Using splitters

Ludvig Renbo Olsen
Cognitive Science, Aarhus University
mail@ludvigolsen.dk | http://ludvigolsen.dk

10/29/2016

Contents

1 Including splitters.R in your session

```
source('splitters.R')
```

2 General information

splitters is a set of functions for easy grouping or splitting of dataframes or vectors. They work by dividing the data up into windows, e.g. 111222333, using different choosable methods.

There are 3 main functions:

group_factor()
This function returns a factor with window numbers.
This can be used to subset, aggregate, group_by, etc.

group()
This function returns the given data as a dataframe with the mentioned grouping factor. The dataframe is grouped by the grouping factor for easy use with dplyr's pipelines.

splt()
This function splits the given data (dataframe or vector) into the specified windows and returns them in a list.

There are currently 6 methods for windowing the data.

3 Methods

It is possible to create windows based on window size, step size or number of windows. These can be given as whole number or percentage. Here we will take a look at the different methods.

3.1 Method: 'greedy'

'greedy' uses window **size** for splitting the data.
Greedy means that each window grabs as many elements as possible (up to size), meaning that there might be less elements available to the last window, but that all other windows than the last are guaranteed to have the size specified.

Example

> We have a vector with 57 values. We want to have window sizes of 10.
>
> The greedy splitter will return windows with this many values in them:
> 10, 10, 10, 10, 10, 7

By setting **force_equal** to TRUE, we discard the last window if it contains fewer values than the other windows.

Example

We have a vector with 57 values. We want to have window sizes of 10.

The greedy splitter with force_equal set to TRUE will return windows with this many values in them:
10, 10, 10, 10, 10

meaning that 7 values have been discarded.

3.2 Method: 'n_dist' (Default)

'n_dist' uses a specified number of windows to split the data.
First it creates equal groups as large as possible. Then, if there are any excess data points, it distributes them across the windows.

Example

We have a vector with 57 values. We want to get back 5 windows.

'n_dist' with default settings would return windows with this many values in them:

11, 11, 12, 11, 12

By setting **force_equal** to TRUE, 'n_dist' will create the largest possible, equally sized windows by discarding excess data elements.

Example

'n_dist' with **force_equal** set to TRUE would return windows with this many values in them:

11, 11, 11, 11, 11

meaning that 2 values have been discarded.

3.3 Method: 'n_fill'

'n_fill' uses a specified number of windows to split the data.
First it creates equal groups as large as possible. Then, if there are any excess data points, it places them in the first windows.
By setting **descending** to TRUE, it would be the last windows though.

Example

We have a vector with 57 values. We want to get back 5 windows.

'n_fill' with default settings would return windows with this many values in them:

12, 12, 11, 11, 11

By setting **force_equal** to TRUE, 'n_fill' will create the largest possible, equally sized windows by discarding excess data elements.

Example

'n_fill' with **force_equal** set to TRUE would return windows with this many values in them:

11, 11, 11, 11, 11

meaning that 2 values have been discarded.

3.4 Method: 'n_last'

'n_last' uses a specified number of windows to split the data.

With *default settings*, it tries to make the windows as equal as possible, but notice that the last window might contain fewer or more elements, if the length of the data is not divisible with the number of windows. All, but the last, windows are guaranteed to contain the same number of elements.

Example

We have a vector with 57 values. We want to get back 5 windows.

'n_last' with default settings would return windows with this many values in them:

11, 11, 11, 11, 13

By setting **force_equal** to TRUE, 'n_last' will create the largest possible, equally sized windows by discarding excess data elements.

Example

'n_last' with **force_equal** set to TRUE would return windows with this many values in them:

11, 11, 11, 11, 11

meaning that 2 values have been discarded.

Notice that 'n_last' will always return the given number of windows. It will never return a window with zero elements. For some situations that means that the last window will contain a lot of elements. Asked to split a vector with 57 elements into 20 windows, the first 19 windows will contain 2 elements, while the last window will itself contain 19 elements. Had we instead asked it to split the vector into 19 windows, we would have had 3 elements in all windows.

3.5 Method: 'n_rand'

'n_fill' uses a specified number of windows to split the data.
First it creates equal groups as large as possible. Then, if there are any excess data points, it places them randomly in the windows.
N.B.: It only places one extra element per window.

Example

We have a vector with 57 values. We want to get back 5 windows.

