Solutions to Homework 3

36-462

Spring 2009

1. (a) First, we need to generate a time series from the logistic map; we already have code to do this. Second, we need to check in which cell of the generating partition the series is at each time. Then we need to return that sequence of cells.

   The first part uses some code for the logistic map (taken from `03.R`, plus the discretization function. (Fully-commented code is online in the accompanying file.)

```
logistic.genpart <- function(x) {
  ifelse(x<0.5,'L','R')
}

logistic.symbseq <- function(timelength,r) {
  x <- logistic.map.ts(timelength,r)
  s <- logistic.genpart(x)
  return(s)
}
```

(b) Start from the end and work backwards. We want to run a test for independence of two random variables, and the hint tells us that the easiest test to use is χ^2. To run such a test, we need a contingency table. What needs to go into the table are counts of how many times a given block of length L was followed by another block of length L. That means we need to divide a time series into those non-overlapping blocks, and then tabulate them. Finally, to do this to a time series, we need to actually have the time series. That last part was handled in part (a) of the problem.

So our solution should look something like this:

```
logistic.map.indep.test <- function(L,n=min(1e4,10*(2^(2*L))),r=1) {
  s <- logistic.symbseq(n,r)
  successive.blocks <- symbseq.to.successive.blocks(s,L)
  my.tab <- table(successive.blocks)
  my.test <- chisq.test(my.tab)
  return(list(p.value=my.test$p.value,test=my.test,count.table=my.tab))
}
```

where `n` is the length of the time series to simulate. (See below for the complicated-looking default value.) Notice that the only thing we actually have to write — the only thing which is currently vaporware — is the `symbseq.to.successive.blocks` function; we wrote `logistic.symbseq` in part (a), and `table` and `chisq.test` are already part of R.

What we need to do is take a window of length L (not to be confused with the symbol "L"!), slide it along the symbol sequence, and record all the patterns we see. We also need to keep track of which pattern followed that one, in the next L block. The common element here is finding all the length L blocks, which we'll push back a step to another function. We'll assume that this function gives us a vector of blocks. (See online code for detailed comments.)

```
symbseq.to.successive.blocks <- function(s,L) {
  n <- length(s)
  all.blocks <- symbseq.to.blocks(s,L)
  # The "leaders" begin at positions 1, 2, ... n-2L+1 (because there
  # needs to be another, following block of length L after each of them)
  max.index.leaders <- n-2*L+1
  # The "followers" begin at positions L+1, L+2, ... n-L+1 (because there
  # needs to be a "leader" block of length L before each of them)
  min.index.followers <- L+1
  max.index.followers <- n-L+1
  leaders <- all.blocks[1:max.index.leaders]
  followers <- all.blocks[min.index.followers:max.index.followers]
  return(list(leaders=leaders,followers=followers))
}
```

Now we write the `symbseq.to.blocks` function.

```
symbseq.to.blocks <- function(s,L) {
  n <- length(s)
  # We need to take blocks from the sequence and collapse them into
  # single strings; make this operation a local function
  collapser <- function(i) {paste(s[i:(i+L-1)],collapse="")}
  # A length L block can't start at any position whose index > n-L+1,
  # though it could start there.
  max.index <- n -L+1
  blocks <- sapply(1:max.index,collapser)
  return(blocks)
}
```

At this point it's a good idea to check that everything is working right with a small example.

```
> ss <- c("L","R","L","R","L","R","L")
> symbseq.to.blocks(ss,2)
[1] "LR" "RL" "LR" "RL" "LR" "RL"
> symbseq.to.successive.blocks(ss,2)
$leaders
[1] "LR" "RL" "LR" "RL"

$followers
[1] "LR" "RL" "LR" "RL"
```

You can check by hand that the code works on this example, and on this one:

```
> rr <- c("L","L","L","R","L","R","R")
> symbseq.to.blocks(rr,2)
[1] "LL" "LL" "LR" "RL" "LR" "RR"
> symbseq.to.successive.blocks(rr,2)
$leaders
[1] "LL" "LL" "LR" "RL"

$followers
[1] "LR" "RL" "LR" "RR"
```

Ideally at this point I'd check an $L = 3$ case, but I'm just the teacher here.

