SIT718 Real World Analytics
Prac. 05.

Generalised Mean / Power Mean

Generalised Mean is also know as Power Mean or Holder Mean. The power mean is a way of expressing most of the means into one formula,

$$g_\lambda(x_1, x_2 \ldots, x_n) = \left(\frac{1}{n}\sum_{i=1}^{n} x_i^\lambda\right)^{1/\lambda}, \qquad \lambda \neq 0$$

By changing the λ to the values shown below, we will the formula for one of the common means,

λ = -1 , Harmonic Mean

λ = 0 , Geometric Mean

λ = 1 , Arithmetic Mean

λ = 2 , Quadratic Mean

In fact λ can be any non-zero number from -∞ to ∞, other real and hypothetical means calculations are possible.

$$M_{-\infty}(x_1, \ldots, x_n) = \lim_{p \to -\infty} M_p(x_1, \ldots, x_n) = \min\{x_1, \ldots, x_n\}$$ minimum

$$M_{-1}(x_1, \ldots, x_n) = \frac{n}{\frac{1}{x_1} + \cdots + \frac{1}{x_n}}$$ harmonic mean

$$M_0(x_1, \ldots, x_n) = \lim_{p \to 0} M_p(x_1, \ldots, x_n) = \sqrt[n]{x_1 \cdot \cdots \cdot x_n}$$ geometric mean

$$M_1(x_1, \ldots, x_n) = \frac{x_1 + \cdots + x_n}{n}$$ arithmetic mean

$$M_2(x_1, \ldots, x_n) = \sqrt{\frac{x_1^2 + \cdots + x_n^2}{n}}$$ root mean square or quadratic mean[4][5]

$$M_3(x_1, \ldots, x_n) = \sqrt[3]{\frac{x_1^3 + \cdots + x_n^3}{n}}$$ cubic mean

$$M_{+\infty}(x_1, \ldots, x_n) = \lim_{p \to \infty} M_p(x_1, \ldots, x_n) = \max\{x_1, \ldots, x_n\}$$ maximum

Defining power means

We now have all the tools necessary to define our power means! This time we will have two inputs to the function, 'x', which will be a vector of inputs and 'p', which will be the power used.

```
PM <- function(x,p)        # 1. pre-defining the function inputs
{ (mean(x^p))^(1/p)        # 2. our calculation which will also
}                          #    be the output
```

However this function will not work if $p = 0$. So we need to create a special case. For this, we will use the `if()` function.

```
PM <- function(x,p)        # 1. pre-defining the function inputs
{ if(p == 0)               # 2. condition for 'if' statement
{ prod(x)^(1/length(x)) }  # 3. what to do when (p==0) is TRUE
else { (mean(x^p))^(1/p)
}
}                          # 4. what to do when (p==0) is FALSE
```

Note here that for '=' conditions, we use a double equals sign '==='. So the possible conditions we can use inside the `if()` brackets are '==', '<', '>', '<=' and '>='.

R Exercise 7 *Define the power mean as a function in* R *and check the following.*

Input	*Expected Output*
`PM(c(3,2,7),2)`	*4.546061*
`PM(c(1,0,7),0)`	*0*
`PM(c(0.28,0.4,0.47),-557)`	*0.2805528*
`PM(c(0.28,0.4,0.47),-558)`	*Inf*

Note that in the last case, 'Inf' should not actually be the result. It's just that at this point, the system cannot tell the difference between one of the transformed inputs and infinity, i.e. if you compare `0.28^(-557)` and `0.28^(-558)`, the latter will be 'Inf' and then all of the operations become absorbed by this value.

We now need to incorporate weights into our previously obtained functions for means. Our new functions will have multiple arguments, some of which will be vectors, and some of which will be single values.