'n_rand' with default settings **could** return windows with this many values in them:

12, 11, 11, 11, 12

By setting **force_equal** to TRUE, 'n_rand' will create the largest possible, equally sized windows by discarding excess data elements.

Example

'n_rand' with **force_equal** set to TRUE would return windows with this many values in them:

11, 11, 11, 11, 11

meaning that 2 values have been discarded.

3.6 Method: 'staircase'

'staircase' uses step_size to split the data.
For each window, the window size will be step size multiplied with the window index.

Example

We have a vector with 57 values. We specify a step size of 5.

'staircase' with default settings would return windows with this many values in them:

5, 10, 15, 20, 7

By setting **force_equal** to TRUE, 'staircase' will discard the last group if it does not contain the expected values (step size multiplied by window index).

Example

'staircase' with **force_equal** set to TRUE would return windows with this many values in them:

5, 10, 15, 20

meaning that 7 values have been discarded.

4 Arguments

4.1 data

Type: dataframe or vector

The data to process.

4.2 n

Type: integer or numeric

n represents either window size, step size or number of windows, depending on which method is specified.
n can be given as a **whole number** (n > 1) or as **percentage** (0 < n < 1)

4.3 force_equal

Type: logical (TRUE or FALSE)

If you need windows with the exact same size, set force_equal to TRUE.
Implementation is different in the different methods. Read more in their sections above.
Be aware that this setting discards excess datapoints!

4.4 allow_zero

Type: logical (TRUE or FALSE)

If you set n to 0, you get an error.
If you don't want this behavior, you can set allow_zero to TRUE, and (depending on the function) you will get the following output:

group_factor() will return the factor with NAs instead of numbers. It will be the same length as expected.

group() will return the expected dataframe with NAs instead of a grouping factor.

splt() functions will return the given data (dataframe or vector) in the same list format as if it had been split.

4.5 descending

Type: logical (TRUE or FALSE)

In methods like 'n_fill' where it makes sense to change the direction of the method, you can use this argument.
In 'n_fill' it fills up the excess data points starting from the last window instead of the first.
NB. Only some of the methods can use this argument.

4.6 randomize

Type: logical (TRUE or FALSE)

After creating the the grouping factor using the chosen method, it is possible to randomly reorganize it before returning it. Notice that this **applies to all the functions**, as group() and splt() uses the grouping factor!

5 Using Functions

We will be using 'n_dist' on a dataframe to showcase the functions. Afterwards we will show and compare the methods.
Notice that you can also use vectors with all the functions.

5.1 group_factor()

1. We create a dataframe

```
df = data.frame("x"=c(1:12),
                "species" = rep(c('cat','pig', 'human'), 4),
                "age" = sample(c(1:100), 12))
```

2. Using group_factor()

```
df$groups = group_factor(df, 5, method = 'n_dist')

df
```

```
##     x species age groups
## 1   1     cat  40      1
## 2   2     pig  49      1
## 3   3   human  34      2
## 4   4     cat  56      2
## 5   5     pig  61      3
## 6   6   human  91      3
## 7   7     cat  39      3
## 8   8     pig  24      4
## 9   9   human  73      4
## 10 10     cat  36      5
## 11 11     pig  52      5
## 12 12   human  22      5
```

3. We could get the mean age of each group

```
aggregate(df[, 3], list(df$groups), mean)
```

```
##   Group.1        x
## 1       1 44.50000
## 2       2 45.00000
## 3       3 63.66667
## 4       4 48.50000
## 5       5 36.66667
```

5.1.1 force_equal

Getting an equal number of elements per window with group_factor().

Notice that we discard the excess values so all groups contain the same amount of elements. Since the grouping factor is shorter than the dataframe, we can't combine them as they are. A way to do so would be to shorten the dataframe to be the same length as the grouping factor.

