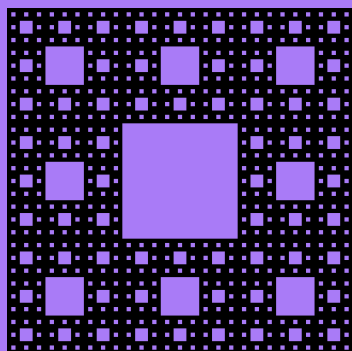
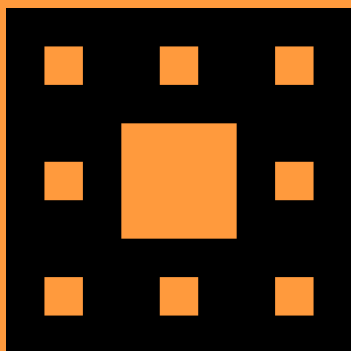
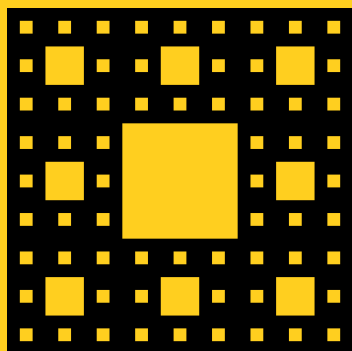
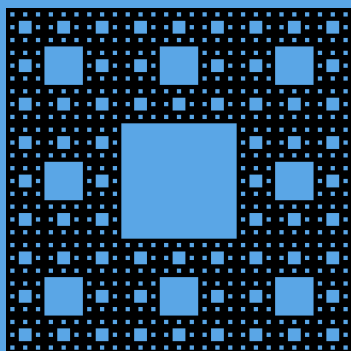


USER! 2026

Semantic vectors for safer statistics

9th July 2026 @ useR! (Warsaw, Poland)

Mitchell O'Hara-Wild, Monash University

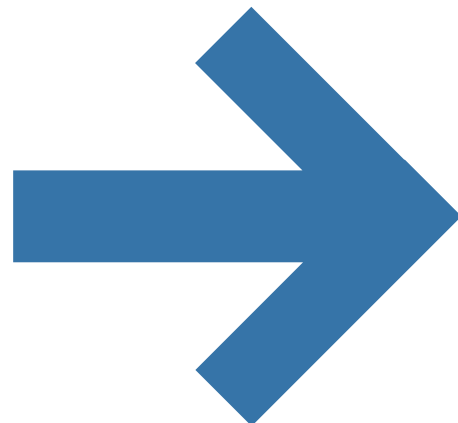


Useful links

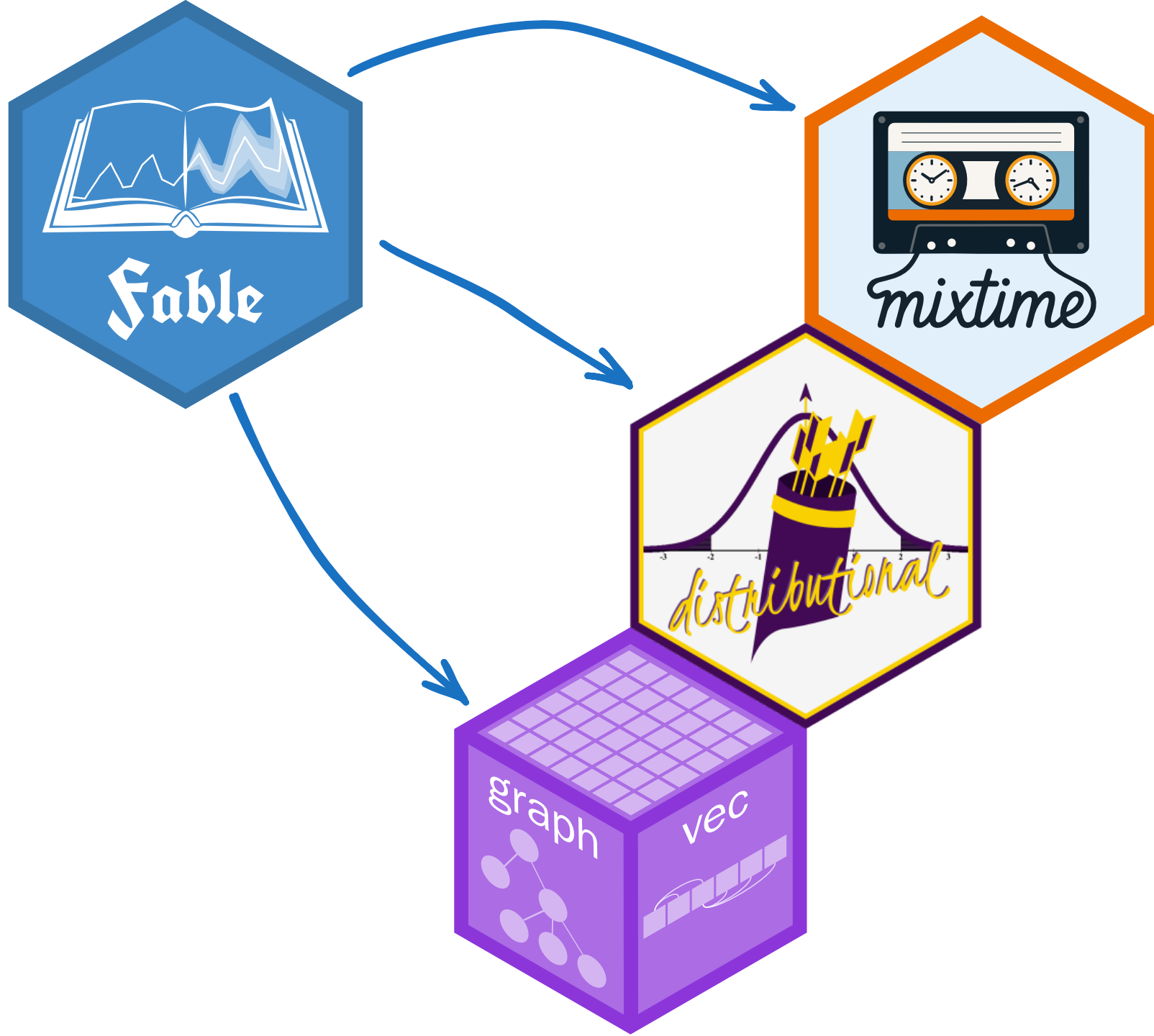
 social.mitchelloharawild.com

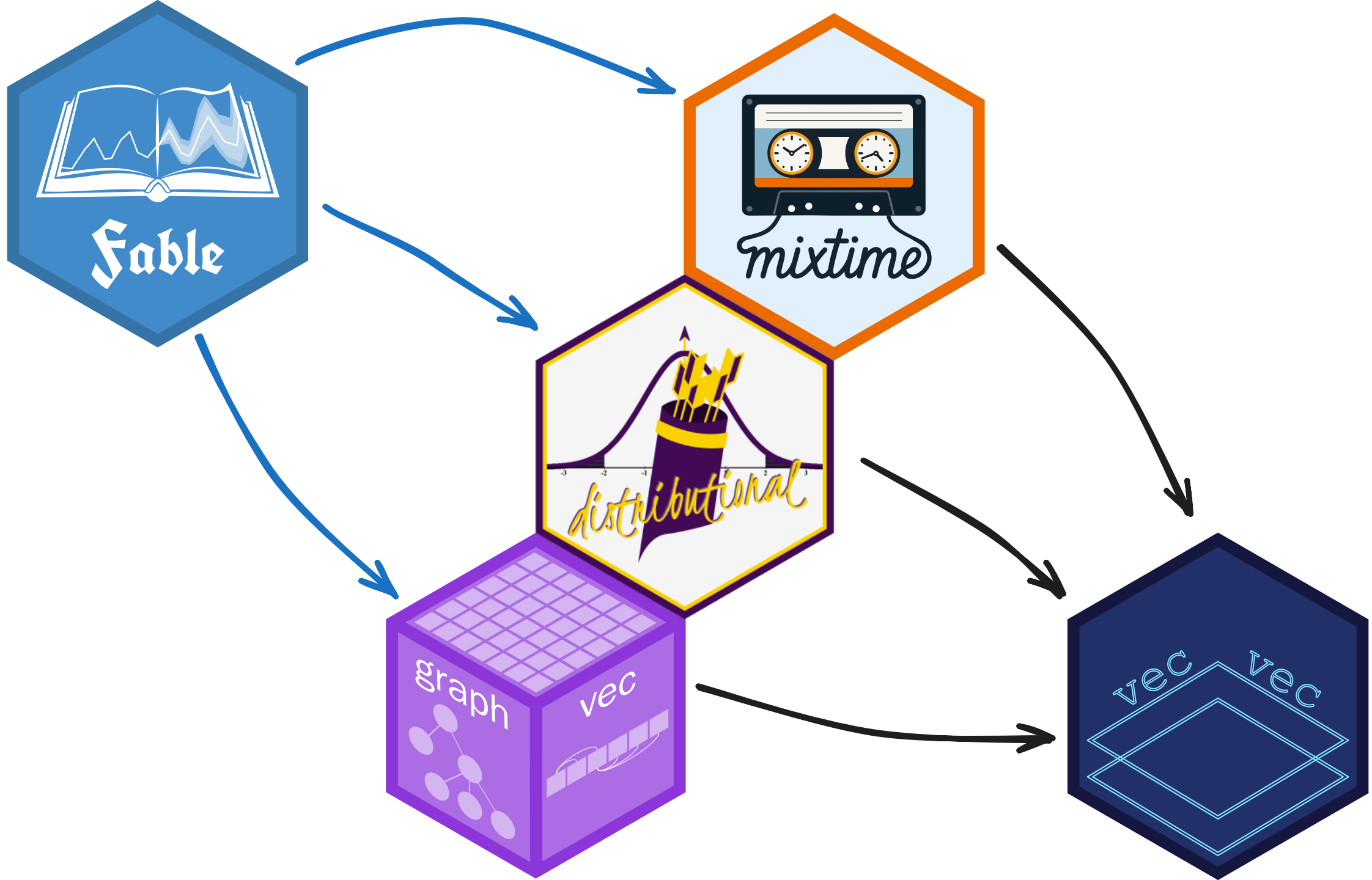
 slides.mitchelloharawild.com/user2026

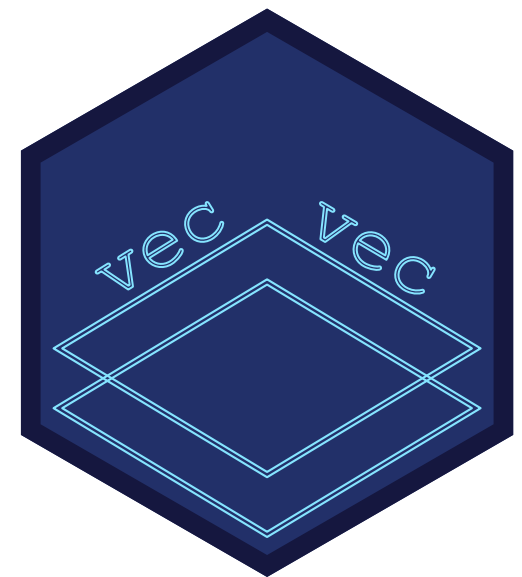
 mitchelloharawild.com/talk-vecvec-user2026

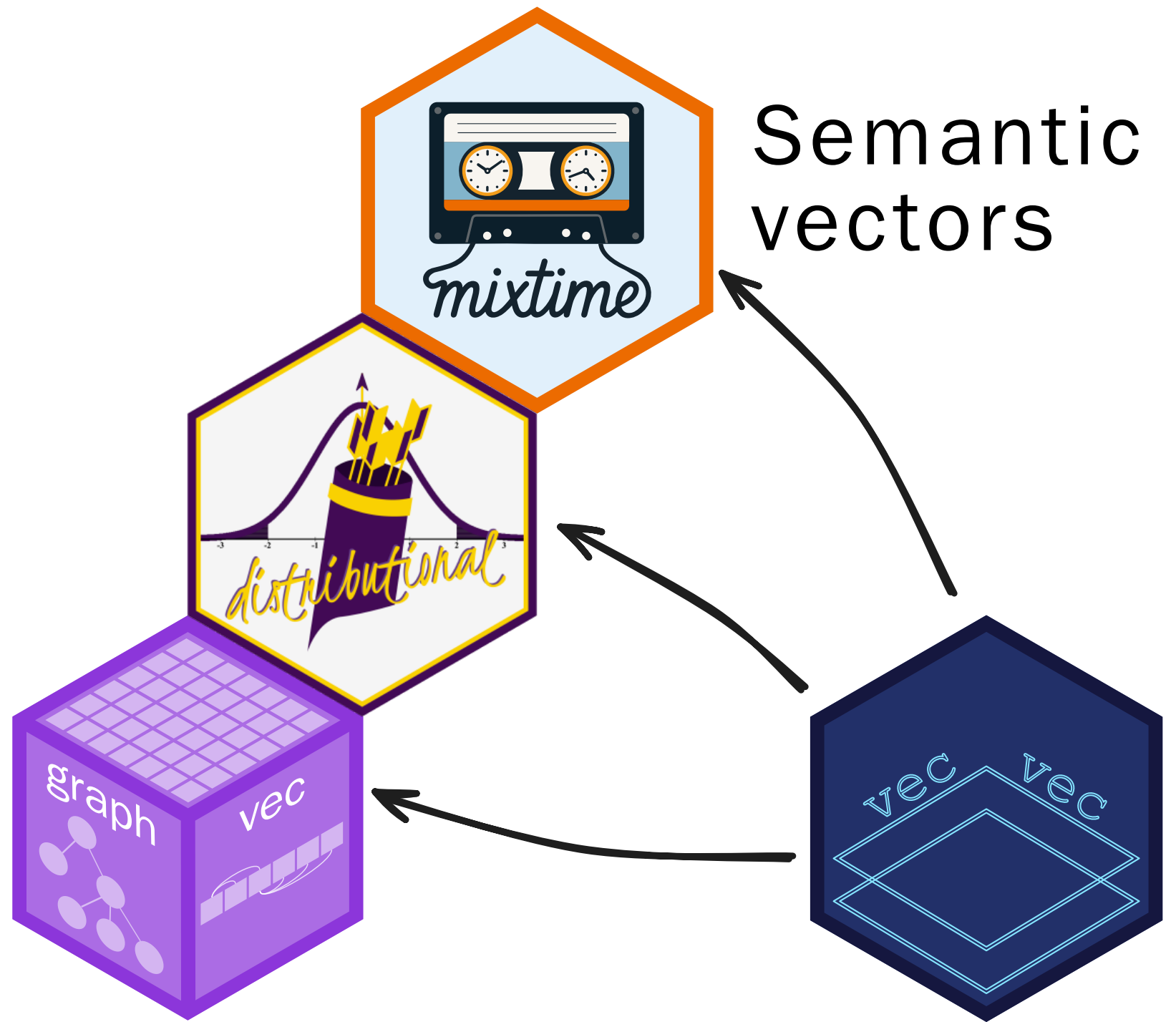


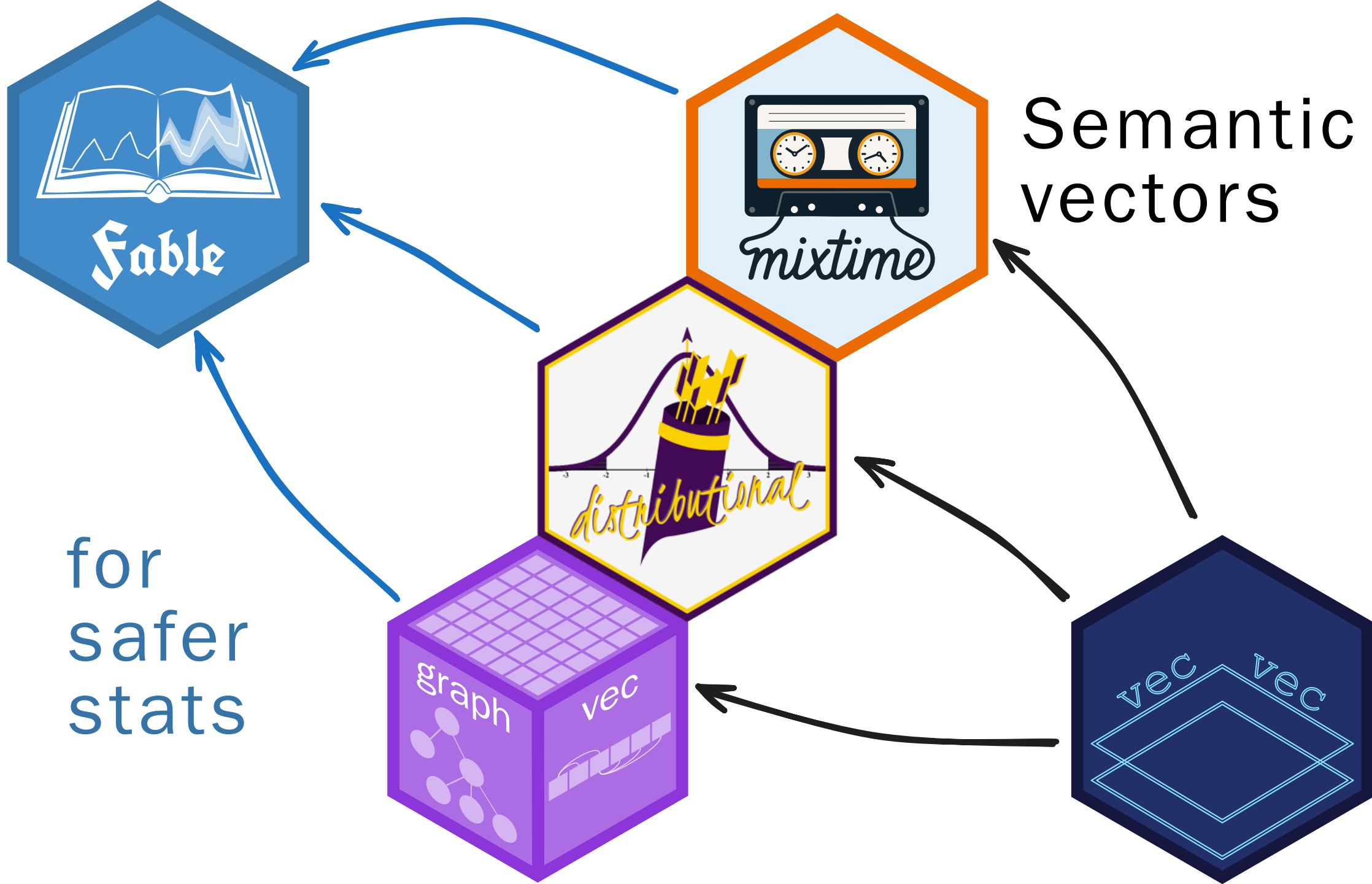












Safer statistics with semantics



Safer statistics with semantics

Many data types have *intrinsic structure*,
requiring special care in statistical analysis.



