SIT718 Real World Analytics
Prac. 4.

This week we will be working on transforming variables and how to implement transformations using R.

Transforming variables

Simple transformations of variables can be achieved using operations that we have already learned. We can either choose to simply replace our values with the transformed ones or we can create a new matrix.

Let's first load our volleball data and then make a copy which we will store as `original`.

```
V <- read.table("volley.txt")
original <- V
```

The second of these lines simply copies the table "V" and assigns the copy to "original". The following command replaces the *Sprint* variable with transformed values according to our negation function.

```
V[,1] <- 51.24 - V[,1]
```

We've used `V[,1]` to access the whole first column and when we replace it with `51.24 - V[,1]`, each entry in the column is subtracted from 51.24. We can then transform it to the unit interval with the following

```
V[,1] <- (V[,1]-min(V[,1]))/(max(V[,1])-min(V[,1]))
```

Be careful with brackets and make sure your operation is working correctly. Sometimes it's good to manually check the first few values that it is working.

Now let's transform the height variable using standardisation. For this, we will use the `sd()` function to calculate the standard deviation.

```
V[,2] <- (V[,2]-mean(V[,2]))/sd(V[,2])
```

R Exercise 5 *Use the linear feature scaling technique to get this column and the remaining columns, 3 and 4, to range between 0 and 1.*

Rank-based scores

There are three functions in R that can help us with situations where we are interested in the order of arguments in a vector. These are `sort()`, `order()` and `rank()`.

The `sort()` function re-orders a vector into increasing (or non-decreasing) order. So with the vector $\langle 1, 6, 2, 3 \rangle$,

```
sort(c(1,6,2,3))
```

would have an output of $1\ 2\ 3\ 6$.

Order, on the other hand, tells us the indices from highest to lowest.

```
order(c(1,6,2,3))
```

would have the output $1\ 3\ 4\ 2$ because the ordering is $x_1 < x_3 < x_4 < x_2$. If there are ties then the `order()` function will output then sort them according to the index, i.e. $\langle 3, 2, 3 \rangle$ would be sorted $2\ 1\ 3$ and not $2\ 3\ 1$. With both of these functions, we can change to descending order by adding the additional argument `decreasing = TRUE`, i.e.

```
order(c(1,6,2,3), decreasing = TRUE)
```

will produce the output $2\ 4\ 3\ 1$.
Rank tells us the relative ranking of the variables. So

```
rank(c(1,6,2,3))
```

will have an output of $1\ 4\ 2\ 3$ because the 6 is ranked fourth, the 2 is ranked second etc. The decreasing option is not available for `rank()`, however by using a negative in front of the input vector the opposite ranking will be obtained, i.e.

```
rank(-c(1,6,2,3))
```

would be $4\ 1\ 3\ 2$.

The most useful function for us in order to convert our *Sprint* times to rank-scores would hence be the `rank()` function. The following ranks the times and then scales them to the unit interval (we get our original times back first).

```
V[,1] <- original[,1]
V[,1] <- (length(V[,1]) - rank(V[,1]))/(length(V[,1])-1)
```

In this case we used `length(V[,1]) - rank(V[,1])` because the lowest value is given the rank 1 but we want it to have the highest score.

Using `if()` for cases

Using the `if()` function requires a careful consideration of the sequence and logic of our function. Let's first consider an example of a piecewise function for values between 0 and 1 that has its join at $(0.5, 0.7)$. So if the input is 0.5, then the output is 0.7. If the input is below 0.5, then it gets increased at the same ratio, and if it is above 0.5, then the ratio of increase drops off so that it still has the output of 1 if the input is 1.

We express the function as an equation in the following way

$$f(t) = \begin{cases} 0.7\frac{t}{0.5}, & 0 \leq t < 0.5 \\ 0.7 + 0.3\frac{t-0.5}{0.5}, & 0.5 \leq t \leq 1. \end{cases}$$

As a function in `R`, we need to create a clause that transforms it using the first equation if the value is less then 0.5, and using the second equation if it is above 0.5. There are a few different ways to do this. The easiest is to use `if(...) {...} else {...}`. Inside the `if()` brackets, we have something like `t < 0.5` or `t >= 0.5` (the latter means greater than or equal to 0.5). Then in the first case brackets $\{...\}$, we tell the function what to do if the `if()` statement is true, and the second case brackets tells the function what to do otherwise. The following programs our piecewise function above.

```
pw.function <- function(t) {
if(t < 0.5) {0.7*t/0.5} else {0.7+0.3*(t-0.5)/0.5}
}
```

R Exercise 6 *Enter in the function and try out a few entries to see that it makes sense and is working correctly.*

We can repeat this process in nested form to define more cases. In the following we interpolate the points $(0.5, 0.7)$ and $(0.8, 0.9)$. In this case our intervals in terms of the input cases are $[0, 0.5]$, $[0.5, 0.8]$ and $[0.8, 1]$ respectively.

```
pw.function.2 <- function(t) {
if(t < 0.5) {0.7*t/0.5}
else {if(t <0.8) {0.7+0.2*(t-0.5)/0.3}
else {0.9+0.1*(t-0.8)/0.2} }
}
```

Another way to have a look at our variables and make sure our functions are working correctly is to plot them.