1. We create a dataframe

```
df = data.frame("x"=c(1:12),
                "species" = rep(c('cat','pig', 'human'), 4),
                "age" = sample(c(1:100), 12))
```

2. Using group_factor() with force_equal

```
groups = group_factor(df, 5, method = 'n_dist', force_equal = TRUE)

groups
```

```
##  [1] 1 1 2 2 3 3 4 4 5 5
## Levels: 1 2 3 4 5
```

```
count(groups)
```

```
##   x freq
## 1 1    2
## 2 2    2
## 3 3    2
## 4 4    2
## 5 5    2
```

3. Combining dataframe and grouping factor

First we make the dataframe the same size as the grouping factor. Then we add the grouping factor to the dataframe.

```
df = head(df, length(groups))

df$group = groups

df
```

```
##     x species age group
## 1   1     cat  83     1
## 2   2     pig  79     1
## 3   3   human  59     2
## 4   4     cat  47     2
## 5   5     pig  18     3
## 6   6   human  86     3
## 7   7     cat  62     4
## 8   8     pig  84     4
## 9   9   human  28     5
## 10 10     cat  15     5
```

5.2 group()

1. We create a dataframe

```
df = data.frame("x"=c(1:12),
                "species" = rep(c('cat','pig', 'human'), 4),
                "age" = sample(c(1:100), 12))
```

2. Using group()

```
df_grouped = group(df, 5, method = 'n_dist')

df_grouped
```

```
## Source: local data frame [12 x 4]
## Groups: .groups [5]
##
##        x species   age .groups
##    <int>  <fctr> <int>  <fctr>
## 1      1     cat    21       1
## 2      2     pig    43       1
## 3      3   human    38       2
## 4      4     cat    30       2
## 5      5     pig    32       3
## 6      6   human     3       3
## 7      7     cat    18       3
## 8      8     pig    60       4
## 9      9   human    52       4
## 10    10     cat    88       5
## 11    11     pig    75       5
## 12    12   human     7       5
```

2.2 Using group() with dplyr pipeline to get mean age

```
df_means = df %>%
  group(5, method = 'n_dist') %>%
  dplyr::summarise(mean_age = mean(age))

df_means
```

```
## # A tibble: 5 <U+00D7> 2
##   .groups mean_age
##    <fctr>    <dbl>
## 1       1 32.00000
## 2       2 34.00000
## 3       3 17.66667
## 4       4 56.00000
## 5       5 56.66667
```

5.2.1 force_equal

Getting an equal number of elements per window with group().

Notice that we discard the excess rows/elements so all groups contain the same amount of elements.

1. We create a dataframe

```
df = data.frame("x"=c(1:12),
                "species" = rep(c('cat','pig', 'human'), 4),
                "age" = sample(c(1:100), 12))
```

2. Using group() with force_equal

```
df_grouped = df %>%
  group(5, method = 'n_dist', force_equal = TRUE)

df_grouped
```

```
## Source: local data frame [10 x 4]
## Groups: .groups [5]
##
##        x species   age .groups
## *  <int>  <fctr> <int>  <fctr>
## 1      1     cat    93       1
## 2      2     pig    56       1
## 3      3   human     1       2
## 4      4     cat    95       2
## 5      5     pig    47       3
## 6      6   human     7       3
## 7      7     cat    37       4
## 8      8     pig    91       4
## 9      9   human    55       5
## 10    10     cat    39       5
```

5.3 splt()

1. We create a dataframe

```
df = data.frame("x"=c(1:12),
                "species" = rep(c('cat','pig', 'human'), 4),
                "age" = sample(c(1:100), 12))
```

2. Using splt()

```
df_list = splt(df, 5, method = 'n_dist')

df_list
```

```
## $`1`
##   x species age
## 1 1     cat  13
## 2 2     pig  11
##
## $`2`
##   x species age
## 3 3   human  85
## 4 4     cat  61
##
## $`3`
##   x species age
## 5 5     pig  57
## 6 6   human  42
## 7 7     cat  62
##
## $`4`
##   x species age
## 8 8     pig  23
## 9 9   human  46
##
## $`5`
##     x species age
## 10 10     cat  71
## 11 11     pig  98
## 12 12   human  52
```

5.3.1 force_equal

Getting an equal number of elements per window with splt().

Notice that we discard the excess rows/elements so all groups contain the same amount of elements.