Safer statistics with semantics

Many data types have *intrinsic structure*, requiring special care in statistical analysis.

-  ordinal: `ordered`, `{forcats}`



Safer statistics with semantics

Many data types have *intrinsic structure*, requiring special care in statistical analysis.

-  ordinal: `ordered`, `{forcats}`
-  uncertainty: `p/d/q/r`, `{distributional}`



Safer statistics with semantics

Many data types have *intrinsic structure*, requiring special care in statistical analysis.

-  ordinal: `ordered`, `{forcats}`
-  uncertainty: `p/d/q/r`, `{distributional}`
-  time: `Date`, `POSIXt`, `{hms}`, `{mixtime}`



Safer statistics with semantics

Many data types have *intrinsic structure*, requiring special care in statistical analysis.

- 🗃 ordinal: `ordered`, `{forcats}`
- 🍀 uncertainty: `p/d/q/r`, `{distributional}`
- ⌚ time: `Date`, `POSIXt`, `{hms}`, `{mixtime}`
- 🕸 graph: `{igraph}`, `{tidygraph}`, `{graphvec}`



Safer statistics with semantics

Many data types have *intrinsic structure*, requiring special care in statistical analysis.

-  ordinal: `ordered`, `{forcats}`
-  uncertainty: `p/d/q/r`, `{distributional}`
-  time: `Date`, `POSIXt`, `{hms}`, `{mixtime}`
-  graph: `{igraph}`, `{tidygraph}`, `{graphvec}`
-  spatial: `{sp}`, `{sf}`



Safer statistics with semantics

Statistical operations leverage their structure to prevent analysis that violate semantics.

```
1 Jan31 <- as.Date("2025-01-31")  
2 Jan31 + 1
```

```
[1] "2025-02-01"
```



Safer statistics with semantics

Statistical operations leverage their structure to prevent analysis that violate semantics.

```
1 Jan31 <- as.Date("2025-01-31")  
2 Jan31 + 1
```

```
[1] "2025-02-01"
```

```
1 Jan31 * 2
```

```
Error in `Ops.Date()`:  
! * not defined for "Date" objects
```



Safer statistics with semantics

Statistical operations leverage their structure to prevent analysis that violate semantics.

```
1 Jan31 <- as.Date("2025-01-31")
2 Jan31 + 1
```

```
[1] "2025-02-01"
```

```
1 Jan31 * 2
```

```
Error in `Ops.Date()`:
! * not defined for "Date" objects
```

```
1 log(Jan31)
```

```
Error in `Math.Date()`:
! log not defined for "Date" objects
```



Safer statistics with semantics

Statistical operations leverage their structure to prevent analysis that violate semantics.

```
1 Jan31 <- as.Date("2025-01-31")
2 Jan31 + 1
```

```
[1] "2025-02-01"
```

```
1 Jan31 * 2
```

```
Error in `Ops.Date()` :
! * not defined for "Date" objects
```

```
1 log(Jan31)
```

```
Error in `Math.Date()` :
! log not defined for "Date" objects
```

```
1 Jan31 + lubridate::period(1, "months")
```

```
[1] NA
```



Safer statistics with semantics

Statistical operations leverage their structure to prevent analysis that violate semantics.

```
1 Jan31 <- as.Date("2025-01-31")
2 Jan31 + 1
```

```
[1] "2025-02-01"
```

```
1 Jan31 * 2
```

```
Error in `Ops.Date()` :
! * not defined for "Date" objects
```

```
1 log(Jan31)
```

```
Error in `Math.Date()` :
! log not defined for "Date" objects
```

```
1 Jan31 + lubridate::period(1, "months")
```

```
[1] NA
```

```
1 Sys.Date() + Sys.time()
```

```
Warning: Incompatible methods ("+.Date", "+.POSIXt") for
"+"
```

```
[1] "4884653-08-21"
```



Unsafe Stats: R's distributions

The included distributions in R (and many extension packages) use p/d/q/r functions for statistical operations on distributions.

The p/d/q/r functions

These functions allow you to calculate statistical operations from distributions:

- **p**: The 'probability' (CDF)
- **d**: The 'density' (PDF)
- **q**: The 'quantiles'
- **r**: The 'random' samples

Some packages also define **m** functions for moments!



Unsafe Stats: R's distributions

Using p/d/q/r functions

These operation prefixes are used in conjunction with the distribution's shape. The general form is:

`<op><shape> (<args>, <parameters>)`



Unsafe Stats: R's distributions

Using p/d/q/r functions

These operation prefixes are used in conjunction with the distribution's shape. The general form is:

`<op><shape> (<args>, <parameters>)`

For example, the density (**d**) at **0.5** of a Normal (**norm**) distribution with mean **1** and standard deviation **3** is:

```
dnorm(q = 0.5, mean = 1, sd = 3)
```

```
[1] 0.1311466
```



Unsafe Stats: R's distributions

Using p/d/q/r functions

These operation prefixes are used in conjunction with the distribution's shape. The general form is:

`<op><shape> (<args>, <parameters>)`

For example, the density (**d**) at **0.5** of a Normal (**norm**) distribution with mean **1** and standard deviation **3** is:

```
dnorm(q = 0.5, mean = 1, sd = 3)
```

```
[1] 0.1311466
```

Semantics not included

These functions are very **error-prone** since users have the burden of performing valid distributional operations.



Unsafe Stats: R's distributions

⚠ Short and confusing function names

The p/d/q/r functions need memorisation for each shape.

⚠ Risky recycling

These p/d/q/r functions are **vectorised** and **fast** 🎉



Unsafe Stats: R's distributions

⚠ Short and confusing function names

The p/d/q/r functions need memorisation for each shape.

⚠ Risky recycling

These p/d/q/r functions are **vectorised** and **fast** 🎉

How these functions vectorise inputs is surprising 🤪



Unsafe Stats: R's distributions

⚠ Short and confusing function names

The p/d/q/r functions need memorisation for each shape.

⚠ Risky recycling

These p/d/q/r functions are **vectorised** and **fast** 🎉

How these functions vectorise inputs is surprising 🤪

```
1 # 95% intervals from N(1,9)
2 qnorm(c(0.025, 0.975), mean = 1, sd = 3)

[1] -4.879892  6.879892
```



Unsafe Stats: R's distributions

⚠ Short and confusing function names

The p/d/q/r functions need memorisation for each shape.

⚠ Risky recycling

These p/d/q/r functions are **vectorised** and fast 🎉

How these functions vectorise inputs is surprising 🤪

```
1 # 95% intervals from N(1,9)
2 qnorm(c(0.025, 0.975), mean = 1, sd = 3)

[1] -4.879892  6.879892
```

```
1 # 2.5% from N(1,9) and 97.5% from N(1,16)
2 qnorm(c(0.025, 0.975), mean = 1, sd = c(3,4))

[1] -4.879892  8.839856
```



Unsafe Stats: R's distributions

⚠ Short and confusing function names

The p/d/q/r functions need memorisation for each shape.

⚠ Risky recycling

These p/d/q/r functions are **vectorised** and fast 🎉

How these functions vectorise inputs is surprising 🤪

```
1 # 95% intervals from N(1,9)
2 qnorm(c(0.025, 0.975), mean = 1, sd = 3)
```

```
[1] -4.879892  6.879892
```

```
1 # 2.5% from N(1,9) and 97.5% from N(1,16)
2 qnorm(c(0.025, 0.975), mean = 1, sd = c(3,4))
```

```
[1] -4.879892  8.839856
```

```
1 # What is happening here!
2 qnorm(c(0.025, 0.975), mean = c(1,2), sd = c(3,4,5))
```

```
[1] -4.879892  9.839856 -8.799820
```



Unsafe Stats: R's distributions

! R's model prediction output

Worst of all: obtaining distributions from models.



Unsafe Stats: R's distributions

! R's model prediction output

Worst of all: obtaining distributions from models.

Predicting penguins

Consider the output when using `predict()` on a `lm()` for the length of penguin bills using depth and species.

```
1 fit <- lm(  
2   bill_length_mm ~ species*bill_depth_mm,  
3   data = palmerpenguins::penguins  
4 )  
5 predict(fit, tail(palmerpenguins::penguins))
```

	1	2	3	4	5	6
	46.10333	51.48517	48.21763	48.40983	49.94750	49.37088



Unsafe Stats: R's distributions

❗ R's model prediction output

Worst of all: obtaining distributions from models.

🐧 Predicting penguins

Consider the output when using `predict()` on a `lm()` for the length of penguin bills using depth and species.

```
1 fit <- lm(  
2   bill_length_mm ~ species*bill_depth_mm,  
3   data = palmerpenguins::penguins  
4 )  
5 predict(fit, tail(palmerpenguins::penguins))
```

1	2	3	4	5	6
46.10333	51.48517	48.21763	48.40983	49.94750	49.37088

⚠️ Where's the uncertainty!

By default, predictions only return the expected value.



Unsafe Stats: R's distributions

💡 Finding uncertainty!

Use `se.fit = TRUE` in `predict()` to get standard errors.

```
1 predict(fit, tail(palmerpenguins::penguins),  
2         se.fit = TRUE)
```

```
$fit  
      1      2      3      4      5      6  
46.10333 51.48517 48.21763 48.40983 49.94750 49.37088
```

```
$se.fit  
      1      2      3      4      5  
6  
0.4769975 0.4685605 0.3082198 0.3020842 0.3333431  
0.3054341
```

```
$df  
[1] 336
```

```
$residual.scale  
[1] 2.444663
```



Unsafe Stats: R's distributions

 Time to compute intervals!

(as a naive first-year stats student)

```
1 pred <- predict(fit, tail(palmerpenguins::penguins),
2   sprintf(
3     "[%f, %f] 95",
4     qnorm(0.025, mean = pred$fit, pred$se.fit),
5     qnorm(0.975, mean = pred$fit, pred$se.fit)
6   )
```

```
[45.168436, 47.038232] 95
[50.566807, 52.403530] 95
[47.613526, 48.821726] 95
[47.817760, 49.001909] 95
[49.294161, 50.600842] 95
[48.772237, 49.969516] 95
```



Unsafe Stats: R's distributions

Here's the code to obtain 95% prediction intervals for the penguins data:

```
1 pred <- predict(fit, tail(palmerpenguins::penguins),  
2 sprintf(  
3   "[%f, %f]95",  
4   pred$fit + qt(0.025, df = pred$df) * pred$se.fit,  
5   pred$fit + qt(0.975, df = pred$df) * pred$se.fit  
6 )
```

```
[45.165056, 47.041611]95  
[50.563487, 52.406850]95  
[47.611342, 48.823910]95  
[47.815620, 49.004049]95  
[49.291799, 50.603204]95  
[48.770072, 49.971680]95
```



Unsafe Stats: R's distributions

Here's the code to obtain 95% prediction intervals for the penguins data:

```
1 pred <- predict(fit, tail(palmerpenguins::penguins),  
2 sprintf(  
3   "[%f, %f]95",  
4   pred$fit + qt(0.025, df = pred$df) * pred$se.fit,  
5   pred$fit + qt(0.975, df = pred$df) * pred$se.fit  
6 )
```

```
[45.165056, 47.041611]95  
[50.563487, 52.406850]95  
[47.611342, 48.823910]95  
[47.815620, 49.004049]95  
[49.291799, 50.603204]95  
[48.770072, 49.971680]95
```

! Not quite right

The above calculations are for **confidence intervals**, not prediction intervals! Did you notice?



Unsafe Stats: R's distributions

The correct code for prediction intervals is:

```
1 pred <- predict(fit, tail(palmerpenguins::penguins),
2 sprintf(
3   "[%f, %f]95",
4   pred$fit + qt(0.025, df = pred$df) * sqrt(pred$se.f
5   pred$fit + qt(0.975, df = pred$df) * sqrt(pred$se.f
6 )
```

[41.203878, 51.002789]95

[46.588864, 56.381473]95

[43.370784, 53.064468]95

[43.564487, 53.255182]95

[45.094230, 54.800773]95

[44.524716, 54.217036]95



Unsafe Stats: R's distributions

The correct code for prediction intervals is:

```
1 pred <- predict(fit, tail(palmerpenguins::penguins),
2 sprintf(
3   "[%f, %f]95",
4   pred$fit + qt(0.025, df = pred$df) * sqrt(pred$se.f
5   pred$fit + qt(0.975, df = pred$df) * sqrt(pred$se.f
6   )
```

```
[41.203878, 51.002789]95
[46.588864, 56.381473]95
[43.370784, 53.064468]95
[43.564487, 53.255182]95
[45.094230, 54.800773]95
[44.524716, 54.217036]95
```

! Error-prone analysis

There's a lot to know about regression, distributions, and R functions to get correct prediction intervals.

It's easy to make mistakes (or ignore uncertainty).



A better alternative?



A better alternative?

What if model predictions could directly produce a vector of distributions?

```
1 # Hypothetical predict() method for lm()
2 pred <- predict(fit, tail(palmerpenguins::penguins))
3 pred
```

```
<distribution[6]>
[1] t(336, 46, 2.5) t(336, 51, 2.5) t(336, 48, 2.5)
[4] t(336, 48, 2.5) t(336, 50, 2.5) t(336, 49, 2.5)
```



A better alternative?

What if model predictions could directly produce a vector of distributions?

```
1 # Hypothetical predict() method for lm()
2 pred <- predict(fit, tail(palmerpenguins::penguins))
3 pred
```

```
<distribution[6]>
[1] t(336, 46, 2.5) t(336, 51, 2.5) t(336, 48, 2.5)
[4] t(336, 48, 2.5) t(336, 50, 2.5) t(336, 49, 2.5)
```

```
1 hilo(pred, 95)
```

```
<hilo[6]>
[1] [41.20388, 51.00279]95 [46.58886, 56.38147]95
[3] [43.37078, 53.06447]95 [43.56449, 53.25518]95
[5] [45.09423, 54.80077]95 [44.52472, 54.21704]95
```



Making better distributions



`{vctrs}` offers two vector types:

 **Vectors:** `new_vctr()`

Vectors add attributes to existing vector types.

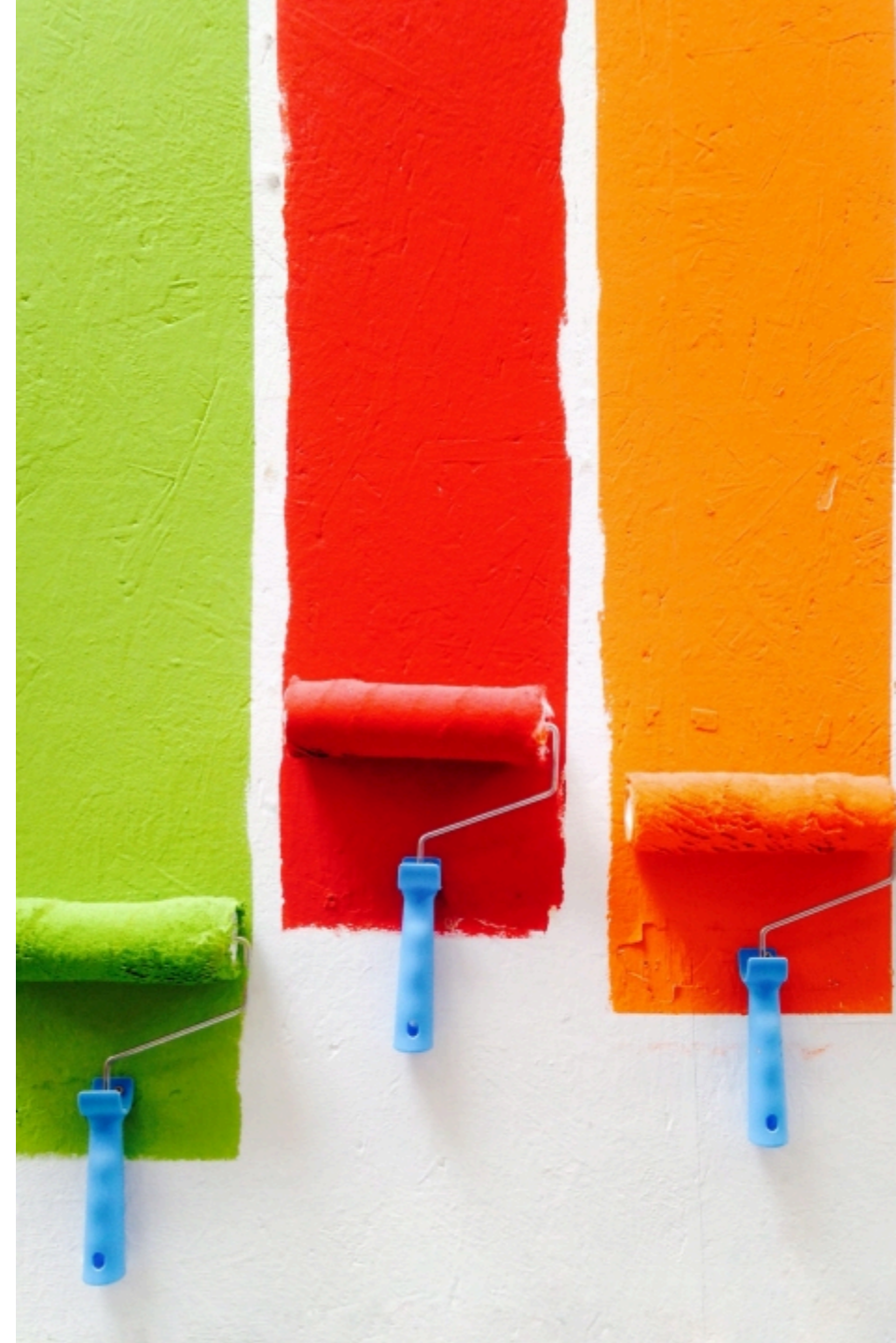
e.g. `POSIXct` is the seconds since 1970-01-01.

```
1 unclass(as.POSIXct("2025-12-04 11:00:00"))
```

```
[1] 1764842400
```

```
attr(,"tzone")
```

```
[1] ""
```



Making better distributions



`{vctrs}` offers two vector types:

 **Vectors:** `new_vctr()`

Vectors add attributes to existing vector types.

e.g. `POSIXct` is the seconds since 1970-01-01.

```
1 unclass(as.POSIXct("2025-12-04 11:00:00"))
```

