Einführung in die Programmierung von Monte Carlo Simulationen

Simulationsstudien sind eine wichtige Methode in der Statistik, um zu überprüfen, unter welchen Bedingungen statistische Verfahren funktionieren und wie robust sie gegenüber Verletzung ihrer Annahmen sind. Das Grundprinzip ist, dass man das Gedankenexperiment der Inferenzstatistik "viele Stichproben ziehen und die Parameter berechnen, um dann deren Verteilung zu betrachten" empirisch (wenn auch am Computer) durchführt. In erster Linie sind Simulationsstudien mit Programmierarbeit verbunden - auch wenn es natürlich (wie bei jedem guten Experiment) darauf ankommt, theoretisch sinnvoll hergeleitete Bedingungen zu vergleichen. Die folgenden Aufgaben sollen Sie schrittweise an die Durchführung einer eigenen Simulationsstudie heranführen.

Mit der Funktion `rnorm(n = 1, mean = 0, sd = 0)` kann man eine Zufallsstichprobe der Größe n aus einer Normalverteilung mit vorgegebenem Mittelwert und vorgegebener SD ziehen. Wir wollen Simulationsstudien mit dem Modell der linearen Regression durchführen, dabei geht man zunächst von einem Populationsmodell aus, in dem man die Regressionkoeffizienten und die Fehlervarianz festlegt. Es gilt:

$$y = X\beta + \epsilon \tag{6}$$

$$y \sim NV(X\beta, \sigma^2) \tag{7}$$

$$\epsilon \sim NV(0, \sigma^2) \tag{8}$$

Das heißt, y ist normalverteilt mit den vorhergesagten Werten als Erwartungswert und der Fehlervarianz σ^2 als Varianz. Die Fehlerterme sind ebenfalls normalverteilt (und unkorreliert mit allen anderen Variablen) mit Erwartunsgwert 0. Das heißt, wir zerlegen y in einen systematischen Varianzanteil, der durch die Prädiktoren erklärt wird und einen nicht-systematischen Fehleranteil.

Hinsichtlich der Verteilung der Prädiktoren steckt im Modell keine Annahme, aber nehmen wir der Einfachheit halber an, dass die Prädiktoren multivariat-normalverteilt sind:

$$X \sim NV(\mu_X, \Sigma_X) \tag{9}$$

Da wir multiple Prädiktoren haben, ist das eine multivariate Verteilung. μ_X ist der Vektor mit den Populationsmittelwerten der X-Variablen und Σ_X ist die Kovarianzmatrix der Prädiktoren, die sowohl deren Varianzen als auch deren Kovarianzen bestimmt. In den Kovarianzen steckt dann die Multikollinearität.

Auf diesen Allgemeinfall werden wir später zurückkommen. Im Folgenden reduzieren wir uns zur Vereinfachung der Programmierarbeit zunächst auf eine einfache lineare Regression, d.h.:

$$y = \beta_0 + \beta_1 \cdot x + \epsilon \tag{10}$$
$$y \sim NV(\beta_0 + \beta_1 \cdot x, 15) \tag{11}$$
$$x \sim NV(100, 15) \tag{12}$$
$$\epsilon \sim NV(0, 15) \tag{13}$$

Hinweis: Fehlende Simulationsparameter dürfen Sie frei wählen Ich würde aber empfehlen, viel aus dem vorigen Abschnitt zu übernehmen, um die Vergleichbarkeit zu erhöhen.

1. Mit der Funktion `rnorm(n = 1, mean = 0, sd = 0)` kann man eine Zufallsstichprobe der Größe n aus einer Normalverteilung mit vorgegebenem Mittelwert und vorgegebener SD ziehen. **Ziehen Sie eine Stichprobe beliebiger (veränderbarer) Stichprobengröße mit Werten für x und ϵ und berechnen Sie die y-Werte mithilfe der Regressionsgleichung.** Halten Sie dabei die Werte β_0 und β_1 flexibel, indem Sie am Anfang des Skripts Variablen einführen. Setzen Sie zunächst $\beta_0 = 50$ und $\beta_1 = 0$ ein.

