Notat 6 - ST1301

22. februar 2005

1 Instruksjoner som data

I begynnelsen av kurset definerte vi data som informasjon uttrykkt i et programmeringsspråk. Slike data kan være av ulik type, f.eks. enkle skalarer eller logiske data, eller sammensatt data som vektorer eller matriser. I de fleste programmeringsspråk skiller det vi definerer som data seg fra de instruksjoner (i form av ulike operatorer eller funksjoner) vi kan uttrykke i programmeringsspråket. Normalt tar en funksjon visse data som inn-argument, funksjonen behandler disse og returnerer så visse utdata. Det samme kan sies om operatorer.

I R er det ikke noe skarpt skille mellom instruksjoner (f.eks. i form av en funksjonsdefinisjon) og data—de uttrykk som en funksjon består av kan også betraktes som en type data i R. Mer generelt sier vi vanligvis at alt (både data, funksjoner, og operatorer) er objekter i R. Vi har allerede sett at vi bruker tilordningsoperatoren `<-` både når vi tilordner verdien av et ordinært uttrykk til en variabel, f.eks. med uttrykket

```
> a <- 5+2
```

og når vi definerer en funskjon, f.eks.

```
> f <- function(x) 5-x-x^2
```

I stedet for å gjøre et kall til funsjonen, f.eks.

```
> f(1)
[1] 3
```

kan gjøre en referanse til objektet `f` ved å skrive

```
> f
function(x) 5-x-x^2
```

Som vi ser vises da selve uttrykket som utgjør funksjonsdefinisjonen. En slik referanse til en funksjonsdefinisjon kan også oppgis som argument i kallet til en annen funksjon.

Hva kan dette brukes til? I mange tilfeller vil vi ha behov for å gjøre ulike bestemte ting med en gitt funksjon. Om vi har en funksjon $f(x)$ kan vi ønske å beregne et integral på formen $\int_a^b f(x)dx$, vi kan ønske å finne $f(x)$ sitt maksimum, eller vi kan ønske å finne hvilke nullpunkter $f(x)$ måtte ha. Dette er beregninger som i det praktiske liv ofte må gjøres ved hjelp av numeriske metoder. I slike tilfelle vil algoritmen vi bruker for å utføre numeriske beregninger ofte være den samme uavhengig av hvilken funksjon vi ønsker å finne f.eks. maksimum til—det som derimot vil variere vil være selve definisjonen av den funksjonen vi ønsker å finne maksimum til. Det vi ønsker er å la funksjonensdefinisjonen (som kan variere) være innargument til en annen funksjon som gjennomfører de numeriske beregningene på en generell måte.

La oss se på følgende eksempel. I øving 4 og 5 undersøkte vi dekningsgraden til to typer konfidensinterval for parameteren p i binomisk modell. Begge konfidensintervallene programmerte vi som funksjoner i R (funksjonene `konfint` og `konfint2`) som tok antall observerte suksesser x ut av antall delforsøk n som innargument og som returnerte endepunktene i intervallet i form av en vektor med lengde 2. For å undersøke dekningsgraden til `konfint2` skrev vi om funksjonen `dekningsgrad` slik at denne i stedet kalte `konfint2`. En mer elegant løsning er imidlertid å skrive om `dekningsgrad` slik at denne undersøker dekningsgraden slik at hvilket konfidensintervall vi skal beregne dekningsgraden til er spesifisert ved et nytt innargument med navn `konfintfunk`:

```
dekningsgrad <- function(konfintfunk,p,n,alpha,nsim=10000) {
  ntreff <- 0
  for (i in 1:nsim) {
     X <- rbinom(n=1,size=n,prob=p)
     ki <- konfintfunk(X,n,alpha)
     if (ki[1]<p & p<ki[2]) {
        ntreff <- ntreff + 1
     }
  }
  ntreff/nsim
}
```

Når vi nå gjør et kall til `dekningsgrad` må en referanse til funksjonen som skal beregne konfidensintervallet fra x og n oppgis som første argument. Kallet blir dermed seende slik ut.

```
> dekningsgrad(konfintfunk=konfint2,p=.1,n=50)
```

eller

```
> dekningsgrad(konfintfunk=konfint,p=.1,n=50)
```

hvis vi i stedet ønsker å finne dekningsgraden til det "vanlige" intervallet.

2 Optimeringsalgoritmer i R

Generelt vil Newton's metode ikke alltid konvergere. I praksis er det også arbeidskrevende å programmere algoritmen dersom vi jobber med ligningssystem med mange ukjente, f.eks. i forbindelse med sannsynlighetsmaksimering for modeller med et stort antall ukjente parametere.