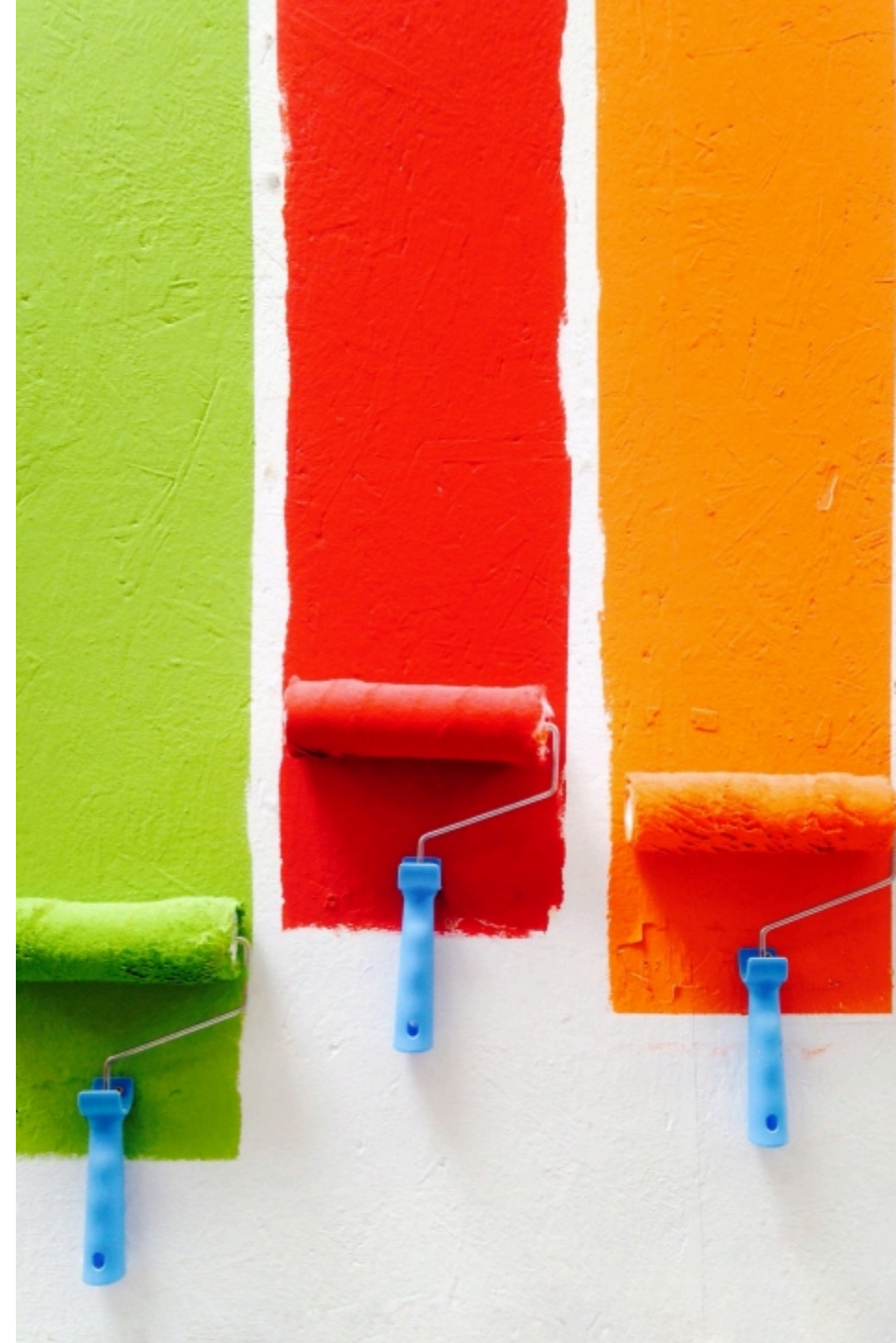
```
[1] 1764842400  
attr(,"tzone")  
[1] ""
```

This enables single-parameter distributions:

```
1 library(distributional)  
2 dist_poisson(c(4, 2, 6))
```

```
<distribution[3]>
```

```
[1] Pois(4) Pois(2) Pois(6)
```



Making better distributions



`{vctrs}` offers two vector types:

Vectors: `new_vctr()`

Vectors add attributes to existing vector types.

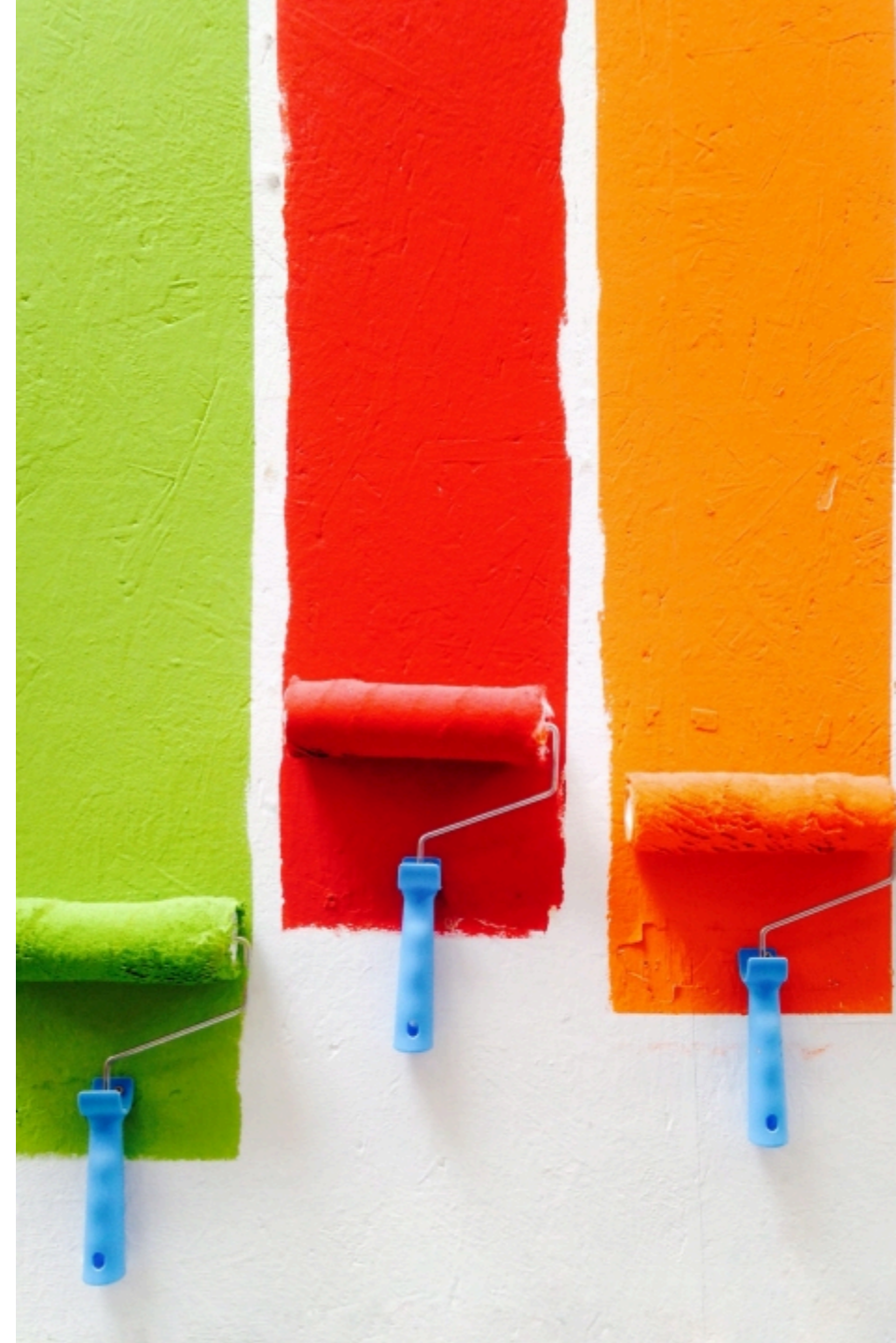
e.g. `POSIXct` is the seconds since 1970-01-01.

```
1 unclass(as.POSIXct("2025-12-04 11:00:00"))  
[1] 1764842400  
attr(,"tzone")  
[1] ""
```

This enables single-parameter distributions:

```
1 library(distributional)  
2 dist_poisson(c(4, 2, 6))  
  
<distribution[3]>  
[1] Pois(4) Pois(2) Pois(6)
```

What about multi-parameter distributions?



Making better distributions



`{vctrs}` offers two vector types:

Records: `new_rcrd()`

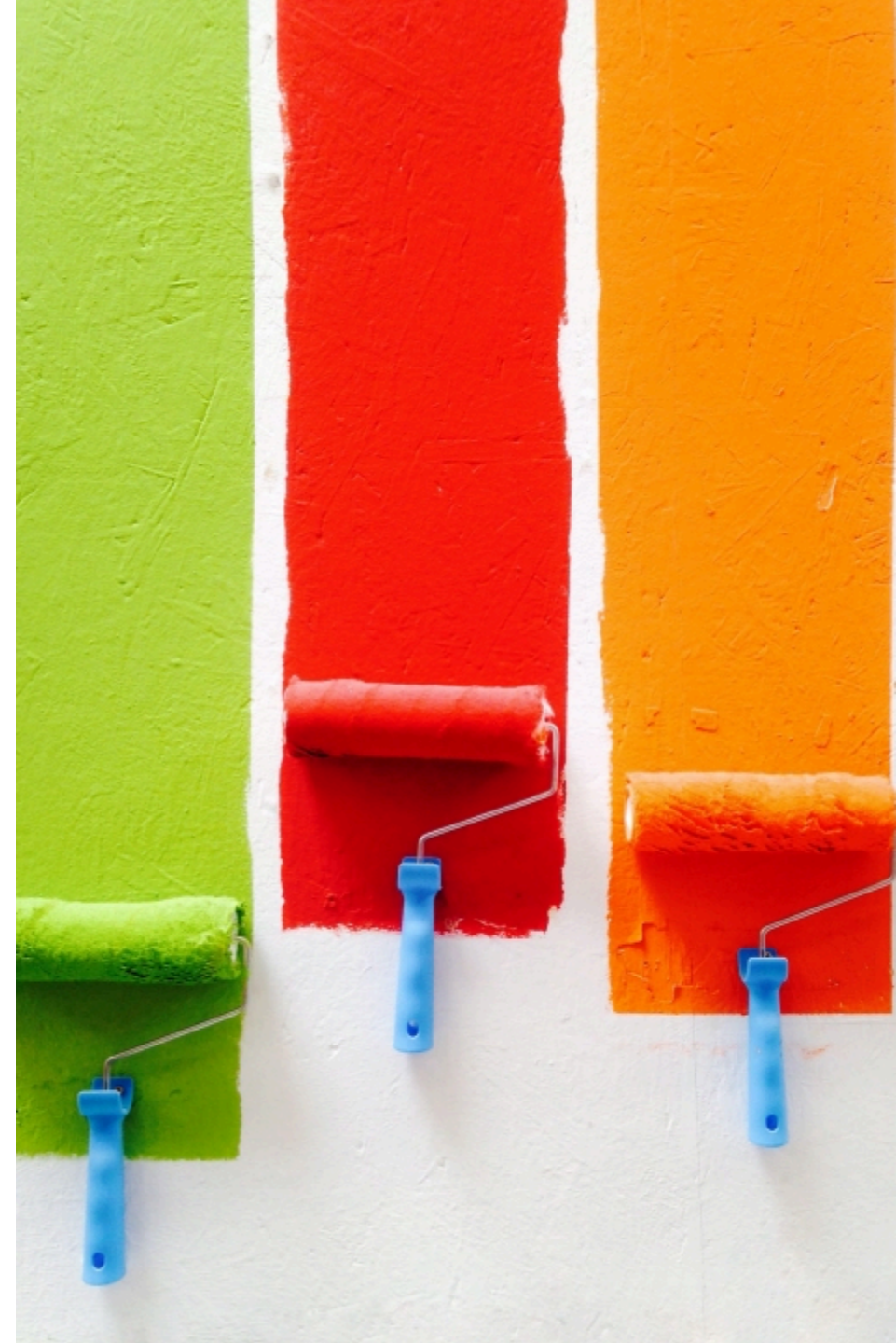
Records use information from multiple vectors (of same length).

e.g. `POSIXlt` contains the parts of a datetime.

```
1 unclass(as.POSIXlt("2025-12-04 11:00:00")) |> str()
```

List of 11

```
$ sec      : num 0
$ min      : int 0
$ hour     : int 11
$ mday     : int 4
$ mon      : int 11
$ year     : int 125
$ wday     : int 4
$ yday     : int 337
$ isdst    : int 0
$ zone     : chr "CET"
$ gmtoff   : int NA
- attr(*, "tzone")= chr [1:3] "" "CET" "CEST"
- attr(*, "balanced")= logi TRUE
```



Making better distributions



`{vctrs}` offers two vector types:

 Records: `new_rcrd()`

Records use information from multiple vectors (of same length).

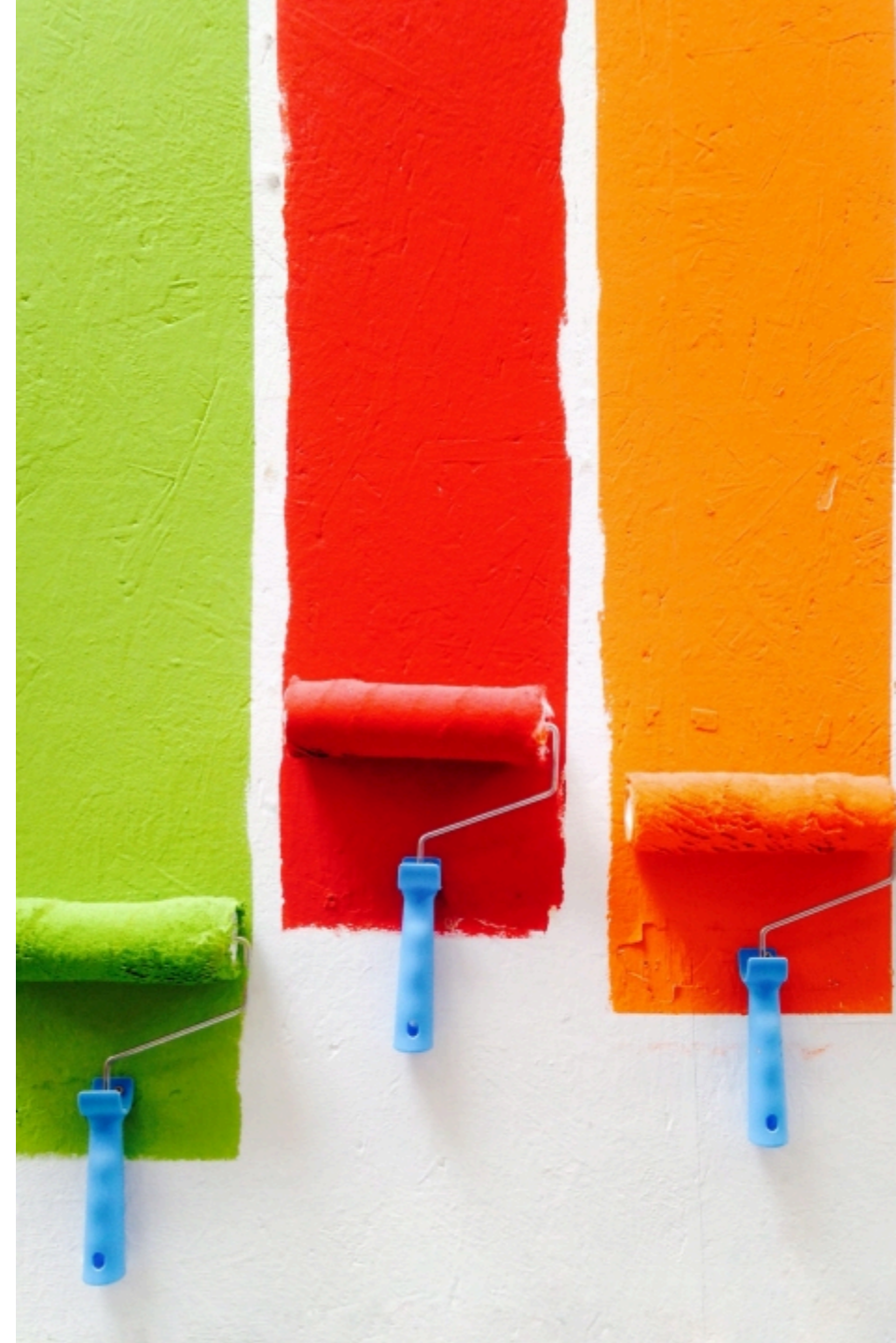
e.g. `POSIXlt` contains the parts of a datetime.

This enables multi-parameter distributions:

```
1 library(distributional)
2 dist_normal(mu = c(1, 3, -1), sigma = c(3, 2, 4))
```

```
<distribution[3]>
```

```
[1] N(1, 9)    N(3, 4)    N(-1, 16)
```



Making better distributions



`{vctrs}` offers two vector types:

 Records: `new_rcrd()`

Records use information from multiple vectors (of same length).

e.g. `POSIXlt` contains the parts of a datetime.

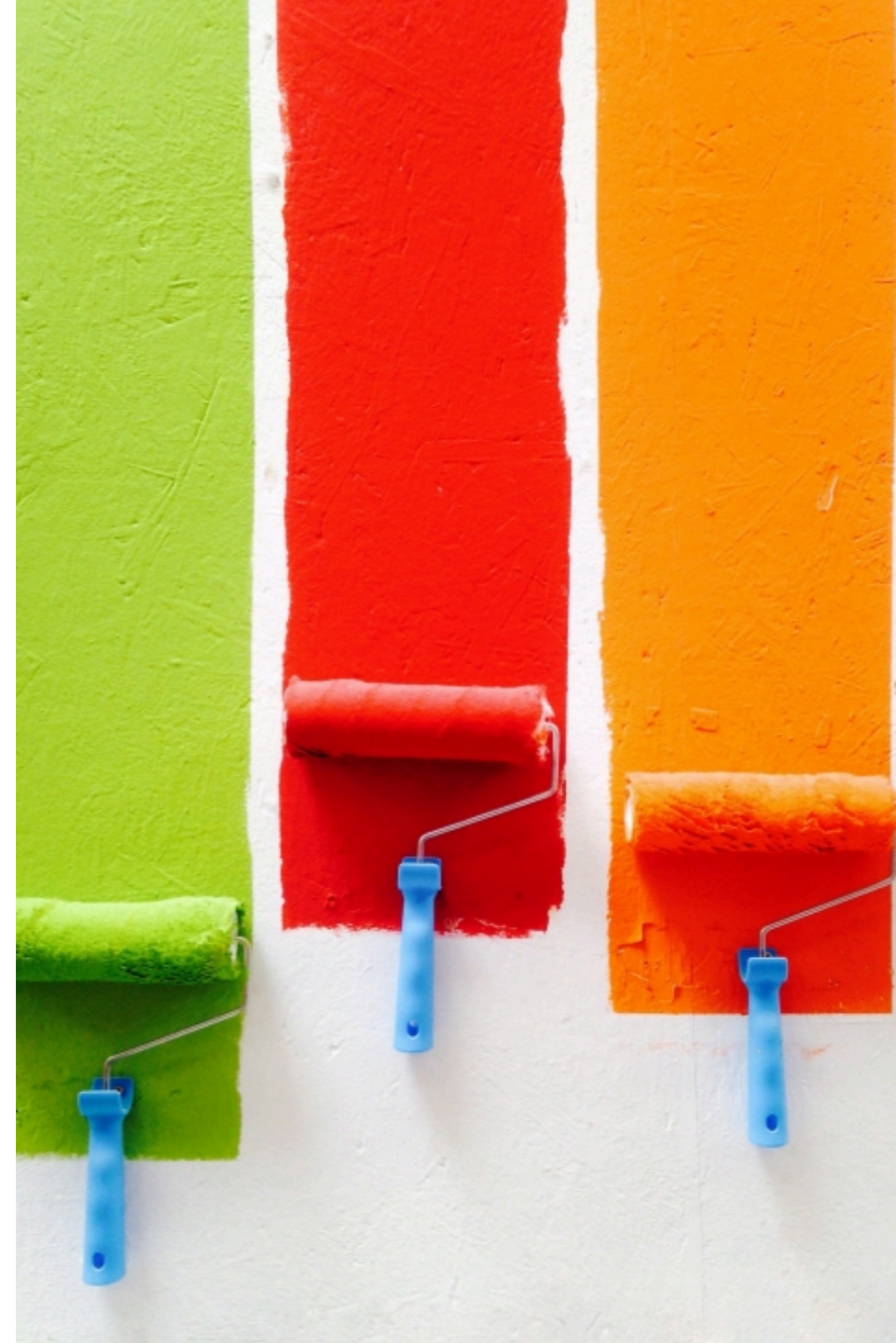
This enables multi-parameter distributions:

```
1 library(distributional)
2 dist_normal(mu = c(1, 3, -1), sigma = c(3, 2, 4))
```

```
<distribution[3]>
```

```
[1] N(1, 9)    N(3, 4)    N(-1, 16)
```

How can different shaped distributions coexist within the same vector?



