Homework Week 9

Chris Messer

2022-10-22

Question 12.1

Question 12.1

Describe a situation or problem from your job, everyday life, current events, etc., for which a design of experiments approach would be appropriate.

> At work I frequently work with UI designers. They put together the webpage for a customer interface, and often times there are small design decisions like should we use a light red font over a dark red background, or White over black for the Sign up button? There are many small decisions on on e page that need to be made, but testing all of them would mean we get only a few sample responses for each combination of factors. Using a DoE approach would allow us to test only a few combinations and see which provides the best customer response.

Question 12.2

To determine the value of 10 different yes/no features to the market value of a house (large yard, solar roof, etc.), a real estate agent plans to survey 50 potential buyers, showing a fictitious house with different combinations of features. To reduce the survey size, the agent wants to show just 16 fictitious houses. Use R's FrF2 function (in the FrF2 package) to find a fractional factorial design for this experiment: what set of features should each of the 16 fictitious houses have? Note: the output of FrF2 is "1" (include) or "-1" (don't include) for each feature.

Answer

First, load the package and set the seed

```
library('FrF2')
```

```
## Loading required package: DoE.base
```

```
## Loading required package: grid
```

```
## Loading required package: conf.design
```

```
## Registered S3 method overwritten by 'DoE.base':
##   method           from
##   factorize.factor conf.design
```

```
##
## Attaching package: 'DoE.base'
```

```
## The following objects are masked from 'package:stats':
##
##     aov, lm
```

```
## The following object is masked from 'package:graphics':
##
##     plot.design
```

```
## The following object is masked from 'package:base':
##
##     lengths
```

```
set.seed(1)
```

According to the documentation, we just need to supply two variables here. 1st, we need to supply the number of runs, which in our example is the number of houses we want to show. Then we need to supply the numbers of factors, which in our example is the number of features we are considering.

```
FrF2(nruns = 16,nfactors = 10)
```

```
##     A  B  C  D  E  F  G  H  J  K
## 1  -1 -1 -1  1  1  1  1 -1  1 -1
## 2   1  1 -1 -1  1 -1 -1 -1  1  1
## 3  -1  1  1 -1 -1 -1  1  1 -1  1
## 4  -1 -1 -1 -1  1  1  1 -1 -1  1
## 5   1 -1 -1 -1 -1  1 -1 -1 -1 -1
## 6   1 -1  1  1 -1  1 -1  1 -1 -1
## 7   1  1 -1  1  1 -1 -1  1 -1 -1
## 8  -1  1 -1 -1 -1  1 -1  1  1 -1
## 9  -1 -1  1  1  1 -1 -1 -1 -1  1
## 10 -1 -1  1 -1  1 -1 -1  1  1 -1
## 11 -1  1 -1  1 -1  1 -1 -1 -1  1
## 12  1 -1 -1  1 -1 -1  1  1  1  1
## 13  1 -1  1 -1 -1  1 -1 -1  1  1
## 14 -1  1  1  1 -1 -1  1 -1  1 -1
## 15  1  1  1  1  1  1  1  1  1  1
## 16  1  1  1 -1  1  1  1 -1 -1 -1
## class=design, type= FrF2
```

Now what does this output tell us? Each row is a house we will show, and each column is a feature of that house we should show. For example, the first house we should show should not have features with values of negative 1 in each of the columns.

This simplistic example does ignore one issue in real life though. This example assumes we have 10 factorial number of houses to choose from (3,628,800), i.e. one house for every combination of features, so that we can choose our 16 houses to show. That is a lot of houses!

Question 13.1

For each of the following distributions, give an example of data that you would expect to follow this distribution (besides the examples already discussed in class).

a. Binomial - Asking someone if they like grapes or oranges better. There is an equal liklihood that they choose either option.
b. Geometric - Number of times we had to roll a dice before a 4 is rolled
c. Poisson - Radioactive decay. Poisson is helpful when we know the average time betweenevents, but the exact time between each event is unknown. In radioactive decay, we may know that on average 3 atoms decay every second, but we don't know the exact spacing between each decay
d. Exponential - the amount of time until an earthquake occurs. This is similar to poisson, but instead of the number of occurences in a time frame, exponential distribution deals with the time between occurrences.
e. Weibull - Time until a part of a machine fails. This distribution is similar to geometric distribution, but it models time until an event occurs rather than occurences until an event occurs.