1. We create a dataframe

```
df = data.frame("x"=c(1:12),
                "species" = rep(c('cat','pig', 'human'), 4),
                "age" = sample(c(1:100), 12))
```

2. Using splt() with force_equal

```
df_list = splt(df, 5, method = 'n_dist', force_equal = TRUE)

df_list
```

```
## $`1`
##   x species age
## 1 1     cat  52
## 2 2     pig  27
##
## $`2`
##   x species age
```

```
## 3 3   human  96
## 4 4     cat  19
##
## $`3`
##   x species age
## 5 5     pig  37
## 6 6   human  60
##
## $`4`
##   x species age
## 7 7     cat  71
## 8 8     pig  93
##
## $`5`
##     x species age
## 9   9   human  47
## 10 10     cat  58
```

5.4 Extra arguments showcase

5.4.1 randomize

1. We create a dataframe

```
df = data.frame("x"=c(1:12),
                "species" = rep(c('cat','pig', 'human'), 4),
                "age" = sample(c(1:100), 12))
```

2. We use group_factor() with randomize set to TRUE

```
groups = group_factor(df, 5, method = 'n_dist', randomize = TRUE)

groups
```

```
##  [1] 4 3 2 5 3 3 5 5 1 4 2 1
## Levels: 1 2 3 4 5
```

3. We use splt() with randomize set to TRUE
   Notice that the index has been shuffled but the window sizes are the same as before!

```
df_list = splt(df, 5, method = 'n_dist', randomize = TRUE)

df_list
```

```
## $`1`
##   x species age
## 3 3   human  54
## 8 8     pig  68
```

```
##
## $`2`
##   x species age
## 2 2     pig  11
## 6 6   human  35
##
## $`3`
##     x species age
## 1   1     cat  59
## 4   4     cat  31
## 10 10     cat  89
##
## $`4`
##   x species age
## 7 7     cat  84
## 9 9   human  39
##
## $`5`
##     x species age
## 5   5     pig   8
## 11 11     pig  26
## 12 12   human  15
```

6 Examples of method differences

In this section we will take a look at the outputs we get from the different methods.

6.1 n_ methods

6.1.1 Vector with 57 elements split in 6 windows

Below you'll see a dataframe with counts of window elements when splitting the same data with the different n_ methods. The forced_equal column is simply with the force_equal set to TRUE.

forced_equal: Since this is a setting to make sure that all windows are of the same size, it makes sense that all the windows have the same size.

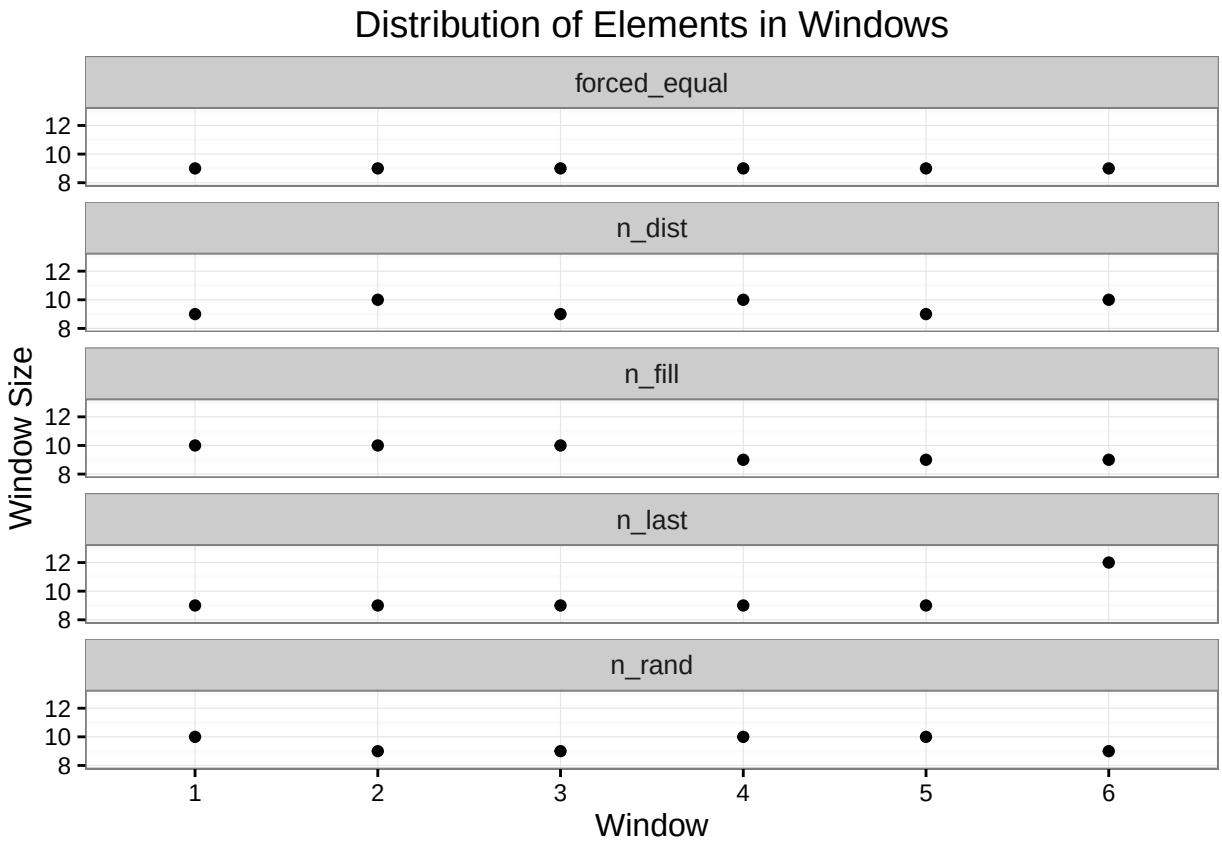
n_dist: compared to forced_equal we see the 3 datapoints that forced_equal had discarded. These are distributed across the windows (in this example window 2,4 and 6)