Going back to `logistic.map.indep.test`, once we have the successive blocks we need to get a contingency table from them, so they need to work sensible with the `table` function. Unsurprisingly (because otherwise, would I have it here?), they do:

```
> table(symbseq.to.successive.blocks(ss,2))
       followers
```

```
leaders LR RL
     LR  2  0
     RL  0  2
```

Finally, the output of `table` needs to work as input to `chisq.test`, but it does:

```
> chisq.test(table(symbseq.to.successive.blocks(ss,2)))

  Pearson's Chi-squared test

data:  table(symbseq.to.successive.blocks(ss, 2))
X-squared = 1, df = 1, p-value = 0.3173

Warning message:
In chisq.test(table(symbseq.to.successive.blocks(ss, 2))) :
  Chi-squared approximation may be incorrect
```

`chisq.test` is giving us a warning here, because the χ^2 approximation to the distribution of the test statistic is only valid if there are a fairly large number of counts for each cell in the table. The usual rule of thumb is that the expected number of counts must be at least 5; let's say 10 to be safe. Each cell in the table corresponds to a word of length $2L$, and we expect (for IID coin-tossing) that each such word is equally likely, so we want $10 = n/2^{2L}$, or $n = 10 \times 2^{2L}$. Of course this grows very rapidly with L, so our default will be to cap it at say 10^4. This is only a default so it can be over-ridden.

Returning the full test results and the count table as well as the p-value is not strictly necessary but doesn't hurt.

How do we know if this is working? If the "leader" and "follower" blocks *are* independent, then the p value of the test should be uniformly distributed on $[0, 1]$, and their CDF should be a straight diagonal line. Let's check that by re-running the test a bunch of times and plotting the empirical CDF (Figure 1).

```
> plot(ecdf(replicate(1000,logistic.map.independence.test(2)$p.value)),
  xlab="Nominal p-value",ylab="True p-value",main="Distribution of
  p-values")
> abline(a=0,b=1,col="blue",lty=2)
```

(c) At $r = 0.966$, the symbolic dynamics are not a Bernoulli process by any reasonable test:

```
> summary(replicate(1000,logistic.map.independence.test(2,3e3,0.966)$p.v
      Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
3.000e-281 4.478e-239 3.061e-230 2.796e-192 5.070e-221 2.269e-189
```

This is as close to the test saying "inconceivable" as you could hope.

