etwas Effekt haben, und die schwachen Prädiktoren auf 0 eingeschrumpft und somit aus dem Modell entfernt werden. Im Englischen heißt Regularisierung deshalb auch *shrinkage*.

Mathematisch versuchen wir ja beim Anpassen der Parameter $\theta$ eines Modells den Unterschied zwischen Antwortvariable $y$ und Modellvorhersage $f_\theta(x) = \hat{y}$ zu minimieren, etwa

[8] Voraussetzung ist, dass alle Prädiktoren standardisiert werden und weiterhin die Antwortvariable zentriert wird, indem der Mittelwert $\bar{y}$ von jedem Antwortvariablenwert $y_i$ abgezogen wird. Dadurch fällt der $y$-Achsenabschnitt weg, der nicht regularisiert wird.

durch Minimierung der Abweichungsquadrate:

$$\min_{\theta} \sum_{i=1}^{N} (y_i - f_\theta(x_i))^2 \tag{A.6}$$

Allgemeiner betrachten wir nicht die Abweichungsquadrate, sondern die zur Verteilungsannahme gehörige *loss function* $V$, die sich aus der Dichtefunktion ergibt:[9]

$$\min_{\theta} \sum_{i=1}^{N} V(y_i, f_\theta(x_i)), \tag{A.7}$$

etwa mit *quadratic loss* $V(a, b) = (a - b)^2$ oder *absolute loss* $V(a, b) = |a - b|$.

Soweit die Situation ohne Regularisierung. Mit Regularisierung erhält die *loss function* einen weiteren Term:

$$\min_{\theta} \sum_{i=1}^{N} \left\{ V(y_i, f_\theta(x_i)) + \lambda |\hat{\beta}|^\alpha \right\}, \tag{A.8}$$

wobei $\hat{\beta}$ die Parameterschätzer, $\lambda$ ein noch zu bestimmender Regularisierungsparameter, und $\alpha$ entweder 0 (AIC), 1 („*lasso*“) oder 2 („*ridge*“) ist. Für den AIC sind nur die Anzahl Parameter wichtig, und $\alpha = 0$. *Ridge regression* bestraft am stärkesten für hohe absolute Parameterschätzer. Uns interessiert hier vor allem das Lasso (Tibshirani 1996), bei dem der Absolutwert der Parameter als Bestrafungsmaß benutzt wird. Wie stark diese Bestrafung ausfällt hängt vom Wert von $\lambda$ ab. Und genau dieses $\lambda$ können wir durch einen speziellen *prior* in der Bayesschen Analyse wählen.

Indem wir einem Parameter $\beta$ eine mit der Distanz zu 0 stark abfallende Verteilung als *prior* zur Seite stellen, führen wir ein Regularisierung durch. So ein beidseitig exponentiell abfallender *prior* wird auch Laplace *prior* genannt (oder eben doppelt exponentielle Verteilung: Abb. A.6). $\lambda$ ist der Parameter dieser auf 0 zentrierten Laplace-Verteilung.[10]

Der Effekt dieses Lasso-*priors* ist eine Eliminierung unwichtiger Prädiktoren aus dem Modell, indem ihre Schätzer auf 0 geschrumpft werden. Das ist der einfachste Bayessche Ansatz zur Modellvereinfachung (O'Hara & Sillanpää 2009), und auch im „normalen“ GLM kann er Gewinn bringend benutzt werden (Hastie et al. 2009).

## A.4 Ergebnisse interpretieren

Bevor es an die Interpretation der Ergebnisse geht, müssen zwei technische Schritte durchgeführt werden: die Überprüfung des MCMC-Verfahrens und die Modelldiagnostik (wie beim GLM). Beide Schritte sind eher technischer Natur und deshalb Gegenstand des nächsten Kapitels.

Grundsätzlich sind die Ergebnisse der Bayesschen Analyse denen der frequentistischen sehr ähnlich. Wir erhalten für jeden Parameter einen Schätzwert, dessen Unsicherheit, und über das 95%-*credible interval* auch einen P-Wert. Trotzdem gibt es ein paar philosophisch-fundamentale Unterschiede bei der Ergebnisinterpretation.

[9] Für die Normalverteilung ist die *loss function* das Quadrat der Abweichungen. Für binäre Daten einfach eine Indikatorfunktion, die 1 ist, wenn $y_i \neq \hat{y}_i$ und 0 sonst. Diese *loss function*-Geschichte ist wichtig für die gesamte *statistical learning*-Theorie und das *machine learning* (Hastie et al. 2009), aber wir gehen hier nicht weiter darauf ein.

[10] Die dazugehörige zentrierte Dichtefunktion lautet $P(x = X|\lambda) = \frac{1}{2\lambda} e^{-\frac{|x|}{\lambda}}$. Je größer der Wert von $\lambda$, desto langsamer fällt die Funktion mit dem Abstand zur $y$-Achse ab. Ein kleiner Wert führt also zu einer starken Bestrafung in der Regularisierung.

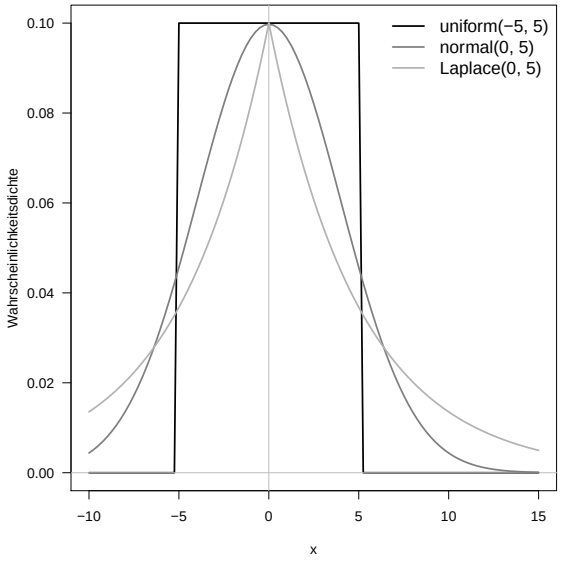



**Abb. A.6:** Drei mögliche *prior* für Modellparameter: uniform, normal und Laplace. Es wird deutlich, dass der Laplace-*prior* die Schätzer am stärksten Richtung 0 zieht. (Für die Normalverteilung wären Werte um 10 oder höher typisch, während für die Laplace-Verteilung kleinere Werte üblich wären, die zu stärkerem Abfall und somit stärkerer Bestrafung führen.)

Bayessche Statistik macht eine Aussage über einen fixen Datensatz: die Parameter sind Zufallsvariablen, aber die Daten sind "gesetzt". Das unterscheidet Bayes deutlich vom typischen GLM, in dem die Daten die Zufallsvariablen sind, und deshalb der *maximum likelihood*-Schätzer *einen* Wert hat (den wir allerdings nur mit einer gewissen Unsicherheit bestimmen können).

In der Praxis ist dieser philosophische Unterschied nicht so groß. Auch die Bayes-Gleichung benutzt die *likelihood* als Abstandsmaß, und häufig sind die *prior* uninformativ, so dass die mit Bayesschen Verfahren berechneten Parameter ähnlich denen der *maximum likelihood* sind. Jaynes (2003) betont allerdings immer wieder, dass der Bayessche Ansatz logisch, vollständig und konsistent ist, während *maximum likelihood* zu inkorrekten Ergebnissen führen kann (wenngleich diese Beispiele häufig etwas konstruiert wirken).

### A.4.1 Marginale *posteriors* plotten

Ein wichtiger Unterschied zu den frequentistischen Ansätzen ist die Betrachtung der *posterior*-Dichte der Schätzer. Da die Parameter ja Zufallsvariablen sind, interessieren uns deren Verteilungen. Je nach Parameter kann diese normalverteilt sein, oder auch sehr schief. Eine Zusammenfassung durch Mittelwert und Standardabweichung ist im letzteren Fall nicht sinnvoll. Entsprechend erhalten wir in der Textausgabe zumeist auch noch einige Quantilen angezeigt, oder wir plotten direkt die *posterior* (Abb. A.7).

Die dazugehörige Textausgabe sieht etwa so aus (hier aus JAGS):

```
 n.sims = 10000 iterations saved
         mu.vect sd.vect    2.5%     25%     50%     75%   97.5%  Rhat n.eff
beta0      1.465   0.196   1.077   1.346   1.468   1.596   1.830 1.003  1300
beta1      0.147   0.058   0.044   0.113   0.147   0.183   0.253 1.006  1400
```

Neben den Mittelwerten und Standardfehlern (= Standardabweichung der MCMC-Werte) ist es üblich mehrere Quantilen anzugeben, die für schiefe Verteilungen sinnvoller sind. Die letzten beiden Spalten stellen die Gelman-Rubin-Statistik dar, die im nächsten Abschnitt als

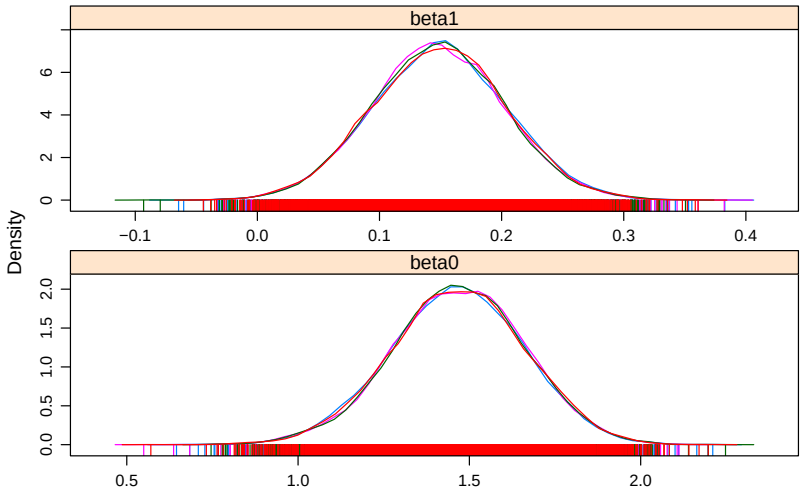



**Abb. A.7:** Posterior density zweier Parameter ($\beta_0$ und $\beta_1$) aus vier Ketten. Jede Linie stellt eine separate MCMC-Kette dar, die Strichlein am unteren Rand die einzelnen Werte der MCMC-Kette.

Konvergenzdiagnostik vorgestellt wird.

Was dieser *posterior* zeigt, ist eine Projektion der mehrdimensionalen *posterior*-Wahrscheinlichkeitsdichte (mit einer Dimension je Parameter) auf die eine Dimension des ausgewählten Parameters. Sprich, aus einer k-dimensionalen Landschaft wird ein 1-dimensionales Profil. Man bezeichnet entsprechend diese Abbildung als *marginal posterior*, also als Randsumme über alle anderen Dimensionen hinweg: diese werden "ausmarginalisiert" oder ausintegriert:

$$P(\theta_i|Y) = \int_{\theta_{[-i]}} P(\theta|Y) d\theta_{[-i]} \tag{A.9}$$

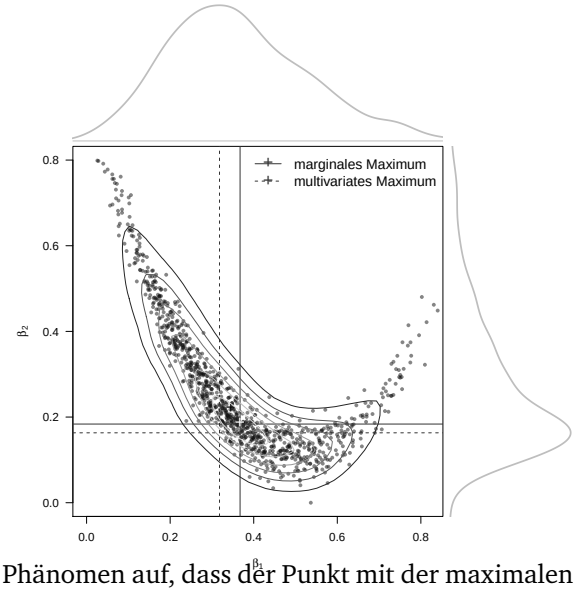
Solche *marginal posteriors* können in die Irre führen, vor allem, wenn die Form der *likelihood* bananenförmig durchgebogen ist (siehe Box). Im nächsten Kapitel schauen wir uns am konkreten Beispiel auch den Korrelationsplot der beiden Parameter an.

### A.4.2 Parameterkorrelation und Effizienz

Probleme gibt es bei den MCMC-Verfahren in der praktischen Umsetzung vor allem aus zwei Gründen. Zum einen sind manche Parameter korreliert, so dass die *likelihood*-Oberfläche schmale Grate aufweisen kann, die schwierig effektiv zu beproben sind. Zum anderen kann es sein, dass unsere Vorschläge für den nächsten Wert der MCMC-Kette zu dicht am gegenwärtigen Wert liegen (unsere *proposal distribution* also zu eng ist), und die übernächsten Werte nicht wie gefordert unabhängig von den alten sind. Dies bezeichnen wir als „Autokorrelation" in den Ketten.

Das erste Problem, Parameterkorrelation, können wir im Prinzip einfach durch längere

**Exkurs 2: Bananen, mehrdimensionale und irreführende *marginal posteriors***
Im MCMC werden ja, wie in Box 1 dargestellt, alle Modellparameter gleichzeitig verändert. (Oder, wie man korrekterweise sagen würde: Es wird aus der gemeinsamen Verteilung („sampling the joint posterior distribution") gezogen.) Nun ist diese gemeinsame *posterior* nicht notwendigerweise multivariate normal, sondern gerne auch „bananenförmig" (siehe Abb.).




In diesem Fall taucht das Phänomen auf, dass der Punkt mit der maximalen *gemeinsamen* Wahrscheinlichkeitsdichte nicht der gleiche Punkt ist, wie der mit den marginalen maximalen Wahrscheinlichkeitsdichten. Sprich, die Verteilung von Parameter $\beta_1$ alleine hat einen anderen Maximalwert, als der Maximalwert beider Parameter zusammen. In unserer Abbildung ist dies besonders sichtbar für den Parameter der x-Achse, $\beta_1$, dessen Maximum (gestrichelt) niedriger liegt als das gemeinsame Maximum von $\beta_1$ und $\beta_2$ (durchgezogene Linien). Auch die marginalen Mediane (schwarzes $+$) sind nicht besser.
Da wir in typischen Modellen immer nur marginale Parameter anschauen können (wir können ja nicht 12-dimensional plotten), müssen wir uns gewahr sein, dass diese Parameterzusammenfassung *nicht* unbedingt auch den maximalen *joint posterior* darstellen!
Der sog. *maximum a posteriori*-Wert (MAP) ist der Modalwert der *joint likelihood*. Hier haben wir mittels kernel-Schätzern den MAP bestimmt, wie durch die Kontourlinien angedeutet. Da der MAP ein Punktschätzer ist, verlässt er den für Bayessche Statistik üblichen Verteilungen. Er entspricht etwa dem *maximum likelihood*-Schätzer im GLM (bei uninformativem *prior*).