n_fill: The 3 extra datapoints are located at the first 3 windows. Had we set descending to TRUE, it would have been the last 3 windows instead.

n_last: We see that n_last creates equal window sizes in all but the last window. This means that the last window can sometimes have a window size, which is very large or small compared to the other windows. Here it is a third larger than the other windows.

n_rand: The extra datapoints are placed randomly and so we would see the extra datapoints located at different windows if we ran the script again. *Unless we use set.seed() before running the function.*

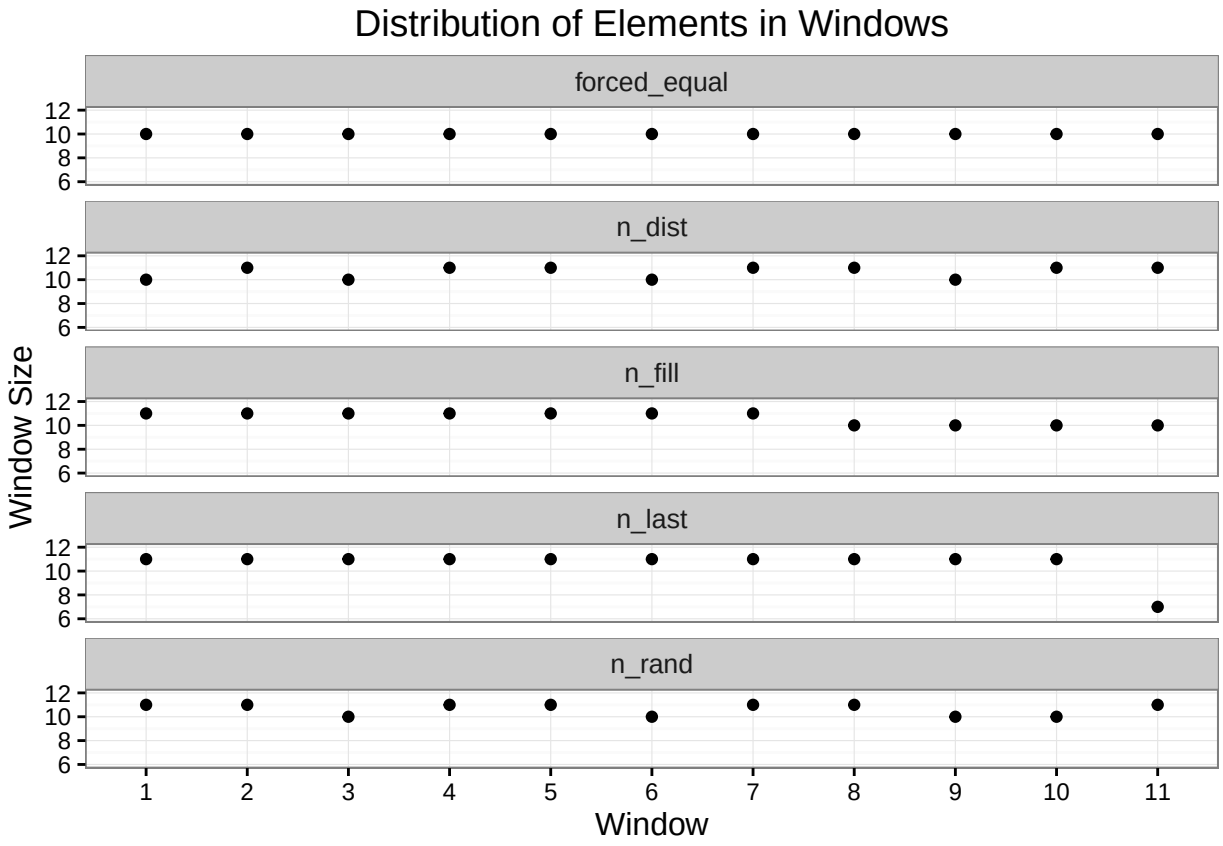
```
##   x n_dist n_fill n_last n_rand forced_equal
## 1 1      9     10      9     10            9
## 2 2     10     10      9      9            9
## 3 3      9     10      9      9            9
## 4 4     10      9      9     10            9
## 5 5      9      9      9     10            9
## 6 6     10      9     12      9            9
```