Making better distributions



`{vctrs}` offers two vector types:

 Records: `new_rcrd()`

Records use information from multiple vectors (of same length).

e.g. `POSIXlt` contains the parts of a datetime.

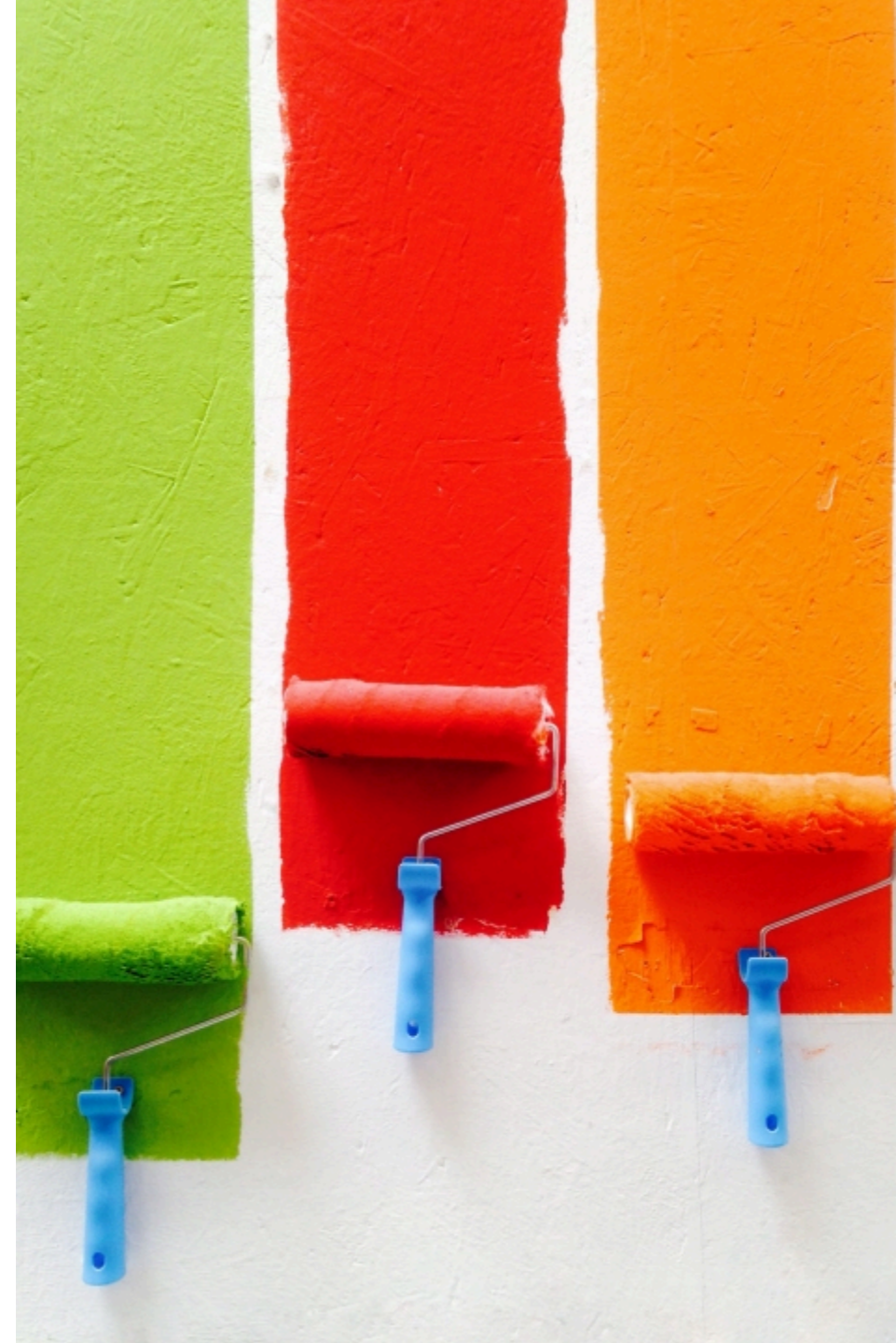
This enables multi-parameter distributions:

```
1 library(distributional)
2 dist_normal(mu = c(1, 3, -1), sigma = c(3, 2, 4))
```

```
<distribution[3]>
```

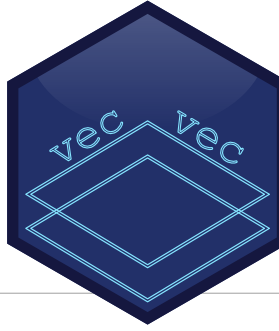
```
[1] N(1, 9)    N(3, 4)    N(-1, 16)
```

How can different shaped distributions coexist within the same vector? **That's trickier!**



Mixed-type vectors

`{vecvec}` creates vectors of vectors:



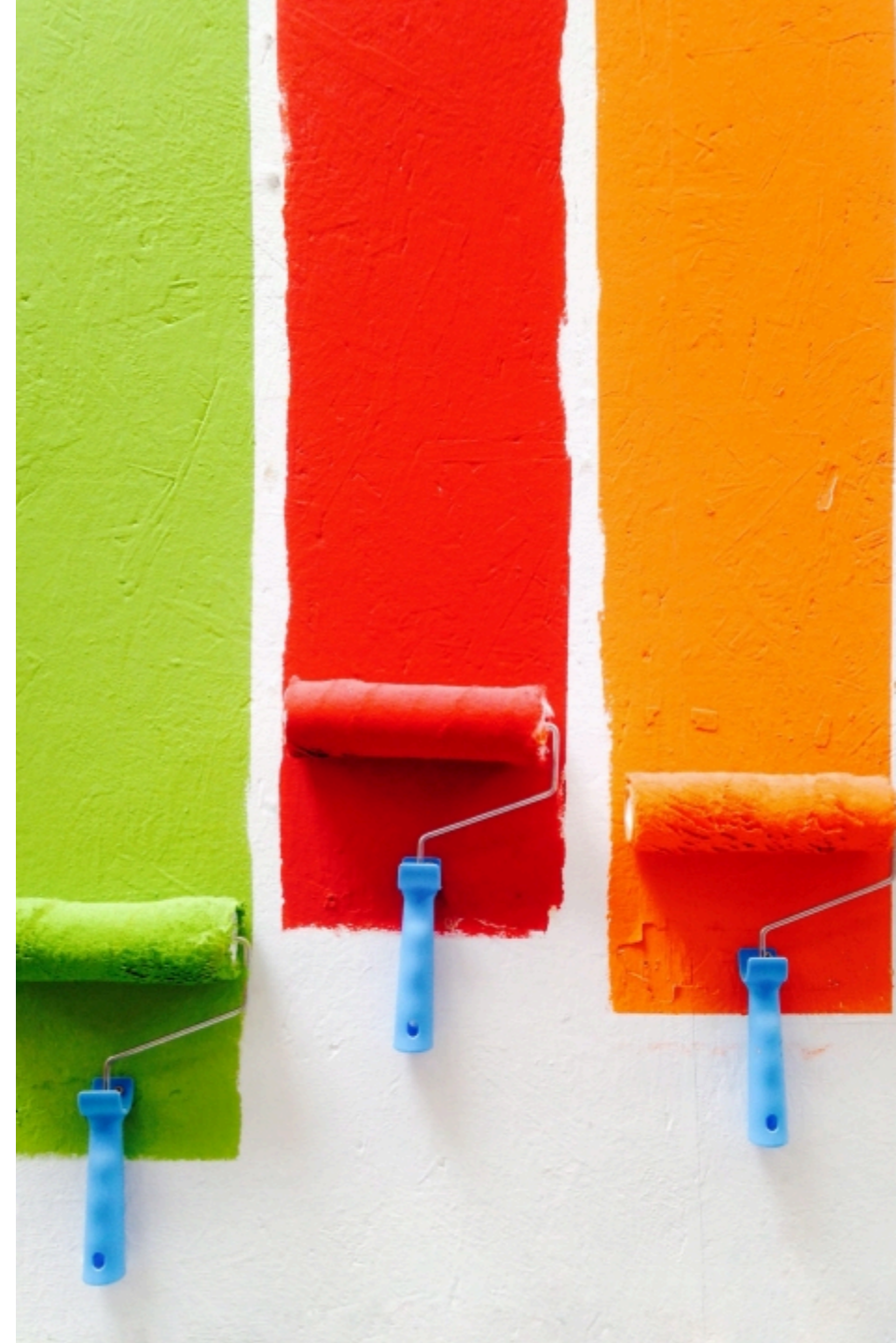
 **vecvec:** `vecvec()`

These vectors mix different vector types together.

```
1 library(vecvec)
2 vecvec(c(1,2), c("a","b"), c(TRUE, FALSE))
```

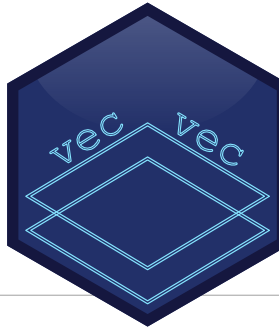
```
<vecvec[6]>
```

```
[1] 1      2      a      b      TRUE FALSE
```



Mixed-type vectors

`{vecvec}` creates vectors of vectors:



 **vecvec:** `vecvec()`

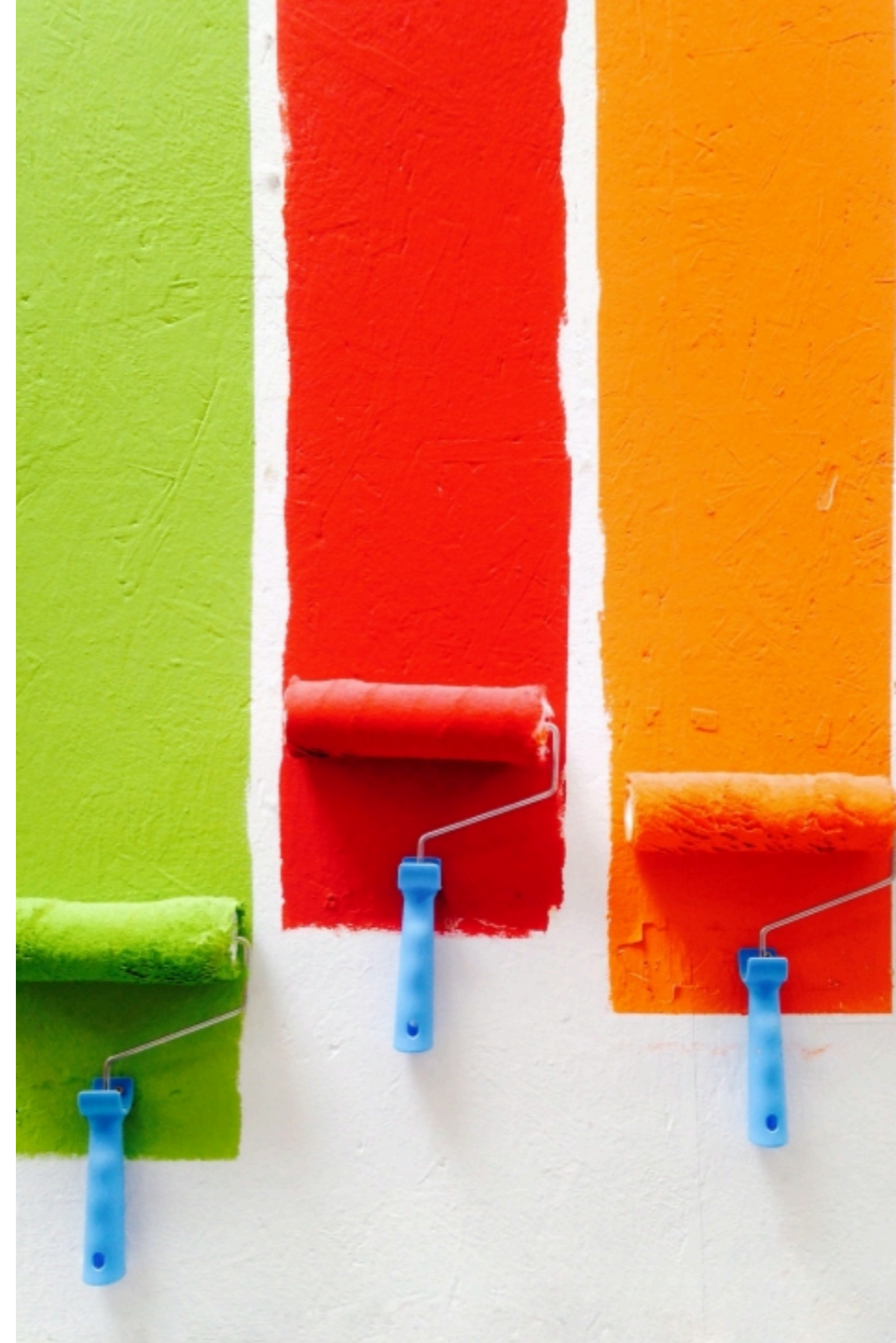
These vectors mix different vector types together.

```
1 library(vecvec)
2 vecvec(c(1,2), c("a","b"), c(TRUE, FALSE))
```

```
<vecvec[6]>
[1] 1      2      a      b      TRUE FALSE
```

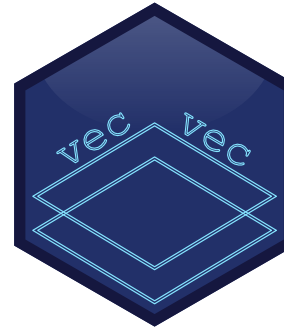
 **A *bad* idea**

Almost always, this causes more problems than it solves.



Mixed-type vectors

These vectors are essentially clever indexing of vectors in lists.

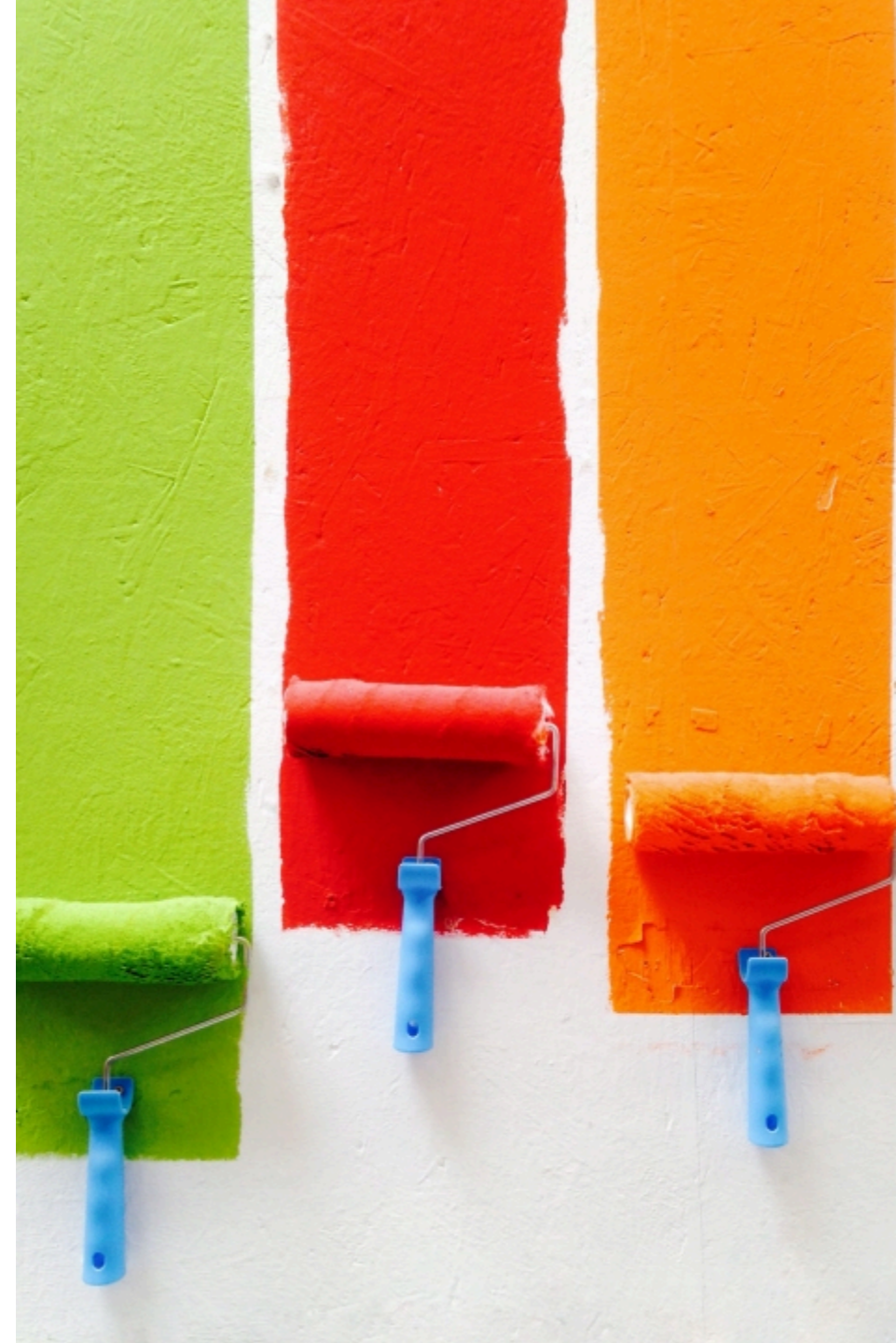


The **structure** of vecvec

Conceptually similar to `factor()` but indexed over list values.