I praksis bruker vi derfor gjerne mer generelle algoritmer som er innebygd i R når vi skal finne sannsynlighetsmaksimeringsestimater eller når vi jobber med andre optimeringsproblemer. Slike optimeringsalgoritmer finnes i en rekke varianter tilgjengelig i R bl.a. gjennom funksjonen `optim`. Noen av disse metodene er også modifikasjoner av Newton's metode, f.eks. den såkalte kvasi-Newton metoden. Denne bygger på at de deriverte beregnes numerisk ved at funksjonen som skal maksimaliseres evalueres gjentatte ganger i området rundt $\mathbf{z}_n$ (se figur 1c). Generelt virker de ulike optimeringsalgoritmene ved at den funksjonen som skal minimaliseres (eller maksimaliseres) evalueres i et antall punkter i parameterrommet slik at algoritmen kan "danne seg et bilde" av funksjonens form og slik jobbe seg i retning av funksjonens minimum.

La oss anta at vi ønsker å finne funksjonen

$$h(x, y) = (x - 2.5)^2 + (y - 1.5)^2 \tag{1}$$

sitt minimum ved bruk av `optim` i R. Vi trenger først å programmere h som en funksjon `h` i R. For at `h` skal kunne virke sammen med `optim` må den ta i mot de av sine innargumenter som `optim` skal optimalisere med hensyn til i form av en vektor, i dette tilfelle av lengde to, hvor verdiene av x og y ligger i første og andre element av denne vektoren. Vi kan programmere `h` f.eks. på følgende måte:

```
h <- function(p) {
  x <- p[1]
  y <- p[2]
  (x-2.5)^2+(y-1.5)^2
}
```

På samme måte som når vi bruker Newton's metode trenger vi å oppgi startverdier for x og y når vi bruker `optim`. Vi oppgir disse verdiene i form av vektor med startverdier. Neste argument til `optim` er en referanse til funksjonsdefinisjonen av på den funksjonen vi ønsker å minimalisere, vi oppgir med andre ord bare navnet på den funksjonen vi ønsker å minimalisere (uten etterfølgende parenteser slik som ved funksjonskall). Det er altså selve funksjonsdefinisjonen som er argument.

```
> optim(c(1,1),h)
$par
[1] 2.499977 1.499952

$value
[1] 2.88779e-09

$counts
function gradient
      61       NA

$convergence
[1] 0

$message
NULL
```

Resultatet av kallet til `optim` er en liste av ulike komponenter. Komponenten `$par` er vektoren som inneholder verdiene av x og y i h sitt minimum, og `$value` inneholder funksjonsverdien i dette minimumet. Vi ser også at `optim` har gjort 61 kall til vår funksjon `h` (listekomponenten `$counts`).

Dersom `optim` ikke finner noe minimum vil komponenten `$convergence` få verdi 1 og ikke 0.

Merk at `optim` generelt vil søke etter minimum. Oppgir vi argumentet `control=list(fnscale=-1)` søker `optim` etter maksimum.

3 Sannsynlighetsmaksimering

Anta at vi har observert dataene $x_1, x_2, \ldots, x_n$ fra en modell med parametervektor $\theta = (\theta_1, \theta_2, \ldots, \theta_k)$. Generelt vil modellen, dersom vi har kontinuerlig fordelte data, spesifisere simultantettheten til dataene, $f_{X_1,\ldots,X_n}(x_1, \ldots, x_n)$. Likelihoodfunksjonen vil generelt være definert som denne simultantettheten

beregnet i punktet $(x_1, x_2, \ldots, x_n)$, og betraktet som en funksjon av de ukjente parameterne, altså

$$L(\theta) = f_{X_1,\ldots,X_n}(x_1, \ldots, x_n). \tag{2}$$

Dersom dataene er uavhengig fordelt får vi at

$$L(\theta) = \prod_{i=1}^{n} f_{X_i}(x_i). \tag{3}$$

Dersom alle X_i'ene også er identisk fordelt (dette vil ikke være tilfelle f.eks. i en regresjonsmodell) har vi at

$$L(\theta) = \prod_{i=1}^{n} f_X(x_i). \tag{4}$$

For diskrete fordelte data for vi tilsvarende uttrykk.