Weighted arithmetic means

Incorporating weights into the arithmetic mean is quite simple, since you will recall that when we multiply two vectors of the same length, the result is the component-wise products (i.e. $\langle x_1, x_2, x_3\rangle * \langle y_1, y_2, y_3\rangle = \langle x_1y_1, x_2y_2, x_3y_3\rangle$). Once we have this product, taking the sum gives us exactly what we need.

```
WAM <- function(x,w) {
sum(w*x)
}
```

We can ensure that our weighting vector $\mathbf{w}$ is normalised, if we need to, by adding an additional line.

```
WAM <- function(x,w) {
w <- w/sum(w)
sum(w*x)
}
```

Weighted power means

This is not particularly any more difficult than for the mean. Recall from the previous topic that we had

```
PM <- function(x,p)        # 1. pre-defining the function inputs
{ if(p == 0)               # 2. condition for 'if' statement
{ prod(x)^(1/length(x)) }  # 3. what to do when (p==0) is TRUE

else {(mean(x^p))^(1/p)}   # 4. what to do when (p==0) is FALSE
}
```

Rather than use the mean, we can use the multiple of our w and input vector, which we raise to the power p.

```
WPM <- function(x,w,p) {  # 1. pre-defining the inputs
if(p == 0) {prod(x^w)}    # 2. weighted geometric mean if p=0
else {sum(w*(x^p))^(1/p)} # 3. our calculation which will also }
#    be the output
```

Default values for functions

We can adapt our power mean function so that we can more easily use it for the geometric mean and arithmetic mean. We do this by including default values, which can be included in the parameter specification. For the default value of p we can use 1, while as a default value for a weighting vector, we can use a vector the same length as $\mathbf{x}$ whose inputs are all $1/n$. So instead of `function(x,w,p)`, we can use

```
function(x,w=array(1/length(x),length(x)),p=1)
```

which means that now we can just input the value for $\mathbf{x}$, and are not required to include a weighting vector or value for p in order to obtain an output. If we do include a weighting vector, but not a value for p, then we have a weighted arithmetic mean. If we want to leave the weighting vector as is, then we can still change between different kinds of power means, however when we input our function, we need to say that we're using p, i.e.

```
WPM(c(0.6,0.7,0.8,0.9),p=2)
```

will evaluate the quadratic mean (unweighted) of $\mathbf{x} = \langle 0.6, 0.7, 0.8, 0.9 \rangle$.

R Exercise 1 *Enter in the weighted power mean as a function and verify that you can obtain correct results for different values of p, $\mathbf{w}$, and using the defaults.*

Weighted median

Programming a weighted median can be a little more complicated than applying weights to other functions, since we need to find the ordered value and check the sums of weights above and below. The following gives one way. The weighting vectors and input vectors are sorted according to $\mathbf{x}$. For $\mathbf{x}$, this can be achieved using the sort function, however for $\mathbf{w}$, we use `order(x)` as it's indices. Recall that `order(x)` gives the indices corresponding with the sorting of $\mathbf{x}$ into increasing order. By listing these inside the square brackets, $\mathbf{w}$ will be reordered accordingly, e.g. if the ordering of the input vector is $2, 3, 1$ (i.e. the lowest input is the second argument, the next lowest is the third and the highest is the first), then the weights will now be given in the order w_2, w_3, w_1.

```
Wmed <- function(x,w) {              # 1. function inputs
w <- w/sum(w)                        # 2. normalise weights
n <- length(x)
w <- w[order(x)]                     # 3. sort weights by inputs
x <- sort(x)                         # 4. sort inputs
out <- x[1]                          # 5. set starting value
for(i in 1:(n-1)) {                  # 6. for each i, we check
if(sum(w[1:i]) < 0.5) out <- x[i+1] #    to see if the weights
}                                    #    are still below 50%,
out                                #    once they're not, we stop
}                                  # updating the output
```

The above calculates the lower weighted median. Basically it keeps replacing the final output with $x_{(i)}$ until the weights are above 0.5, at which point it stops replacing the output. This is not overly efficient, however for the moment it is fine for us.

R Exercise 2 *Verify the weighted median examples given in the topic chapter using the above coded function.*

Borda counts

Recall that we were able to use the `order()` and `rank()` operations to obtain rank-based scores. To define a function that calculates the Borda count, we need to know what form the data will come in. If we have an input vector $\mathbf{x}$ and each x_i indicates the number or proportion of votes in that position, then we can use a weighted arithmetic mean as we did previously. However if we have input scores that we want to transform into a Borda count scoring system, we will need to do a little more.

If we have rows (representing each candidate or alternative) we can transform the scores they have received from each judge or source (corresponding with the columns) into ranks using our assign function.

We need to replace all columns in the matrix/array with the rank scores. The following will replace the first column with the corresponding rankings.

```
x[,1] <- rank(x[,1])
```

Once this is done, we could then sum the scores in each row (since `rank()` gives the highest value to the highest score, however if we want to use alternative weightings, we need to replace these ranks with those scores.