Ketten lösen. Früher oder später müssen die Ketten mischen, und müssen die MCMC-Proben die wahre *likelihood*-Oberfläche beschreiben, so will es der Beweis von Metropolis. Nur ist das u.U. nicht effizient. Wenn also nach 100.000 Vorschlägen immer noch keine Konvergenz in Sicht ist, dann sollten wir die Option erwägen, das Modell neu zu formulieren. Dafür können wir z.B. zwei stark korrelierte Variablen $A$ und $B$ ersetzen durch die beiden Hauptkomponenten einer PCA von $(A, B)$, die ja orthogonal und damit unkorreliert sind.

Die Autokorrelation ist zwar theoretisch lästiger, aber praktisch unproblematisch. Unser Konvergenzkriterium ($\widehat{R}$) stellt sicher, dass wir ausreichend lange Ketten haben. Wenn in diesen Ketten jetzt Autokorrelation vorliegt, dann könnten wir versuchen, diese durch ausdünnen zu reduzieren: wir nehmen einfach nur jeden 4. oder 10. Wert. Allerdings verlieren wir dadurch viele Schätzer, und können uns doch nicht sicher sein, dass im ausgedünnten Datensatz die Autokorrelation beseitigt ist. Deshalb wird typischerweise vom Ausgedünnen abgeraten (Link & Eaton 2012) – was es für uns einfacher macht.

## A.5 Warum das Ganze? Wegen der Flexibilität!

Über viele Seiten haben wir jetzt Ideen und Teile der Techniken Bayesscher Statistik kennengelernt. Ziemlich viel Neues, zum Teil Verwirrendes, und das alles, um eine Regression durchzuführen, die wir bisher ohne Probleme als GLM gerechnet habe?

Nahezu alle Beispiele im Buch sind einfach, sogar sehr einfach, und die Wirklichkeit ist komplizierter. Was, wenn unsere Daten nicht durch Normal-, Poisson- oder Binomialverteilung beschrieben werden können? Tatsächlich gibt eine ganze Reihe an statistischen Problemen,[11] die mit Bayesschen Verfahren deutlich einfacher (oder gar ausschließlich so) zu lösen sind, als frequentistisch (siehe etwa die Bücher von Clark & Gelfand 2006; Clark 2007; Gelman & Hill 2007; Kéry 2010; Kéry & Schaub 2011; Gelman et al. 2013; Kéry & Royle 2015).

Neben dieser technischen Begründung für Bayessche Statistik zählt aber nochmehr, dass Bayessche Statistik den Blick auf die Daten verändert. Wir sehen sie als Evidenz für verschiedene Hypothesen/Theorien, wir überprüfen unsere Sichtweise mit neuen Daten, und tun nicht so, als wäre jeder Datensatz in einem Vakuum entstanden. Es würde mich nicht wundern, wenn in 10 oder 20 Jahren eine Situation erreicht ist wie vor den Fisher'schen Frequentismusbemühungen: das wieder alle Statistik Bayessch ist.

Dieses Kapitel stellt einen ersten Schritt dar. Für weitere Schritte siehe Gelman & Hill (2007); Kéry (2010); Kruschke (2015). Wer dann mehr über die Grundlagen und die Mathematik lernen will, sollte Jaynes (2003) und natürlich Gelman et al. (2013) lesen.

[11] Etwa: gekreuzte Zufallseffekte; Mischung von Verteilungen; Überschuss an 0-Werten (*zero inflation*); *state-space*-Modelle (mit verschiedenen Fehlerquellen).

# Anhang B

# Bayessche Statistik mit R

*Bayesian statistics is difficult in the exactly the same way that thinking is difficult.*
Unknown

Wir haben uns inzwischen an die Benutzung von R gewöhnt, und jetzt müssen wir unsere Kenntnisse um einen R-Dialekt erweitern. Es gibt eine Reihe Bayesscher R-Pakete für spezifische Fragestellungen, auf die wir später kurz eingehen. Die flexiblen Ansätze, die fragestellungsunabhängig sind, benutzen Software außerhalb von R.

## B.1 Bayes mit JAGS

Am einfachsten zu lernen ist der Dialekt BUGS (*Bayesian inference Using Gibbs Sampling*), eine in S programmierte Bayes-Software. Diese BUGS-Sprache ist in verschiedenen Programmen implementiert, wir benutzen JAGS (*Just Another Gibbs Sampler*).[1]

Wir beginnen damit, dass wir JAGS installieren.[2] JAGS ist (wie R) ein *open source*-Programm und hier erhältlich: `http://mcmc-jags.sourceforge.net`. JAGS wird wie eine normale Software installiert und dann von im Weiteren von R aufgerufen.[3]

JAGS hat zwei substantielle Unterschiede zu R: Es gibt keine Vektorisierung, und wir müssen das durch `for`-Schleifen ersetzen. Das macht den Code etwas hässlicher. Der zweite Unterschied ist, dass die Definitionen von Verteilungen (etwa `dnorm`) anders sind: ähnlich genug, um sie sofort zu begreifen, aber unterschiedlich genug, um immer wieder Fehler zu machen. Konkret ist wichtig zu wissen, dass die Normalverteilung durch Mittelwert und *Präzision* definiert ist (nicht durch Standardabweichung). Präzision ist einfach 1/Varianz, und diese Art der Parametrisierung hat den Grund, dass dann ein $\gamma$*-prior* für die Präzision sinnvoll ist.

Wir nähern uns diesem Dialekt wie immer am besten durch Beispiele, in denen wir Beispiele aus früheren Kapiteln in JAGS fitten.

[1] Der Code sollte aber ohne Probleme auf WinBUGS oder OpenBUGS oder auch Nimble übertragbar sein. JAGS ist Java-basiert und betriebssystemunabhängig. Außerdem sind die Fehlermeldungen dort am wenigsten unverständlich. Der gegenwärtige Star am Bayesschen Softwarehimmel ist STAN, aber der Syntax dort ist etwas komplizierter und der Geschwindigkeitsgewinn für unsere Zwecke gering.

[2] Hier im Detail beschrieben: `https://sourceforge.net/projects/mcmc-jags/files/Manuals/4.x/jags_installation_manual.pdf/download`

[3] Ob die Installation geklappt hat, kann man dadurch überprüfen, dass man auf der Komandozeile `jags` eingibt und hoffentlich eine Rückmeldung bekommt. Dann JAGS mit Strg-C wieder abbrechen.

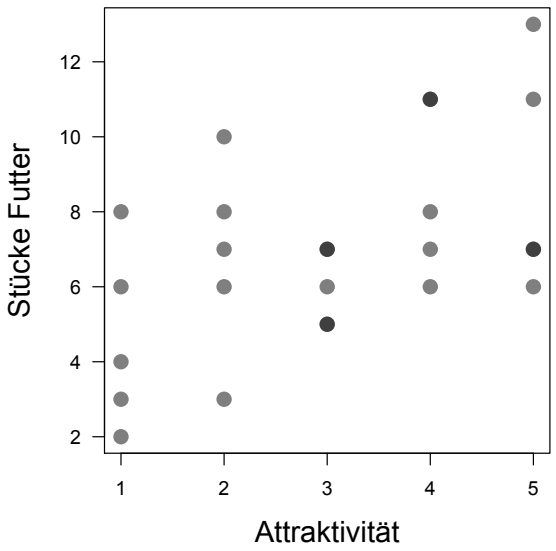



**Abb. B.1:** Fütterungsaktivität von Halsbandschnäppermännchen in Abhängigkeit von ihrer Attraktivität. Diese einfachen Daten dienen als Beispiel für ein Poisson-GLM in JAGS.

### B.1.1 Eine einführende Poisson-Regression

#### Das Modell zum Laufen bringen

Am Beispiel der Halsbandschnäpperdaten (siehe Abb. B.1), die wir zunächst einladen und plotten:

```
> library(R2jags)
> schnaepper <- read.table("schnaepper.txt", header=T)
> par(mar=c(5,5,1,1))
> plot(stuecke ~ attrakt, data=schnaepper, las=1)
```

Als ersten Schritt müssen wir JAGS die Daten als Liste aufbereiten. Wir übergeben einen Vektor mit den Antwortvariablen (hier fantasielos `Y` genannt, eine Matrix mit den Prädiktoren (hier nur einer: `X`) und der Anzahl Datenpunkte, über die wir nachher eine Schleife laufen lassen (`N`).

```
> # 1. Daten ins JAGS-Format: X, Y, und N
> jagsDaten <- list(X=schnaepper$attrakt, Y=schnaepper$stuecke, N=nrow(schnaepper))
```

Jetzt müssen wir das Regressionsmodell in einer für JAGS verständlichen Art beschreiben. Diese ist in R verfasst, aber hat eben die oben erwähnten Besonderheiten.

Unsere Regression sieht ja so aus: $Y \sim \text{Pois}(\lambda = e^{\beta_0 + \beta_1 X})$. Wir zerlegen diese Formulierung in zwei Zeilen. Die erste beschreibt, dass $Y$ aus einer Poisson-Verteilung gezogen wird: $Y \sim \text{Pois}(\widehat{\lambda})$, wobei $\widehat{\lambda}$ der Schätzwert aus dem Modell ist (deshalb das Dach). In der zweiten Zeile steht, wie wir diesen Schätzwert aus `X` bestimmen: $\log(\widehat{\lambda}) = \beta_0 + \beta_1 X$.

Schließlich müssen wir noch die *prior* für die beiden Modellparameter $\beta_0$ und $\beta_1$ festlegen. Diese können ja grundsätzlich beliebige Werte annehmen, und wir haben jetzt erst einmal keine Ahnung, welche. Als wenig informative *prior* benutzen wir deshalb die Normalverteilung zentriert auf 0, mit einer weiten Standardabweichung (= geringer Präzision).

Das zusammen sieht dann in JAGS so aus:

```
> # 2. Modell formulieren
> schnaepperModel <- function(){
```

```
+   # likelihood function
+   for (i in 1:N){
+     Y[i] ~ dpois(lambdaHat[i])
+     log(lambdaHat[i]) <- beta0 + beta1 * X[i]
+   }
+   # priors
+   beta0 ~ dnorm(0, 0.01)
+   beta1 ~ dnorm(0, 0.01)
+ }
```

Jetzt legen wir die Startwerte fest (am besten mit einer Funktion, die diese zufällig erzeugt), die Einstellungen des MCMC-Algorithmus fest (Anzahl Ketten, Anzahl Schritte, Ausdünnung der Ketten), sowie welche Modellparameter wir mitschreiben wollen.

```
> # 3. Parameter einstellen
> Parameter <- c("beta0", "beta1")
> Startwerte <- function(N) replicate(N, as.list(c("beta0"=rnorm(1, 0, 4),
+  "beta1"=rnorm(1, 0, 4))), simplify=F)
> n.chains <- 4; n.iter <- 1000; n.thin <- 1
```

Nun haben wir alles zusammen und können JAGS aufrufen und mittels der MCMC-Method die beiden Parameter schätzen:

```
> # 4. JAGS aufrufen
> fjags <- jags(data = jagsDaten, inits = Startwerte(n.chains), parameters.to.save =
+  Parameter, model.file = schnaepperModel, n.chains = n.chains,
+  n.iter = n.iter, n.thin = n.thin)

Compiling model graph
   Resolving undeclared variables
   Allocating nodes
Graph information:
   Observed stochastic nodes: 25
   Unobserved stochastic nodes: 2
   Total graph size: 72

Initializing model

  |**************************************************| 100%
```

Die Informationen, die wir als Rückmeldung von JAGS erhalten, berichten, dass das Modell erfolgreich zusammengesetzt wurde, wie groß es ist, und der *progress bar* begleitet den tatsächlichen MCMC-*sampling*-Prozess. In diesem Schritt treten die meisten Fehler auf, die wir dann mit Hilfe des JAGS-Manuals[4] und verschiedener Foren im Internetz zu lösen versuchen.

```
> # 5. Modellzusammenfassung
> fjags

Inference for Bugs model at "/var/folders/cc/3jfhfx190rb2ptxnqrqxj94m0000gp/T//
  RtmpgXmZt2/modela99b5c1243ce.txt", fit using jags,
 4 chains, each with 1000 iterations (first 500 discarded)
 n.sims = 2000 iterations saved
         mu.vect sd.vect    2.5%     25%     50%     75%   97.5%  Rhat n.eff
```

[4] https://sourceforge.net/projects/mcmc-jags/files/Manuals/4.x/jags_user_manual.pdf/download

```
beta0      1.440   0.346   1.025   1.334   1.465   1.584   1.862 1.069   390
beta1      0.148   0.094   0.038   0.113   0.147   0.185   0.277 1.110  1600
deviance 118.900 110.156 111.470 111.960 112.802 114.228 120.969 1.176   420

For each parameter, n.eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor (at convergence, Rhat=1).

DIC info (using the rule, pD = var(deviance)/2)
pD = 6052.1 and DIC = 6171.0
DIC is an estimate of expected predictive error (lower deviance is better).
```

An diese Ausgabe müssen wir uns gewöhnen, wenn wir mit JAGS arbeiten. Die erste Zeile ist unwichtig. JAGS schreibt eine Textdatei mit dem Modell irgendwo auf unseren Rechner, und das ist der Pfad dahin. Diese Datei wird typischerweise automatisch wieder gelöscht.

Interessanter ist die nächste Zeile, die uns angibt, wie viele Ketten für wie lange gelaufen sind. Die erste Hälfte wird bei Benutzung der Grundeinstellungen weggeworfen, da hier der MCMC-Algorithmus noch von den Startwerten seinen Weg Richtung wahrer *posterior*-Verteilung finden muss. `n.sims` gibt die daraus resultierende Anzahl MCMC-*samples* über alle Ketten hinweg an.