Question 13.2

In this problem you, can simulate a simplified airport security system at a busy airport. Passengers arrive according to a Poisson distribution with λ1 = 5 per minute (i.e., mean interarrival rate ⍁1 = 0.2 minutes) to the ID/boarding-pass check queue, where there are several servers who each have exponential service time with mean rate ⍁2 = 0.75 minutes. [Hint: model them as one block that has more than one resource.] After that, the passengers are assigned to the shortest of the several personal-check queues, where they go through the personal scanner (time is uniformly distributed between 0.5 minutes and 1 minute).

Use the Arena software (PC users) or Python with SimPy (PC or Mac users) to build a simulation of the system, and then vary the number of ID/boarding-pass checkers and personal-check queues to determine how many are needed to keep average wait times below 15 minutes. [If you're using SimPy, or if you have access to a non-student version of Arena, you can use λ1 = 50 to simulate a busier airport.]

> There are a large number of combinations that could work to keep wait times under 15 minutes. For a slow airport, you need at least 3 scanners/4 Checkers or 4 scanners and 3 checkers.
>
> For a fast airport, you need closer to at least 35 scanners and 31 checkers when lambda = 50.

Analysis

I have used SimPy for this analysis. I will include the code at the bottom of this document. The output of that code is two matricies, one for a slow airport (lambda = 5) and one for a busy airport (lambda = 50). Here, in R, I'll analyze those matricies to answer the question asked.

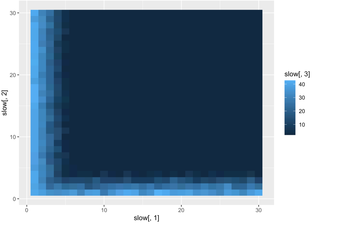
First, lets bring in the data.

```
slow <- read.csv('airport/slow.csv')
fast <- read.csv('airport/fast.csv')
```

I think a heat map would be a good way to visualize this to see what combination of checkers and scanners would work. Lets try it.

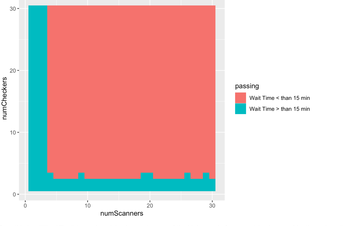
```
library(ggplot2)

ggplot(slow, aes(x=slow[,1], y=slow[,2], fill= slow[,3])) +
  geom_tile()
```



The above is a little difficult to read, so lets make the heat map a binary map, where instead of showing us all values of wait times, it just tells us if that combination of scanners/checkers resulted in a wait time less that 15 minutes.

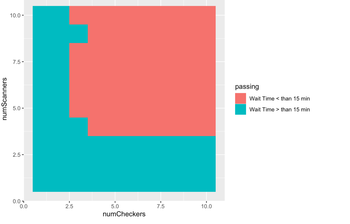
```
slow$passing <- 'Wait Time > than 15 min'
slow[slow$wait_time <= 15,]$passing <- 'Wait Time < than 15 min'
slow$passing <- factor(slow$passing)
ggplot(slow, aes(x=numScanners, y=numCheckers, fill= passing)) +
  geom_tile()
```



Even this is a little difficult to read, so lets chop out some of the higher end of scanner/checker combinations since they all pass.

```
slow.ten <- slow[(slow$numScanners <=10 & slow$numCheckers <= 10),]

ggplot(slow.ten, aes(x=numCheckers, y=numScanners, fill= passing)) +
  geom_tile()
```

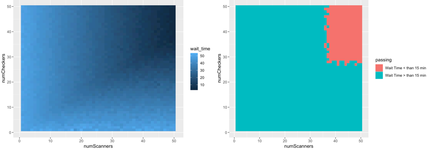


Now we can clearly see what combinations of checkers and scanners result in wait times less than 15 minutes.

