Risk Modelling Activities

1. Visit the ‘Introduction to Monte Carlo Simulation in Excel’ page by Winston (see the reading list) and work through the exercises provided.
2. Visit Allen Downey’s Think Bayes 2 pages (See the reading list).

   Download the notebooks.zip file and then try the following exercises (note these can be run in Codio using the Jupyter Notebook, or via your download from the University of Essex Software Hub) Complete the exercises labelled chapter 1 and chapter 2.

2. Bayes' Theorem in Python:

```python
import thinkbayes2
from thinkbayes2 import Pmf

# 'pmf' = probability mass function

# Chapter 2

# Dice prob
pmf = Pmf()
for x in [1,2,3,4,5,6]:
    pmf.Set(x, 1/6)
print("Here are your dice chances:")
print(pmf)

# The cookie problem

pmf = Pmf()
# Prior distribution
pmf.Set('Bowl 1', 0.5)
pmf.Set('Bowl 2', 0.5)

# Update distribution based on new data (likelihood)
pmf.Mult('Bowl 1', 0.75)
pmf.Mult('Bowl 2', 0.5)
# With update, dist. no longer normalized, but because H1 and H2 are mutually
# exclusive and collectively exhaustive, can renormalize:
pmf.Normalize()
#Next, find the posterior distribution based on the above:
print(f"Bowl 1 {pmf.Prob('Bowl 1')}")
print(f"Bowl 2 {pmf.Prob('Bowl 2')}")
# Let's make the above code more general:

# First, define a class
# 'Cookie' maps hypos to their probabilities
class Cookie(Pmf):
    # def __init__ method gives each hypo the same prior probability
    def __init__(self, hypos):
        Pmf.__init__(self)
        for hypo in hypos:
            self.Set(hypo, 1)
        self.Normalize()

    # To update info in light of new data (likelihood)
    # Loops through each hypo in the suite and x's its prob by the likelihood of the data under the hypo
    # Is computed by Likelihood
    def Update(self, data):
        for hypo in self.Values():
            like = self.Likelihood(data, hypo)
            self.Mult(hypo, like)
        self.Normalize()

    # Likelihood calculation
    mixes = {
    'Bowl 1' :dict(vanilla=0.75, chocolate=0.25),
    'Bowl 2' :dict(vanilla=0.5, chocolate=0.5),
    }
    # Uses dictionary 'mixes' to map the name of a bowl to the mix of cookies in the bowl
    def Likelihood(self, data, hypo):
        mix = self.mixes[hypo]
        like = mix[data]
        return like

# Now the novel info:
hypos = ['Bowl 1', 'Bowl 2']
pmf = Cookie(hypos)

# Update with likelihood
pmf.Update('vanilla')

#Can print the posterior prob of each hypo:
for hypo, prob in pmf.Items():
    print(hypo, prob)

# Advantage 1: provides a framework for solving similar problems
# Advantage 2: Generalizes to the case if we draw more cookies from same bowl:
```

```python
dataset = ['vanilla', 'chocolate', 'vanilla']
for data in dataset:
    pmf.Update(data)

for hypo, prob in pmf.Items():
    print(hypo, prob)

# The Monty Hall Problem

# To solve, define a new class:

class Monty(Pmf):

    # Same as Cookie Problem
    def __init__(self, hypos):
        Pmf.__init__(self)
        for hypo in hypos:
            self.Set(hypo, 1)
        self.Normalize()

    # Same as Cookie Problem
    def Update(self, data):
        for hypo in self.Values():
            like = self.Likelihood(data, hypo)
            self.Mult(hypo, like)
        self.Normalize()

    # Not the same as Cookie Problem
    def Likelihood(self, data, hypo):
        if hypo == data:
            return 0
        elif hypo == 'A':
            return 0.5
        else:
            return 1

# Novel info for Monty Hall Problem:

hypos = 'ABC'
pmf = Monty(hypos)

# Calling Update is pretty much the same:
data = 'B'
pmf.Update(data)

# Printing the results is the same:
for hypo, prob in pmf.Items():
    print(hypo, prob)

# Can see which elements of the framework are the same, can encapsulate them in an object:
# A Suite is a Pmf that provides __init__, Update, and Print:

class Suite(Pmf):
    '''Represents a suite of hypotheses and their probabilities.'''

    def __init__(self, hypo=tuple()):
        '''Initializes the distribution'''

    def Update(self, data):
        '''Updates each hypothesis based on the data'''

    def Print(self):
        '''Prints the hypothesis and their probabilities'''

# Can implement through thinkbayes2.py:
# Should write a class that inherits Suite and provides Likelihood
# For example:

from thinkbayes2 import Suite

class Monty(Suite):

    def Likelihood(self, data, hypo):
        if hypo == data:
            return 0
        elif hypo == 'A':
            return 0.5
        else:
            return 1

# Novel info for this class:
suite = Monty('ABC')
suite.Update('B')
suite.Print()

# The M&M Problem

class M_and_M(Suite):

# Likelihood is tricky here.
    # First, Need to encode mixes from 1994 and 1996:
    mix94 = dict(brown = 30,
        yellow = 20,
        red = 20,
        green = 10,
        orange = 10,
        tan = 10)

    mix96 = dict(blue = 24,
        green = 20,