Es folgt die standardisierte Darstellung aller Modellparameter, die wir uns ausgeben ließen plus der *deviance*. Angegeben werden jeweils der Schätzer und sein Standardfehler (wie im GLM), sowie verschiedene Perzentilen des *posterior*, für den Fall, dass dieser asymmetrisch ist und der Standardfehler deshalb wenig hilfreich ist. Wenn die 2.5% und 97%-Perzentilen *beide* größer oder beide kleiner 0 sind, dann ist dieser Parameter signifikant von 0 verschieden und würde im GLM ein Sternchen erhalten. Es folgen zwei sehr wichtige Spalten: der `Rhat`-Wert, der Konvergenz der Schätzung anzeigt und $< 1.02$ sein sollte. Und die effektive Anzahl unabhängiger MCMC-*samples*, wenn man die zeitliche Autokorrelation der Ketten berücksichtigt: `n.eff`.

### MCMC-Diagnostik & Feinabstimmung

Wir sind jetzt leider noch nicht fertig. Es folgen noch (mindestens) zwei Schritte: die MCMC-Anpassung (MCMC-Diagnostik), und die Modelldiagnostik.

Dass der `n.eff`-Wert deutlich niedriger liegt für $\beta_0$ als `n.sims` ist uns ein Hinweis, dass die Ketten mit zeitlicher Autokorrelation zu kämpfen haben. Also erhöhen wir das Ausdünnen der Ketten zunächst auf 3, d.h. nur jeder dritte Wert wird gespeichert.

```
> n.thin <- 5
> fjags <- jags(data = jagsDaten, inits = Startwerte(n.chains), parameters.to.save =
+  Parameter, model.file = schnaepperModel, n.chains = n.chains, n.iter = n.iter,
+  n.thin = n.thin)
... # snip: JAGS output nicht wiedergegeben
> fjags

...
 4 chains, each with 1000 iterations (first 500 discarded), n.thin = 5
 n.sims = 400 iterations saved
         mu.vect sd.vect    2.5%     25%     50%     75%   97.5%  Rhat n.eff
beta0      1.405   0.636   0.961   1.323   1.471   1.598   1.840 1.174   350
beta1      0.139   0.180   0.037   0.111   0.147   0.190   0.256 1.134   400
deviance 142.281 348.594 111.451 112.046 112.945 114.289 120.518 1.073   400
```

```
For each parameter, n.eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor (at convergence, Rhat=1).

DIC info (using the rule, pD = var(deviance)/2)
pD = 60940.4 and DIC = 61082.6
```

Jetzt ist `n.eff` sehr ähnlich `n.sim`, d.h. die Ausdünnung ist so in Ordnung.

Wir wenden uns dem zweiten MCMC-Problem zu, der Konvergenz. Diese wir mit der $\widehat{\mathrm{R}}$-Statistik quantifiziert, und der Wert sollte unter 1.02 liegen. Das ist hier nicht der Fall, weshalb wir die Ketten länger laufen lassen müssen. Wir erhöhen also die Kettenlänge auf 5000:

```
> n.iter <- 5000
> fjags <- jags(data = jagsDaten, inits = Startwerte(n.chains), parameters.to.save =
+  Parameter, model.file = schnaepperModel, n.chains = n.chains, n.iter = n.iter,
+  n.thin = n.thin)

... # snip
 4 chains, each with 5000 iterations (first 2500 discarded), n.thin = 5
 n.sims = 2000 iterations saved
         mu.vect sd.vect    2.5%     25%     50%     75%   97.5%  Rhat n.eff
beta0      1.464   0.223   1.080   1.342   1.465   1.595   1.835 1.016  2000
beta1      0.147   0.058   0.040   0.111   0.148   0.182   0.254 1.001  2000
deviance 114.465  41.263 111.467 111.965 112.838 114.161 118.816 1.223  2000

For each parameter, n.eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor (at convergence, Rhat=1).

DIC info (using the rule, pD = var(deviance)/2)
pD = 851.4 and DIC = 965.8
```

Jetzt sehen unsere `Rhat`-Werte gut aus, und wir haben immer noch genügend effektive MCMC-Schritte ($> 500$).

Ein weiteres diagnostisches Werkzeug zur Beurteilung der Kettenmischung ist der *trace plot* (in diesem Fall ohne den *trace plot* für die *deviance*: Abb. B.2).

```
> fjagsWindow <- window(as.mcmc(fjags), start=0.51*n.iter)
> par(mfrow=c(3,1))
> traceplot(fjagsWindow, las=1, lty=1)
```

Was wir erkennen (sollten) ist, dass die 4 Ketten sehr gut mischen, alle um die gleichen Mittelwerte streuen, so dass offensichtlich egal ist, von wo aus die Ketten gestartet sind. Solange die gleitenden Mittelwerte nicht horizontal sind, läuft etwas falsch (meist zu wenig Schritte bei ungenügender Ausdünnung). Diese *trace plots* sehen sehr gut aus.

Wir können uns nun um das letzte Elemente der JAGS-Ausgabe kümmern, pD und DIC. Der DIC, *deviance information criterion*, ist das Bayessche Äquivalent zum AIC, beruhend auf der Variabilität in DIC gegenüber Veränderungen in den Parametern (siehe Spiegelhalter et al. 2002, für Details). Beim AIC geht die Anzahl Parameter ein, und pD ist das aquivalente Maß für Modellkomplexität bei MCMC. Wenn wir die drei Modellfits bisher vergleichen sehen wir, dass der DIC hier nichts taugt: je nach Modell variiert der DIC um eine Größenordnung! Das ist in der Tat eine häufig geäußerte Kritik, und selbst viel längere Ketten lösen das Problem häufig nicht. (Eine Erhöhung auf 50000 Schritte liefert immer noch Werte zwischen 120 und

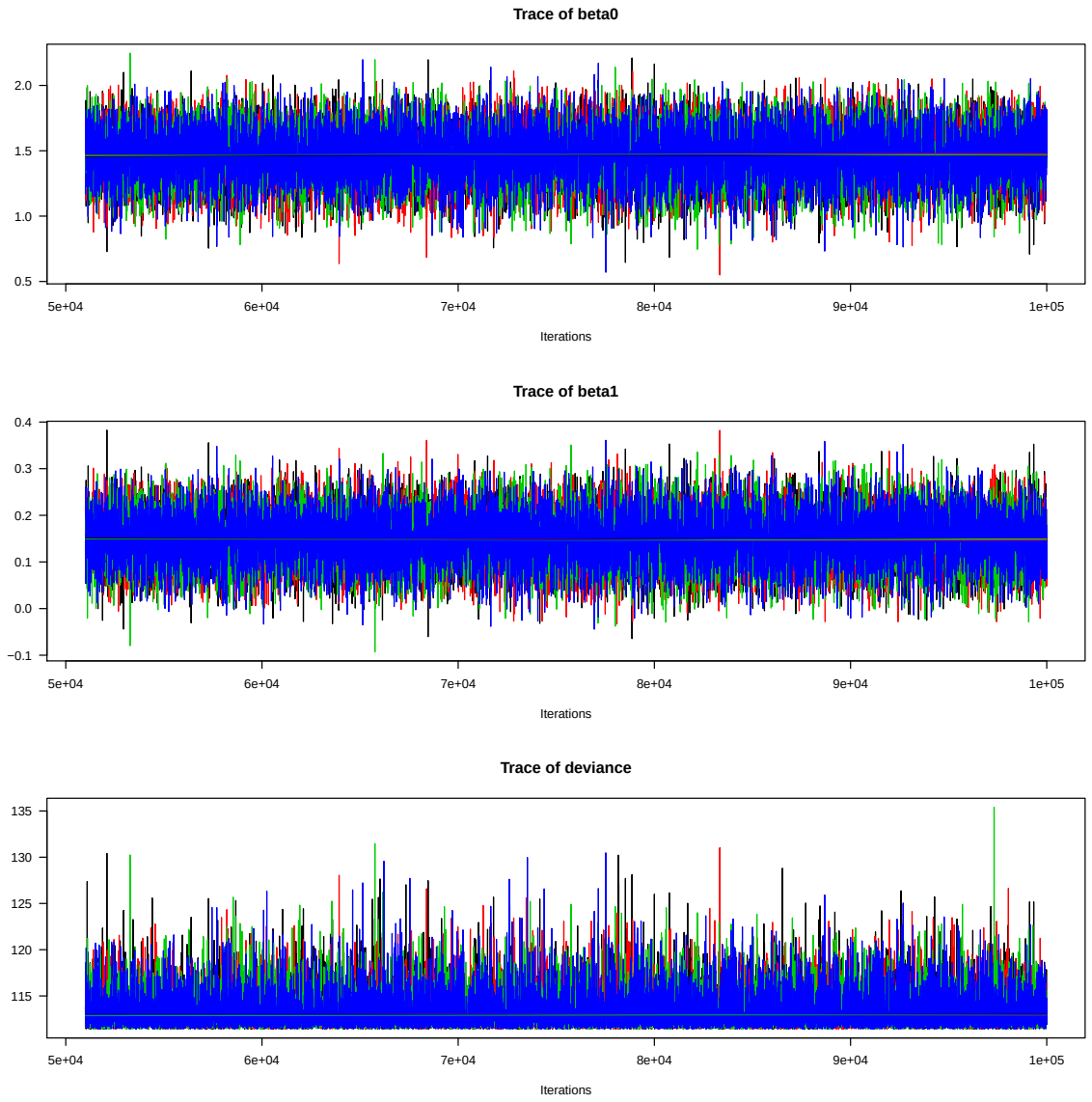



**Abb. B.2:** Auf der y-Achse ist der Parameterwert aufgetragen, auf der x-Achse die (ausgedünnten) MCMC-Schritte. Jede Kette ist durch eine andere Farbe repräsentiert. Die sogenannte *burn-in*-Phase ist abgeschnitten. Die horizontalen Linien sind ein gleitender Mittelwert jeder Kette.

1300. Bei 1E5 Schritten sind wir immer noch bei über 100% Variation bei nur 2 Läufen.) Deshalb werden wir im weiteren mit dem Argument `DIC=FALSE` die Ausgabe unterdrücken.[5]

### Modellinterpretation

Zunächst einmal sollten wir überprüfen, ob ein normales GLM und unser Modell ähnliche Ergebnisse liefern.

```
> summary(glm(stuecke ~ attrakt, data=schnaepper, family=poisson))

Call:
glm(formula = stuecke ~ attrakt, family = poisson, data = schnaepper)

Deviance Residuals:
```

[5]Das Konzept des DIC ist schon ziemlich klever, und eine Verallgemeinerung der Freiheitsgrade und des AIC (Wood 2015). Allerdings ist die Schätzung des Wertes numerisch sehr instabil, wodurch es wenig nützlich wird.

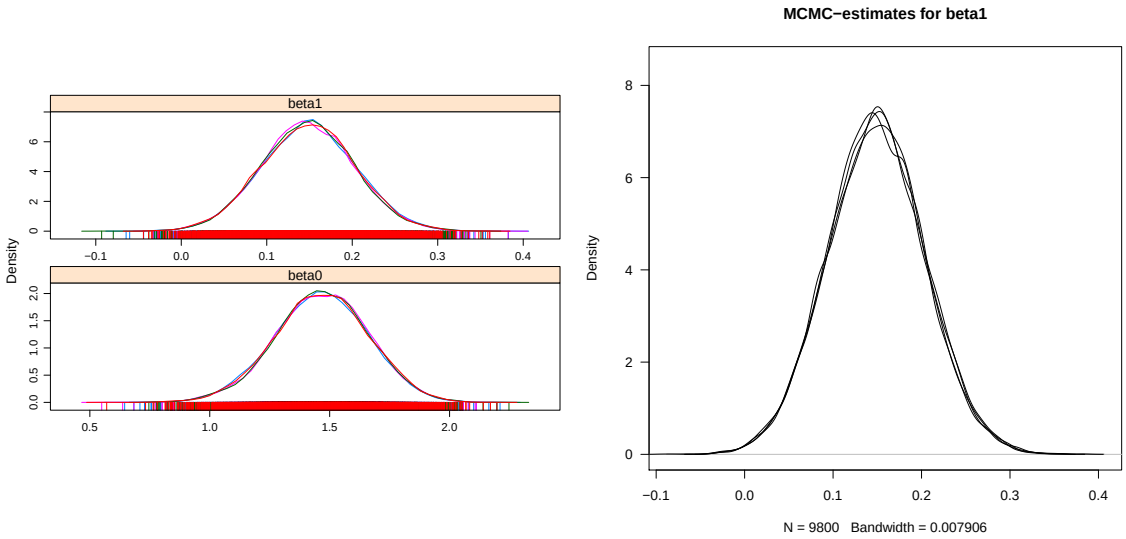



**Abb. B.3:** Dichte der *posteriors* für den Schätzer der Modellparameter $\widehat{\beta}$. Links, mittels eines Befehls aus **lattice**, rechts das obere linke Feld mit Standardbefehlen. Linien sind die vier unterschiedlichen Ketten.

```
     Min        1Q    Median        3Q       Max
-1.55377  -0.72834   0.03699   0.59093   1.54584

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  1.47459    0.19443   7.584 3.34e-14 ***
attrakt      0.14794    0.05437   2.721  0.00651 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 25.829  on 24  degrees of freedom
Residual deviance: 18.320  on 23  degrees of freedom
AIC: 115.42
```

Diese Werte sehen denen der JAGS-Ausgabe sehr ähnlich. Die *deviance* ist allerdings sehr anders, und auch sehr ungenau geschätzt. Entsprechend sind AIC und DIC praktisch nicht vergleichbar.

Eine typische Art und Weise, Ergebnisse eines MCMC-Laufs zu visualisieren, sind die Dichten der *posterior*. Wir haben ja aus dem letzten Modell 2000 Ziehungen, die wir darstellen können. Dazu gibt es (mindestens) einen einfachen und (mindestens) einen etwas umständlicheren Weg. Beide seien hier vorgestellt (Abb. B.3).

```
# einfach: links
> library(lattice)
> densityplot(fjagsWindow[,1:2]) # nicht deviance
# umständlich: rechts
> plot(density(fjagsWindow[[1]][,2]), las=1, ylim=c(0, 8.5), main="MCMC-estimates
+  for beta1")
> for (i in 2:length(fjagsWindow)) lines(density(fjagsWindow[[i]][,2]))
```

Nach der langen Diskussion zum Thema *prior* wäre es doch naheliegend, die gewählten

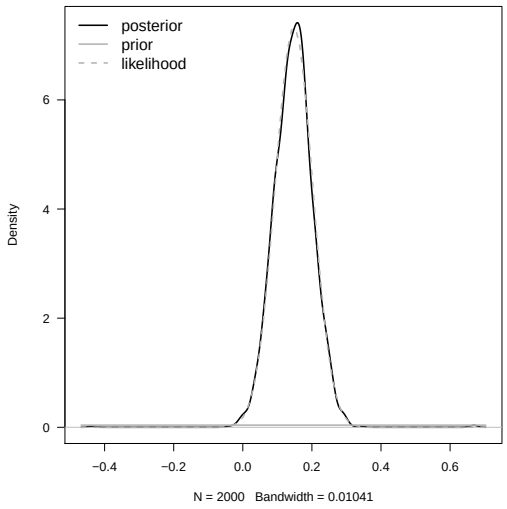



**Abb. B.4:** Vom *prior* über die *likelihood* zum *posterior*: in diesem Fall ist der *prior* wirklich ohne Einfluss auf die *posterior*, und alle Information kommt aus den Daten (d.h. der *likelihood*).