We can do this using a special indexing command. For a vector, or matrix, `x[x==7]` calls all entries in `x` that are equal to 7. This means we can use the following to convert all rankings of 3 to 10 points.

```
x[x==3] <- 10
```

In order to convert all the rankings, we need to be careful not to assign a score that is equal to one of the unconverted rankings. For example, if we had rankings $1, 2$ and 3 and we wanted to assign the respective scores $2, 4, 10$, we should do this in the following order.

```
x[x==3] <- 10
x[x==2] <- 4
x[x==1] <- 2
```

Otherwise, after assigning `x[x==1] <- 2` in the next step both the 2 and original 1 rankings will be changed to 4.

In the context of group decision making, the following assumes `x` is a matrix where each row is a candidate and each column is the set of scores given by one judge. If all the values in the weighting vector `w` are less than 1, there should be no confusion when converting the integer ranks.

```
Borda <- function(x,w) {                  # 1. function inputs
  total.scores <- array(0,nrow(x))        # 2. vector to hold final scores
  for(j in 1:ncol(x)) {                   # 3. convert each column to ranks
    x[,j] <- rank(-x[,j])                 # Using negation to get the descending order
  }
  y <- x-x                                # initiate score matrix
  for(i in 1:length(w)) {                 # 4. convert all of the ranks to the w scores
    y[x==i] <- w[i]
  }
  for(i in 1:nrow(y)) {                   # 5. add the scores for each candidate
    total.scores[i] <- sum(y[i,])
  }
  total.scores
}

## Example for Weekly Resource 5.9 Borda Count

x=rbind(c(9,6,4),c(7,7,6),c(4,8,8))
w=c(2,1,0)
Borda(x,w)
```

OWA operator

Recall that we defined our weighted arithmetic mean using the following.

```
WAM <- function(x,w) {sum(w*x)}
```

The OWA function should be very similar, however first we need to re-order the vector **x** from lowest to highest. This is achieved using the `sort()` function, which reorders the vector non-decreasingly[2].

```
OWA <- function(x,w) {
sum(w*sort(x))
}
```

We can make sure our weighting vector is normalized by including the line '`w <- w/sum(w)`' at the beginning. This is also a situation where we may want to include a default value for the weighting vector. As with the weighted power mean, we could update the above to

```
OWA <- function(x,w=array(1/length(x),length(x)) ) {
  w <- w/sum(w)
  sum(w*sort(x))
}
```

The special cases of the OWA, the trimmed mean and Winsorized mean can also be defined. In fact, the trimmed mean can be implemented in `R` using the mean function with an additional argument `trim`. Recall that the trimmed mean was defined in terms of a parameter 'h', which was an even number determining how many values are removed from calculation. However sometimes it makes more sense to define the trimmed mean in terms of the percentage that is removed. For example, with 10 data, a value of `trim=0.1` would remove the highest and lowest entry (10% from the top and 10% from the bottom) and for an input vector **x** we have

```
mean(x,trim=0.1)
```

We can calculate the Winsorized mean by replacing the upper and lower h% of entries in the vector which is equivalent to allocating higher weights to the upper and lower non-trimmed inputs. We first determine how many values to replace (we will assume h is given as a percentage here too and then find hn). We will require the floor function for this step in case hn does not result in an integer value. As a default, we set the h percentage to zero.

```
Wins.mean <- function(x,h=0) {     # 1. pre-defining the inputs
  n <- length(x)                     # 2. store the length of x
  repl <- floor(h*n)                 # 3. how many data to replace
```

[2] To define the OWA in terms of a non-increasing permutation, i.e. if we want the first weight to be allocated to the highest input, we can use `decreasing = TRUE` as an optional argument when we sort, i.e. `sort(x,decreasing = TRUE)`.

```
  x <- sort(x)                          # 4. sort x
  x[1:repl] <- x[repl+1]                # 5. replace lower values
  x[(n-repl+1):n] <- x[n-repl]          # 6. replace upper values
  mean(x)                               # 7. calculate the mean
}
```

With both of these functions, we could set them up so that we input the `rmv` and `repl` values instead of h. Either way is fine, as long as it is taken into account when using the functions to calculate outputs.