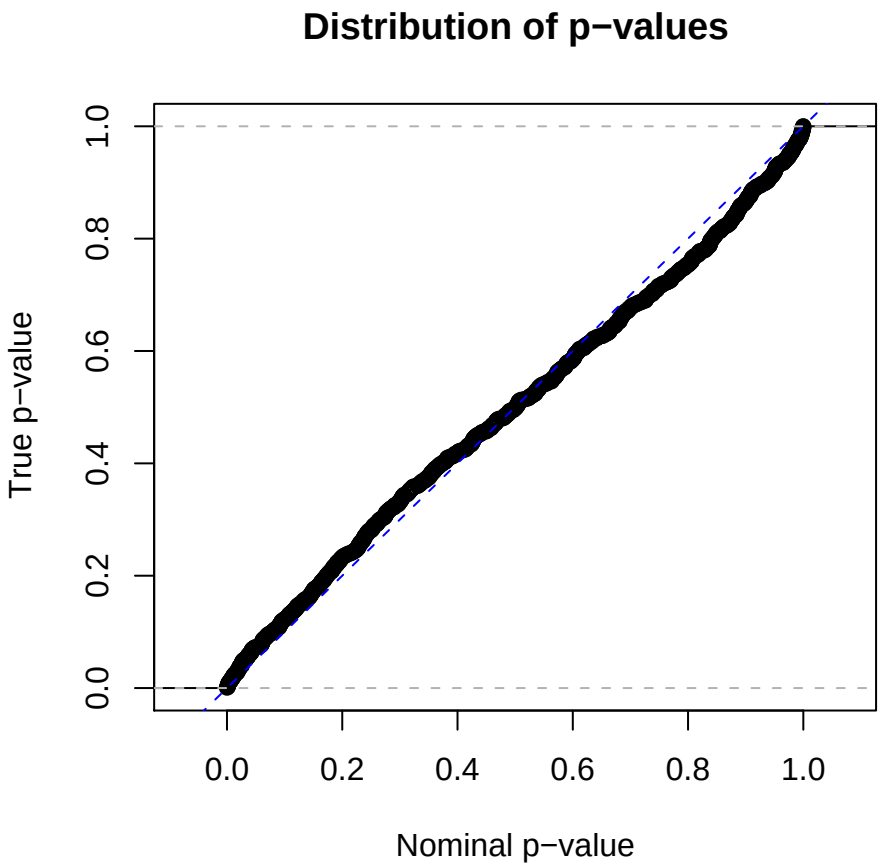

Figure 1: Distribution of p-values obtained from question 2b (black circles), with theoretical uniform distribution (dashed blue line).

2. Recall that the topological entropy rate is defined as

$$h_0 \equiv \lim_{L\to\infty} \frac{1}{L} \log W_L \tag{1}$$

where W_L is the number of allowed words of length L.

(a) There are at least three ways to do this. The simplest one is to just count the number of distinct *observed* words of length L, $\widehat{W}_L$, and estimate by division:

$$\widehat{h}_0^{division} \equiv \frac{1}{L} \log \widehat{W}_L \tag{2}$$

for some large L, as our estimate of h_0.

The second way is to notice that if the limit exists, then for large L we must have

$$\log W_L \approx C + h_0 L \tag{3}$$

where C is some constant we don't care about. So if we regress $\widehat{W}_L$ on L, the slope will be an estimate of h_0. Call this the *regression* estimate.

The third way is to use the fact, mentioned in the on-line notes about the topological entropy rate, that

$$h_0 = \lim_{L\to\infty} \log W_L - \log W_{L-1} \tag{4}$$

(Notice that this also follows from the linear expression I gave above.) So yet another estimate of h_0 is to take

$$\log \widehat{W}_L - \log \widehat{W}_{L-1} \tag{5}$$

for some large L. Call this the *difference* estimate.

(There is also a fourth way, which is to fit a model, like a finite-state machine, to the symbol dynamics and then calculate the topological entropy rate of the model analytically. This is often the most reliable approach, but involves first building a good model-fitter.)

All three estimates will ultimately converge on the same value, if you feed them enough data. In principle, all of them work best when the value of L is large. In practice, if L is too large relative to n, we see only a very small sample of the allowed words, i.e., $\widehat{W}_L$ becomes much smaller than W_L, introducing systematic errors into our estimate. At the very least, when $\widehat{W}_L > \widehat{W}_{L+1}$, we do not have enough data to say what is happening with the longer words.

For the logistic map, we use a binary alphabet (symbols set), so there are at most 2^L words of each length. To give us some chance of seeing each of them, we should use a symbol sequence which is at

least a few times longer than the number of words we might run into, say 10×2^L.

Here's how to do the division estimate.[1]

```
logistic.TER.division <- function(r,L,n=10*(2^L)) {
  s <- logistic.symbseq(n,r)
  blocks <- symbseq.to.blocks(s,L)
  word.table <- table(blocks)
  W.L <- dim(word.table) # Counts number of distinct allowed words
  return(log(W.L)/L)
}
```

Notice the trick with using the `table` function to identify all the *distinct* words. Let's re-cycle that for the regression estimate.

```
logistic.TER.regression <- function(r,L,n=10*(2^L)) {
  s <- logistic.symbseq(n,r)
  W <- vector(mode="numeric",length=L)
  for (i in (1:L)) {
    blocks <- symbseq.to.blocks(s,i)
    W[i] <- dim(table(blocks))
  }
  my.regression <- lm(logcounts ~ lengths,
                      data.frame(logcounts=log(W),lengths=(1:L)))
  return(as.vector(my.regression$coefficients[2]))
}
```