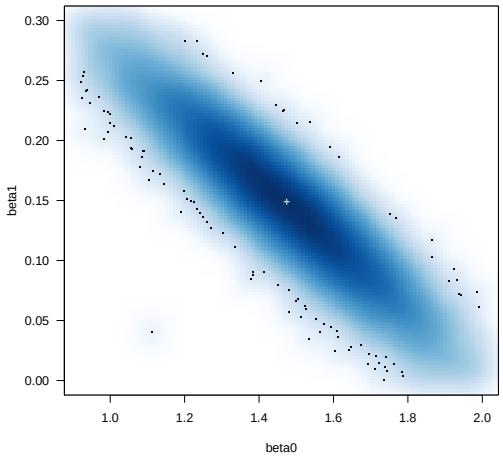



**Abb. B.5:** Die Parameterschätzer für $\beta_0$ und $\beta_1$ sind in den MCMC-Ketten negativ korreliert ($r = -0.88$). Diese Korrelation wird von JAGS berücksichtigt, um nicht den sowieso viel schlechteren weißen Bereich zu beproben. Das weiße Kreuz ist der *maximum likelihood*-Schätzer aus dem GLM.

*prior* zu variieren, um ihren Einfluss abzuschätzen. Dafür gibt es ein einfachere Alternative: das Gegenüberstellen von *prior*, *likelihood* und *posterior*. Die *likelihood* können wir aus dem GLM extrahieren, das ja mittels *maximum likelihood* die Parameter geschätzt hat. Wie Abb. B.4 zeigt, hat unser *prior* überhaupt keinen Effekt auf den *posterior*. Damit erübrigt sich ein Variieren des *priors*, was ganz anders aussähe, wenn der *posterior* dem *prior* ähnlich wäre.

Wir können uns das Ergebnis des MCMC-Verfahrens anschauen, indem wir die ausgetesteten Kombinationen von $\beta_0$ und $\beta_1$ als Dichteplot anschauen (Abb. B.5). Ganz klar erkennen wir, dass wann immer ein höherer Wert für $\beta_0$ ausgewählt wird, dann passt dazu nur ein niedrigerer Wert von $\beta_1$. Offensichtlich wurden gar nicht alle Kombinationen dieser beiden Parameter ausprobiert, sondern nur die um diese Linie liegende. Und das ist auch so. JAGS benutzt die erste Hälfte (die sogeannte *burning*-Phase) der Ketten um abzuschätzen, wie sinnvoll beprobt werden kann. D.h., intern ist JAGS diese negative Korrelation „bewusst“, und aus Effizienzgründen werden um diese Linie hinweg neue Wertepaare für $\beta_0$ und $\beta_1$ generiert. Nebenbei bemerkt ist der Ausdruck “burnin” irreführend. In dieser Anpassungsphase

optimiert JAGS die Verteilung, aus der die Vorschläge für den nächsten Parametersatz gezogen werden (siehe Box 1 im vorigen Kapitel), und sie wird deshalb im JAGS-Manual auch als *adaptation phase* bezeichnet. Ein Einbrennen ist nicht nötig, da eine MCMC-Kette ja kein Gedächtnis hat und entsprechend sich nicht erst zum richtigen Wert hin entwickeln muss. (Tatsächlich gibt es über die laxe Verwendung von *burnin* und *adaptation* einige semantische Verwirrungen, die wir hier nicht lösen können.)

### Abbildung der geschätzten Funktion

Wir wollen jetzt natürlich auch die gefittete Funktion in die Daten einzeichnen. Und genau wie beim GLM hätten wir gerne ein 95%-Konfidenzintervall, das hier als *credible interval* bezeichnet wird.

Wir machen das in drei Schritten. Zunächst berechnen wir den Median für die beiden Parameter $\beta_0$ und $\beta_1$ und daraus die Regressionslinie. Zur Berechnung selbst erinnern wir uns an die Lineare Algebra aus der Schule: Wenn wir einen Datensatz X (zwei Spalten: eine mit Einsern für den Achsenabschnitt, eine mit Werten von 1 bis 5 für die Attraktivität) mit Parameterschätzervektor $\hat{\beta}$ Matrix-multiplizieren, erhalten wir einen Vektor Vorhersagewerte: $X\hat{\beta} = \hat{y}$.

```
> fjagsMedian <- apply(fjags$BUGSoutput$sims.matrix[, 1:2], 2, median)
> neuAttrakt.mat <- cbind(1, seq(1, 5, len=50))
> preds <- neuAttrakt.mat %*% fjagsMedian
> plot(stuecke ~ jitter(attrakt, 0.5), data=schnaepper, las=1, xlab="attrakt")
> lines(neuAttrakt.mat[,2], exp(preds), lwd=2)
```

Im zweiten Schritt machen wir das für alle 2000 MCMC-Werte $\widehat{\beta}$, und erhalten eine Matrix mit Vorhersagewerten: $X\widehat{\beta} = \widehat{Y}$.

```
> predsAlle <- neuAttrakt.mat %*% t(fjags$BUGSoutput$sims.matrix[, 1:2])
> matlines(neuAttrakt.mat[,2], exp(predsAlle), col=rgb(0.2, 0.2, 0.2, 0.02), lty=1)
```

Im dritten Schritt dampfen wir diese 2000 Vorhersagewerte je X-Wert wieder ein, und zwar auf ihre 95%-Quantilen. Normalerweise würden wir nur die Regressionslinie und dieses *credible interval* abbilden, aber hier der Vollständigkeit halber auch die 2000 einzelnen Linien (Abb. B.6).

```
> CIs <- apply(predsAlle, 1, quantile, c(0.025, 0.975))
> matlines(neuAttrakt.mat[,2], exp(t(CIs)), col="white", lty=1)
```

### Modelldiagnostik

Wie bei jedem normalen Modell müssen wir natürlich auch für Bayessche Modelle Modelldiagnostik durchführen. Sind die Daten entsprechend unser Annahmen verteilt (oder z.B. *overdispersed*)? Ist die Streuung in den Residuen so, wie wir das erwarten würden? Gibt es ein Muster in den Residuen?

Dies ist für Hunderte, Tausende MCMC-Proben ungleich lästiger zu tun, als für ein GLM, in dem es nur eine Lösung gibt.

Genau wie bei der Abbildung der Regressionslinie im letzten Abschnitt können wir zunächst die Residuen von der mittleren Regressionlinie berechnen (Abb. B.7).

```
> neuAttrakt.mat <- cbind(1, schnaepper$attrakt)
> predsX <- neuAttrakt.mat %*% fjagsMedian
```

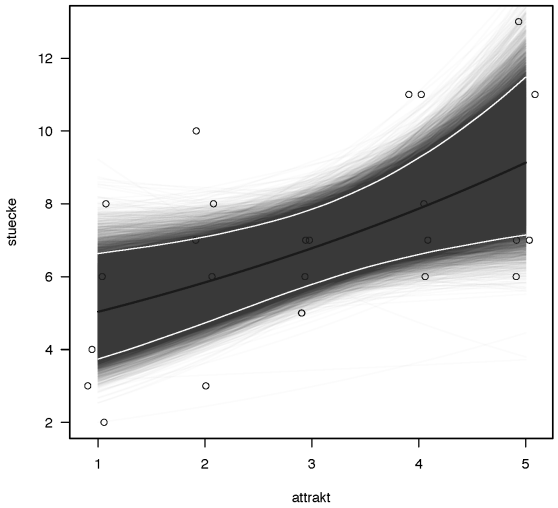



**Abb. B.6:** Geschätzte Poisson-Regression der Schnäpperdaten. Die Datenpunkte wurden etwas in x-Richtung verrauscht, damit sie nicht übereinanderliegen. Schwarze Linie ist die Regression aus dem Median der Schätzwerte, während die grauen Linie die 2000 einzelnen Regressionlinien jeder MCMC-Probe darstellen. Das 95%-Glaubwürdigkeitsintervall ist in weiß eingezeichnet.

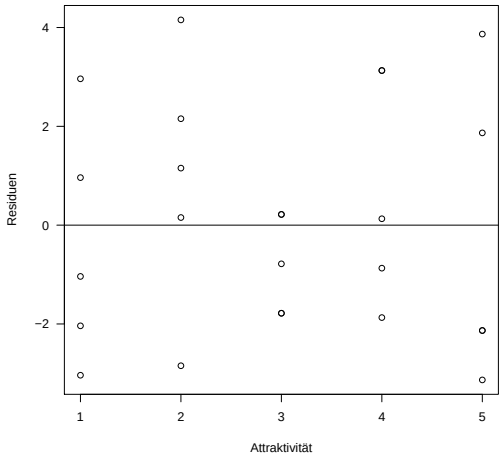



**Abb. B.7:** Residuen der Poisson-Regression. Kein Muster ist erkennbar, dabei ist doch zu erwarten, dass bei höheren Erwartungswert auch die Varianz höher ist.

```
> resids <- schnaepper$stuecke - exp(predsX)
> plot(resids ~ schnaepper$attrakt, las=1, xlab="Attraktivität", ylab="Residuen")
> abline(h=0)
```

Eine andere Möglichkeit der Modelldiagnostik ist uns aus dem **DHARMa**-Paket bekannt: Wir können aus dem Modell Daten simulieren, und diese mit den beobachteten Daten vergleichen. Für jeden der beobachteten Datenpunkte können wir 2000 Vorhersagen berechnen, und die Lage der Beobachtung darin abbilden:

```
> predsXAlle <- neuAttrakt.mat %*% t(fjags$BUGSoutput$sims.matrix[, 1:2])
> plot(density(exp(predsXAlle[1,])), main="", lwd=2, las=1)
> abline(v=schnaepper$stuecke[1], lwd=2, col="grey")
```

Mit Hilfe der `ecdf`-Funktion können wir berechnen, die uns die Quantile angibt, auf der sich

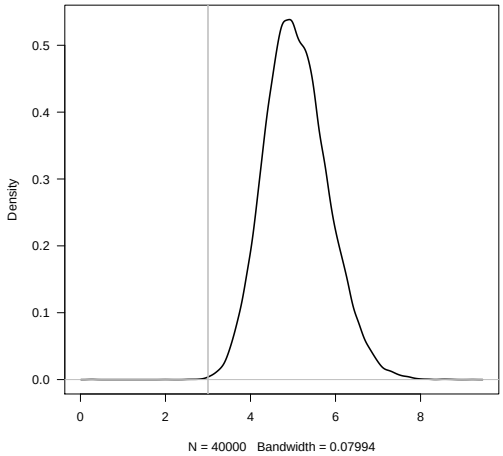



**Abb. B.8:** Verteilung der 2000 Vorhersagen für den ersten Datenpunkt (vertikale Linie).

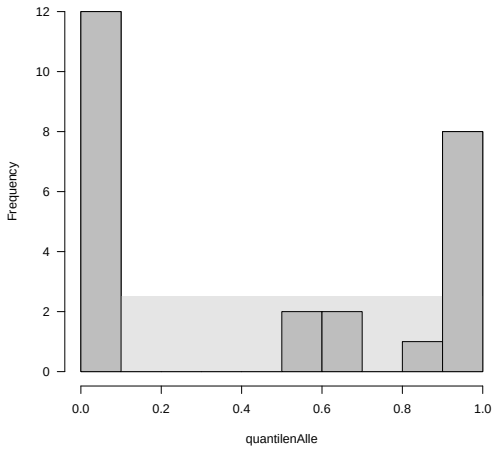



**Abb. B.9:** Verteilung der 25 Quantilenwerte gemäß Abb. B.8. Bei einer uniformen Verteilung wäre die vom grauen Rechteck angegebene Verteilung zu erwarten.

der beobachtete Punkt innerhalb der Simulationen befindet:

```
> ecdf(exp(predsXAlle[1,]))(schnaepper$stuecke[1])

[1] 0.00045
```

Das sagt uns so zunächst wenig. Aber wir würden erwarten, dass über alle Datenpunkte hinweg alle Quantilen gleichförmig vorkommen. Also berechnen wir diesen Quantilenwert für alle Daten und machen daraus ein Histogramm (Abb. B.9):

```
> quantilenAlle <- NA
> for (i in 1:nrow(schnaepper)){
+   quantilenAlle[i] <- ecdf(exp(predsXAlle[i,]))(schnaepper$stuecke[i])
+ }
> hist(quantilenAlle, breaks=11, las=1, col="grey", main="")
> rect(0, 0, 1, 1*2.5, col="grey90", lty=0)
> hist(quantilenAlle, breaks=11, las=1, col="grey", main="", add=T)
```

Ganz offensichtlich liegen die beobachteten Punkte entweder am unteren oder oberen Rand der 2000 Vorhersagen, aber kaum jemals dazwischen. Das ist bei so wenigen Datenpunkten

nun noch kein Drama, aber es deutet darauf hin, dass unser Regressionsmodell die Annahmen nicht ideal erfüllt. Für größere Datensätze wird dieses Werkzeug verlässlicher.

Häufig wird dieses Histogramm auf eine einzige Zahl reduziert: den Mittelwert. Dieser sogenannte „Bayessche P-Wert“ gibt an, wo im Mittel der Beobachtungswert im Simulationswert liegt, und sollte idealerweise 0.5 betragen.

```
> summary(quantilenAlle)

    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
0.000075 0.001475 0.578125 0.461888 0.965500 1.000000
```

Der Mittelwert liegt ein klein wenig unter 0.5, ist aber sehr vertretbar. Wir sehen aber auch, dass diese Minimalzusammenfassung auf einen Wert mehr versteckt als offenbart.

## B.2 Tipps & Tricks mit JAGS

Die Arbeit mit JAGS, und auch mit BUGS, erfordert manchmal ein paar Tricks, um das Ziel zu erreichen. In den folgenden Abschnitten werden wir uns ein paar Ideen und Tricks ansehen, die nicht zentral für das Verständnis der Bayesschen Statistik sind, aber sehr nützlich in der Anwendung mit JAGS.