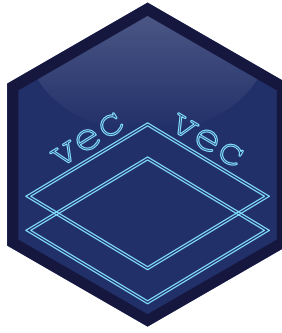
```
1 library(vecvec)
2 str(
3   vecvec(c(1,2), c("a","b"), c(TRUE, FALSE))
4 )
```

```
<vecvec::vecvec> int [1:6] 1 2 3 4 5 6
@ x:List of 3
.. $ : num [1:2] 1 2
.. $ : chr [1:2] "a" "b"
.. $ : logi [1:2] TRUE FALSE
```



Mixed-type *arrays*

The shape (dimensions) of the indices are the shape of the `vecvec` vector.

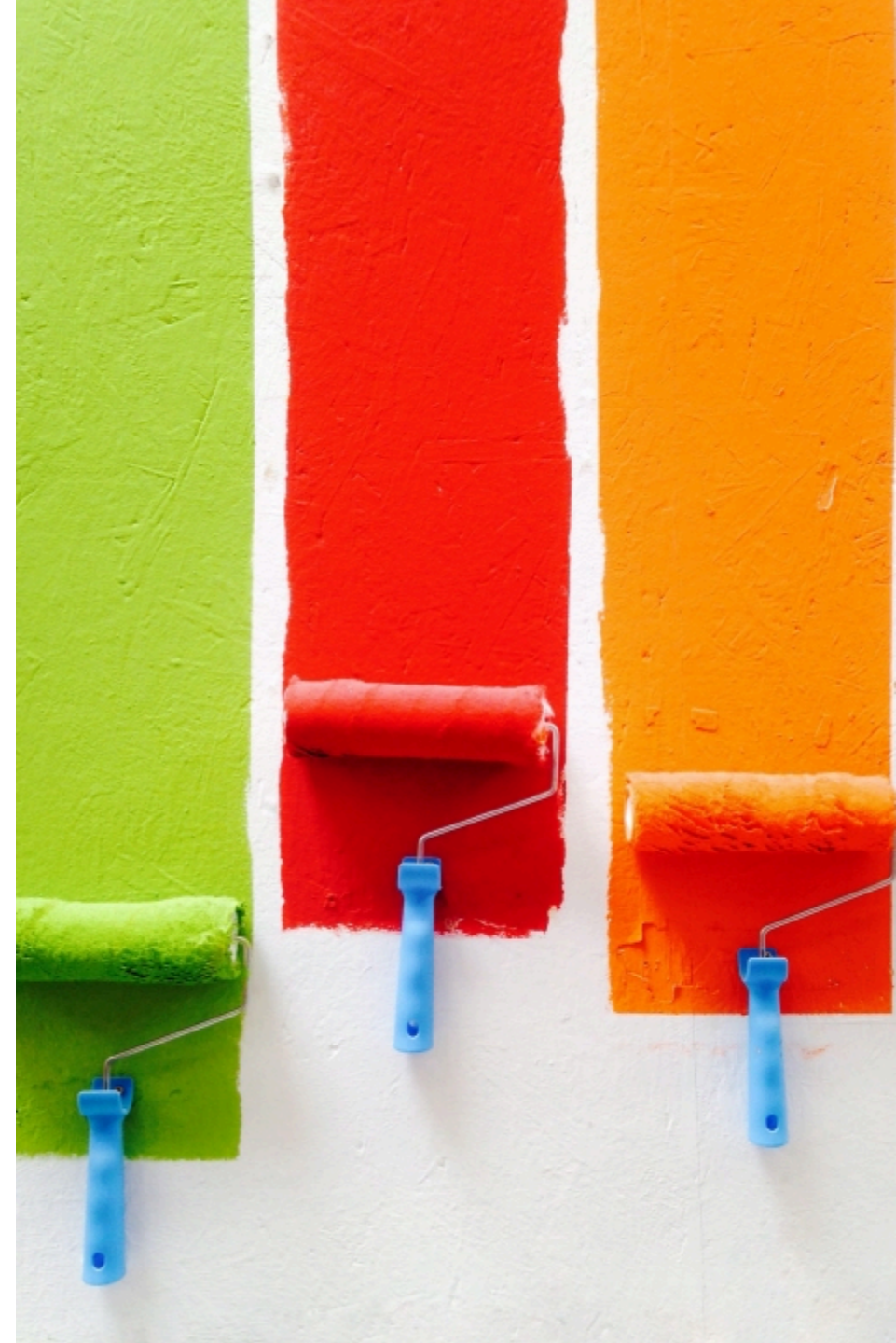


1 2 3 4 Multi-dimensional

Unlike many data types, `vecvec` can be matrices or arrays.

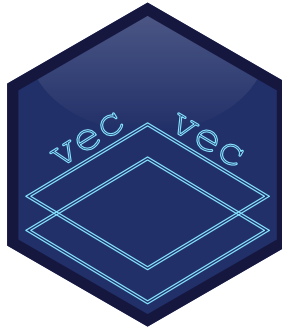
```
1 library(vecvec)
2 array(
3   vecvec(c(1,2), c("a","b"), c(TRUE, FALSE)),
4   dim = c(3, 2)
5 )
```

```
<vecvec[3,2]>
      [,1] [,2]
[1,] 1    b
[2,] 2    TRUE
[3,] a    FALSE
```



Mixed-type operations

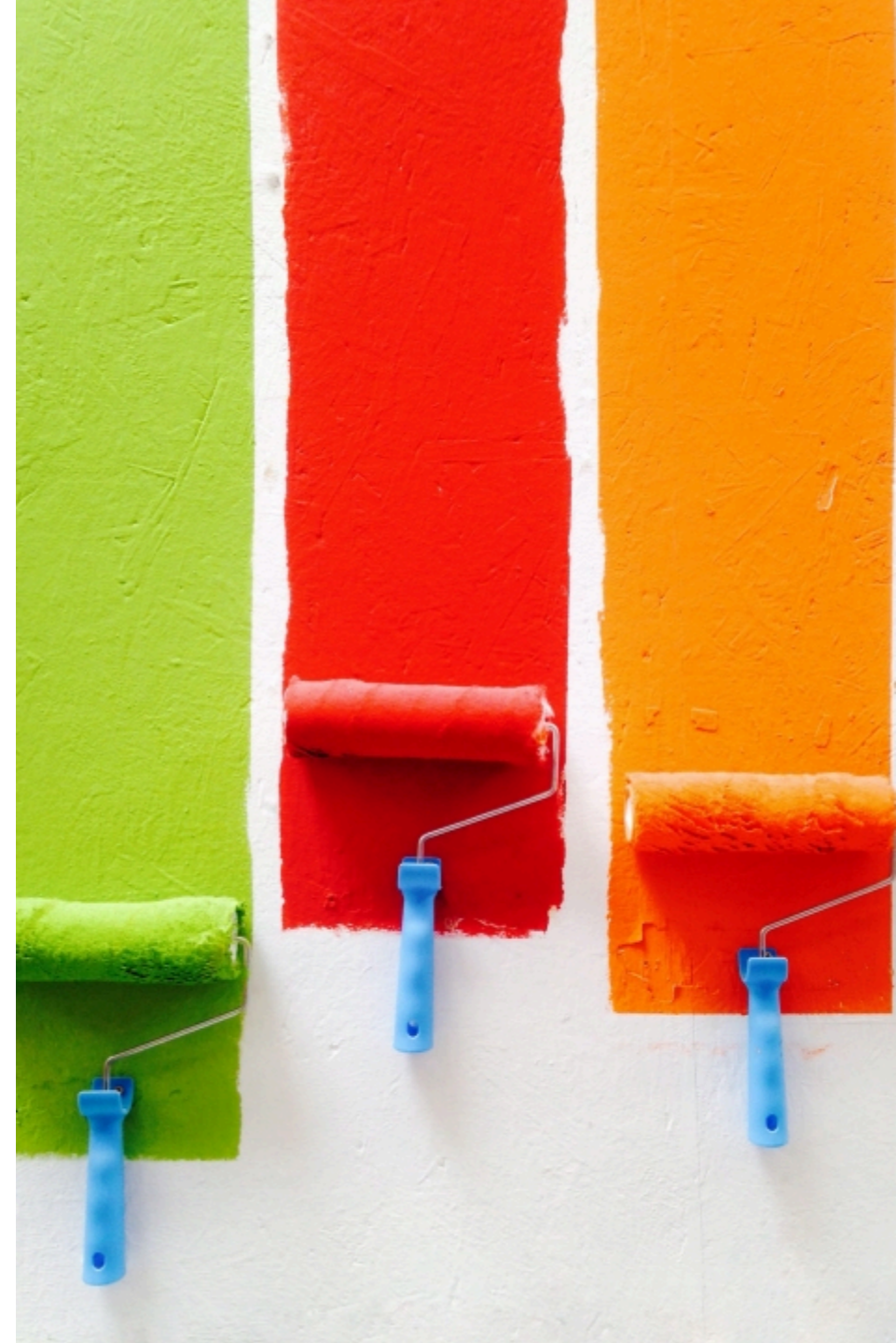
Operations apply to each component vector (fast and vectorised).



Analysis across vectors

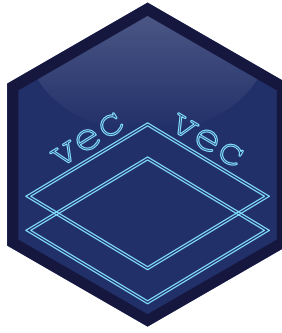
```
1 library(vecvec)
2 vecvec(c(1,2), c("a","b"), c(TRUE, FALSE)) == 1L
```

```
[1] TRUE FALSE FALSE FALSE TRUE FALSE
```



Mixed-type operations

Operations apply to each component vector (fast and vectorised).



Analysis across vectors

```
1 library(vecvec)
2 vecvec(c(1,2), c("a","b"), c(TRUE, FALSE)) == 1L
```

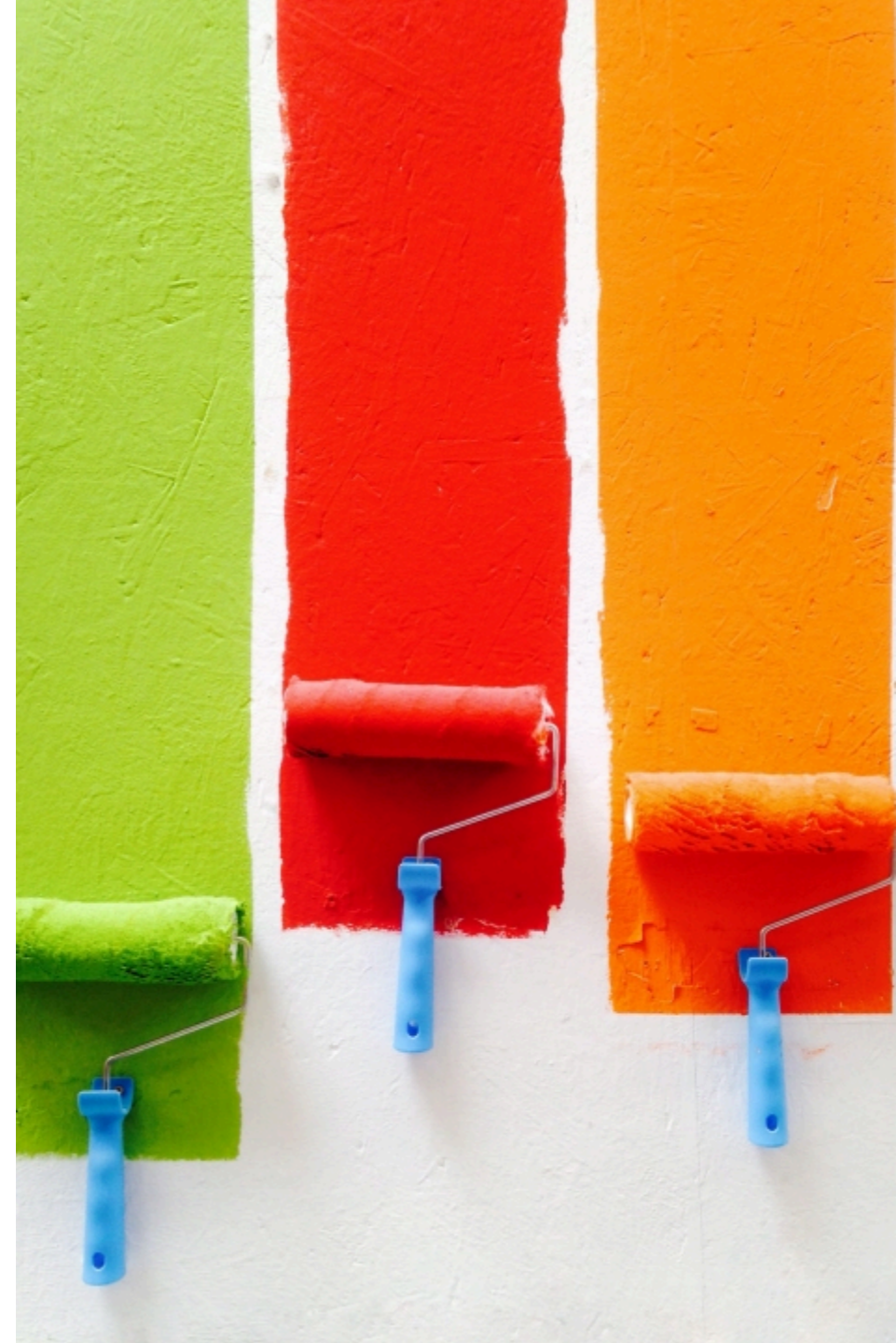
```
[1] TRUE FALSE FALSE FALSE TRUE FALSE
```



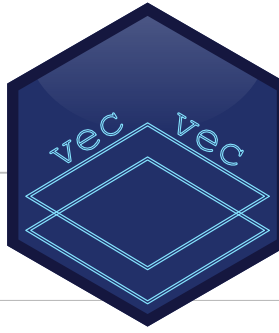
Recycling and casting

Operations respect safe recycling rules.

Casting rules match underlying vector types.

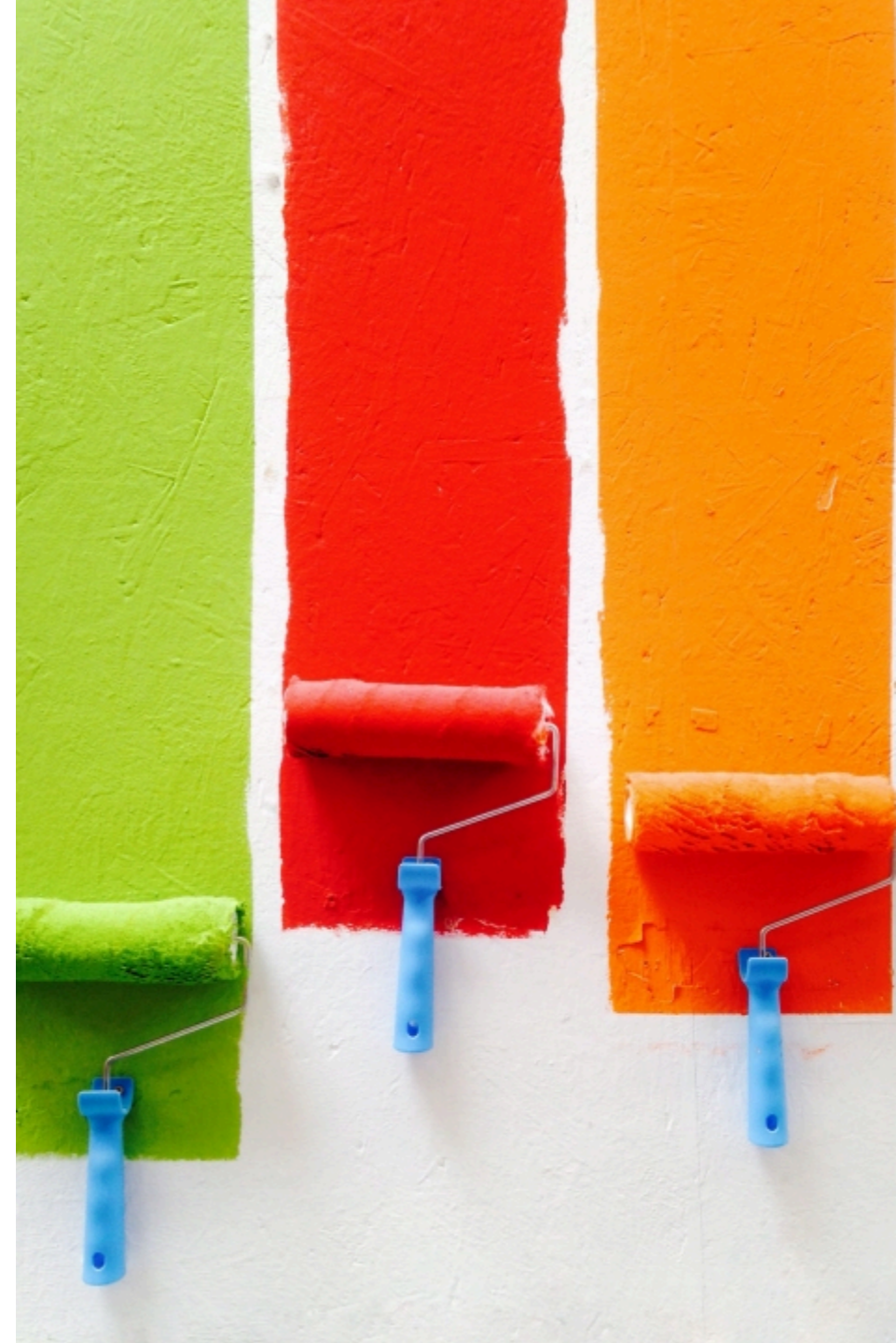


Mixed-type manipulation

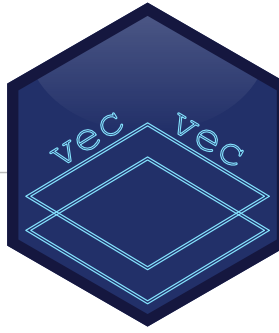


```
1 library(tidyr)
2 household
```

```
# A tibble: 5 × 5
  family dob_child1 dob_child2 name_child1 name_child2
  <int> <date>      <date>      <chr>      <chr>
1     1 1998-11-26 2000-01-29 Susan      Jose
2     2 1996-06-22 NA          Mark      <NA>
3     3 2002-07-11 2004-04-05 Sam       Seth
4     4 2004-10-10 2009-08-27 Craig     Khai
5     5 2000-12-05 2005-02-28 Parker    Gracie
```