        orange = 16,
        yellow = 14,
        red = 13,
        brown = 13)

    # Then encode the hypotheses:
    hypoA = dict(bag1 = mix94, bag2 = mix96)
    hypoB = dict(bag1 = mix96, bag2 = mix94)

    # Then map from the name of the hypos to the representation
    hypotheses = dict(A = hypoA, B = hypoB)

    # Now, can write Likelihood:
    def Likelihood(self, data, hypo):
        bag, color = data
        mix = self.hypotheses[hypo][bag]
        like = mix[color]
        return like

# Novel code for M&M Problem:

#This creates the suite and updates it:
suite = M_and_M('AB')

suite.Update(('bag1', 'yellow'))
suite.Update(('bag2', 'green'))

suite.Print()

# Exercise 2.1:

#Chapter 3

# Dice problem
# Three step strategy:
# 1. Choose a representation for the hypotheses
# 2. Choose a rep for the data
# 3. Write the likelihood function

class Dice(Suite):
    def Likelihood(self, data, hypo):
        # Must be 0 bc hypo can't be more than sides of die
        if hypo < data:
            return 0
        else:
            return 1.0/hypo

# Hypotheses = the individual die itself by face
suite = Dice([4, 6, 8, 12,20])
```

```
# Data = integers 1 - 20
# 6 is the data

suite.Update(6)
suite.Print()

for roll in [6,8,7,7,5,4]:
    suite.Update(roll)
suite.Print()

# The Locomotive Problem
# " A Railroad numbers its locomotives in order 1...N. One day you see a loco.
# with the number 60. Estimate how many locomotives the railroad has."

# Can break into 2 steps:
# 1. What did we know about N before we saw the data? (This is the prior.)
# 2. For any given value of N, what is the likelihood of seeing the data (a loco.
# with the number 60)?

# For 1. Let's assume that N is equally likely to be any value from 1 to 1000:
# hypos = range(1, 1001)

# Now we need a likelihood function.
# If we assume there's only/only pay attention to Company X
# The likelihood of seeing any of Company X's locomotives is 1/N

# So here's the likelihood function:
class Train(Suite):
    def Likelihood(self, data, hypo):
        if hypo < data:
            return 0
        else:
            return 1.0/hypo

    def Mean(suite):
        total = 0
        for hypo, prob in suite.Items():
            total += hypo * prob
        return total

    def Percentile(pmf, percentage):
        p = percentage / 100.0
        total = 0

        for val, prob in pmf.Items():
            total += prob
            if total >=p:
                return Value


hypos = range(1, 1001)

# Here's the data update:
suite = Train(hypos)
suite.Update(60)

# This results in the guessed number having the highest probability.
# Maybe that's not the goal.

# Let's compute the mean of the posterior distribution (see Train)

# Print the mean using Pmf as ALT
print(suite.Mean())


# What if we want less arbitary assumptions?
# With more data, posterior dist. based on different priors tend to converge

hypos = range(1, 501)
suite = Train(hypos)

for data in [60, 30, 90]:
    suite.Update(data)
print(suite.Mean())
```

```
hypos = range(1, 1001)
suite = Train(hypos)

for data in [60, 30, 90]:
    suite.Update(data)
print(suite.Mean())

hypos = range(1, 2001)
suite = Train(hypos)

for data in [60, 30, 90]:
    suite.Update(data)
print(suite.Mean())


# If more data not available, improve priors by getting more background info
# A company with 1000 trains is not as likely as a company with only 1
# Distribution of company sizes tends to follow a power of law
# If there are 1000 companies with fewer than 10 locomotives, might be 100 comp-
# anies with 100 locos, 10 companies with 1000m and maybe 1 with 10,000

# Can construct a power law prior like this:

class Train(Dice):
    def __init__(self, hypos, alpha=1.0):
        Pmf.__init__(self)
        for hypo in hypos:
            self.Set(hypo, hypo**(-alpha))
        self.Normalize()

# This is the code that constructs the prior:
hypos = range(1, 1001)
suite = Train(hypos)

suite.Update(60)
print(suite.Mean())

hypos = range(1, 501)
suite = Train(hypos)

for data in [60, 30, 90]:
    suite.Update(data)
print(suite.Mean())

hypos = range(1, 1001)
suite = Train(hypos)

for data in [60, 30, 90]:
    suite.Update(data)
print(suite.Mean())

hypos = range(1, 2001)
suite = Train(hypos)

for data in [60, 30, 90]:
    suite.Update(data)
print(suite.Mean())

# Once post. dist. found, useful to summarize the results with a single point
# estimate (or an interval).
# Common to use the mean, median, or value with max likelihood
# Usually report 2 values; 90% chance the unknown value falls between them =
# a CREDIBLE INTERVAL

#Thinkbayes provides a function that computes percentiles:
   # def Percentile(pmf, percentage):
    #    p = percentage / 100.0
     #   total = 0

      #  for val, prob in pmf.Items():
       #     total += prob
        #    if total >=p:
         #       return Value

# Here is the code that uses it:
```