3.1 Programmering av likelihoodfunksjonen

Anta at $t_1, t_2, \ldots, t_n$ er uavhengige identisk Weibullfordelte data med tetthet

$$f_T(t) = \frac{a}{b}\left(\frac{t}{b}\right)^{a-1} \exp\left(-\left(\frac{t}{b}\right)^{a}\right). \tag{5}$$

Da kan vi skrive log-likelihoodfunksjonen på formen

$$\ln L(a, b) = \sum_{i=1}^{n} \ln f_T(t_i), \tag{6}$$

Sannsynlighetstettheten $f_T(t_i)$ og logaritmen til denne kan beregnes i R med funksjonen `dweibull` ved å bruke tilleggsargumentet `log=T`. Eventuellt kunne vi først beregnet tettheten og så log-transformert med da ville vi risikert numeriske problemer hvis noen av sannsynlighetstetthetene var svært små (noe de vil kunne bli hvis vi befinner oss på feil sted i parameterrommet).

Alle leddene i summen kan beregnes ved at `dweibull` virker elementvis på datavektoren. Når vi skal programmere likelihoodfunksjonen i R slik at denne kan virke sammen med `optim` må også de variabler som funksjonen skal maksimaliseres med hensyn på oppgis som argument i form av en vektor. Hele likelihoodfunksjonen kan dermed programmeres f.eks. slik:

```
lnL <- function(par,t) {
   a <- par[1]
   b <- par[2]
   -sum(dweibull(t,shape=a,scale=b,log=T))
}
```

Minustegnet i siste linje gjør at vi slipper å oppgi tilleggsargumentet `control=list(fnscale=-1)` når vi skal beregne maksimum — `optim` vil i stedet finne minimum til minus log-likelihoodfunksjonen, altså maksimum til pluss log-likelihoodfunksjonen.

3.2 Sannsynlighetsmaksimering ved bruk av `optim`

Leser vi først inn levetidsdataene vi har brukt tidligere kan vi nå finne sannsynlighetsmaksimeringsestimatene til både a og b:

```
> t <- scan("/home/jarlet/undervisning/bioberegninger/spurv.dat")
> optim(c(1.3, 2), lnL, t=t)
$par
[1] 1.333679 1.787965

$value
[1] 1242.049

$counts
function gradient
      69       NA

$convergence
[1] 0

$message
NULL
```

Det er fornuftig å oppgi startverdier på parameterne i nærheten av det vi tror er sannsynlighetsmaksimeringsestimatene (f.eks. `c(1.3,2)` som her).

Vi ser at estimatet av formparameteren nå blir $\hat{a} = 1.33$ og at estimatet av skalaparameteren blir $\hat{b} = 1.78$ når begge estimeres fritt (til forskjell fra resultatene i øving 6 hvor vi antok at at $b = 2$).

Når vi gjør et kall til en funksjon (her `optim`) som i sin tur kaller en annen funksjon (her `lnL`) vil vi i mange tilfelle ha behov for å oppgi argumenter til den andre funksjonen (i dette tilfelle datavektoren `t`) i funksjonskallet til

den første. I eksempelet over har vi oppgitt argumentet `t=t` i funksjonskallet til `optim`. Dette argumentet sendes videre til `lnL` gjennom en spesiell type mekanisme, et såkalt "..."-argument, når `optim` i sin tur kaller `lnL`. Vi skal se nærmere på hvordan denne mekanismen fungerer nedenfor.

Alternativt kunne vi latt `lnL` referere til dataene gjennom en global variabel `t` men dette er uhensiktsmessig i tilfeller hvor vi ønsker å beregne sannsynlighetsmaksimeringsestimater både for observerte og simulerte data.

3.3 Detaljer

La oss se litt mer på detaljene i hva som skjer ved framgangsmåten brukt over. La oss først lage et konturplot av log-likelihoodfunksjonen. Funksjonen `contour` virker på samme måte som funksjonen `persp` (øving 7) men lager i stedet for en tredimensjonal overflate ett enklere todimensjonalt kart med "høydekvoter" som representerer punkter i parameterrommet med samme log-likelihood. En generell funksjon som lager et slikt konturplot kan se slik ut:

```
lnLplot <- function(lnL,min1,max1,min2,max2,xlab="",ylab="",gridsize=50,...){
   seq1 <- seq(min1,max1,length=gridsize)
   seq2 <- seq(min2,max2,length=gridsize)
   lnLmatrise <- matrix(NA, gridsize, gridsize)
   for (i in 1:gridsize) {
      for (j in 1:gridsize) {
         lnLmatrise[i,j] <- lnL(c(seq1[i],seq2[j]),...)
      }
   }
   contour(seq1,seq2,-lnLmatrise,xlab=xlab,ylab=ylab)
}
```