Mixed-type manipulation

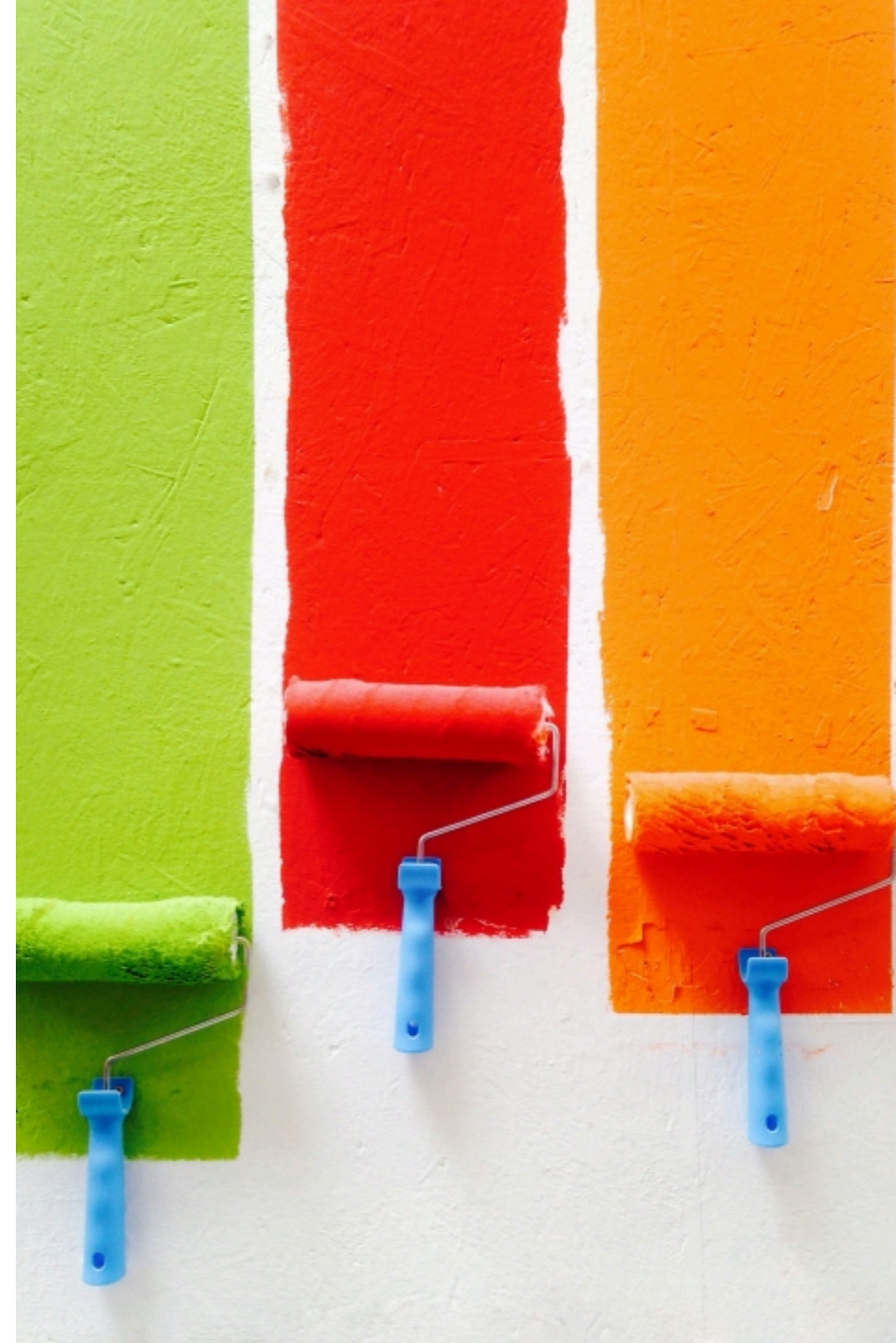


```
1 library(tidyr)
2 household |>
3   pivot_longer(
4     everything(),
5     values_transform = vecvec::vecvec
6   )
```

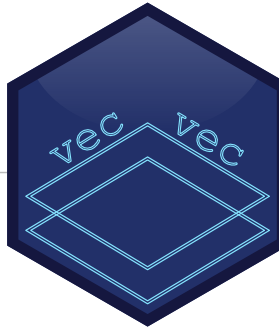
A tibble: 25 × 2

	name	value
	<chr>	<vecvec>
1	family	1
2	dob_child1	2
3	dob_child2	3
4	name_child1	4
5	name_child2	5
6	family	1998-11-26
7	dob_child1	1996-06-22
8	dob_child2	2002-07-11
9	name_child1	2004-10-10
10	name_child2	2000-12-05

i 15 more rows



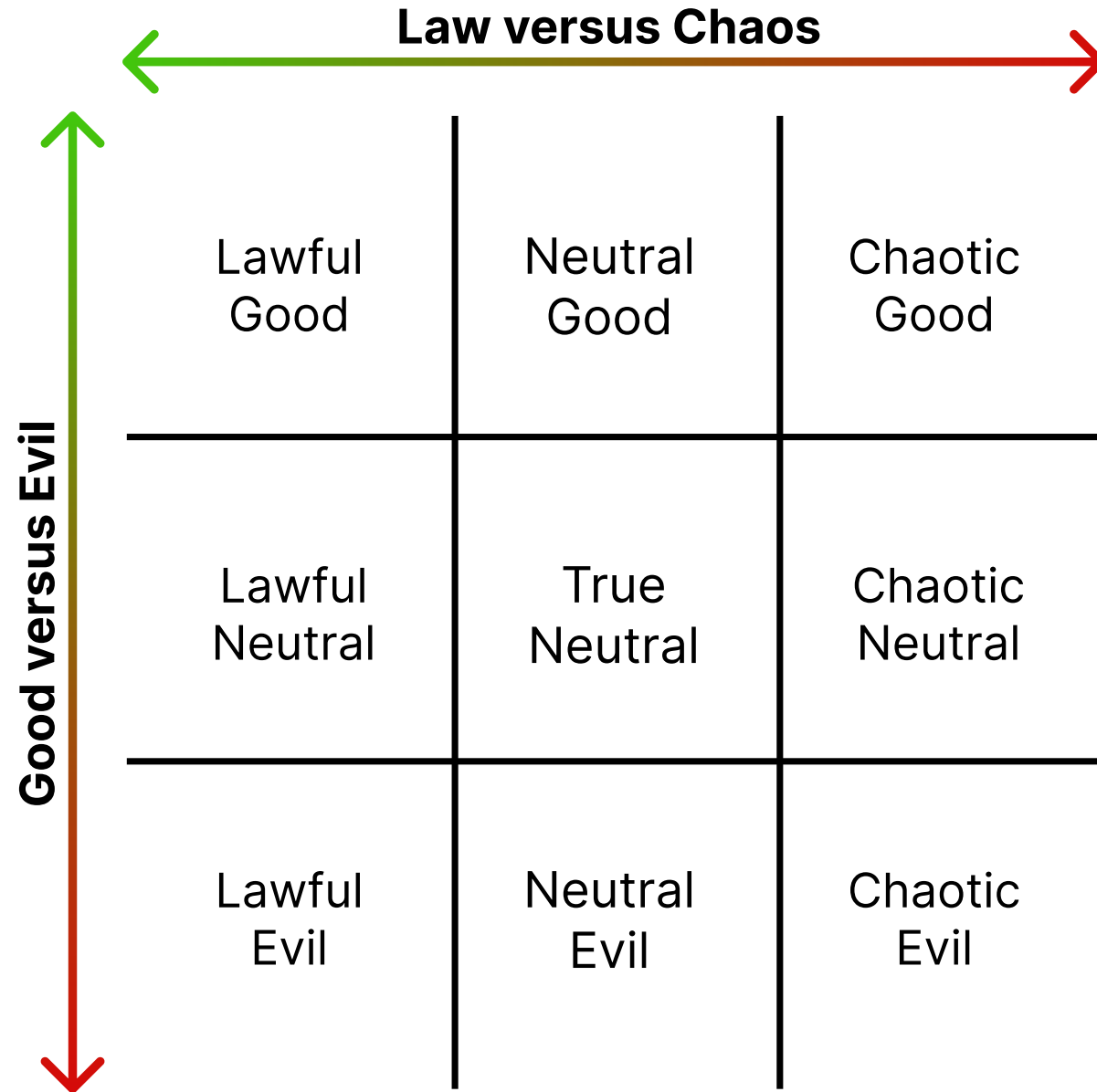
Mixed-type manipulation

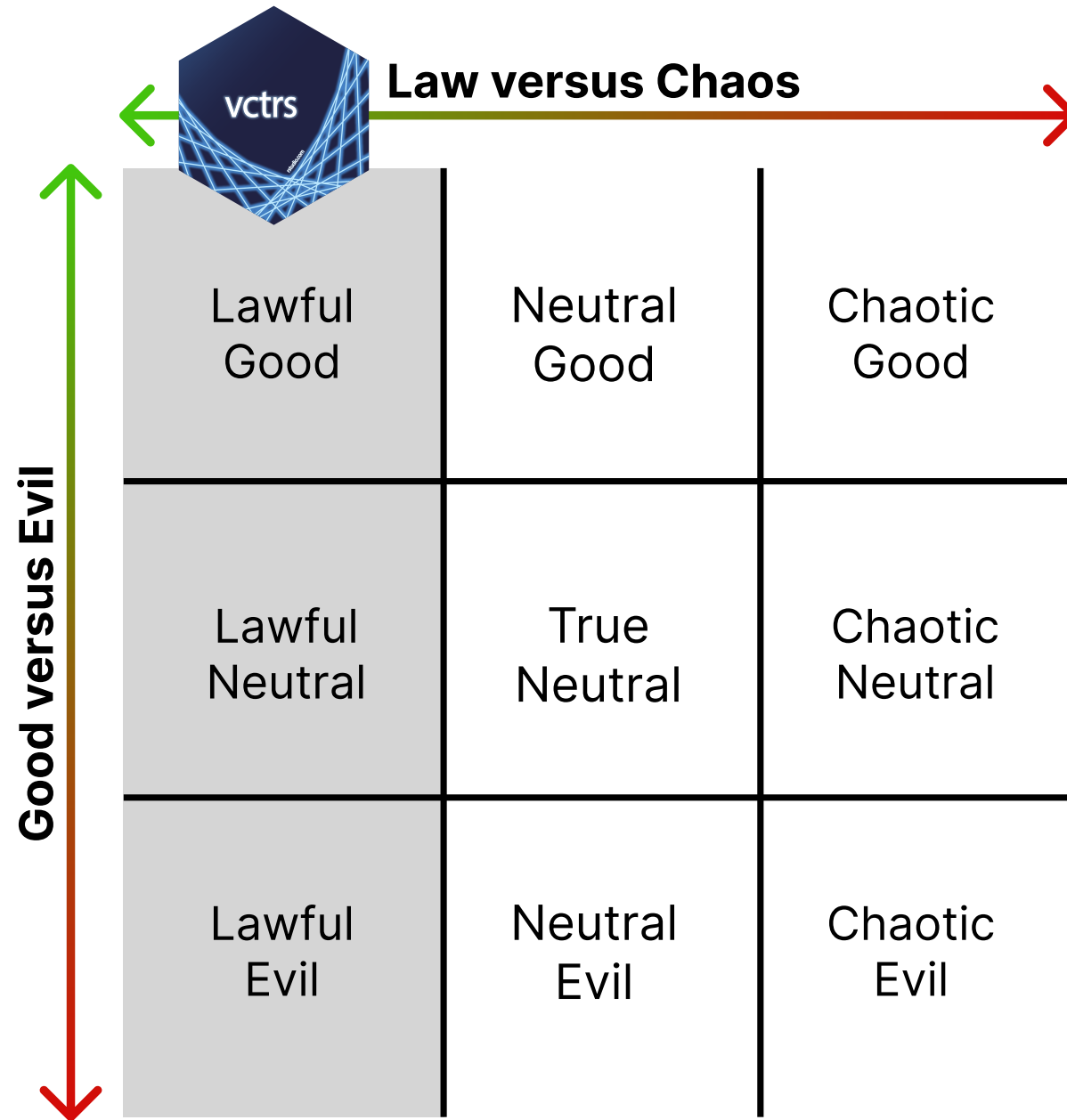


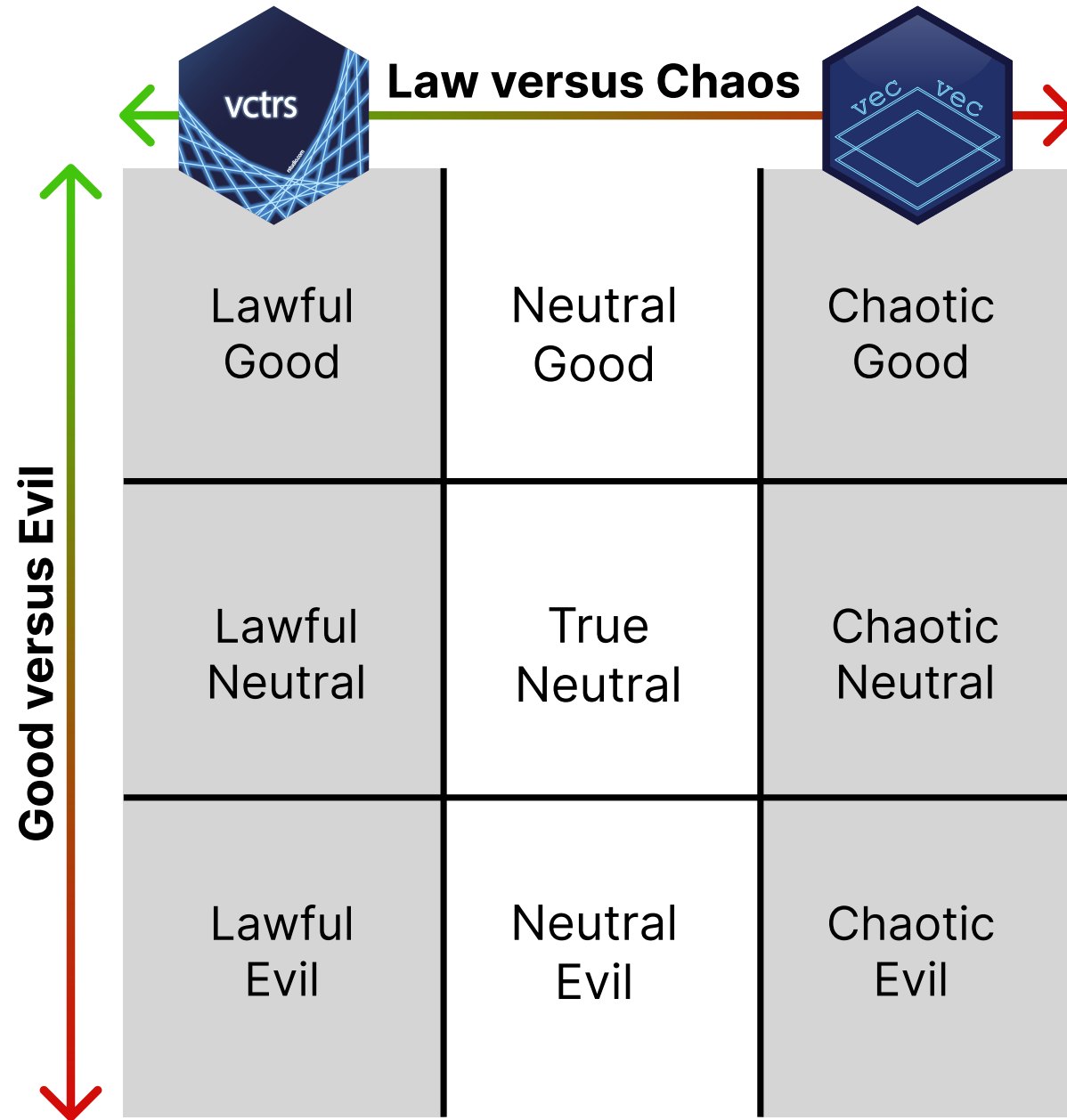
```
1 library(tidyr)
2 household |>
3   pivot_longer(
4     everything(),
5     values_transform = vecvec::vecvec
6   )
```

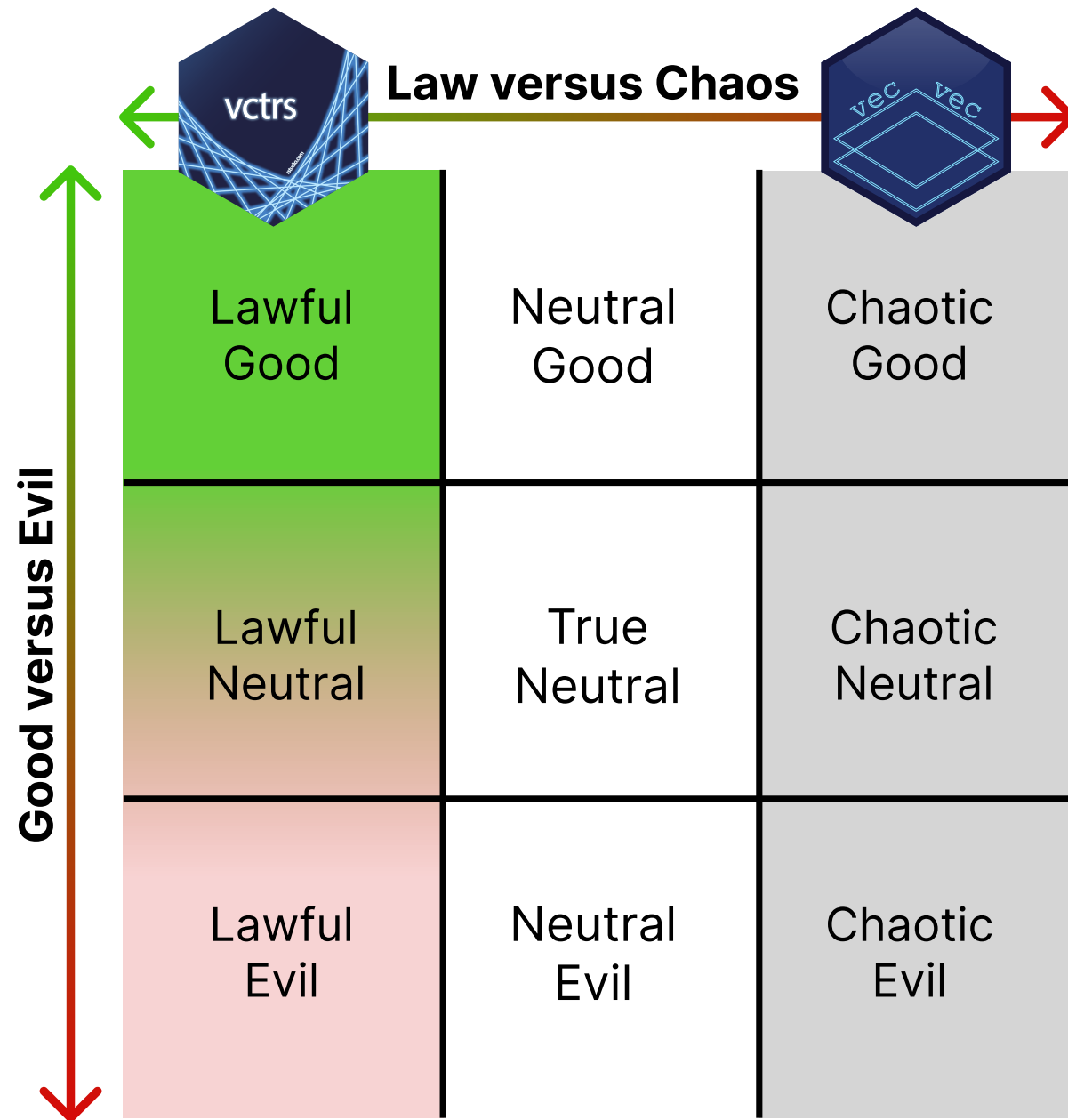
```
# A tibble: 25 × 2
  name      value
  <chr>    <vecvec>
1 family      1
2 dob_child1  2
3 dob_child2  3
4 name_child1 4
5 name_child2 5
6 family      1998-11-26
7 dob_child1  1996-06-22
8 dob_child2  2002-07-11
9 name_child1 2004-10-10
10 name_child2 2000-12-05
# i 15 more rows
```

