STA 032 R Homework 2

Hardy Jones
999397426
Professor Melcon
Winter 2015

1. (a)

n	probability
100	0.07000
1000	0.04600
10000	0.05650
100000	0.05903

(b)

n	probability
100	0.0800
1000	0.0550
10000	0.0577
100000	0.0588

(c)

n	probability
100	0.00000
1000	0.00900
10000	0.01080
100000	0.00946

2. (a)

n	probability
100	0.31000
1000	0.32000
10000	0.33200
100000	0.33213

(b)

n	probability
100	0.44000
1000	0.52900
10000	0.52750
100000	0.53243

(c)

n	probability
100	0.6100
1000	0.6390
10000	0.6344
100000	0.6324

(d)

n	probability
100	0.97000
1000	0.89100
10000	0.90090
100000	0.90126

(e)

n	probability
100	0.1700
1000	0.2240
10000	0.2336
100000	0.2336

(f)

n	probability
100	0.55000
1000	0.71600
10000	0.69830
100000	0.70237

(g)

n	probability
100	0.60000
1000	0.64000
10000	0.62840
100000	0.63189

Appendix A R code

Problem 1

We start by loading the representation of the deck of cards.

```
CardValue = rep(c("A", 2:10, "J", "Q", "K"), times = 4)
CardSuit  = rep(c("S", "C", "H", "D"),  each = 13)
CardDeck  = t(rbind(CardValue, CardSuit))
```

Next we create some helper functions to keep this code comprehensible.

```
source("./preamble.R")

# We want a function that takes a deck,
# draws one card at random,
# and returns both the deck after being drawn from, and the drawn card.
# Draw : Deck -> (Card, Deck)
Draw <- function(deck) {
  len <- length(deck) / 2
  i <- sample(len, 1)
  list(card = deck[i, ], deck = deck[-i, ])
}

# Using ‘Draw‘, we draw two cards.
# Draw2 : Deck -> (Card1, Card2, Deck)
Draw2 <- function(deck) {
  drawn1 <- Draw(deck)
  drawn2 <- Draw(drawn1$deck)
  list(card1 = drawn1$card, card2 = drawn2$card, deck = drawn2$deck)
}

# We simulate ‘n‘ runs of supplied predicate,
# returning only the number that pass the predicate.
# Simulate : (Deck -> Boolean, Int) -> Int
Simulate <- function(p, n) {
  filtered <- Filter(function(x) { p(CardDeck) }, 1:n)
  length(filtered)
}

# The actual probability runner.
# Probability : (Deck -> Boolean) -> {n : [Float], probability : [Float]}
Probability <- function(f) {
  sim.ns <- c(100, 1000, 10000, 100000)
  sims <- sapply(sim.ns, function(n) { Simulate(f, n) / n })

  data.frame(n = sim.ns, probability = sims)
}
```

(a)

Now, we can very simply create a predicate to test whether we have a pair, and run the simulation.

```r
source("./preamble.R")
source("./1.R")

# We want a function that checks if two cards are pairs.
# ArePair : (Card, Card) -> Boolean
ArePair <- function(card1, card2) {
  card1[1] == card2[1]
}

# We draw two cards and check if they're a pair.
# DrawPair : Deck -> Boolean
DrawPair <- function(deck) {
  drawn <- Draw2(deck)
  ArePair(drawn$card1, drawn$card2)
}

# We actually run the simulation and return a data frame with probabilities.
probability.pair <- Probability(DrawPair)
```

(b)

We can do similarly for hearts.

```r
source("./preamble.R")
source("./1.R")

# We want a function that checks if a card is a heart.
# IsHeart : Card -> Boolean
IsHeart <- function(card) {
  card[2] == "H"
}

# We want a function that checks if two cards are hearts.
# AreHearts : (Card, Card) -> Boolean
AreHearts <- function(card1, card2) {
  IsHeart(card1) && IsHeart(card2)
}

# We draw two cards and check if they're both hearts.
# DrawHearts : Deck -> Boolean
DrawHearts <- function(deck) {
  drawn <- Draw2(deck)
  AreHearts(drawn$card1, drawn$card2)
}

# We actually run the simulation and return a data frame with probabilities.
probability.hearts <- Probability(DrawHearts)
```

(c)