Still to do:

1. splines
2. data augmentation
3. posterior predictive and Bayesian p-values –> steht schon z.T. in "Diagnostik"
4. mixed effect in JAGS
5. temporal and spatial autocorrelation (AR1 vs geht irgendwie nicht)

### B.2.1 Umgang mit `NA`s

Wir müssen unterscheiden zwischen NAs in der Antwortvariablen und denen in den Prädiktoren. Wenn manche Werte der Antwortvariablen fehlen, so können wir den Datensatz einfach mit den NAs an JAGS übergeben, und diese NAs werden vom Modell geschätzt! Im Modell gibt es ja eine Regression, und solange wir die $\mathbf{X}_i$-Werte eines Datenpunktes $i$ bereitstellen, wird automatisch von JAGS auch der Erwartungswert $\hat{y}_i$ berechnet. Für beobachtete Datenpunkte bestimmt ja der Vergleich von $y_i$ und $\hat{y}_i$ die *likelihood* (d.h. $y_i \sim V(\theta = \hat{y}_i)$); NAs werden dabei einfach nicht berücksichtigt. D.h., wir erhalten als Nebeneffekt des MCMC auch Vorhersagen für NAs. Genau so können wir auch effizient Vorhersagen machen: wir hängen die Trainingsdaten $\mathbf{X}_h$ einfach an unseren Datensatz an und übergeben als $y$-Werte NAs. Dann erhalten wir im Modelloutput direkt Verteilungen der *posterior* jedes Datenpunktes (den wir dann z.B. als Median oder Mittelwert zusammenfassen können).

Komplizierter wird es, wenn wir NAs in den Prädiktoren haben. Das ist in JAGS nicht möglich. Wir haben jetzt drei Optionen: (1) Alle Zeilen unserer Daten löschen, in denen ein NA vorkommt. (2) Daten vor der Benutzung von JAGS auffüllen („imputation“). Und (3) Prädiktorwerte als Teil des Modells schätzen. Option (3) ist ganz klar die beste, allerdings ist das nicht ganz einfach. Tatsächlich müssen wir nun ein zweites Modell kodieren, nämlich eines, dass die fehlenden X-Werte schätzt. Dieses zweite Modell wird gleichzeitig mit unserem

eigentlichen Modell für y in JAGS gefittet. Im einfachsten Fall ziehen wir die fehlenden X-Werte einfach aus einer Verteilung. Im besseren Modell versuchen wir mögliche Korrelationen innerhalb der Prädiktoren zu benutzen, um die fehlenden X-Werte zu schätzen.

Im folgenden Code-Schnipsel fehlen ein paar der binären X-Werte. Wir nehmen einfach an, dass alle X aus einer Binomialverteilung kommen, und wir ziehen die Wahrscheinlichkeit dieser Binomialverteilung aus einer uniformen Verteilung. Die so gezogenen X-Werte werden dann „einfach“ für X im Modell eingesetzt:[6]

```
model {
        Y[i] ~ dbern(p[i])
        logit(p[i]) <- b1 + b2*X[i]

        X[i] ~ dbern(pX[i])
        pX[i] ~ dunif(0,1)
}
...
}
```

Dies ist ein ziemlich „dummes“ Modell für die fehlenden X. Kompliziertere Schätzer für fehlende X würden den Rahmen hier sprengen (und ich fand leider auch keine vernünftigen, einfachen Beispiele im Internetz).

### B.2.2 Bayessches updating

In der Logik der Bayesschen Statistik ist ja die *posterior*-Verteilung das Ergebnis einer jeden Analyse. Sie enthält alle Informationen der Auswertung. Wenn nun neue Daten hinzukommen, dann müssen wir nicht die gesamte Analyse wiederholen, sondern wir nehmen einfach die zuvor berechnete posterior-Verteilung als prior für die neue Analyse. Dies bezeichnet man als „Bayessches updating“.

Wir schauen uns das jetzt mit einem Beispiel an, um das Prinzip zu verstehen, und ein wichtiges Detail. Allerdings wird Bayessches updating vor allem dann verwendet, wenn die Berechnungen sehr rechenaufwändig sind, und eine Analyse aller Daten „in Echtzeit“ nicht möglich ist.

Wir beginnen damit, dass wir einen Datensatz erfinden und in zwei Teile zerlegen, die wir dann fitten: erst Teil 1, dann Teil 2 mit den posterior aus Modell 1 als prior für Modell 2.

```
> set.seed(1)
> x <- seq(0, 2, len=100)
> y <- rnorm(100, mean=2 + 2*x - 1*x*x, sd=.5)
> index1 <- sample(100, 50)
> index2 <- setdiff(1:100, index1)
```

Jetzt fitten wir das erste Modell mit den durch `index1` ausgewählten Daten.

```
> jagsDaten <- list(x=x[index1], y=y[index1], N=50)
> model1 <- function(){
        +   # likelihood function
        +   for (i in 1:N){
        +     y[i] ~ dnorm(muHat[i], tau)
        +     muHat[i] <- beta[1] + beta[2] * x[i] + beta[3] * pow(x[i], 2)
        +   }
        +   # priors
```

[6] Quelle: `https://sourceforge.net/p/mcmc-jags/discussion/610036/thread/9bf85a85/`

```
        +   for (j in 1:3){
        +     beta[j] ~ dnorm(0, 0.01)
        +   }
        +   tau ~ dgamma(0.01, 0.01)
        + }
> # 3. Parameter einstellen
> Parameter <- c("beta", "tau")
> #Startwerte <- function(N) replicate(N, list("beta"=as.list(rnorm(3, 0, 4))))
> n.chains <- 4; n.iter <- 1000; n.thin <- 1
> # 4. JAGS aufrufen
> fjags <- jags(data = jagsDaten, inits = NULL, parameters.to.save = Parameter,
+   model.file = model1, n.chains = n.chains, n.iter = n.iter, n.thin = n.thin)

Compiling model graph
Resolving undeclared variables
Allocating nodes
Graph information:
Observed stochastic nodes: 50
Unobserved stochastic nodes: 4
Total graph size: 363

Initializing model

|**************************************************| 100%

> # 5. Modellzusammenfassung
> fjags

Inference for Bugs model at "/var/folders/cc/3jfhfx190rb2ptxnqrqxj94m0000gp/T//RtmpcR4vhL/modelab4d34e9
4 chains, each with 1000 iterations (first 500 discarded)
n.sims = 2000 iterations saved
         mu.vect sd.vect   2.5%    25%    50%    75%  97.5%  Rhat n.eff
beta[1]    1.967   0.233  1.507  1.812  1.968  2.124  2.424 1.001  2000
beta[2]    2.423   0.553  1.293  2.056  2.428  2.782  3.484 1.003  2000
beta[3]   -1.189   0.265 -1.719 -1.363 -1.192 -1.018 -0.643 1.001  2000
tau        4.066   0.841  2.623  3.471  3.988  4.601  5.817 1.003   930
deviance  72.919   2.924 69.184 70.780 72.252 74.446 80.395 1.004   620

For each parameter, n.eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor (at convergence, Rhat=1).

DIC info (using the rule, pD = var(deviance)/2)
pD = 4.3 and DIC = 77.2
DIC is an estimate of expected predictive error (lower deviance is better).
```

Die Parameterschätzer, konkret Mittelwerte und Standardabweichungen, stehen uns im Output zur Verfügung:

```
> fjags$BUGSoutput$mean$beta

[1]  1.967003  2.422593 -1.189001

> fjags$BUGSoutput$mean$tau

[1] 4.066097
```

```
> fjags$BUGSoutput$sd$beta

[1] 0.2327433 0.5532290 0.2645074

> fjags$BUGSoutput$sd$tau

[1] 0.8412583
```

Hieraus bauen wir die neuen prior für Modell 2, das wir mit dem 2. Teil der Daten füttern:

```
> jagsDaten <- list(x=x[index2], y=y[index2], N=50, betaMean=fjags$BUGSoutput$mean$beta, betaSD
> model2 <- function(){
        +   # likelihood function
        +   for (i in 1:N){
                +     y[i] ~ dnorm(muHat[i], tau)
                +     muHat[i] <- beta[1] + beta[2] * x[i] + beta[3] * pow(x[i], 2)
                +   }
        +   # INFORMATIVE priors !!
        +   for (j in 1:3){
                +     beta[j] ~ dnorm(betaMean[j], betaSD[j])
                +   }
        +   tau ~ dnorm(tauMean, tauSD)
        + }
> Parameter <- c("beta", "tau")
> fjags2 <- jags(data = jagsDaten, inits = NULL, parameters.to.save = Parameter,
+   model.file = model2, n.chains = n.chains, n.iter = n.iter, n.thin = n.thin)
[...]
```

Wir lassen die JAGS-Rückmeldung zum Modell aus Platzgründen weg.

```
> fjags2

Inference for Bugs model at "/var/folders/cc/3jfhfx190rb2ptxnqrqxj94m0000gp/T//RtmpcR4vhL/model
4 chains, each with 1000 iterations (first 500 discarded)
n.sims = 2000 iterations saved
         mu.vect sd.vect   2.5%    25%    50%    75%  97.5%  Rhat n.eff
beta[1]    2.036   0.165  1.703  1.929  2.035  2.149  2.353 1.001  2000
beta[2]    1.900   0.381  1.150  1.643  1.901  2.148  2.632 1.001  2000
beta[3]   -0.974   0.189 -1.335 -1.098 -0.974 -0.848 -0.597 1.001  2000
tau        5.172   0.754  3.811  4.653  5.123  5.654  6.716 1.002  1100
deviance  52.136   3.090 47.961 49.844 51.571 53.738 59.640 1.001  2000

For each parameter, n.eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor (at convergence, Rhat=1).

DIC info (using the rule, pD = var(deviance)/2)
pD = 4.8 and DIC = 56.9
DIC is an estimate of expected predictive error (lower deviance is better).
```

Wie wir sehen, ist durch die Verdoppelung der Datenbasis die Fehler auf die Schätzer (Spalte `sd.vect`) kleiner geworden. Entspricht dies dem Fehler, den wir erhalten hätten, wenn wir den gesamten Datensatz analysiert hätten? Das sollte ja eigentlich der Fall sein, wenn Bayessches updating funktioniert.

Schauen wir einmal nach:

```
> # Im Vergleich dazu alle Daten zusammen:
> jagsDaten <- list(x=x, y=y, N=100)
> model3 <- function(){
        +   # likelihood function
        +   for (i in 1:N){
                +     y[i] ~ dnorm(muHat[i], tau)
                +     muHat[i] <- beta[1] + beta[2] * x[i] + beta[3] * pow(x[i], 2)
                +   }
        +   # INFORMATIVE priors !!
        +   for (j in 1:3){
                +     beta[j] ~ dnorm(0, 0.01)
                +   }
        +   tau ~ dnorm(0.01, 0.01)
        + }
> Parameter <- c("beta", "tau")
> fjags3 <- jags(data = jagsDaten, inits = NULL, parameters.to.save = Parameter,
+   model.file = model3, n.chains = n.chains, n.iter = n.iter, n.thin = n.thin)
[...]

> fjags3

Inference for Bugs model at "/var/folders/cc/3jfhfx190rb2ptxnqrqxj94m0000gp/T//RtmpcR4vhL/modelab4d541d
4 chains, each with 1000 iterations (first 500 discarded)
n.sims = 2000 iterations saved
mu.vect sd.vect    2.5%     25%     50%     75%   97.5%  Rhat n.eff
beta[1]    2.042   0.133   1.779   1.957   2.045   2.130   2.302 1.002  1400
beta[2]    2.061   0.312   1.431   1.855   2.064   2.269   2.674 1.002  1300
beta[3]   -1.036   0.151  -1.332  -1.139  -1.038  -0.934  -0.735 1.002  1100
tau        4.952   0.705   3.647   4.491   4.925   5.412   6.452 1.001  2000
deviance 126.695   2.974 123.078 124.486 126.045 128.139 134.729 1.002  1500

For each parameter, n.eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor (at convergence, Rhat=1).

DIC info (using the rule, pD = var(deviance)/2)
pD = 4.4 and DIC = 131.1
DIC is an estimate of expected predictive error (lower deviance is better).
```

Nein! Das Modell 3 mit allen Daten hat sichtlich geringere Standardfehler als das aktualisierte Modell 2! Was stimmt hier nicht?

Nun, der Grund, weshalb unser Bayessches updating nicht funktioniert hat, ist, dass wir es uns mit dem prior zu einfach gemacht haben. Tatsächlich sind die Parameterschätzer für beta stark miteinander korreliert (u.a. weil wir die Prädiktoren nicht standardisiert hatten). Wir haben aber unabhängige, univariate prior übergeben.

```
> cor(fjags$BUGSoutput$sims.matrix[, c(1:3, 5)])

             beta[1]      beta[2]       beta[3]           tau
beta[1]  1.000000000 -0.890820380  0.7966635120 -0.0053647351
beta[2] -0.890820380  1.000000000 -0.9752879368  0.0036952257
beta[3]  0.796663512 -0.975287937  1.0000000000  0.0001640259
tau     -0.005364735  0.003695226  0.0001640259  1.0000000000
```

Richtiger wäre es, die Kovarianz der Schätzer zu berücksichtigen, und entsprechend den prior für die beta-Schätzer *multivariat normal* zu gestalten! Dazu machen wir zwei Schritte:

Zunächst spezifizieren wir die Mittelwerte und Kovarianz der betas, dann verändern wir den JAGS-Code so, dass der prior für beta multivariate normal ist.

```
> # Definition der multivariaten Normalverteilung als Prior für Model 2 aus Model 1
# (nur für die 3 beta-Koeffizienten, da tau unkorreliert ist):
> Sigma <- cov(cor(fjags$BUGSoutput$sims.matrix[, c(1:3)]))
> mus <- colMeans(fjags$BUGSoutput$sims.matrix[, c(1:3)])
```

Es folgt Schritt 2, die Umformulierung der prior im JAGS-Code:

```
> ## Jetzt 2. Hälfte des Datensatzes, mit multivariaten Prior aus 1. Hälfte als prior:
> jagsDaten <- list(x=x[index2], y=y[index2], N=50, betaMean=mus, betaSigma=Sigma,
+   tauMean=fjags$BUGSoutput$mean$tau, tauSD=fjags$BUGSoutput$sd$tau)
> model2mv <- function(){
        +   # likelihood function
        +   for (i in 1:N){
                +     y[i] ~ dnorm(muHat[i], tau)
                +     muHat[i] <- beta[1] + beta[2] * x[i] + beta[3] * pow(x[i], 2)
                +   }
        +   # INFORMATIVE priors !!
        +   beta ~ dmnorm(betaMean, betaSigma)
        +   tau ~ dnorm(tauMean, tauSD)
        + }
> Parameter <- c("beta", "tau")
> fjags2mv <- jags(data = jagsDaten, inits = NULL, parameters.to.save = Parameter,
+   model.file = model2mv, n.chains = n.chains, n.iter = n.iter, n.thin = n.thin)
[...]
> fjags2mv

Inference for Bugs model at "/var/folders/cc/3jfhfx190rb2ptxnqrqxj94m0000gp/T//RtmpcR4vhL/model
4 chains, each with 1000 iterations (first 500 discarded)
n.sims = 2000 iterations saved
mu.vect sd.vect   2.5%    25%    50%    75%  97.5%  Rhat n.eff
beta[1]    1.990   0.143  1.713  1.889  1.987  2.086  2.278 1.002  1400
beta[2]    2.027   0.299  1.451  1.826  2.028  2.229  2.599 1.000  2000
beta[3]   -1.032   0.151 -1.329 -1.131 -1.032 -0.926 -0.737 1.000  2000
tau        5.155   0.755  3.733  4.628  5.144  5.651  6.713 1.003  1200
deviance  52.077   2.918 48.044 49.925 51.545 53.693 59.053 1.000  2000

For each parameter, n.eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor (at convergence, Rhat=1).

DIC info (using the rule, pD = var(deviance)/2)
pD = 4.3 and DIC = 56.3
DIC is an estimate of expected predictive error (lower deviance is better).
```

Ah! Diese Standardfehler sind in der Tat sehr vergleichbar mit denen des Gesamtdatensatzes. Tatsächlich überschätzen wir also den Fehler, wenn wir die Korrelation der Schätzer nicht berücksichtigen.