And here's the difference estimate:

```
logistic.TER.difference <- function(r,L,n=10*(2^L)) {
  s <- logistic.symbseq(n,r)
  lastW <- dim(table(symbseq.to.blocks(s,L)))
  nextotlastW <- dim(table(symbseq.to.blocks(s,L-1)))
  return(log(lastW) - log(nextotlastW))
}
```

To double-check these, notice that when $r = 1$, we have IID coin-tossing, and every sequence of length L is allowed, so $W_L = 2^L$. This means that h_0 should be $\log 2 = 0.6931472$.

```
> logistic.TER.division(1,3)
[1] 0.6931472
> logistic.TER.regression(1,3)
[1] 0.6931472
> logistic.TER.difference(1,3)
[1] 0.6931472
```

[1] Wouldn't it be better programming practice to write a *separate* function to estimate h_0 from an *arbitrary* symbol sequence, so that it could be used with other dynamical systems? Yes, but we'd still need something which took r, not a symbol sequence, as its argument, in order to plot h_0 versus r.

which checks out.

Finally, let's plot these estimates as functions of r to see if we're getting something reasonable. We know that h_0 should be zero whenever the logistic map goes to a limit cycle (see the online notes for details). I use $L = 6$ simply for reasons of speed. The plot is Figure 2.

```
> r.values <- seq(from=0,to=1,length.out=200)
> difference.values <- sapply(r.values,logistic.TER.difference,L=6)
> plot(r.values,difference.values,type="l",xlab="r",ylab="Estimated h0",
       main="topological entropy rate estimates")
> division.values <- sapply(r.values,logistic.TER.division,L=6)
> lines(r.values,division.values,lty=2)
> regression.values <- sapply(r.values,logistic.TER.regression,L=6)
> lines(r.values,regression.values,lty=3)
```

(b) The easiest way to get a value for the standard error here is simply to re-run the estimator multiple times and take the standard deviation. This only captures the error associated with the fluctuations from one run of the simulation to another, rather than the systematic errors which come from biases in the estimator, etc.