R Exercise 15 *Enter in the OWA, trimmed mean and Winsorized mean and verify that you get the same results for* $\mathbf{x} = \langle 0.3, 0.6, 0.8, 0.3, 0.4, 0.5 \rangle$ *when they are equivalent, e.g. for the trimmed mean and* $h = 0.17$*, the OWA weights should be* $\mathbf{w} = \langle 0, 0.25, 0.25, 0.25, 0.25, 0 \rangle$*, and for the Winsorized mean, the outputs should be the same when the OWA weights are* $\mathbf{w} = \langle 0, 1/3, 1/6, 1/6, 1/3, 0 \rangle$.

Choquet integral (Optional)

The Choquet integral is another function that could be coded in a number of ways. Our aim is to have a function that takes an input vector $\mathbf{x}$, a fuzzy measure v and then produces the output. However note here that the fuzzy measure v needs be entered as an argument, and so we will need to decide on how this is to be done.

The most straightforward way is to code v as a vector, and so we just need to decide on the ordering (because there is no linear order on subsets). The two most common orderings are either by cardinality, so that the order of the set values would be,

$$v(\emptyset), v(\{1\}), v(\{2\}), \ldots, v(\{n\}), v(\{1,2\}), v(\{1,3\}), \ldots, v(\{1,2,\ldots,n\}),$$

or another type of ordering is called 'binary ordering'. Binary ordering uses the form of a binary number to decide which elements are in the set at a given position. For example, the first 6 binary numbers are 000, 001, 010, 011, 100, and 101. If the digit on the far right is a 1, then the 1st element/variable is in the subset. If it is a zero, then 1 is not in the subset. If the digit second from the right is a 1, then the 2nd element/variable is in the set and so on. So corresponding with 000 is $v(\emptyset)$, corresponding with 001 is $v(\{1\})$, corresponding with 011 is $v(\{1,2\})$, and corresponding with 101 is $v(\{1,3\})$. This means the order of the values would be,

$$v(\emptyset), v(\{1\}), v(\{2\}), v(\{1,2\}), v(\{3\}), v(\{1,3\}), v(\{2,3\}), v(\{1,2,3\}), v(\{4\}), \ldots$$

Although this might seem more complicated, using this ordering actually makes it much easier for us to define our function, because the position of a set in the vector can be directly calculated and will not change when n changes.

The following makes use of the `order()` function.

```
Choquet <- function(x,v) { # 1. pre-defining the inputs
  n <- length(x)             # 2. store the length of x
  w <- array(0,n)            # 3. create an empty weight vector
  for(i in 1:(n-1)) {        # 4. define weights based on order
    v1 <- v[sum(2^(order(x)[i:n]-1))]     #
                           # 4i. v1 is f-measure of set of all
                           #     elements greater or equal to
                           #     i-th smallest input.
    v2 <- v[sum(2^(order(x)[(i+1):n]-1))]  #
                           # 4ii. v2 is same as v1 except
                           #     without i-th smallest
    w[i] <-  v1 - v2           # 4iii. subtract to obtain w[i]
    }                       #
  w[n] <- 1- sum(w)          # 4iv. final weight leftover
  x <- sort(x)               # 5. sort our vector
  sum(w*x)                   # 6. calculate as we would WAM
}
```

The steps for 4i to 4iii could be combined into a single step,

```
w[i] <- v[sum(2^(order(x)[i:n]-1))] - v[sum(2^(order(x)[(i+1):n]-1))]
```

Recall that the `order()` function tells us the indices in **x** corresponding with the lowest to highest inputs. So a result of $\langle 3,1,2,4\rangle$ would mean that x_3 is the lowest, x_1 is the second lowest, x_2 is the second highest and x_4 is the highest. Then taking `order(x)[3:4]` would give us the indices of the two highest inputs (`2  4`). Subtracting 1 and raising 2 to the power of this vector gives the corresponding position of our subset. So in the case of `2 4`, subtracting 1 makes it `1 3` and 2 to the power of these values gives 2 and 8. Summing these together gives 10, which in binary is 1010, i.e. the 4th and 2nd element (don't forget that we read the binary number from right to left). So the fuzzy measure value associated with this subset should have been stored in the 10th position.

The last weight is just found by subtracting the previously defined weights from 1 rather than redefining v2 so that it doesn't try to take `order(x)[(n+1):n]` when what we actually want there is the empty set.