Glücklicherweise sind maximum likelihood-Schätzer asymptotisch normalverteilt, so dass wir zumeist die multivariate Normalverteilung nutzen können, um posterior als neue prior zu übergeben. Allerdings sollten wir immer die posterior plotten (marginal), um zu sehen, ob sie nicht multimodal sind.

### Beschränkte Verteilungen

In JAGS kann man sehr einfach existierende Verteilungen beschränken, etwa eine Normalverteilung auf $\mathbb{R}_0$ oder ein Poisson-Verteilung auf $\mathbb{N}$ (also ohne 0). Dafür muss man nur der Verteilung die Beschränkung nachstellen. So ist eine 0-positiv-beschränkte Normalverteilung z.B. durch `y[i] ~ dnorm(mu[i], sd) T(0,)` darstellbar. `T(,)` erlaubt die Angabe einer unteren (vor dem Komma) und/oder einer oberen Schranke (nach dem Komma). Diese Schranke kann auch von einer Variablen abhängen, wenn z.B. der Fehler immer positiv sein muss: `y[i] ~ dnorm(mu[i], sd) T(y[i],)`, oder ähnlich.[7]

Der JAGS-Code für das Schnäpperbeispiel mit einer positiv beschränkten („zero-truncated“) Poisson-Verteilung wäre etwa:

```
> schnaepperModel <- function(){
+   # likelihood function
+   for (i in 1:N){
        +     Y[i] ~ dpois(lambdaHat[i]) T(1,)
        +     log(lambdaHat[i]) <- beta0 + beta1 * X[i]
        +   }
+   # priors
+   beta0 ~ dnorm(0, 0.01)
+   beta1 ~ dnorm(0, 0.01)
+ }
```

(Dies dient nur zur Illustration. Natürlich könnte ein Schnäpper auch kein Futterstückchen im Betrachtungszeitrum anbringen, deshalb wäre hier eine untere Schranke falsch.)

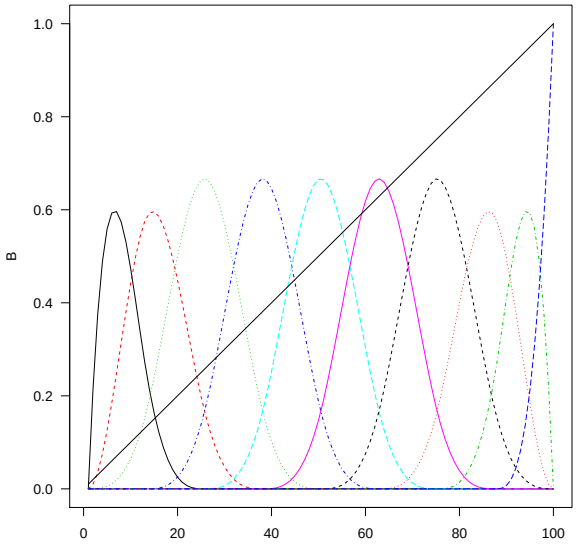
### Splines

Splines sind flexible Kurven, die man anstelle etwa einer Geraden durch seine Daten legen möchte. Die Theorie der splines ist z.B. in Wood (2017) ausgeführt, und in sogenannten Additiven Modellen (GAMs) umgesetzt. Wer also weiß, was GAMs sind, und den Gedanken von splines auch in JAGS umsetzen möchte, dem sei hier kurz der entsprechende Code zur Verfügung gestellt.

Splines sind im Prinzip mehrere nicht-lineare Prädiktoren, die den eigentlichen Prädiktor x ersetzen. Die Form der splines kann man sich aussuchen, typischerweise werden quadratische oder kubische Funktionen gewählt. Aus der Variablen x werden dann mehrere solche kubischen Funktionen gemacht, und diese werden dann im Modell als Prädiktor verwendet. Das sieht dann z.B. so aus:

```
> library(splines)
> x <- 1:100
> J <- 10
> B <- bs(x, J)
> par(mar=c(4,4,1,1))
> matplot(B, type="l", las=1)
> lines(1:100, (1:100)/100)
```

[7] Ich meine mich zu erinnern, dass es ein Problem mit `T(,)` gab, und man vor das `T` noch ein "';'ßetzen musste. Vielleicht war das in R2jags, oder doch in STAN?




Aus der Geraden wurden jetzt also 10 splines (von denen nur der positive Teil benutzt wird). Deren Werte stecken in der Matrix `B`, die wir an JAGS übergeben und dort als Prädiktormatrix nutzen können:

[to be continued ...]

## B.3 Spezielle Bayes-Pakete in R

Die Bayessche Statistik wächst und mit ihr das Angebot an R-Paketen zum Thema. Eine nicht immer aktuelle Übersicht bietet die *CRAN-task view* **Bayesian**.[8]

Von den vielen speziellen Anwendungen (räumliche Modelle, genetische Analysen, Finanzstatistik, ...) und allgemeinen MCMC-artigen Verfahren abgesehen, schauen wir uns vier flexible Pakete an, die es uns ermöglichen mit fast normalem R-Code Bayessche GLMs zu fitten.

### B.3.1 arm: Data Analysis Using Regression and Multilevel/Hierarchical Models

Dieses Paket wird von den Autoren zur Begleitung ihres Buchs *Data Analysis Using Regression and Multilevel/Hierarchical Models* bereitgestellt (Gelman & Hill 2007), und ist inzwischen um einige Funktionen erweitert worden. Die wichtigste Funktion für uns ist `bayesglm`, die wir identisch zu `glm` benutzen können. Ohne weitere Angaben nimmt `bayesglm` sehr flache *prior* an (genauer gesagt t-Verteilungen), die wir aber mit dem Argument `prior.df=Inf` auf eine Normalverteilung ändern werden (sonst hätten wir ja gar nichts gemacht).

```
> library(arm)
> fbglm <- bayesglm(stuecke ~ attrakt, data=schnaepper, family=poisson, prior.df=Inf)
> summary(fbglm) # praktisch identisch zu GLM
```

Die Ausgabe ist bewusst eng an die von `summary(glm(.))` angelehnt:

```
Call:
bayesglm(formula = stuecke ~ attrakt, family = poisson, data = schnaepper,
    prior.df = Inf)

Deviance Residuals:
```

[8] https://cran.r-project.org/web/views/Bayesian.html

```
     Min        1Q    Median        3Q       Max
-1.55352  -0.72783   0.03768   0.59181   1.54628

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  1.47449    0.19440   7.585 3.33e-14 ***
attrakt      0.14790    0.05436   2.721  0.00651 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 25.829  on 24  degrees of freedom
Residual deviance: 18.320  on 23  degrees of freedom
AIC: 115.42
```

Wenn wir also nur einen bestimmten *prior* benutzen wollen, um Vorinformationen einzupflegen, dann ist **arm** unser Freund. Auch gemischte Modelle kann arm sehr gut bedienen. Alle weiteren Funktionen in diesem Paket sind für uns hier irrelevant.

### B.3.2 MCMCglmm: Multivariate Generalised Linear Mixed Models

**MCMCglmm** ist entwickelt worden, um die gelegentlich sehr fragilen gemischten Modelle robuster zu fitten, und um mehr Flexibilität für gemischte Modelle zuzulassen. Wir benutzen von der Hauptfunktion `MCMCglmm` hier nur den Teil, der äquivalent zu `glm` ist. Zur Illustration verändern wir die Grundeinstellung für die *prior* ein wenig, so dass sie denen des JAGS-Modells gleichen (Mittelwert 0, Standardabweichung 10). Da die *prior* multivariat eingegeben werden müssen, übergeben wir die Standardabweichung als eine Matrix mit nur diagonalen Elementen (`diag(10, 2)`). Außerdem stellen wir den MCMC-*sampler* ähnlich ein wie in JAGS.

```
> library(MCMCglmm)
> fMCMCglm <- MCMCglmm(stuecke ~ attrakt, data=schnaepper, family="poisson", thin=5,
+ nitt=20000, burnin=10000, verbose=F, prior=list(B=list(mu=c(0,0), V=diag(10, 2))))
> summary(fMCMCglm)

 Iterations = 10001:19996
 Thinning interval  = 5
 Sample size  = 2000

 DIC: 114.3513

 R-structure:  ~units

      post.mean l-95% CI u-95% CI eff.samp
units 0.0003786 4.19e-08  0.00212     13.9

 Location effects: stuecke ~ attrakt

            post.mean l-95% CI u-95% CI eff.samp  pMCMC
(Intercept)   1.41573  1.33370  1.55245    3.770 <5e-04 ***
attrakt       0.13142  0.09011  0.17277    3.607 <5e-04 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Dieses Paket ist wirklich hervorragend implementiert, dokumentiert und mit Beispielen versehen. Bei großen Datenmengen wird es sehr langsam, genau wie JAGS und praktisch alle übrigen Bayesschen Verfahren, aber dafür ist es eine sehr gute Alternative für die Funktionen für (frequentistische) gemischte Modelle in **nlme** und **lme4**.

Im Gegensatz zu arm bietet **MCMCglmm** auch noch eine `plot`-Funktion für das gefittet Objekt an, die sowohl die *traceplots* als auch die Dichteplots der *posterior* liefert (`plot(fMCMCglm)`).

### B.3.3 INLA: Integrated Nested Laplace Approximation

Sowohl JAGS wie auch MCMCglmm bedienen sich der klassischen MCMC-Strategie, um die Modellparameter zu schätzen. **arm**s `Bayesglm` benutzt einen Trick, der nur für GLMMs zuverlässig funktioniert, und den wir hier nicht kennenlernen werden.[9] Demgegenüber gibt es noch einen ganz anderen Ansatz die *posterior* zu schätzen, ohne aufwändige Irrfahrt im Parameterraum: die Laplace-Näherung. Das Verfahren ist mathematisch sehr schwierig (tatsächlich sind es partielle Differentialgleichungen, die gelöst werden, und die die *posterior*-Dichte annähern) und hört auf den schönen Namen INLA. INLA ist entwickelt worden, um räumlich abhängige Daten zu analysieren, was natürlich auch mit MCMC geht, aber enorm zeitraubend ist, weil der Raumbezug zu einer Unzahl zu schätzender Parameter führt. INLA ist für diesen Zweck wirklich Größenordnungen schneller.

Mit dieser Bayesschen Kanone schießen wir jetzt auf unseren einfachen Schnäpperdatensatz. Wie auch bei den vorhergehenden Paketen ist der Syntax für dieses Problem sehr einfach. Unsere *prior* werden mit dem Argument `control.fixed` definiert, allerdings für Achsenabschnitt und sonstige Parameter separat. Wie bei JAGS erfolgt die Spezifizierung über die Präzision, nicht über die Standardabweichung! Die vorangehende Installation dauert ein wenig, weil INLA sehr groß ist.

```
> install.packages("INLA", repos="https://www.math.ntnu.no/inla/R/stable")
> library(INLA)
> finla <- inla(stuecke ~ attrakt, data=schnaepper, family="poisson",
+ control.family=list(link="log"), control.fixed=list(mean=0, prec.intercept=0.01))
> summary(finla)

Call:
c("inla(formula = stuecke ~ attrakt, family = \"poisson\", data = schnaepper, ",
 "    control.family = list(link = \"log\"), control.fixed = list(mean = 0, ",
 "        prec.intercept = 0.01))")

Time used:
 Pre-processing    Running inla Post-processing           Total
         0.5867          0.1337          0.0567          0.7770

Fixed effects:
              mean     sd 0.025quant 0.5quant 0.975quant   mode kld
(Intercept) 1.4757 0.1944     1.0838   1.4793     1.8480 1.4864   0
attrakt     0.1481 0.0544     0.0419   0.1479     0.2553 0.1475   0

The model has no random effects

The model has no hyperparameters

Expected number of effective parameters(std dev): 2.001(0.00)
```

[9] *expectation maximisation* mit *data augmentation*, um die *prior* darzustellen.

```
Number of equivalent replicates : 12.50

Marginal log-Likelihood:  -66.97
```

Die Ausgabe ist eher im JAGS-Stil gehalten, mit einer Zusammenfassung der Schätzer.

Diese paar Zeilen sollen nur dazu dienen, einfache und für weitere Analysen möglicherweise sehr nützliche Bayessche Ansätze anzureißen. Selbst eine oberflächliche Beschreibung von INLA würde den Rahmen sprengen. Dafür sei auf Handbuch und viele Code-Beispiele auf der Webseite von INLA verwiesen.[10]

### B.3.4 BayesianTools: ein flexibler Ansatz zur Berechnung Bayesscher Modelle

**BayesianTools** verfolgt den Ansatz, für alle möglichen Arten von Modellen ein konsistentes *sampling*-System zur Verfügung zu stellen. Damit sind dann lineare Regressionen möglich, aber auch das Anpassen komplexer physikalischer Prozessmodelle. Der Nachteil dieser Flexibilität ist, dass wir uns in die Methodik der Schöpfer dieses Pakets eindenken müssen. Als Einstieg ist auf jeden Fall empfohlen, die Vignette des Pakets zu lesen und durchzuarbeiten!