Now lets do the same for the busy airport.

```
fast$passing <- 'Wait Time > than 15 min'
fast[fast$wait_time <= 15,]$passing <- 'Wait Time < than 15 min'
fast$passing <- factor(fast$passing)


ggplot(fast, aes(x=numScanners, y=numCheckers, fill= wait_time)) +
  geom_tile()
ggplot(fast, aes(x=numScanners, y=numCheckers, fill= passing)) +
  geom_tile()
```



I really enjoyed this simulation assignment! Some other cool things we could have done:

- We could build a logistic regression model based on our simulation results, and in real life we could bake this into our staffing tool. For example, I as a manager, would like to know if I were to staff the airport with 13 scanners, and 8 checkers on a Wednesday, what is the likelihood that wait times will be less than 15 minutes?
- We could bring in more data to our simulation like different distributions for different days of the week
- We could use real data from prior history at the airport to estimate passenger arrival times, and smooth the data out with HW exponential smoothing, and so much more

Final.py

```
import simpy
import random
import pandas as pd

# Now we need to define our environment. We are wanting to simulate an
# airport, so we will define a class of Airport. If you are new to python,
# a class is helpful when we are wanting to be able to save state changes of
# an object. In this example, we want to have one airport that has
# attributes about it that change over time but are consistent.

# We will also allow the airport to take in some arguments, for things like
# arrival rate, check rate, etc. Most of these are defined by the question
# in the homework. Some of them we will just have to make a guess for the
# sake of our model. In real life we would collect data from an airport and
# use that as our variable values, but for now we will just make a guess. I
# will annotate my code as I go.

class Airport:
    # call the init method - this means as soon as we actually create an
    # airport in our code, it will do all of the things in the init section.
    # Here we will assign it some things an airport will need to have. It
    # will ne need to know things like do I have scanners? How many? How
    # fast are people arriving at me? etc.
    def __init__(self, env, arrRate, checkRate,
                 numCheckers, numScanners, runTime, minScan, maxScan):
        self.env = env
        self.arrRate = arrRate
        self.checkRate = checkRate
        self.numCheckers = numCheckers
        self.numScanners = numScanners
        self.runTime = runTime
        self.minScan = minScan
        self.maxScan = maxScan
        # here we will give it some empty variables to store information in
        # later.
        self.passengerCount = 0
        self.checkCount = 0
        self.arrivals = 0
        self.passenger_list = []

        # And now we actually create those resources we talked about earlier!
        self.checker = simpy.Resource(env, self.numCheckers)
        # since the scanners all have a queue, and the passengers will be
        # looking for the scanner with the shortest line, lets store all of
        # our scanners in a list
        self.scanners = []
        for i in range(self.numScanners):
            self.scanners.append(simpy.Resource(env, 1))

    # Now lets give our resources some actions. Here we are telling the
    # check function in needs to wait according to an exponential
    # distribution, of 1/checkout rate. This is just how the python package
    # interprets it a poisson distribution of a lambda value.

    def check(self, passenger):
        yield self.env.timeout(random.expovariate(1.0 / self.checkRate))
        # random.expovariate(1.0 / self.checkRate)

    # and now we do the same for scanning time, except this time it takes in
    # a normal distribution.
    def scan(self, passenger):
        yield self.env.timeout(random.uniform(self.minScan, self.maxScan))


# Great! now we do the same for passengers. We will create a passenger
# object so it can store information. Esssentially, once the passenger gets
# through the checking and scanning lines, we will ask each one how long it
# took them to get through, by calling the passenger.checkTime attribute.
class Passenger:
    # give the passengers some attributes
    def __init__(self, name, airport):
        self.airport = airport
        self.name = name

        self.arrTime = self.airport.env.now
        self.checkTime = None
        self.airport.env.process(self._get_boarding_pass_checked())
        # print(f'Boarding check took {self.checkTime} for passenger {
        # self.name}')

    # now we have to tell the passenger what to do.
    def _get_boarding_pass_checked(self):
        with self.airport.checker.request() as request:
            tIn = self.airport.env.now  # first record when the passenger
            # starts to get checked
            yield request  # now request a checker, and don't do anything
            # until they get one.