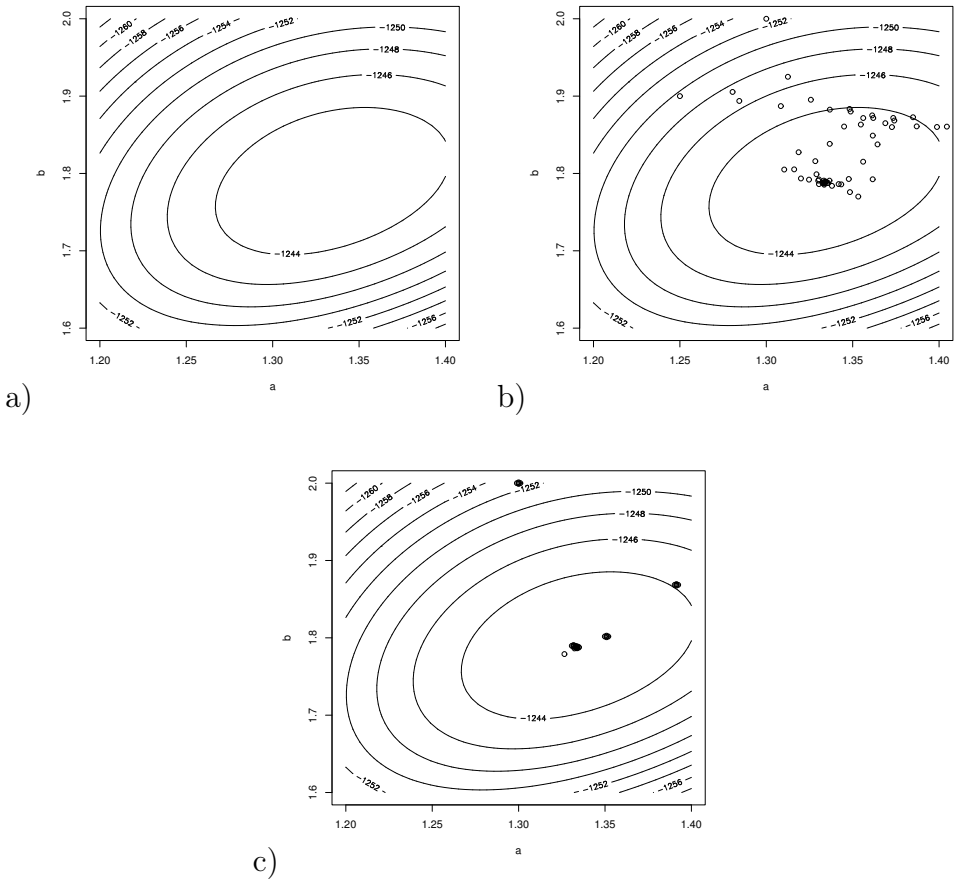
Her lar vi `lnLplot` ta i mot argumenter også gjennom det spesielle "..."-argumentet. Disse argumentene sender `lnLplot` så uendret videre til funksjonen `lnL` inne i den nestede for-løkken.

Vi kan nå lage et konturplot over et passende område av parameterrommet rundt estimatene med kommandoen

```
> lnLplot(lnL,1.2, 1.4, 1.6, 2, t=t, xlab="a",ylab="b")
```

Se figur 1a.

La oss nå se på hvordan `optim` leter seg fram gjennom parameterrommet. Dette kan lett illustreres ved at vi legger inn et par ekstra linjer i `lnL` som legger til punkter i konturplottet hver gang `lnL` kalles:



Figur 1: Konturplot av log-likelihoodfunksjonen. I plot b) er punktene i parameterrommet hvor likelihoodfunksjonen blir evaluert av `optim` plottet (når optimaliseringen gjøres ved bruk av default metoden `method=''Nelder-Mead''`).

```
lnL <- function(par,t,vispunkter=F) {
   a <- par[1]
   b <- par[2]
   if (vispunkter) {
      points(a,b)  # Legg til et punkt med koordinater (a,b)
      Sys.sleep(1) # Vent ett sekund
   }
   -sum(dweibull(t,shape=a,scale=b,log=T))
}
```

Beregner vi nå nye sannsynlighetsmaksimeringsestimater med `optim` ser vi hvordan algoritmen beveger seg gjennom parameterrommet:

```
> optim(c(1.3, 2), lnL, t=t, vispunkter=T)$par
[1] 1.333679 1.787965
```

Se figur 1b.

