Week 4 exercises

Brad McNeney

Short exercises based on chapters from text

1. Explain the output of the following code chunk.

```
f <- function() {
  fe <- environment(f)
  ee <- environment()
  pe <- parent.env(ee)
  list(fe=fe,ee=ee,pe=pe)
}
f()
```

```
## $fe
## <environment: R_GlobalEnv>
##
## $ee
## <environment: 0x7fa0655b5240>
##
## $pe
## <environment: R_GlobalEnv>
```

2. Read the help files on the `exists()` and `get()` functions. Explain the output of the following code chunk.

```
f <- function(xx) {
  xx_parent <- if(exists("xx",envir=environment(f))) {
    get("xx",environment(f))
  } else {
    NULL
  }
  list(xx,xx_parent)
}
f(2)
```

```
## [[1]]
## [1] 2
##
## [[2]]
## NULL
```

```
xx <- 1
f(2)
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 1
```

```
rm(xx)
```

3. Write a function with argument `xx` that tests whether `xx` exists in the parent environment and, if so,
   (a) assigns the value of `xx` in the parent environment to the variable `xx_parent`, and
   (b) tests whether `xx` and `xx_parent` are equal. If the test is FALSE, throw a warning to alert the user to the fact that the two are not equal.
4. Write an infix version of `c()` that concatenates two vectors.

Map-Reduce

Refer to the Wikipedia page on algorithms for computing sample variances and covariances: https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance We will implement the two-pass algorithm for computing the sample variance as a Map-Reduce.

1. Use the following code chunk from the lecture 4 notes to simulate data in 10 chunks and calculate the overall sample mean.

```
rfun <- function(seed,n) { set.seed(seed); return(rnorm(n))}
dat2 <- lapply(1:10,rfun,n=1e5)
mfun2 <- function(x) { return(data.frame(sum=sum(x),n=length(x))) }
sumdat <- lapply(dat2,mfun2)
simple_reduce <- function(x, f) { # Text section 9.5
  out <- x[[1]]
  for (i in seq(2, length(x))) {
    out <- f(out, x[[i]])
  }
  out
}
allres <- simple_reduce(sumdat,rbind)
my_mean <- sum(allres[,"sum"])/sum(allres[,"n"]) # mean
```

2. Write a function that takes a vector and `my_mean` as input and returns (i) the sum of squared deviations between the vector's values and `my_mean` and (ii) the number of vector values (n).
3. Use `lapply()` to call your function from (2) on each element of `dat2`.
4. Use `simple_reduce()` to combine your results into a single data frame, and calculate the sample variance from this data frame. Compare your answer to `var(unlist(dat2))`.

Recursive partitioning

- The following code chunk is the start of an implementation of recursive partitioning using a binary tree data structure to store the partition.
- Binary trees can be implemented as a linked list of nodes that contain
    1. data
    2. a pointer to the left child
    3. a pointer to the right child
- For our recursive partitioning example, the data will be a region of the original covariate space and the response/covariate data in that region.
- The following code chunk establishes node and region data structure.

```
# Constructor for the node data structure:
new_node <- function(data,childl=NULL,childr=NULL){
  nn <- list(data=data,childl=childl,childr=childr)
  class(nn) <- "node"
  return(nn)
}
# The data stored in the node are a partition, or region of the
# covariate space. Constructor for region data structure:
new_region <- function(coords=NULL,x,y){
  if(is.null(coords)) {
    coords <- sapply(x,range)
  }
  out <- list(coords=coords,x=x,y=y)
  class(out) <- "region"
  return(out)
}
```