6.1.2 Vector with 117 elements split in 11 windows

Here is another example.

```
##     x n_dist n_fill n_last n_rand forced_equal
## 1   1     10     11     11     11           10
## 2   2     11     11     11     11           10
## 3   3     10     11     11     10           10
## 4   4     11     11     11     11           10
## 5   5     11     11     11     11           10
## 6   6     10     11     11     10           10
## 7   7     11     11     11     11           10
## 8   8     11     10     11     11           10
## 9   9     10     10     11     10           10
## 10 10     11     10     11     10           10
## 11 11     11     10      7     11           10
```



6.2 Greedy

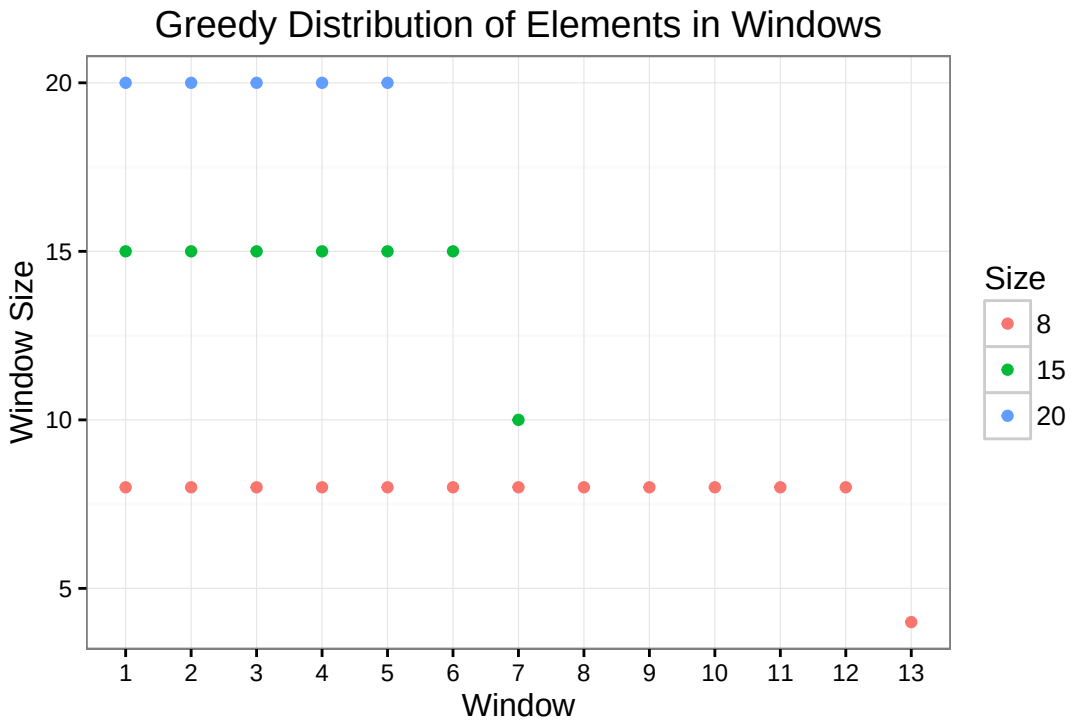
6.2.1 Vector with 100 elements with sizes of 8, 15, 20

Below you will see window sizes when using the method 'greedy' and asking for window sizes of 8, 15, 20. What should become clear is that only the last window can have a different window size than what we asked for. This is important if, say, you want to split a time series into windows of 100 elements, but the time series is not divisible with 100. Then you could use force_equal to remove the excess elements, if you need equal windows.

