Statistical analysis lab #1

Goals:

- build a linear (regression) model from scratch; compare results with more sophisticated ways of implementing the same model
- (optional) graphical model diagnostics
- randomization approaches:
  - permutation tests
  - basic bootstrapping
  - posterior predictive simulation

Preliminaries

R knowledge prerequisites:

- writing R functions
- passing functions as arguments to other functions (e.g. `optim()`)
- using `for` loops, or `sapply`, to run chunks of code multiple times
- basic use of `apply`

Load packages:

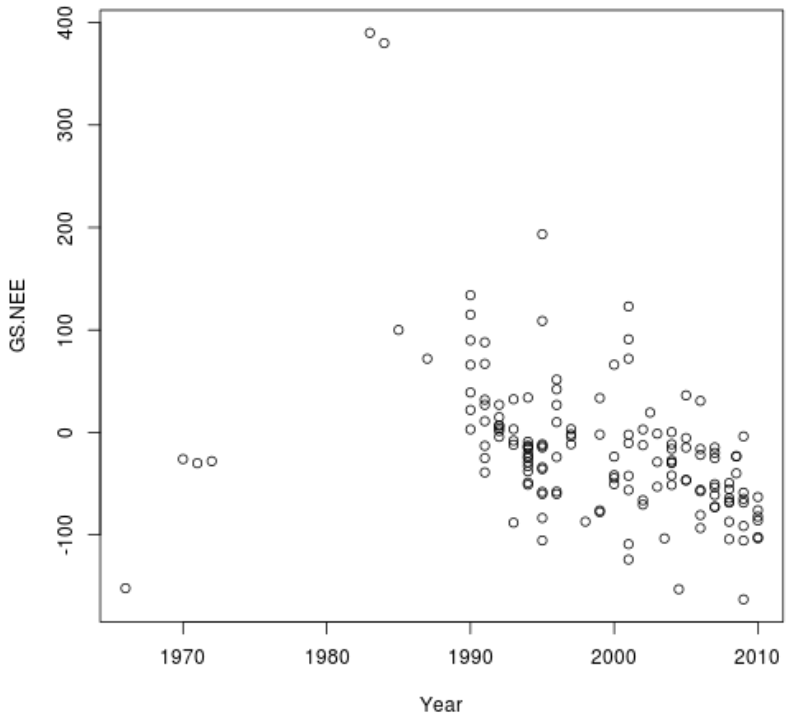
```
library(ggplot2); theme_set(theme_bw())
library(reshape2)  ## melt/dcast
library(plyr)     ## raply/aaply
## lmPerm is not available from CRAN, needs to be installed via
## library(devtools); install_version("lmPerm",version="1.1-2")
library(lmPerm)  ## lmp
library(arm)     ## sim
library(bbmle)
library(rbenchmark)
library(boot)
```

Read data:

```
tdat <- read.csv("tundra.csv",na.strings="-")
## construct a *centered* version of the predictor
tdat <- transform(tdat,cYear=Year-min(Year))
## alternatively tdat$cYear <- tdat$Year-min(tdat$Year)
```

Look at it:

```
plot(GS.NEE~Year,data=tdat)
```



Regression by hand

- Define a function `ssqfun()` that takes a parameter vector `p`; computes the linear prediction `y.hat=p[1]+p[2]*tdat$cYear`; and returns the sum of squared differences between `y.hat` and `tdat$GS.NEE`. (You'll either need to include `na.rm=TRUE` in your `sum()` call, or subset `tdat` so that there are no `NA` values: be careful, a simple `na.omit()` or `complete.cases()` applied to the whole data set will get rid of too much.)
- Look at the plot and guess reasonable starting values for the intercept and slope parameters.
- Put these variables into a vector (`c(slope,intercept)`) and try them out in your `ssqfun()` function: you should get a *finite* value (i.e. not `Inf` or `NA` or `NaN`); if you don't, go back and debug your function.

Now put your function into `optim()` and let the black box do its magic:

```
fit1 <- optim(par=c(400,-10),fn=ssqfun)
```

Check the results – do you know what all the pieces mean? (Hint: `$convergence` should be 0.) Read the `Value:` section of The Fine Manual if you're not sure.