R Exercise 16 *Enter in the Choquet integral function (you can copy and paste if it is easier). Verify the examples used throughout the topic. In the Example from 4.4.4, the fuzzy measure shown would be encoded as*

```
c(0.3, 0.4, 0.4, 0.1, 0.9, 0.5, 1).
```

We don't need to input the empty set (since we don't use it in calculation) and this allows the value of `v[1]` *to correspond with 001,* `v[2]` *to correspond with 002 and so on.*

Orness

Compared with the Choquet integral, the orness calculations for both the OWA and the Choquet integral should seem relatively easy. In the case of the Choquet integral, we will need to use the factorial function, which is calculated using `factorial()` (and not '!').

Firstly for the OWA, we multiply each weight by $\frac{i-1}{n-1}$. The vector of these values can be found just using our vector subtraction and division operations

```
orness <- function(w) {    # 1. the input is a weighting vector
n <- length(w)             # 2. store the length of w
sum(w*(1:n-1)/(n-1))       # 3. orness calculation
}
```

The orness calculation for the Choquet integral requires us to work out the cardinality of the corresponding subsets of the fuzzy measure. To do this, we will use a special function that converts any integer to its binary representation. This is `intToBits()`. The output of this function is a string of double digit values, e.g. `00 01 01 00` ... where a 01 corresponds with a 1 in the binary place value representation. This is done in the reverse order to how we normally write binary numbers. For example, 10011 would be the same as `01 01 00 00 01`. We need to invoke the `as.numeric()` function on this vector in order to sum the values because, as a default, bits are not treated like numbers. This representation makes an operation like finding the cardinality based on the position of the subset very easy.

```
orness.v <- function(v) {    # 1. the input is a fuzzy measure
  n <- log(length(v)+1,2)      # 2. calculates n based on |v|
  m <- array(0,length(v))      # 3. empty array for multipliers
  for(i in 1:(length(v)-1)) { # 4. S is the cardinality of
    S <- sum(as.numeric(intToBits(i))) #    of the subset at v[i]
    m[i] <- factorial(n-S)*factorial(S)/factorial(n)  #
    }                                 #
  sum(v*m)/(n-1)                 # 5. orness calculation
}
```

R Exercise 17 *Verify the following orness calculations (assuming the input is associated with an OWA function).*

Input (`orness(...)`)	*Expected output*
`c(1,0,0,0)`	*0*
`c(0,0,0,0,1)`	*1*
`c(0,0.2,0.6,0.2,0)`	*0.5*
`((1:4)/4)^2 - ((0:3)/4)^2`	*0.7083333*

R Exercise 18 *Verify the following orness calculations (assuming the input is a fuzzy measure associated with a Choquet integral).*

Input (`orness.v(...)`)	*Expected output*
`c(0,0,0,0,0,0,1)`	*1*
`c(1,1,1,1,1,1,1)`	*0*
`c(0.3,0.1,0.4,0.6,0.9,0.7,1)`	*0.5*
`c(0.3,0.4,0.4,0.1,0.9,0.5,1)`	*0.4333333*

Shapley values (Optional)

For determining the Shapley value from a fuzzy measure, we will once again use the `intToBits()` function in order to calculate the cardinality. We also use it to determine whether or not a variable is in the corresponding subset.

```
shapley <- function(v) {    # 1. the input is a fuzzy measure
  n <- log(length(v)+1,2)     # 2. calculates n based on |v|
  shap <- array(0,n)          # 3. empty array for Shapley values
  for(i in 1:n) {             # 4. Shapley index calculation
    shap[i] <- v[2^(i-1)]*factorial(n-1)/factorial(n) #
                                # 4i.  empty set term
    for(s in 1:length(v)) {     # 4ii. all other terms
      if(as.numeric(intToBits(s))[i] == 0) {  #
                                # 4iii.if i is not in set s
        S <- sum(as.numeric(intToBits(s)))    #
                                # 4iv. S is cardinality of s
        m <- (factorial(n-S-1)*factorial(S)/factorial(n))  #
                                # 4v. calculate multiplier
        vSi <- v[s+2^(i-1)]       # 4vi. f-measure of s and i
        vS <- v[s]                # 4vii. f-measure of s
        shap[i]<-shap[i]+m*(vSi-vS) # 4viii. add term
      }                         #
    }                           #
  }                             #
shap                            # 5. return shapley indices
}                               #    vector as output
```