With a size of 8, we get 13 windows. The last window (13) only contains 4 elements, but all the other windows contain 8 elements as specified.

With a size of 15, we get 7 windows. The last window (7) contains only 10 elements, but all the other windows contain 15 elements as specified.

With a size of 20, we get 5 windows. As 20 is divisible with the 100 elements that the splitted vector contained, the last window also contains 20 elements, and so we have equal groups.



6.3 Staircasing

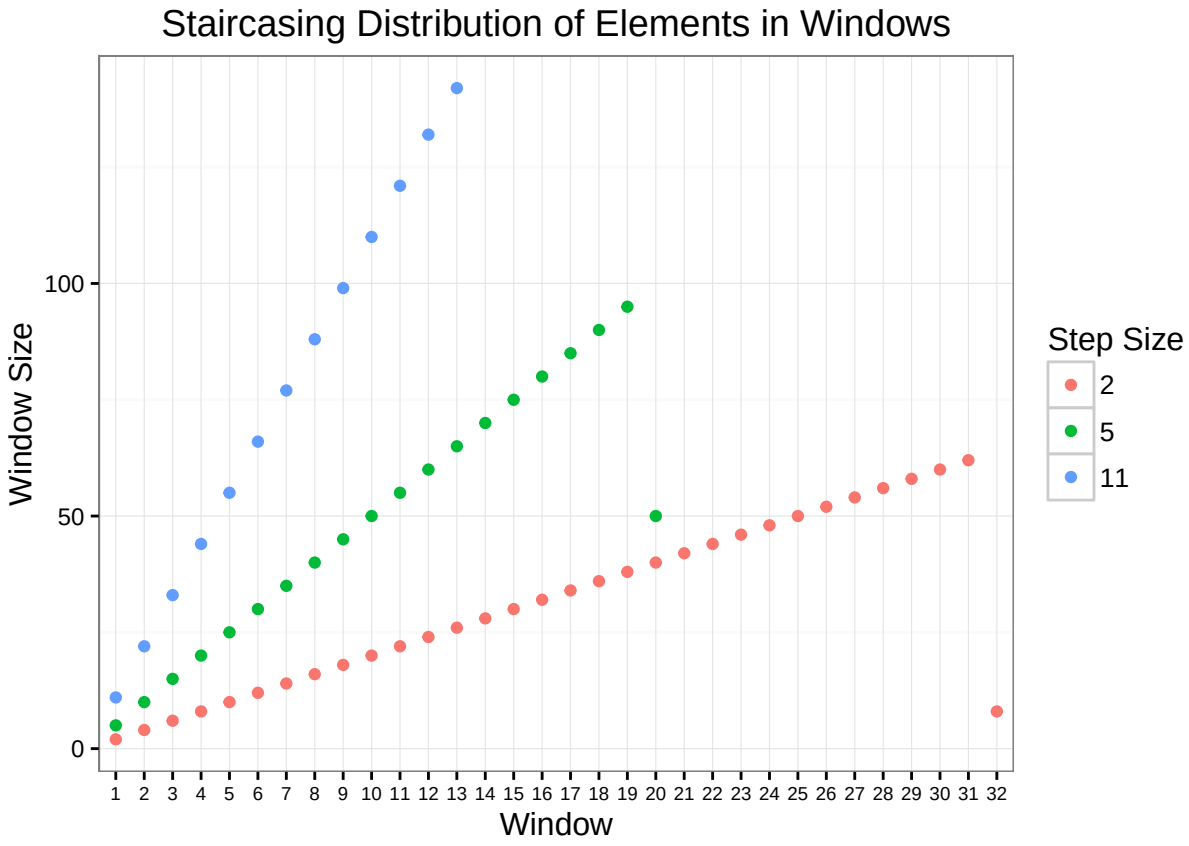
6.3.1 Vector with 1000 elements with step sizes of 2, 5, 11

Below you'll see a plot with the window sizes at each window when using step sizes 2, 5, and 11.