```
> library(BayesianTools)
> vignette("BayesianTools")
```

Für die obige Poisson-Regression, müssen wir mehrere vorbereitende Schritte durchführen. Konkret müssen wir die *likelihood* als Funktion definieren, dann die *prior* (sowohl die Dichtefunktion als auch eine Funktion, die aus dieser Dichtefunktion zufällig zieht: den *sampler*), daraus ein *setup*-Objekt produzieren und schließlich noch die Einstellung für den MCMC vornehmen. Dann erst können wir den eigentlichen MCMC-sampler starten. Im folgenden gehen wir diese Schritte einzeln durch.

#### 1. Definition der log-*likelihood*

Inzwischen kennen wir die *likelihood*-Funktion für die Poisson-Regression ja: $Y \sim \mathrm{Pois}(\lambda = e^{\beta_0+\beta_1 X})$. Diese schreiben wir als R-Funktion der Parameter $\beta_0$ und $\beta_1$ (übergeben als `parms`):

```
> poisResLogLik <- function(parms){ # die logLik-Funktion
+  beta0 <- parms[1]
+  beta1 <- parms[2]
+  yhat <- exp(beta0 + beta1*attrakt) # log-link für Poisson-Regression
+  LL <- sum(dpois(stuecke, yhat, log=TRUE))
+  if (!is.finite(LL)) LL <- -1e5
+  return(LL)
+ }
```

#### 2. Definition der *prior*-Funktion

Gegenwärtig ist **BayesianTools** etwas limitiert, welche *prior*-Verteilungen möglich sind. Zwar kann eine beliebige Verteilung definiert werden, aber nur für alle Parameter die gleiche! Wir brauchen *prior* für unsere beiden Parameter $\beta_0$ und $\beta_1$, und nehmen wie bisher an, dass sie normalverteilt sind mit Mittelwert=0 und Standardabweichung=10.

Neben den Dichteverteilungen (in Funktion `densFun`) müssen wir auch eine sogenannte sampler-Funktion (`sampleFun`) definieren, die aus den *prior* zieht. Gemeinsam werden sie durch die Funktion `createPrior` in `priors` eingebaut.

[10] `http://www.r-inla.org/`

```
> densFun <- function(parms){  # density
+  sum(dnorm(parms, mean=0, sd=10, log=T)) # evaluiert beide Parameter!
+ }
> sampleFun <- function(n=2){ # sampler
+  rnorm(n, mean=0, sd=10) # Ziehung für beide Parameter!
+ }
> priors <- createPrior(density=densFun, sampler=sampleFun)
```

Wichtig ist hier, dass in der *sampler*-Funktion so viele Parameter gezogen werden, wie im Modell sind, in unserem Fall also 2.

### 3. Setup-Objekt produzieren

BayesianTools benötigt eine standardisierte Eingabe der *likelihood*- und *prior*-Funktionen, die wir gerade definiert haben.

```
> mySetup <- createBayesianSetup(likelihood=poisResLogLik, prior=priors)
```

Die Funktionen sind hierin jetzt abrufbar. Mit

```
> mySetup$likelihood$density(c(1, 0.2))
```

erhalten wir die log-*likelihood* für die Parameterkombination $\beta_0 = 1$ und $\beta_1 = 0.2$:

```
[1] -63.2895.
```

Analog liefert uns

```
> mySetup$prior$density(c(1, 0.2))
```

die Summe der log-Wahrscheinlichkeitsdichten für die beiden Parameter:

```
[1] -6.448247.
```

Das heißt, in `mySetup` liegen die verschiedenen Beiträge zur Bayes-Formel als Funktionen abrufbar bereit für den MCMC-Sampler.

### 4. MCMC-Sampler auswählen und einstellen

Schießlich müssen wir noch die Einstellungen des MCMC-Samplers vornehmen, genau wir auch in JAGS (also wie viele Ketten, wie lange jede Kette, ob ausgedünnt werden soll, wie lange burnin usw.). Wir vertrauen einfach einmal den Standardeinstellungen und wählen für den Sampler (Metropolis) die Anzahl und Länge der Ketten.[11]

```
> settings = list(iterations = 10000, nrChains=3, adaptationNotBefore = 1000)
```

(Die letzte Option hat eine Grundeinstellung von 3000, wodurch viele unserer *samples* verloren gingen.)

### 5. MCMC-Sampling

Jetzt haben wir alles zusammen, und starten das Absammeln der *posterior*-Verteilung. BayesianTools (das sich selbst „BT“ nennt), informiert uns über den Fortschritt.

```
> outM <- runMCMC(bayesianSetup = mySetup, sampler="Metropolis", settings = settings)
```

[11] Diese *settings* sind nicht gut dokumentiert, und so muss man im R-Code von `Metropolis` suchen, was man wie nennt.

```
BT runMCMC: trying to find optimal start and covariance values
BT runMCMC: Optimization finished, setting startValues to 1.46545231561603 0.152607894355206
- Setting covariance to 0.0377840373078403 -0.00971328289526969 -0.00971328289526969 0.0029416459350485
 Running Metropolis-MCMC, chain  1 iteration 10000 of 10000 . Current logp:  -62.29117  Please wait!
runMCMC terminated after 1.98499999999876seconds
BT runMCMC: trying to find optimal start and covariance values
BT runMCMC: Optimization finished, setting startValues to 6.76898432414706 -1.90367571650678
- Setting covariance to 0.0101900167922232 -0.0075573109559765 -0.0075573109559765 0.00643321768644396
 Running Metropolis-MCMC, chain  2 iteration 10000 of 10000 . Current logp:  -63.96598  Please wait!
runMCMC terminated after 1.98999999999978seconds
BT runMCMC: trying to find optimal start and covariance values
BT runMCMC: Optimization finished, setting startValues to 1.47363689891921 0.148277620854312
- Setting covariance to 0.0377973870396912 -0.00973154173709437 -0.00973154173709437 0.0029547028549118
 Running Metropolis-MCMC, chain  3 iteration 10000 of 10000 . Current logp:  -62.77431  Please wait!
runMCMC terminated after 2.20300000000134seconds
```

Anstelle von „Metropolis“ gibt es eine Reihe anderer Sampler (siehe `?runMCMC`), von denen wir unten einen weiteren beispielhaft anschauen wollen. Außerdem gibt es Optionen zur Parallelisierung und, sehr wichtig, zur Analyse und Visualisierung der Ergebnisse. Schauen wir uns diese also an.

### 6. Ergebnisse und Modelldiagnostik

Zunächst wollen wir einfach einmal die Schätzer sehen. Dafür schauen wir uns die Zusammenfassung des gerade erschaffenen Objekts `outM` an.

```
> summary(outM)

# # # # # # # # # # # # # # # # # # # # # # # # #
## MCMC chain summary ##
# # # # # # # # # # # # # # # # # # # # # # # # #

# MCMC sampler:  Metropolis
# Nr. Chains:  3
# Iterations per chain:  10000
# Rejection rate:  0.649
# Effective sample size:  2771
# Runtime:  6.178  sec.

# Parameter       MAP      2.5%    median   97.5%
#  par 1 :       1.474    1.068    1.470    1.854
#  par 2 :       0.148    0.040    0.149    0.257

## DIC:  119.588
## Convergence
 Gelman Rubin multivariate psrf:  1.001

## Correlations
       par 1  par 2
par 1  1.000 -0.956
par 2 -0.956  1.000
```

Diese Ausgabe zeigt uns, dass wir 3 Ketten laufen ließen, für jeweils 10000 Schritte, von denen nur ein Anteil von $1 - 0.649 = 0.351$ angenommen wurde. Da unser Ziel mindestens 20% ist, sind wir hiermit zufrieden. Aus den Grundeinstellungen zur Ausdünnung und *adaptation*-Phase ergeben sich somit 2771 *samples*.

Die beiden Parameter werden mit Werten sehr ähnlich zu allen anderen bisherigen Verfahren geschätzt. Neben dem Median und den 95% Glaubwürdigkeitsintervall erhalten wir den „MAP“, den *maximal a-posterior probability*-Wert. Dieser ist praktisch der beste Schätzer für unsere Parameter, wenn man die Korrelation zwischen den Parametern berücksichtigt (siehe auch 2).

Den DIC haben wir schon bei JAGS kennengelernt. Je nach sampler ist dieser mehr oder weniger glaubwürdig, hier sieht er plausibel aus (vergleiche mit den AIC in den anderen Verfahren).

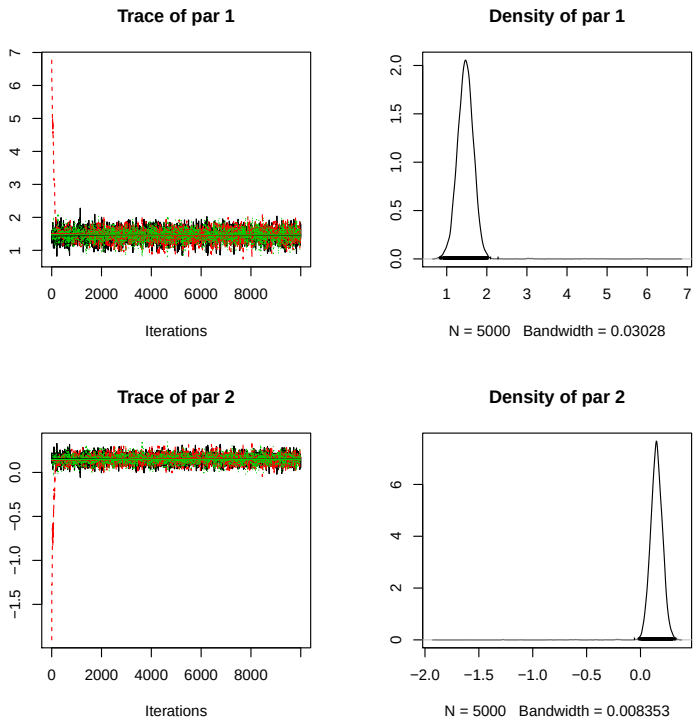
Die Gelman-Rubin-Diagnostik kennen wir bereits als $\hat{R}$. Hier wird ihr multivariates Gegenstück angegeben, also die Konvergenz über alle Parameter, nicht pro Parameter. Mit einem Wert von 1.001 können wir sehr zufrieden sein (sonst müssten wir die Länge der Ketten erhöhen).

Schließlich gibt uns Rdie Korrelation in den Schätzern des MCMC für die beiden Parameter an. Diese ist sehr hoch und negativ, was wir auch für eine lineare Regression erwarten würden.[12]

Im Weiteren schauen wir uns drei einfach Diagnotiken an:

```
> plot(outM)
```

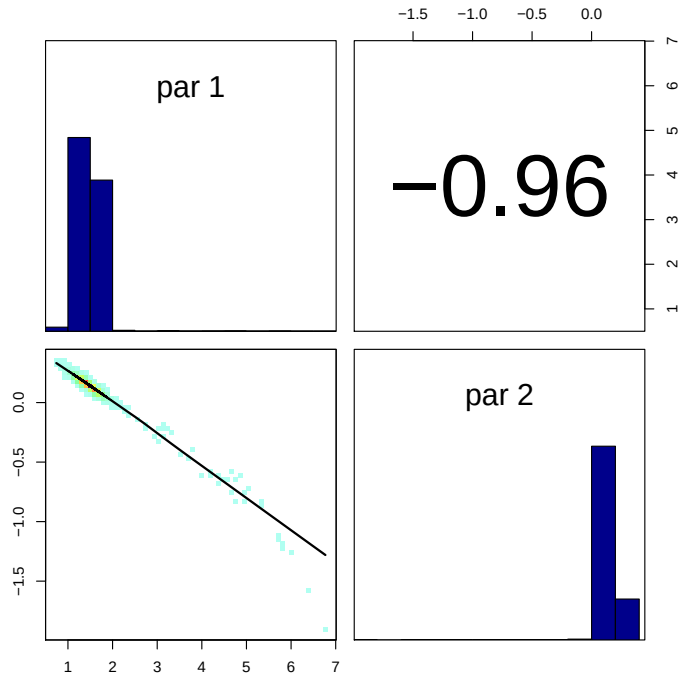
liefert den *trace*- und den *density-plot* für jeden Parameter:



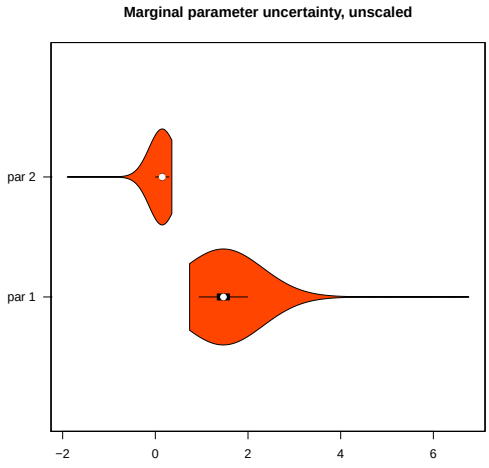

```
> correlationPlot(outM)
```

bildet die Korrelationen der Parameter ab (in diesem Fall wenig informativ, aber eine bananenförmige Korrelation würde sich hier abzeichnen):

[12] Ist das klar? Wenn wir z.B. die Linie durch die Punkte steilen machen würden, dann sollten wir, um den Fit zu retten, den $y$-Achsenabschnitt negativer machen. Genau diese Korrelation ist hier quantifiziert.