            yield self.airport.env.process(self.airport.check(self.name))
            #now that they found a checker, get checked
            tOut = self.airport.env.now  # record when passenger ends being
            # checked
            self.checkTime = (tOut - tIn)  # find total time for passenger
            # to be checked
            self.airport.checkCount += 1  # record in our airport object
            # that someone has been checked
            self.airport.env.process(self._get_scanned())  # call the
            # scanning process to start

    # this is just a function to help our passenger find the shortest line.
    def _find_shortest_scanner_line(self):
        min_queue = 0
        for i in range(1, self.airport.numScanners):
            if len(self.airport.scanners[i].queue) < len(
                    self.airport.scanners[min_queue].queue):
                min_queue = i
        return min_queue

    # I won't annotate line by line, but this is the same process as the
    # checking process.
    def _get_scanned(self):
        shortest_line = self._find_shortest_scanner_line()
        with self.airport.scanners[shortest_line].request() as request:
            tIn = self.airport.env.now
            yield request
            yield self.airport.env.process(self.airport.scan(self.name))
            tOut = self.airport.env.now
            self.departTime = tOut
            self.scanTime = (tOut - tIn)
            self.totalTime = (self.departTime - self.arrTime)
            self.airport.passengerCount += 1
            self.airport.passenger_list.append(self)
            # print(f'Passenger {self.name} waited {self.scanTime}')


# Now that we have our objects created, we have to open our airport for
# business!

# create a function that opens our airport. We will instruct it on how fast
# passengers arrive, etc.
def open_airport(env,
                 airport,
                 passenger=1):
    p_list = []
    airport.passenger_list.append(Passenger(passenger, airport))

    while True:
        yield env.timeout(random.expovariate(airport.arrRate))
        passenger += 1
        airport.arrivals += 1
        Passenger(passenger, airport)


# Think of this as the manager of the airport function. Someone has to
# actually open the doors-that is what the .run function does at the bottom
# there.
def run_sim(arrRate=5,
            checkRate=.75,
            numCheckers=1,
            numScanners=1,
            runTime=720,
            minScan=.5,
            maxScan=1.5):
    env = simpy.Environment()
    airport = Airport(env,
                      arrRate=arrRate,
                      checkRate=checkRate,
                      numCheckers=numCheckers,
                      numScanners=numScanners,
                      runTime=runTime,
                      minScan=minScan,
                      maxScan=maxScan)

    env.process(open_airport(env, airport))
    env.run(until=100)
    return airport

# Now we want to open the airport for several days to really get a feel for
# what the average times are. So we replicate it 10 times
def replicate(replications, arrRate=5, numCheckers=1, numScanners=1):
    average_time = []
    for i in range(replications):
        airport = run_sim(arrRate=arrRate,
                          numCheckers=numCheckers,
                          numScanners=numScanners)
        wait_times = [p.totalTime for p in airport.passenger_list]
        average = sum(wait_times) / len(wait_times)
        average_time.append(average)
    return sum(average_time) / len(average_time)

# Now, we want to see how varying the number of checkers and scanners
# actually affects the wait times of customers. So we will loop over that as
# well
def simSlow():
    df = []
    for numScanners in range(1, 31):
        for numCheckers in range(1, 31):
            average_time = replicate(1,
                                     arrRate=5,
                                     numCheckers=numCheckers,
                                     numScanners=numScanners)
            speed = {'numScanners': numScanners,
                     'numCheckers': numCheckers,
                     'wait_time': average_time}
            df.append(speed)
            #print(speed)
    return pd.DataFrame(df)

#Now do the same for a busy airport by increasing the lambda value to 50
def simBusy():
    df = []
    for numScanners in range(1, 51):
        for numCheckers in range(1, 51):
            average_time = replicate(1,
                                     arrRate=50,
                                     numCheckers=numCheckers,
                                     numScanners=numScanners)
            speed = {'numScanners': numScanners,
                     'numCheckers': numCheckers,
                     'wait_time': average_time}
            df.append(speed)
            #print(speed)
    return pd.DataFrame(df)

#finally, save the results to a csv file.
slow = simSlow()
slow.to_csv('slow.csv', index=False)
fast = simBusy()
fast.to_csv('fast.csv', index=False)
print('done')
```