And finally, for the complex test of a pair with one card a diamond and the other card a heart.

```
source("./preamble.R")
source("./1.R")
source("./1a.R")
source("./1b.R")

# We want a function that checks if a card is a diamond.
# IsDiamond : Card -> Boolean
IsDiamond <- function(card) {
  card[2] == "D"
}

# We want to ensure one card is a heart and the other a diamond.
AreHeartDiamond <- function(card1, card2) {
  (IsHeart(card1) && IsDiamond(card2)) || (IsDiamond(card1) && IsHeart(card2))
}

# We reuse our functions from before
AreComplex <- function(card1, card2) {
  ArePair(card1, card2) && AreHeartDiamond(card1, card2)
}

# We draw two cards and check if they're a pair
# with one a heart and the other a diamond.
# DrawComplex : Deck -> Boolean
DrawComplex <- function(deck) {
  drawn <- Draw2(deck)
  AreComplex(drawn$card1, drawn$card2)
}

# We actually run the simulation and return a data frame with probabilities.
probability.complex <- Probability(DrawComplex)
```

Problem 2

We need a data structure for coins. Again we break down functions so they are simple.

We also have a simulator and a probability reporter.

```
# We want each coin to know its probabilities.
colors <- c("Red", "Blue", "Green")
heads <- c(0.4, 0.7, 0.5)
coins <- cbind(colors, heads)

# We choose one coin with equal probability.
# ChooseCoin : [Coin] -> Coin
ChooseCoin <- function(coins) {
  len <- length(coins) / 2
  i <- sample(len, 1)
```

```
  coins[i, ]
}

# Given some coin, we flip it and report whether it's "H" or "T",
# for `heads` and `tails` respectively.
# FlipCoin : Coin -> H | T
FlipCoin <- function(coin) {
  if (coin[2] > runif(1, 0, 1)) "H" else "T"
}

# Tells whether a coin is blue or not.
# IsBlue : Coin -> Boolean
IsBlue <- function(coin) {
  coin[1] == "Blue"
}

# Tests for whether event A would succeed.
# ProbabilityA : Coin -> Boolean
ProbabilityA <- function(coin) {
  IsBlue(coin)
}

# Tests for whether event B would succeed.
# ProbabilityB : Coin -> Boolean
ProbabilityB <- function(coin) {
  FlipCoin(coin) == "H"
}

# We simulate `n` runs of supplied predicate,
# returning only the number that pass the predicate.
# Simulate : (Deck -> Boolean, Int) -> Int
Simulate <- function(p, n) {
  filtered <- Filter(function(x) { p(ChooseCoin(coins)) }, 1:n)
  length(filtered)
}

# The actual probability runner.
# Probability : (Deck -> Boolean) -> {n : [Float], probability : [Float]}
Probability <- function(f) {
  sim.ns <- c(100, 1000, 10000, 100000)
  sims <- sapply(sim.ns, function(n) { Simulate(f, n) / n })

  data.frame(n = sim.ns, probability = sims)
}
```

(a)

This problem is a direct check of whether the coin was blue or not.

```
source("./2.R")

# We simply simulate the `ProbabilityA` function.
probability.a <- Probability(ProbabilityA)
```

(b)

This problem is a direct check of whether the coin was heads or not.

```
source("./2.R")

# We simply simulate the `ProbabilityB` function.
probability.b <- Probability(ProbabilityB)
```

(c)

Things start to get a bit trickier at this point.

```
source("./2.R")

# Since these two events are not independent,
# we have to convert to logic and test it.
# Logically this is A \/ B
probability.a.cup.b <- Probability(function(coin) {
  ProbabilityA(coin) || ProbabilityB(coin)
})
```

(d)

```
source("./2.R")

# Since these two events are not independent,
# we have to convert to logic and test it.
# Logically this is ~A \/ B
probability.a.comp.cup.b <- Probability(function(coin) {
  !ProbabilityA(coin) || ProbabilityB(coin)
})
```

(e)

```
source("./2.R")

# Since these two events are not independent,
# we have to convert to logic and test it.
# Logically this is A /\ B
probability.a.cap.b <- Probability(function(coin) {
  ProbabilityA(coin) && ProbabilityB(coin)
})
```

(f)

```
source("./2.R")

# Since these two events are not independent,
```

```r
# we have to convert to logic and test it.
# Logically this is B -> A
# which is equivalent to ~B \/ A
probability.a.given.b <- Probability(function(coin) {
  !ProbabilityB(coin) || ProbabilityA(coin)
})
```

(g)

```r
source("./2.R")

# Since these two events are not independent,
# we have to convert to logic and test it.
# Logically this is ~B -> A
# which is equivalent to B \/ A
probability.a.given.b.comp <- Probability(function(coin) {
  ProbabilityB(coin) || ProbabilityA(coin)
})
```