Samme plot med optimeringsalgoritmen kvasi-Newton (`method=''BFGS''`) er vist i figur 1c. I mange tilfeller vil den fungere bra og være noe raskere enn Nelder-Mead algoritmen som er default.

4 Flere eksempler

Anta at vi ønsker å studere hvor raskt et legemiddel nedbrytes etter at det er tatt av n pasienter. Vi antar at konsentrasjonen etter tid t er $(a-b)e^{-ct}+b$ slik at konsentrasjonen går mot b når t går mot uendelig og at den er a ved tid $t=0$. Anta at vi observerer konsentrasjoner X_i ved tidspunkt t_i for ulike pasienter $i=1,2,\ldots,n$. På grunn av måleusikkerhet antar vi videre at observasjonene X_i er normalfordelte med varians σ^2.

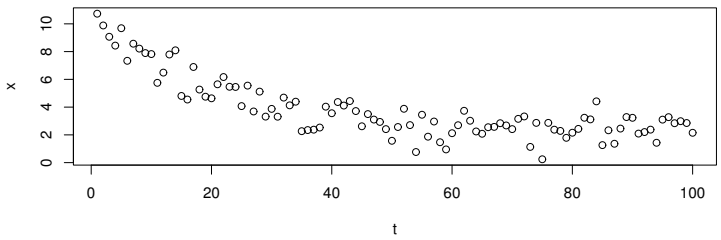
Oppsummerer vi dette har vi følgende statistiske modell:

$$X_i \sim N((a-b)e^{-ct_i}+b, \sigma^2). \tag{7}$$

Her er altså observasjonene uavhengige men ikke identisk fordelt fordi forventingsverdiene er forskjellige.

La oss først simulere ett datasett fra denne modellen. Anta at vi måler konsentrasjonen av legemiddelet i $n=100$ forskjellige pasienter ved de valgte tidspunktene $1,2,\ldots,100$ timer. Anta også at $a=10$, $b=2$, $c=0.05$, og $\sigma^2=1$. Et simulert datasett kan da lages på følgende måte:

```
> t <- 1:100
> x <- rnorm(n=100,mean=8*exp(-.1*t)+2,4)
> plot(t,x)
```



Figur 2: Simulerte data fra modell (7).

Plottet av dataene er vist i figur

Log-likelihoodfunksjonen for denne modellen kan programmeres på følgende måte:

```
lnL <- function(p,x,t) {
  a <- p[1]
  b <- p[2]
  c <- p[3]
  sigma2 <- p[4]
  -sum(dnorm(x,mean=(a-b)*exp(-c*t)+b,sd=sqrt(sigma2),log=T))
}
```

Legg merke til at `dnorm` virker elementvis på vektorene `x` og `(a-b)*exp(-c*t)+b`, forventningene til observasjonene, slik at `dnorm` returnerer logaritmen til alle leddene i summen som inngår i log-likelihoodfunksjonen.

Vi kan nå finne sannsynlighetsmaksimeringsestimatene ved å bruke `optim`. Legg merke til at både `t` og `x` er argumenter som blir sendt videre til `lnL` via `optim`'s "..."-argument:

```
> optim(c(1,1,1,1),lnL,x=x,t=t)
$par
[1] 6.54997289 0.11863612 0.01616672 5.05129293

$value
[1] 191.0224

$counts
function gradient
     501       NA
```

```
$convergence
[1] 1

$message
NULL
```

Vi ser at vi får `$convergence` blir lik 1, som betyr at `optim` ikke fant fram til noe maksimum i løpet av 500 iterasjoner. Velger vi mer en mer fornuftig startverdi på parametervektoren får vi:

```
> optim(c(10,3,.1,1),lnL,x=x,t=t)
$par
[1] 10.50474828  2.14439504  0.05815284  0.94928235

$value
[1] 139.3044

$counts
function gradient
     239       NA

$convergence
[1] 0

$message
NULL
```

Nå ser vi at estimatene blir liggende i nærheten av de sanne verdiene samtidig som `$convergence` blir lik 0. Dette betyr ikke nødvendigvis at noe maksimum er funnet, bare at den siste relative endringen i $\ln L$ er mindre enn `reltol` som har defaultverdi i størrelsesorden 10^{-8}. Se hjelpesiden til `optim`. Dette kan inntreffe også om `optim` havner i ett området i parameterrommet hvor $\ln L$ "flater" ut. I slike tilfeller er det lurt å kontrollere at man har funnet et virkelig maksimum ved å kalle `optim` med forskjellige startverdier av parametervektoren.