```
> marginalPlot(outM)
```




fasst die *posterior*-Verteilung als Violinplot zusammen, in dem der weiße Punkt den MAP darstellt (während das schwarze Quadrat den Median anzeigt).

Schließlich wenden wir einfach noch einen anderen Sampler an, mit Namen *differential evolution* (DE):

```
> settings = list(iterations = 5000, nrChains=3)
> outDE <- runMCMC(bayesianSetup = mySetup, sampler="DE", settings = settings)

 Running DE-MCMC, chain  1 iteration 5004 of 5004 . Current logp  -63.06225 -62.53873
 -62.66202 -63.44435 -65.13555 -62.26143 Please wait!
runMCMC terminated after 0.5seconds
 Running DE-MCMC, chain  2 iteration 5004 of 5004 . Current logp  -62.3407 -62.25045
 -62.25352 -62.56181 -64.37665 -62.68649 Please wait!
runMCMC terminated after 0.496999999999389seconds
 Running DE-MCMC, chain  3 iteration 5004 of 5004 . Current logp  -62.38244 -63.85518
```

```
 -62.89051 -62.44994 -62.26141 -63.03633 Please wait!
runMCMC terminated after 0.5seconds
```

Was genau an DE anders ist als beim Metropolis würde hier zu weit führen. Dies ist in der Vignette von **BayesianTools** sehr schön dargestellt. Nur um zu erklären, weshalb nicht wie bisher eine sondern 6 log-*likelihoods* angegeben werden: Bei DE laufen gekoppelte Ketten, die sich über die *proposal*-Funktion gegenseitig beeinflussen; jede von ihnen hat ihre eigene log-*likelihood*.

```
> summary(outDE)

# # # # # # # # # # # # # # # # # # # # # # # # # # #
## MCMC chain summary ##
# # # # # # # # # # # # # # # # # # # # # # # # # # #

# MCMC sampler:  DE
# Nr. Chains:  18
# Iterations per chain:  834
# Rejection rate:  0.656
# Effective sample size:  5654
# Runtime:  1.497  sec.

# Parameter       MAP      2.5%    median   97.5%
#  par 1 :       1.470   -7.365    1.458    2.258
#  par 2 :       0.149   -0.438    0.150    1.410

## DIC:  4.573804e+51
## Convergence
 Gelman Rubin multivariate psrf:  1.024

## Correlations
       par 1  par 2
par 1  1.000 -0.744
par 2 -0.744  1.000
```

Die Ausgabe ist ähnlich zum Metropolis, mit Ausnahme des DIC: diesem können wir beim Algorithmus DE nicht vertrauen![13]

Alle obigen Abbildungen können für jeden der Sampler verwendet werden.

### B.3.5 Googles TensorFlow mittels greta

Die plattformunabhängige und quelloffene Software TensorFlow[14] wurde von Google entwickelt, um große Datenmengen mit Verfahren der „künstlichen Intelligenz“ zu analysieren. TensorFlow ist darauf ausgelegt, alle verfügbaren Prozessoren einzubinden, automatisch zu parallelisieren, und die ganzen rechentechnischen Tricks anzuwenden, die nötig sind, um auch riesige Datenmengen bewältigen zu können, ohne sich als Nutzer um Details zu kümmern. Das R-Paket **tensorflow** bindet diese Optionen in Rein.

Mit **greta** hat Nick Golding eine weitere Vereinfachung geschaffen, die es erlaubt, auch Bayessche Modelle mittels TensorFlow zu rechnen. Der Syntax ist pures R, allerdings in etwas ungewöhnlicher Interpretation. Als *sampler* wird Hamilton Monte Carlo benutzt. Unser Schnäpper-Beispiel sieht so aus in **greta**:

[13] `DIC(outDE)$Dhat` ist da besser.

[14] `https://www.tensorflow.org/`

```
> #install.packages(c("greta", "bayesplot"))
> # on commandline, type: sudo /usr/local/bin/pip install --upgrade virtualenv
> # install_tensorflow()
> library(greta)
> X = schnaepper$attrakt
> Y = schnaepper$stuecke
>
> # define priors: (BEFORE THE MODEL!!)
> beta0 = normal(0, 10)
> beta1 = normal(0, 10)
>
> # define model:
> mean <- exp(beta0 + beta1*X)
> distribution(Y) = poisson(mean)
>
> fgreta <- model(beta0, beta1)
> draws <- mcmc(fgreta, n_samples = 1000)

warmup ========================================   100/100 | eta:  0s | 7% bad
sampling ======================================== 1000/1000 | eta:  0s

> summary(draws)

Iterations = 1:1000
Thinning interval = 1
Number of chains = 1
Sample size per chain = 1000

1. Empirical mean and standard deviation for each variable,
plus standard error of the mean:

Mean      SD Naive SE Time-series SE
beta0 1.4586 0.20117 0.006361       0.007182
beta1 0.1507 0.05625 0.001779       0.002191

2. Quantiles for each variable:

2.5%   25%   50%   75%  97.5%
beta0 1.04921 1.333 1.459 1.595 1.8363
beta1 0.03762 0.114 0.149 0.184 0.2662
```

Wie bei den früheren Methoden gibt es auch hier eine MCMC-output, den man mit verschiedenen Paketen und Befehlen weiterverarbeiten kann.[15]

**greta** für so ein einfaches Beispiel zu nutzen ist sicherlich übertrieben. Aber die automatische Auslagerung auf mehrere Prozessoren und die einfache Syntax machen die Nutzung einfach. **greta** steckt noch in den Kinderschuhen, und mit der Zeit werden vielleicht noch mehr *sampler* implementiert, wie in **BayesianTools**.

### B.3.6 STAN und brms

JAGS ist eine prima Implementierung der BUGS-Sprache, mit der wir sehr flexibel ein enormes Repertoire an statistischen Modellen fitten können.[16] Ein Nachteil von JAGS ist seine

[15] Etwa mit `bayesplot::mcmc_dens(draws)`

[16] Tatsächlich kann man sogar eigene Funktionen in JAVA schreiben (z.B. ein ganzes Vegetationsmodell), und als Funktion mit in JAGS hineinkompilieren. Damit kann man dann sogar komplizierte Prozessmodelle direkt in

Langsamkeit: wenn wir Zehn- oder Hunderttausende Parameterkombinationen durchrechnen dauert das einfach. Aus diesem Problem entstand eine andere Lösung namens STAN.[17] Darin wird versucht das Problem zu lösen, dass so ein MCMC sehr langsam das Maximum erarbeiten muss. Der Ansatz zur Lösung ist, dass ich an jeder Stelle (d.h. für jeden beliebigen Parametersatz) die lokalen Ableitungen berechnen kann, und dadurch weiß, in welcher Richtung und welcher Steigung es am schnellsten zum (lokalen) Maximum geht. Technisch sprechen wir hier von der Berechnung des Gradienten der log-likelihood, was als Hamiltonian Monte Carlo (HMC) bezeichnet wird. Eine weitere Verfeinerung verhindert, dass der Algorithmus einen beschrittenen Gradienten wieder hinabläuft auf der Suche nach dem globalen Maximum; sie wird als „no-U-turn sampler“ (NUTS) bezeichnet. STAN implementiert diesen Ansatz.[18]

STAN ist im Syntax ähnlich zu JAGS, erfordert aber etwas mehr Disziplin und Variablendefinitionen. Glücklicherweise gibt es aber das hochgelobte Paket **brms**, dass uns einen Zugriff auf STAN erlaubt im Stil der Modellspezifikation von **lme4**. Für sehr komplizierte Analysen muss man zwar noch auf STAN direkt zugreifen, aber unser Beispiel können wir einfach mal in **brms** durchspielen. Die Einstellungen sind für uns jetzt relativ leicht interpretierbar, und sonst in der Hilfe und den Vignetten zu **brms** wirklich sehr gut dokumentiert.

```
> fit1 <- brm(stuecke ~ attrakt, data = schnaepper, family = poisson(),
+             prior = c(set_prior("normal(0,5)", class = "b")),
+             warmup = 1000, iter = 2000, chains = 4,
+             control = list(adapt_delta = 0.95))

Compiling the C++ model
Start sampling

SAMPLING FOR MODEL 'poisson brms-model' NOW (CHAIN 1).

Gradient evaluation took 4.4e-05 seconds
1000 transitions using 10 leapfrog steps per transition would take 0.44 seconds.
Adjust your expectations accordingly!

Iteration:    1 / 2000 [  0%]  (Warmup)
Iteration:  200 / 2000 [ 10%]  (Warmup)
Iteration:  400 / 2000 [ 20%]  (Warmup)
... (snip)
Iteration: 1600 / 2000 [ 80%]  (Sampling)
Iteration: 1800 / 2000 [ 90%]  (Sampling)
Iteration: 2000 / 2000 [100%]  (Sampling)

Elapsed Time: 0.102776 seconds (Warm-up)
0.099226 seconds (Sampling)
0.202002 seconds (Total)
```

Wir erhalten Rückmeldung über den *sampling*-Prozess der 4 Ketten, außer wir setzen `refresh=0`. Interessanter ist dann die tatsächliche Modellausgabe und der Plot:

JAGS fitten (Stephen Higgings, Uni Bayreuth, mündl. Mitteilung).

[17] `mc-stan.org`

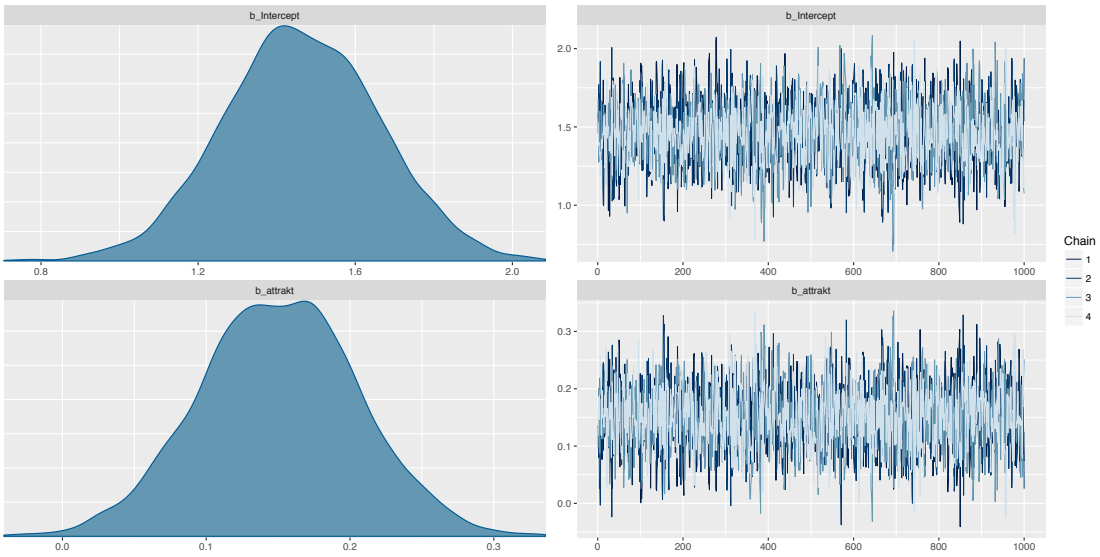
[18] Die Geschichte von STAN wird im Vorwort zu seinem sehr umfangreichen Handbuch beschrieben (`https://github.com/stan-dev/stan/releases/download/v2.17.0/stan-reference-2.17.0.pdf`) und ist recht spannend. Am Anfang war STAN nämlich deutlich langsamer als JAGS/BUGS, was etwas verwunderte, da STAN in C++ übersetzt und kompiliert wird, und somit viel schneller sein sollte.

```
> fit1
> plot(fit1)

Family: poisson
Links: mu = log
Formula: stuecke ~ attrakt
Data: schnaepper (Number of observations: 25)
Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;
total post-warmup samples = 4000
ICs: LOO = NA; WAIC = NA; R2 = NA

Population-Level Effects:
                 Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
Intercept        1.46      0.20     1.08     1.85         2252 1.00
attrakt          0.15      0.06     0.04     0.26         2428 1.00

Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
is a crude measure of effective sample size, and Rhat is the potential
scale reduction factor on split chains (at convergence, Rhat = 1).
```




Wie gehabt schätzt auch STAN-mittels-brms die Koeffizienten korrekt und liefert gleich auch die Konvergenzstatistik mit (`Rhat`). Für Regressionmodelle dieser Art (und weitaus komplizierter) ist **brms** also wirklich eine hübsche Sache.

# Literaturverzeichnis


Clark, J. S. (2007). *Models for Ecological Data: An Introduction*. Princeton, N.J, USA: Princeton Univ. Press. 15

Clark, J. S. & Gelfand, A. E. (2006). *Hierarchical Modelling for the Environmental Sciences: Statistical Methods and Applications*. Oxford: Oxford Univ. Press. 15

Gelman, A., Carlin, J., Stern, H., Dunson, D., Vehtari, A., & Rubin, D. (2013). *Bayesian Data Analysis*. Chapman & Hall/CRC. 15

Gelman, A. & Hill, J. (2007). *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge, UK: Cambridge University Press. 15, 35

Gelman, A. & Rubin, D. B. (1992). Inference from iterative simulation using multiple sequences. *Statistical Science*, 7(4), 457–472. 8

Hastie, T., Tibshirani, R. J., & Friedman, J. H. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Berlin: Springer, 2nd edition. 11

Jaynes, E. T. (2003). *Probability Theory: The Logic of Science*. Cambridge, U.K.: Cambridge University Press. 1, 10, 12, 15

Kéry, M. (2010). *Introduction to WinBUGS for Ecologists: Bayesian Approach to Regression, ANOVA, Mixed Models and Related Analyses*. Salt Lake City, USA: Academic Press. 15

Kruschke, J. K. (2015). *Doing Bayesian Data Analysis. A Tutorial with R, JAGS, and Stan*. Academic Press, 2nd edition. 15

Kéry, M. & Royle, J. A. (2015). *Applied Hierarchical Modeling in Ecology: Analysis of distribution, abundance and species richness in R and BUGS. Volume 1: Prelude and Static Models*. Academic Press. 15

Kéry, M. & Schaub, M. (2011). *Bayesian Population Analysis using WinBUGS: A Hierarchical Perspective*. Academic Press. 15

Link, W. A. & Eaton, M. J. (2012). On thinning of chains in MCMC. *Methods in Ecology and Evolution*, 53, 112–115. 14

O'Hara, R. B. & Sillanpää, M. J. (2009). A review of bayesian variable selection methods: What, how and which. *Bayesian Analysis*, 4(1), 85–118. 11

Raftery, A. E. & Lewis, S. M. (1992). Comment: One long run with diagnostics: Implementation strategies for markov chain monte carlo. *Statistical Science*, 7, 493–497. 8

Spiegelhalter, D. J., Best, N. G., Carlin, B. P., & van der Linde, A. (2002). Bayesian measures of model complexity and fit. *Journal of the Royal Statistical Society B*, 64, 583–639. 21

Tibshirani, R. (1996). A comparison of some error estimates for neural network models. *Neural Computation*, 163, 152–163. 11

Wood, S. N. (2015). *Core Statistics*. Cambridge University Press. 22

Wood, S. N. (2017). *Generalized Additive Models: an Introduction with R*. London: Chapman & Hall/CRC Press, 2nd edition. 34

# Index