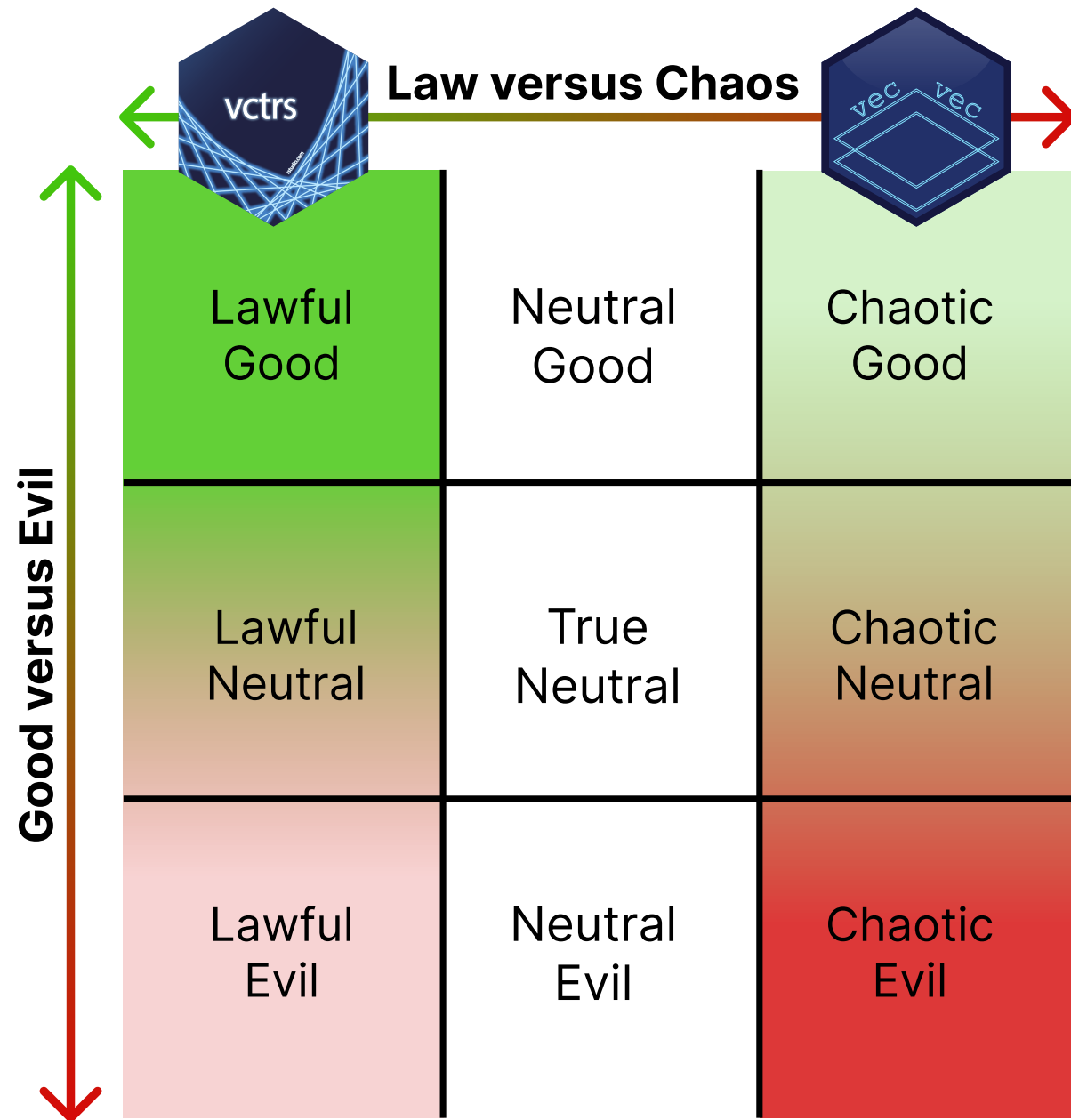








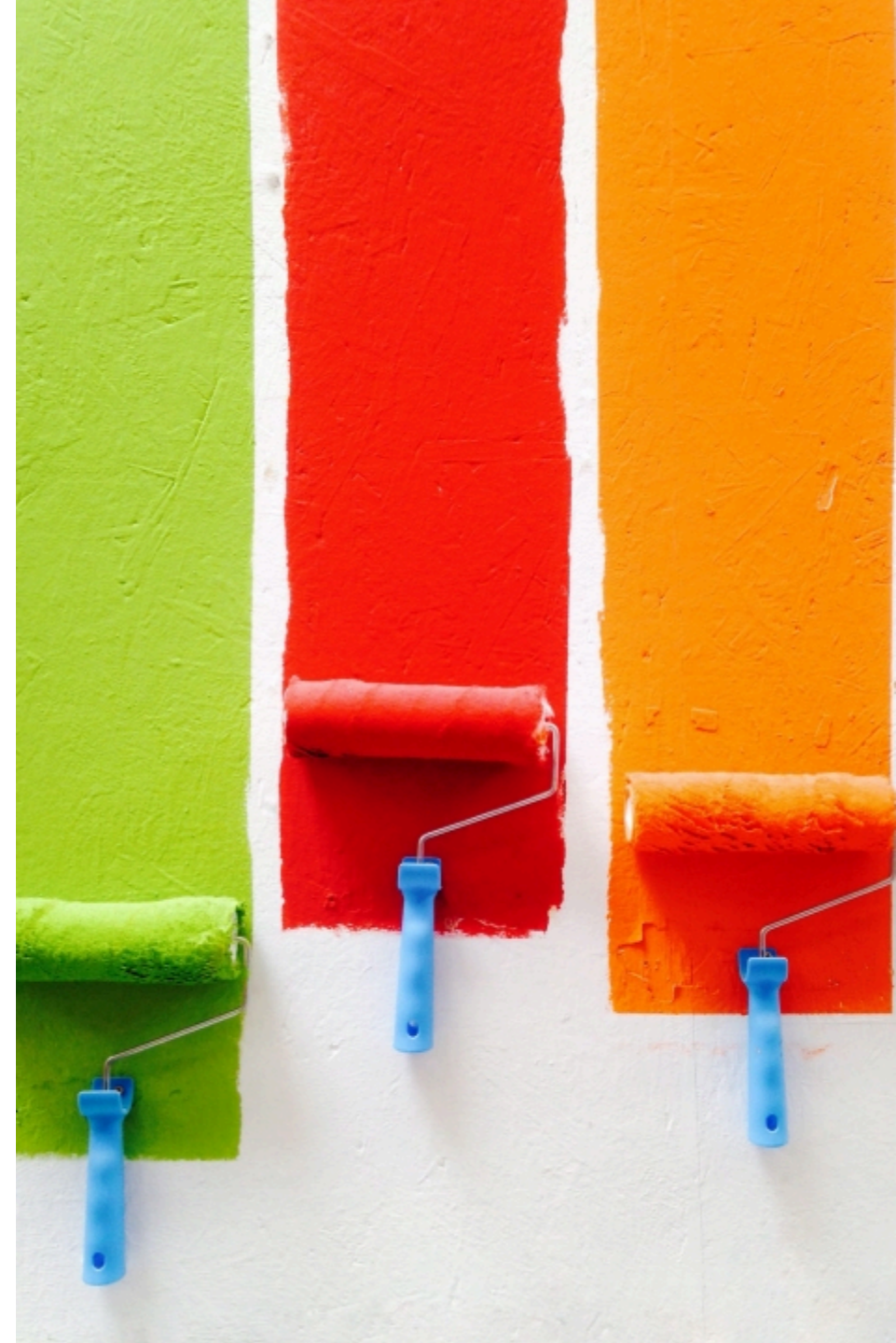




Mixed-type *semantic* vectors

`{vecvec}` is perhaps only useful for mixed-type vectors that share **common semantics**.

Suitable semantic data-types include:



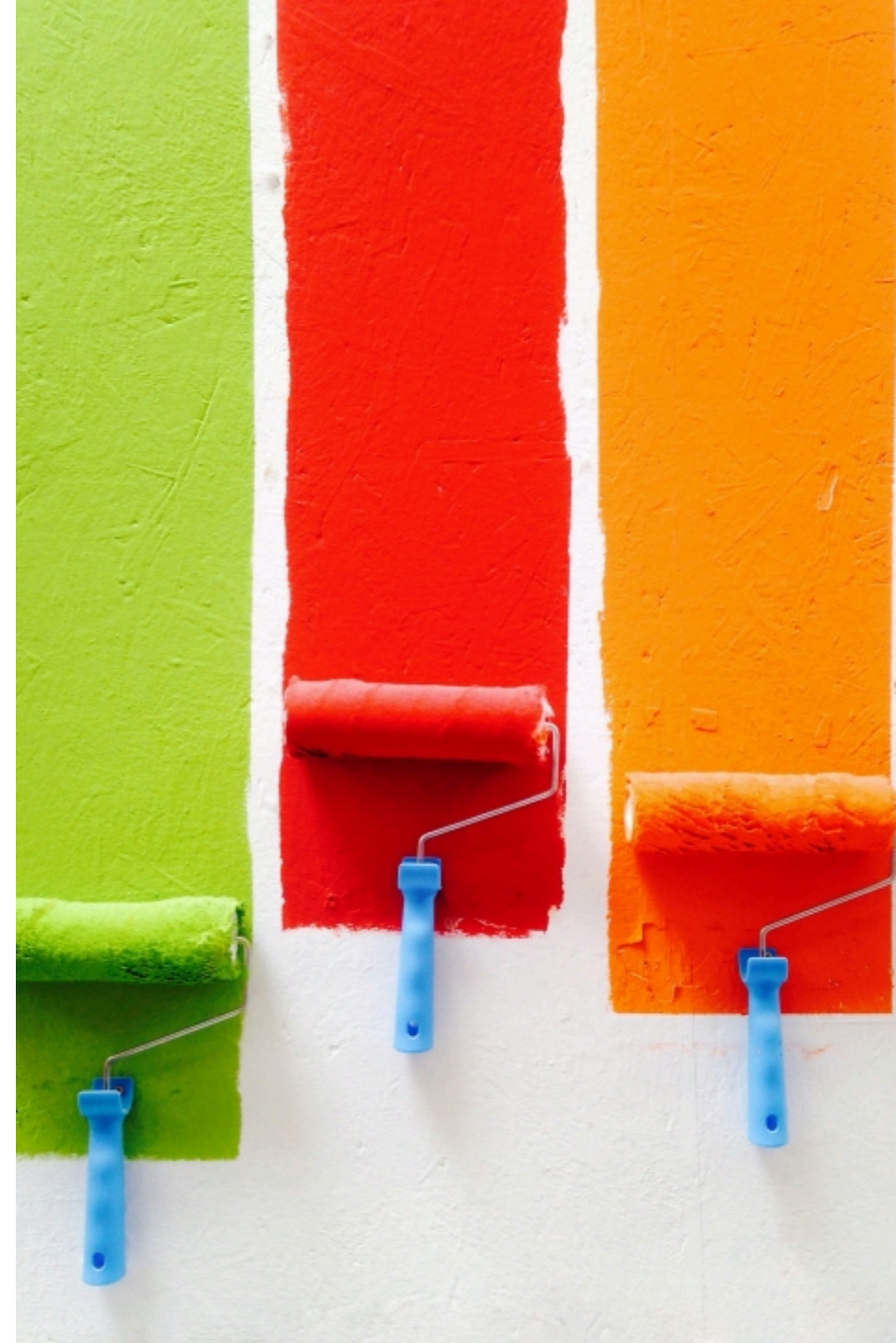
Mixed-type *semantic* vectors

$\{\text{vec} \text{vec}\}$ is perhaps only useful for mixed-type vectors that share **common semantics**.

Suitable semantic data-types include:



Distributional (different shapes)



Mixed-type *semantic* vectors

`{vecvec}` is perhaps only useful for mixed-type vectors that share **common semantics**.

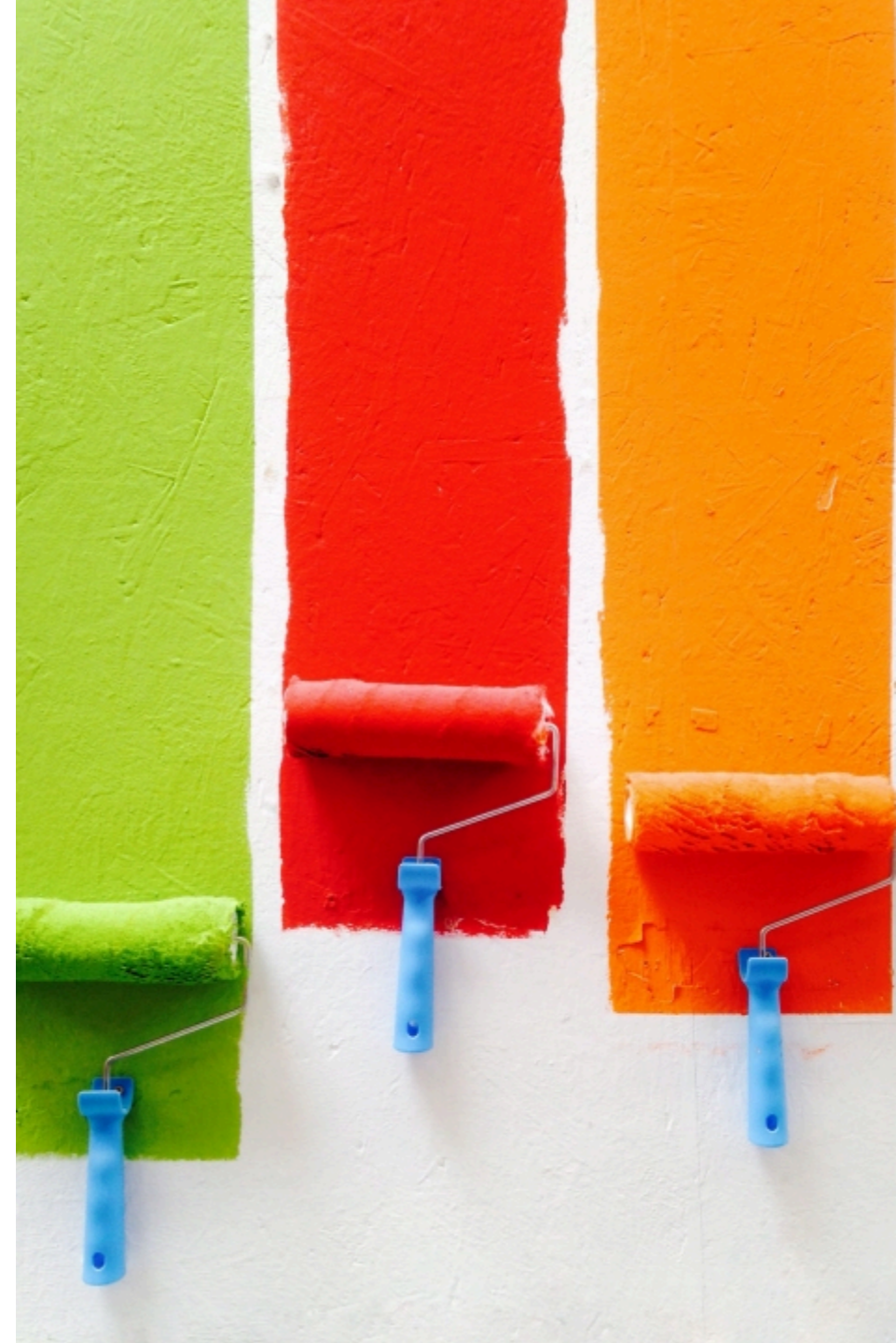
Suitable semantic data-types include:



Distributional (different shapes)



Temporal (different chronons / calendars)



Mixed-type *semantic* vectors

`{vecvec}` is perhaps only useful for mixed-type vectors that share **common semantics**.