Re-plot your data and superimpose the resulting fit:

```
plot(GS.NEE~cYear,data=tdat)
abline(coef=fit1$par,col=2)
```

Does it make sense?

Now we're going to take a little bit of a leap and combine several steps, i.e.:

- using a Normal negative log-likelihood ($= C + n/2 \log \sigma + 1/(2\sigma^2) \sum (y_i - x_i)^2$) rather than just the sum of squares
- using the `bbmle` package to tell R that we are working with a log-likelihood, and to get results that will work with convenience functions like `coef()`, `vcov()`, `confint()`, `profile()` ...
- using the *formula interface* in the `bbmle` package

```
tdat2 <- subset(tdat,!is.na(GS.NEE))
fit2 <- mle2(GS.NEE~dnorm(mean=int+slope*cYear,sd=sd0),
             data=tdat2,
             start=list(int=400,slope=-10,sd0=5))
```

- Try out `summary()`, `coef()` and confirm that you get the same answers. Actually, they're not *quite* the same answer. The `optim()` function uses a different default method by default. You can optionally try the following:
  - add `control=list(parscale=c(100,5))` to the `optim()` fit
  - use `method="BFGS"` for the `optim()` fit
  - use `method="Nelder-Mead"` for the `mle2` fit
- experiment with the `summary`, `vcov`, `predict`, `coef`, `confint`, `simulate` methods to see some of the advantages of using `mle2` over a raw `optim` call.
- Fitting the `sd` parameter is tricky; it shouldn't ever be negative, and if the starting value is too large the optimizer gets confused and heads for large values (the optimizer gives a warning saying that it has reached its maximum number of iterations). There are several solutions to this:

  - play around with the starting values until you don't get warnings any more
  - fit the standard deviation on a log scale, i.e. specify `sd=exp(logsd0)` in the formula (and provide a sensible starting value for `logsd0`)
  - use theory that says that for any value of the mean, the best estimate of the standard deviation is $\sum((y_i - x_i)^2)/(n-1)$ (this is actually a bias-corrected estimate, not the maximum likelihood estimate). You can define a `dnorm2` function that does this automatically and use `GS.NEE ~ dnorm2(int+slope*cYear)`:

```
dnorm2 <- function(x,mean,log=FALSE) {
    rss <- sum((x-mean)^2)
    n <- length(x)
    dnorm(x,mean=mean,sd=sqrt(rss/(n-1)),log=log)
}
```

If you have time, try out some of these methods. * We have skipped several steps between `optim` and `mle2`.

* We could have used `-sum(dnorm(...,log=TRUE))` rather than the sum of squares criterion in a function that we put into `optim()` * Alternatively we could also have written a negative log-likelihood function for `mle2` rather than using the formula interface (using either a single vector `NLLfun <- function(p) {...}` or named arguments `NLLfun <- function(int,slope,sd0) {...}`, although the latter is a little tricky.

Now use `lm` for the same fit as above. Compare the results.

- You can compare the speed of different methods with `benchmark()` from the `rbenchmark()` package (other people like the `microbenchmark` package), e.g.

```
benchmark(lm(GS.NEE~cYear,data=tdat),
          optim(par=c(400,-10),fn=ssqfun))
```

Permutation tests

- Now using `lm`, write a function `permfun()` that permutes the response variable (`GS.NEE`) (see lecture notes); fits a linear model to a specified data set (using `lm` is probably best); and returns the t statistic for the slope (see `coef(summary(...))`)

You should get this result:

```
set.seed(101); permfun()
```

```
## [1] -0.1932
```

- Use a `for` loop, `sapply`, or `plyr::raply` to get 1000 values from your function.
- plot a histogram of your results {`r par(las=1,bty="l") hist(permvec,col="gray")`}
- find the (two-tailed) p-value of the permutation by computing the fraction of time that the absolute value of the observed t statistic is >= the absolute value of the null t statistic.
- compare the results with the results of `lm` and of `lmPerm::lmp`.

Bootstrap tests

- Now write a bootstrap function for the linear regression slope. See the lecture notes; it should work almost the same as your `permfun`, but sampling rows of the data set with replacement rather than elements of the response variable without replacement, and it should return the estimated slope rather than the t statistic.