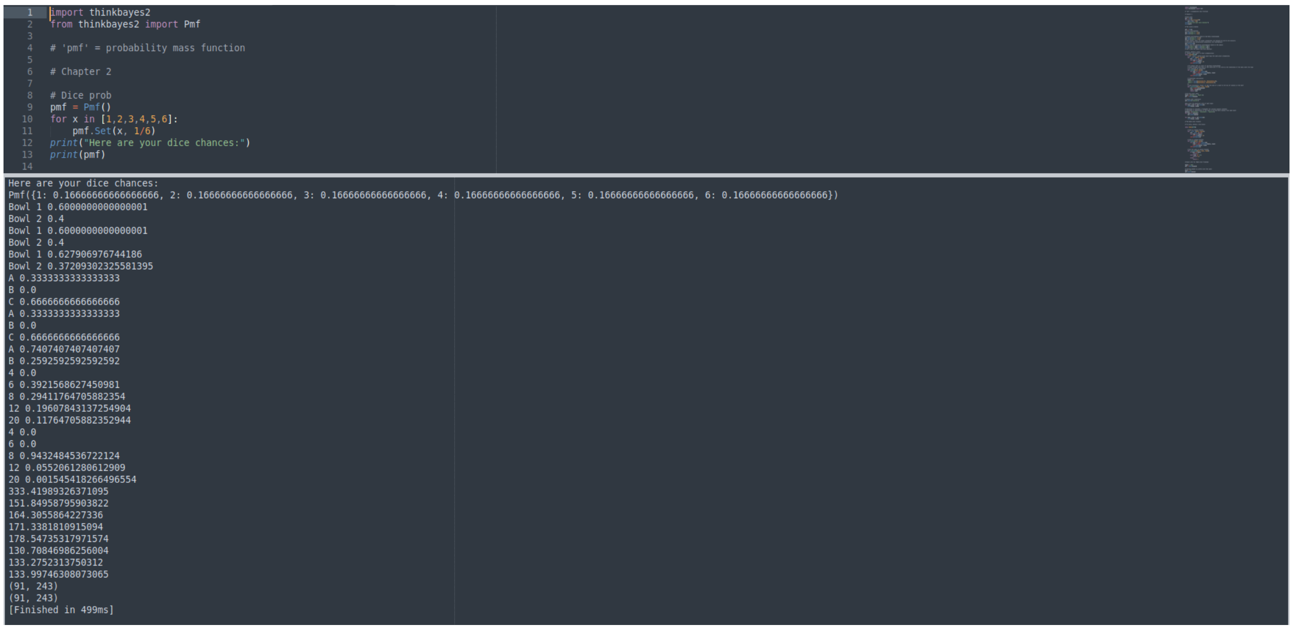
```
interval = suite.Percentile(5), suite.Percentile(95)
print(interval)

# If we need to compute more than a few percentiles, more efficient to use a
# cumulative distribution function (Cdf) rather than Pmf
# Thinkbayes has a Cdf class
#Pmf also provides a method that makes the corresponding Cdf:
cdf = suite.MakeCdf()
interval = cdf.Percentile(5), cdf.Percentile(95)
print(interval)

# Converting Pmf to Cdf takes time proportional to the number of values, len(pmf)
# Also uses "log time" to look up probabilities

# Exercise 3.1
```

Bayes Test code:



```
import thinkbayes2
from thinkbayes2 import Pmf

# 'pmf' = probability mass function

# Chapter 2

# Dice prob
pmf = Pmf()
for x in [1,2,3,4,5,6]:
    pmf.Set(x, 1/6)
print("Here are your dice chances:")
print(pmf)

```

```
Here are your dice chances:
Pmf({1: 0.16666666666666666, 2: 0.16666666666666666, 3: 0.16666666666666666, 4: 0.16666666666666666, 5: 0.16666666666666666, 6: 0.16666666666666666})
Bowl 1 0.6000000000000001
Bowl 2 0.4
Bowl 1 0.6000000000000001
Bowl 2 0.4
Bowl 1 0.627906976744186
Bowl 2 0.37209302325581395
A 0.3333333333333333
B 0.0
C 0.6666666666666666
A 0.3333333333333333
B 0.0
C 0.6666666666666666
A 0.7407407407407407
B 0.2592592592592592
4 0.0
6 0.3921568627450981
8 0.29411764705882354
12 0.19607843137254904
20 0.11764705882352944
4 0.0
6 0.0
8 0.9432484536722124
12 0.0552061280612909
20 0.001545418266496554
333.41989326371095
151.84958795903822
164.3055864227336
171.3381810915094
178.54735317971574
130.70846986256004
133.2752313750312
133.99746308073065
(91, 243)
(91, 243)
[Finished in 499ms]
```

References:

Downey, A. (2016) *Think Bayes*. Sebastopol, CA, USA: O’Reilly.