2. **Berechnen Sie die Parameter der linearen Regression zwischen x und y sowie die Korrelation für die Stichprobe.**

3. Wiederholen Sie die Stichprobenziehung mehrfach und beobachten Sie die extrahierten Parameter. Was fällt auf?

4. Finden Sie heraus, wie man neben der Korrelation und den Regressionskoeffizienten auch die Signifikanztestergebnisse extrahiert. Speichern Sie alle Werte in einem Vektor (mit `c()`).

5. Um effizient zu programmieren, ist es wichtig, den Code zu modularisieren und einzelne Schritte, die oft wiederholt werden, in eigenen Funktionen zusammenzufassen. Zum Beispiel kann man eine Funktion, welche die Summe aus zwei Variablen berechnet, so erzeugen:

```
sum <- function(x,y) {
  sum = x + y # Summe berechnen
  return(sum) # Variable sum wird Output der Funktion
}
```

Hinweis: Funktionen haben einen eigenen Variablenraum, d.h., Variablennamen dürfen innerhalb der Funktion neu vergeben werden, aber die Funktion kennt auch nur die Variablen, die man ihr aktiv gibt. **Schreiben Sie eine Funktion, welche wie in Aufgabe 1 aus vorgegebenen Regressionsparametern eine Stichprobe mit Werten für x und y erzeugt.** (Tipp: Halten Sie möglichst alle benötigten Zahlenwerte flexibel, dann können wir die Funktion lange weiter benutzen.) **Schreiben Sie**

eine zweite Funktion, welche die Regressionskoeffizienten, deren Standardfehler, und die Korrelation aus den gezogenen Daten extrahieren kann und als Vektor zurückgibt.

6. Um eine Simulationsstudie durchzuführen, ist es essentiell, dass man den untersuchten Prozess (z.B. Regression für Zufallsstichproben) auch beliebig oft durchführen kann, ohne den Code explizit so oft hinzuschreiben, dafür benutzt man so genannte `for`-Schleifen. Zum Beispiel kann man die Zahlen von 1 bis 1000 nacheinander in einer Schleife quadrieren:

```
for (i in 1:1000) {
  print(i^2)
}
```

Natürlich möchte man in der Regel die Ergebnisse jedes Durchgangs auch speichern, zum Beispiel so:

```
quadrate = rep(NA, 1000)
for (i in 1:1000) {
  quadrate[i] = i^2
}
```

Jetzt sind Sie wieder dran: **Führen Sie die Stichprobenziehung 10000 mal durch und speichern Sie die Regressionskoeffizienten und deren Standardfehler für jede Stichprobe. Untersuchen Sie Mittelwert, Standardabweichung und Verteilung der Parameter über die Stichproben hinweg.** Welcher Verteilung folgen die Parameter? Wie lauten Mittelwert und Standardabweichung?

7. **Wiederholen Sie Aufgabe 6 für verschiedene Effektstärken** (β_1 = -1, 0.8, ...,0,0.2,0.4,...1). Berechnen Sie jeweils den α-Fehler bzw. die Teststärke. Berechnen Sie außerdem Mittelwert und Standardabweichung der Regressionsgewichte sowie den Mittelwert der Standardfehler der Regressionsgewichte. Was stellen Sie fest? (Tipp: Nutzen Sie eine for-Schleife, um die Simulation für verschiedene Werte β_1 durchzuführen.)

8. Bisher haben wir die for-Schleifen als mächtiges Werkzeug kennengelernt, mit dem man ohne Weiteres vollständige Simulationsstudien durchführen kann. Allerdings wird es mit zunehmender Komplexität und Verschachtelung immer schwieriger, den Überblick zu behalten und die Ergebnisse effizient abzuspeichern. Deswegen hat man die `apply`-Familie erfunden, im Wesentlichen funktioniert das so, dass man den Inhalt der for-Schleife zu einer Funktion zusammenfasst und diese in einen Befehl gibt. Der große Vorteil ist, dass man ohne Indizierung vernünftige Ergebnisobjekte bekommen kann. Für das Beispiel von oben sieht eine apply-Lösung so aus:

```
quadrate = sapply(1:1000, FUN = function(i){i^2},
                  simplify = "array")
```

Man bekommt also dasselbe ohne Indizierung, was die Sache erheblich einfacher und

flexibler macht. **Wiederholen Sie die Aufgaben 6 und 7 und versuchen Sie, ohne for-Schleifen auszukommen.**