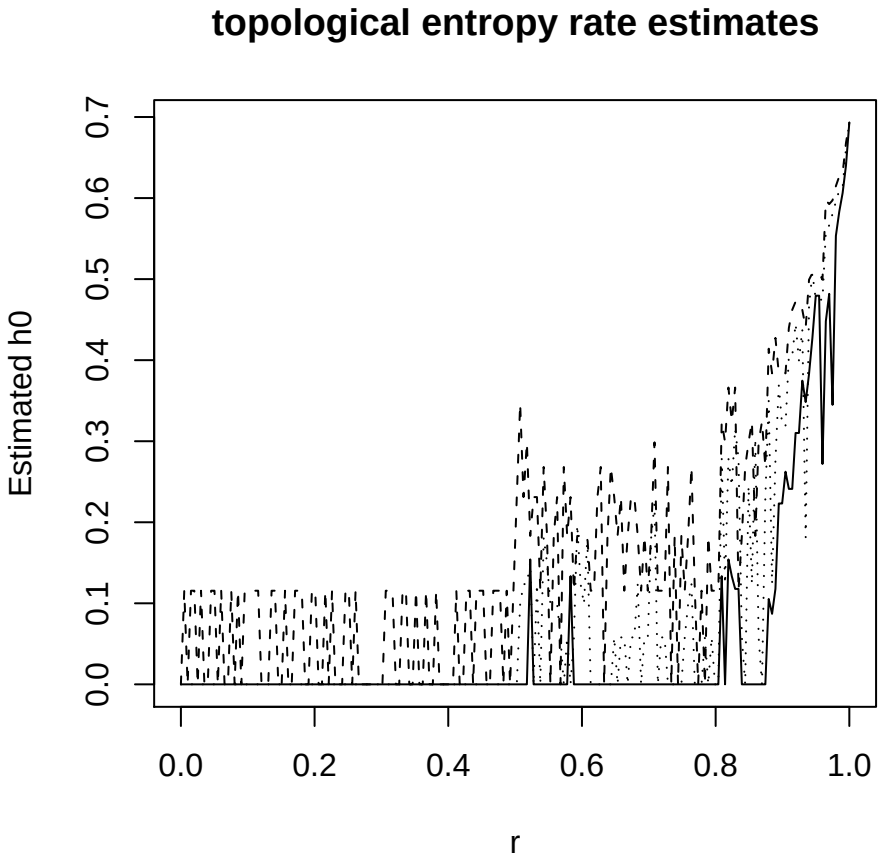

Figure 2: Three estimates of the topological entropy rate of the logistic map. Solid line, difference estimate. Dashed line, division estimate. Dotted line, regression estimate. The true value of h_0 is 0 whenever the map goes to a limit cycle, i.e., whenever $r < 0.866$ or so, suggesting that the division and regression estimates may have a larger upward bias than the difference estimate.

3. Basically all the parts needed for this were already assembled for solving problem 1. We need to estimate the 2×2 transition matrix P_{ij}, and the maximum likelihood estimates are

$$\widehat{P}_{ij} = \frac{N_{ij}}{\sum_{j=0}^{1} N_{ij}} \tag{6}$$

where N_{ij} counts the number of times the symbol i is followed by the symbol j. We can get this from the `symbseq.to.successive.blocks` function we wrote, followed by using `table`.

```
markov.mle.1 <- function(s) {
  blocks <- symbseq.to.successive.blocks(s,1)
  counts <- table(blocks)
  # prop.table converts a count table to proportions, either by rows
  # or by columns, depending on 2nd argument - see its help file
  mle <- prop.table(counts,1)
  log.like <- sum(counts[counts>0]*log(mle[mle>0]))
  return(list(transition.matrix=mle,log.like=log.like))
}
```

We don't *need* the log-likelihood here, but it doesn't hurt to compute it. Remember it's

$$\mathcal{L} = \sum_{i,j} N_{ij} \log \widehat{P}_{ij} \tag{7}$$

Even though $0 \log 0 = 0$, R does not like taking log of 0, so the conditions in the sum restrict us to the strictly-positive terms.

Here's a quick check that this is working properly:

```
> ss <- rbinom(20,1,0.7)
> ss
 [1] 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 0 1
> table(symbseq.to.successive.blocks(ss,1))
       followers
leaders  0  1
      0  2  2
      1  2 13
> markov.mle.1(ss)
$transition.matrix
       followers
leaders         0         1
      0 0.5000000 0.5000000
      1 0.1333333 0.8666667

$log.like
[1] -8.662706
```

There are four positions where the sequence has "0"; two of them are followed by "0" and two of them are followed by "1". There are 15 positions where a "1" is followed by something. (There is also a last "1", at the end, and we don't know what it transitions to.) Of these, two of them are followed by "0". So the counts are correct, and converting the counts to probabilities gives us exactly what the function says. This isn't *right* — the data came from IID coin-tossing, with $p = 0.7$, so the right matrix would be

$$\begin{bmatrix} 0.3 & 0.7 \\ 0.3 & 0.7 \end{bmatrix}$$

but it's not crazy to not get this right with only 20 data points!

To see this program in a better light, let's write a small function to simulate from a binary Markov chain.

```
rbinmarkov <- function(n, p01, p11, p1start=NULL) {
  if (is.null(p1start)) {
    P = matrix(c(1-p01,p01,1-p11,p11),nrow=2)
    # Find the invariant distribution as the leading normalized eignevector
    first.eigenvec = eigen(P)$vectors[,1]
    P.inv = first.eigenvec/sum(first.eigenvec)
    p1start = P.inv[2]
  }
  s = vector(length=n)
  s[1] = rbinom(1,1,p1start)
  for (i in 2:n) {
    s[i] = rbinom(1,1,ifelse(s[i-1]<1,p01,p11))
  }
  return(s)
}
```

Now let's try our estimator on the output from this.

```
> rs <- rbinmarkov(1e4,0.3,0.7)
> markov.mle.1(rs)
$transition.matrix
          followers
leaders         0         1
      0 0.7086676 0.2913324
      1 0.3046236 0.6953764

$log.like
[1] -6088.478
```

Since the true matrix is $\begin{bmatrix} 0.7 & 0.3 \\ 0.3 & 0.7 \end{bmatrix}$, this seems to be working.