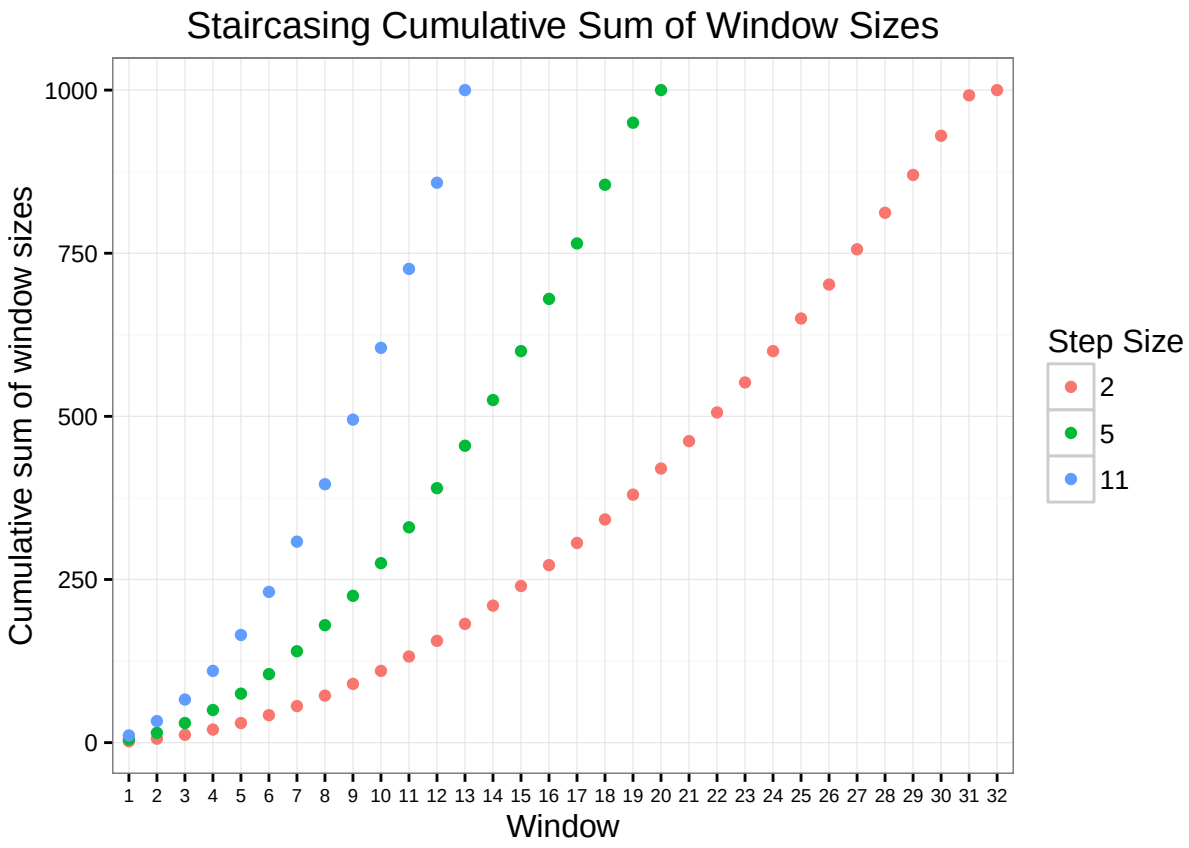
At a step size of 2 elements it simply increases 2 for each window, until the last window (32) where it runs out of elements. Had we set force_equal to TRUE, this last window would have been discarded, because of the lack of elements.

At a step size of 5 elements it increases with 5 every time. Because of this it runs out of elements faster. Again we see that the last window (20) has fewer elements.

At a step size of 11 elements it increases with 11 every time. It seems that the last window is not too small, but it can be hard to see on this scale. Actually, the last window misses 1 element to be complete and so would have been discarded if force_equal was set to TRUE.



Below we will take a short look at the cumulative sum of window elements to get an idea of what is going on under the hood.
Remember that the splitted vector had 1000 elements? That is why they all stop at 1000 on the y-axis. There are simply no more elements left!


Staircasing Cumulative Sum of Window Sizes
Cumulative sum of window sizes
1000
750
500
250
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Window
Step Size
2
5
11