- Some tests of the above constructors are given in the next code chunk.

```
set.seed(123); n <- 10
x <- data.frame(x1=rnorm(n),x2=rnorm(n))
y <- rnorm(n)
new_region(x=x,y=y)
```

```
## $coords
##             x1        x2
## [1,] -1.265061 -1.966617
## [2,]  1.715065  1.786913
##
## $x
##             x1         x2
## 1  -0.56047565  1.2240818
## 2  -0.23017749  0.3598138
## 3   1.55870831  0.4007715
## 4   0.07050839  0.1106827
## 5   0.12928774 -0.5558411
## 6   1.71506499  1.7869131
## 7   0.46091621  0.4978505
## 8  -1.26506123 -1.9666172
## 9  -0.68685285  0.7013559
```

```
## 10 -0.44566197 -0.4727914
##
## $y
##  [1] -1.0678237 -0.2179749 -1.0260044 -0.7288912 -0.6250393 -1.6866933
##  [7]  0.8377870  0.1533731 -1.1381369  1.2538149
##
## attr(,"class")
## [1] "region"
```

```
new_node(new_region(x=x,y=y))
```

```
## $data
## $coords
##             x1        x2
## [1,] -1.265061 -1.966617
## [2,]  1.715065  1.786913
##
## $x
##             x1         x2
## 1  -0.56047565  1.2240818
## 2  -0.23017749  0.3598138
## 3   1.55870831  0.4007715
## 4   0.07050839  0.1106827
## 5   0.12928774 -0.5558411
## 6   1.71506499  1.7869131
## 7   0.46091621  0.4978505
## 8  -1.26506123 -1.9666172
## 9  -0.68685285  0.7013559
## 10 -0.44566197 -0.4727914
##
## $y
##  [1] -1.0678237 -0.2179749 -1.0260044 -0.7288912 -0.6250393 -1.6866933
##  [7]  0.8377870  0.1533731 -1.1381369  1.2538149
##
## attr(,"class")
## [1] "region"
##
## $childl
## NULL
##
## $childr
## NULL
##
## attr(,"class")
## [1] "node"
```

- The recursive partitioning function is shown below. We'll discuss this in class.

```
#-----------------------------------------------------#
# Recursive partitioning function.
recpart <- function(x,y){
  init <- new_node(new_region(x=x,y=y))
  tree <- recpart_recursive(init)
```

```
  class(tree) <- c("tree",class(tree))
  return(tree)
}
recpart_recursive <- function(node) {
  R <- node$data
  # stop recursion if region has a single data point
  if(length(R$y) == 1) { return(NULL) }
  # else find a split that minimizes a LOF criterion
  # Initialize
  lof_best  <- Inf
  # Loop over variables and splits
  for(v in 1:ncol(R$x)){
    tt <- split_points(R$x[,v]) # Exercise: write split_points()
    for(t in tt) {
      gdat <- data.frame(y=R$y,x=as.numeric(R$x[,v] <= t))
      lof <- LOF(y~.,gdat) # Exercise: write LOF()
      if(lof < lof_best) {
        lof_best <- lof
        childRs <- split(R,xvar=v,spt=t) # Exercises: write split.region()
      }
    }
  }
  # Call self on best split
  node$childl <- recpart_recursive(new_node(childRs$Rl))
  node$childr <- recpart_recursive(new_node(childRs$Rr))
  return(node)
}
```

Exercises

1. Write `split_points()`. The function should take a vector of covariate values as input and return the sorted unique values. You will need to trim off the maximum unique value, because this can't be used as a split point. (As yourself why not.) Write a snippet of R code that tests your function.
2. Write the function `LOF()` that returns the lack-of-fit criterion for a model. The function should take a model formula and data frame as input, pass these to `lm()` and return the residual sum of squares. Write a snippet of R code that tests your function.
3. Write `split.region()`. The function should take a region `R`, the variable to split on, `v`, and the split point, `t`, as arguments. Split the region into left and right partitions and return a list of two regions labelled `Rl` and `Rr`. Note: It is tempting to split the x and y data and calculate the coordinates matrix from the x's, as the constructor does when not passed a coordinates matrix. However, this will leave gaps in the covariate space. (Ask yourself why.) Write a snippet of R code that tests your function.
4. Run `recpart()` with your versions of `split_points()`, `LOF()` and `split.region()`. Use the test data `x` and `y` defined in the testing code chunk as input and save your output to an object called `testres`. The object will not print properly at this point. We will write a print method in lab 3. You do not need to check that the output is correct.