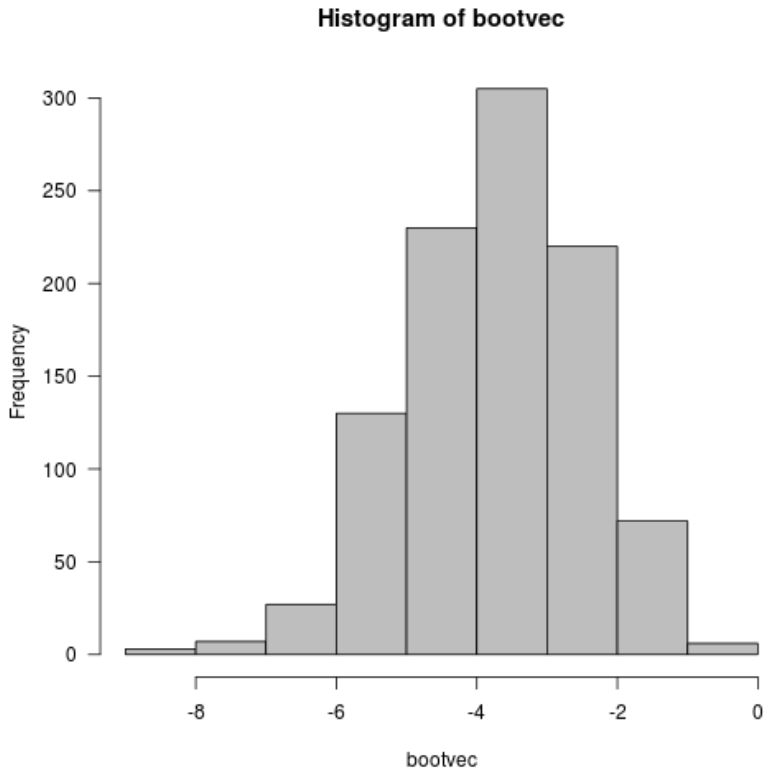
  You should get this result:

```
set.seed(101); bootfun()
```

```
## cYear
## -4.75
```

- Use a `for` loop, `sapply`, or `plyr::raply` to get 1000 values from your function.
- plot a histogram of your results

```
par(las=1,bty="l")
hist(bootvec,col="gray")
```



* find the 95% bootstrap confidence intervals by using the `quantile` function to compute the 0.025 and 0.975 quantiles of the vector.

```
##   2.5%  97.5%
## -6.169 -1.463
```

- Optionally, try out the `boot()` function from the `boot` package. The confusing part here is that `boot()` is called as `boot(data,statistic)` where `statistic` is a function similar to the one you wrote already, but it should take the data as its first argument and a vector of observation numbers as its second argument, something like this ...

```
bootfun2 <- function(data,statistic) {
    bdat <- data[statistic,]
    ## etc.
}
```

  Then you can use `boot` and `boot.ci` as follows:

```
bb <- boot(tdat,bootfun2,500)
boot.ci(bb,type=c("norm","perc"))
```

- compare your results with the results of `confint` applied to your `lm` fit.
- optionally, try out parametric bootstrapping for this example (resample values from the sampling distribution of the parameters; simulate data; re-fit the model to those data; extract the slope estimate; repeat).

Power analysis

Consider the following code for simulating linear, Normally distributed data:

```
p <- c(0,4,1)
x <- runif(20)
y <- rnorm(20,p[1]+p[2]*x,sd=p[3])
```

- Convert this into a function `powfun1` that takes `slope` as its single argument and returns a data frame with columns `x` and `y`:
- Write a function `powfun2` that takes a data frame with columns `x` and `y` and returns a p-value for the slope (use `coef(summary(fit))["x","Pr(>|t|)"]`).
- Set up a vector of possible slopes:

```
slopevec <- seq(0,5,length.out=51)
```

- Define the desired number of simulations:

```
nsim <- 100
```

- Allocate a matrix for results:

```
powmat <- matrix(nrow=length(slopevec),ncol=nsim)
```

- Now set up a `for` loop:

```
for (i in 1:length(slopevec)) {
    for (j in 1:nsim) {
        powmat[i,j] <- powfun2(powfun1(slope=slopevec[i]))
    }
}
```

- Now calculate the power for each slope using `rowMeans(powmat<0.05)` and plot a power curve.