Suitable semantic data-types include:



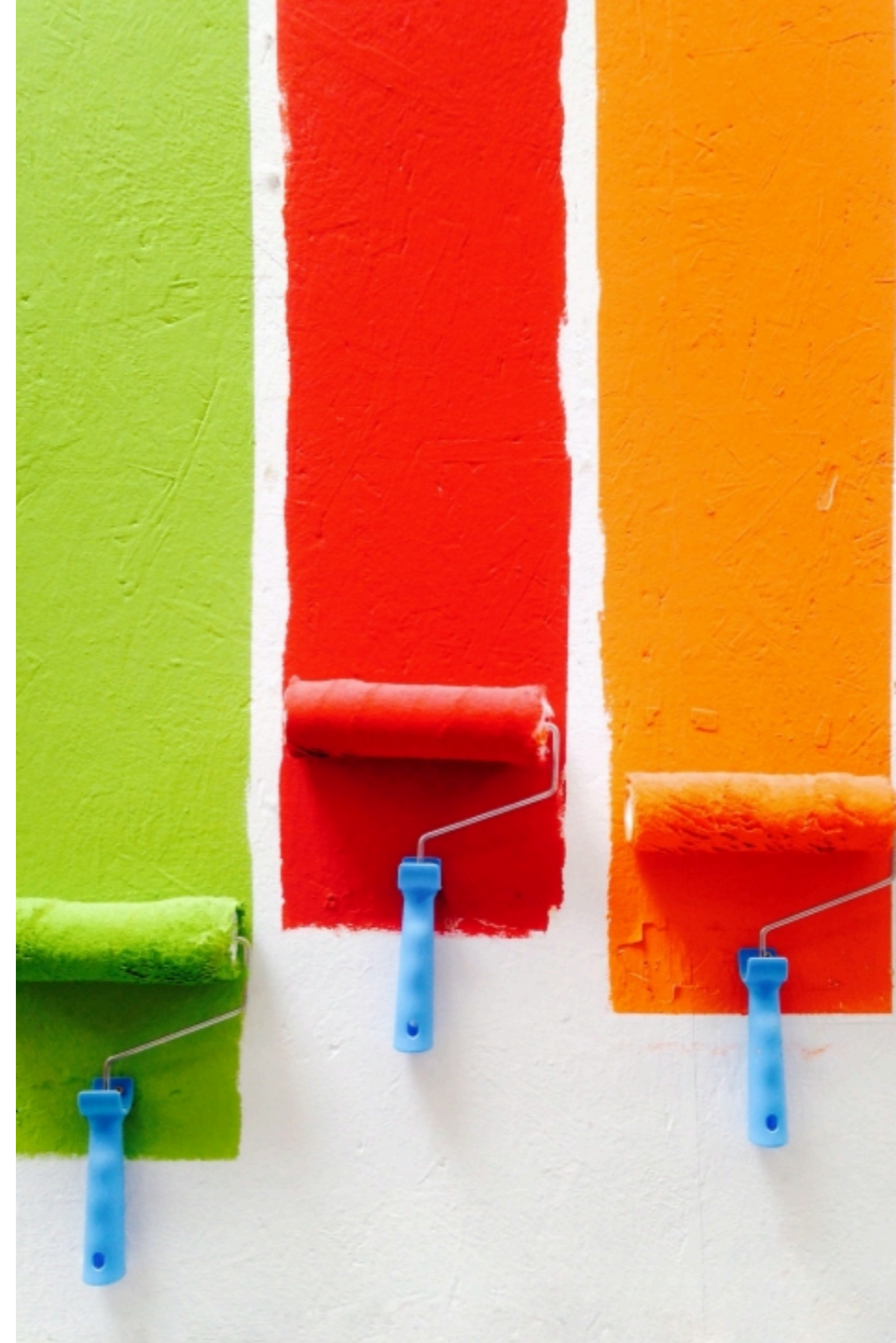
Distributional (different shapes)



Temporal (different chronons / calendars)



Spatial (different geometries)



Mixed-type *semantic* vectors

`{vecvec}` is perhaps only useful for mixed-type vectors that share **common semantics**.

Suitable semantic data-types include:



Distributional (different shapes)



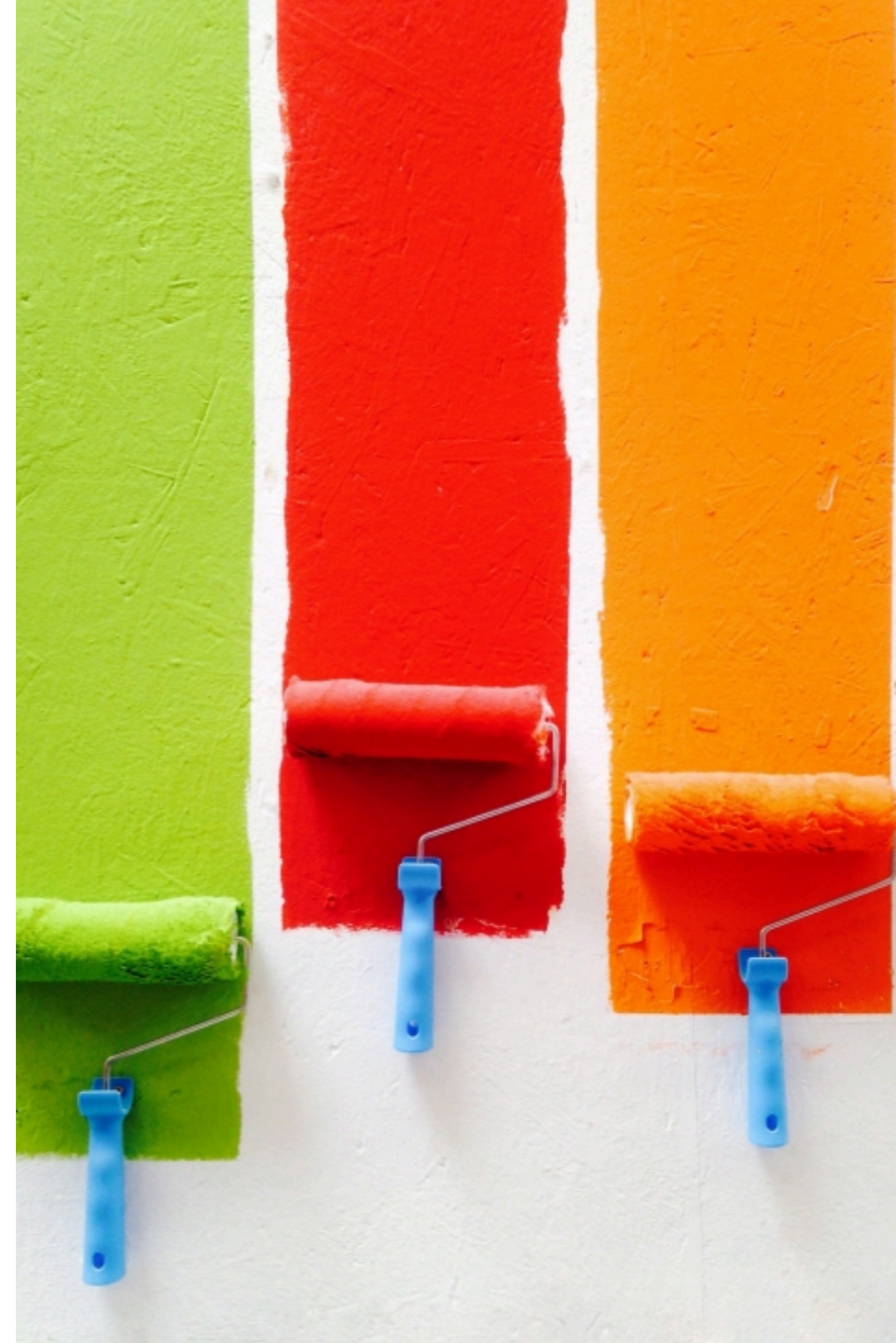
Temporal (different chronons / calendars)



Spatial (different geometries)

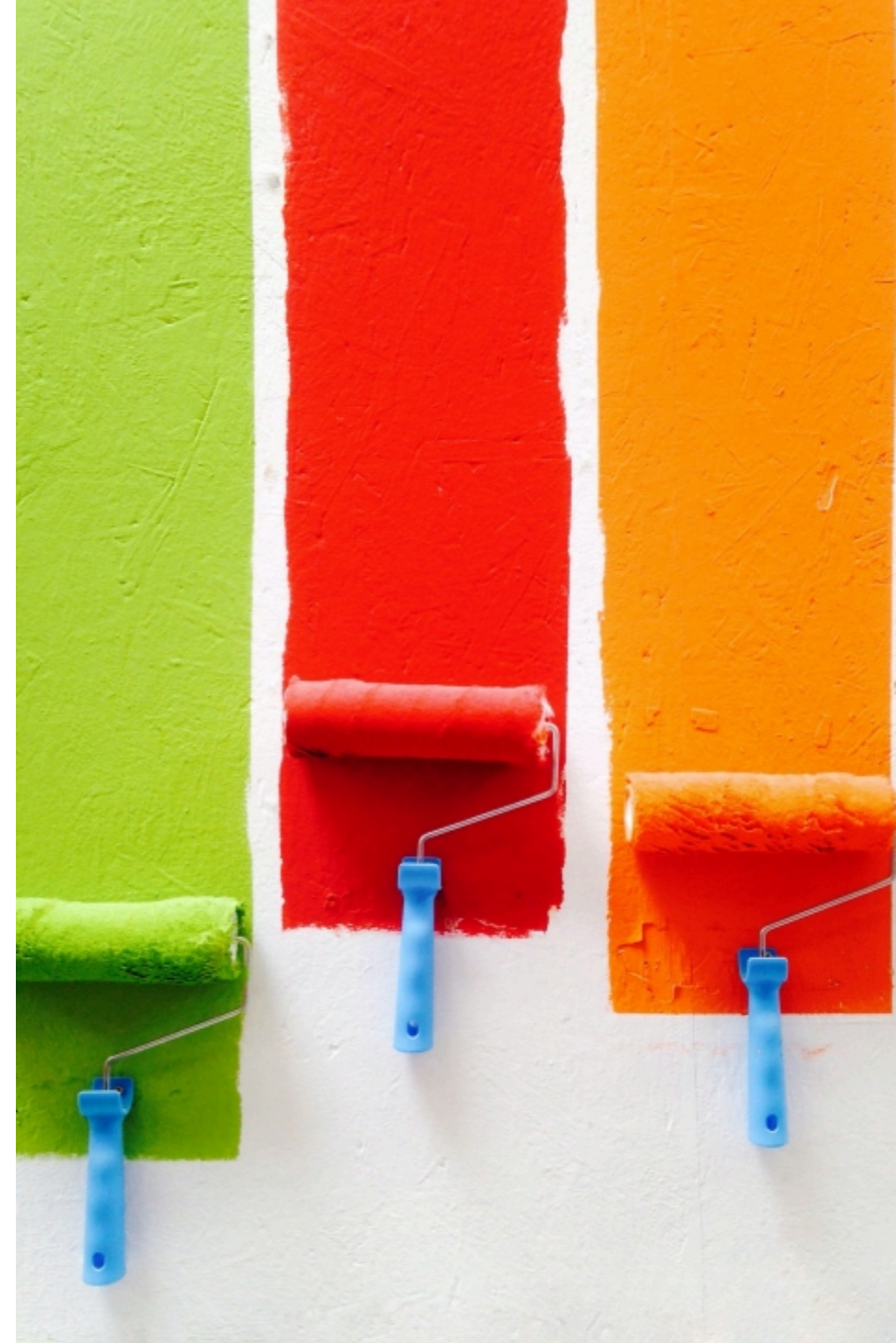


Preferential (different candidates)



Mixed-type *semantic* vectors

`{vecvec}` allows us to efficiently combine distributions of **different shape**.



Mixed-type *semantic* vectors



`{vecvec}` allows us to efficiently combine distributions of **different shape**.



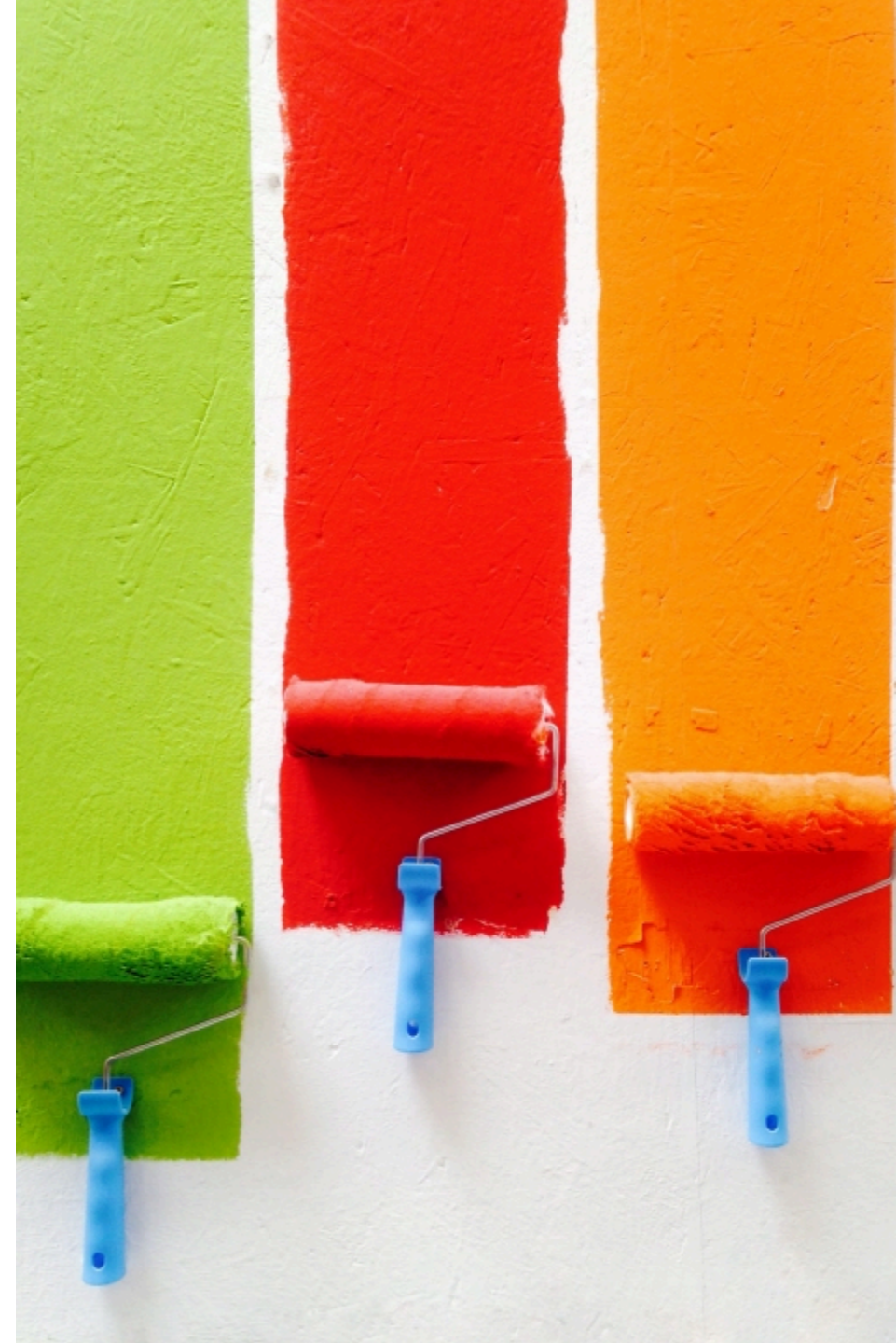
Mixed distributions

Putting it all together...

```
1 library(distributional)
2 dist <- c(
3   dist_poisson(c(4, 2, 6)),
4   dist_normal(mu = c(1, 3, -1), sigma = c(3, 2, 4))
5 )
6 dist
```

```
<distribution[6]>
```

```
[1] Pois(4)    Pois(2)    Pois(6)    N(1, 9)    N(3, 4)    N(-1, 16)
```



Mixed-type *semantic* vectors



Mixed distributions

Putting it all together...

```
1 library(distributional)
2 dist <- c(
3   dist_poisson(c(4, 2, 6)),
4   dist_normal(mu = c(1, 3, -1), sigma = c(3, 2, 4))
5 )
6 dist
```

```
<distribution[6]>
```

```
[1] Pois(4)    Pois(2)    Pois(6)    N(1, 9)    N(3, 4)    N(-1, 16)
```

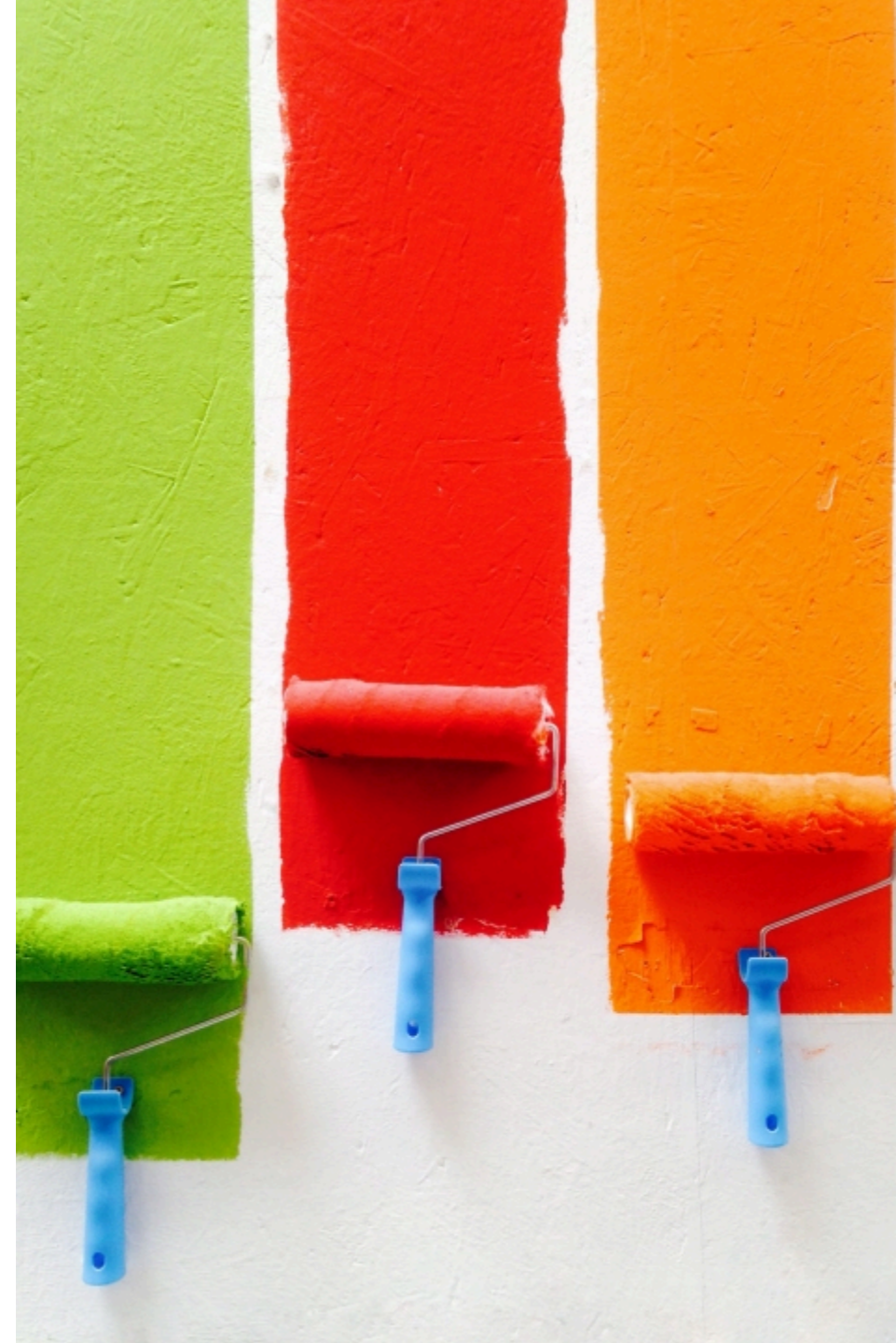


Vectorised statistics across distributions

Since distributions share common semantics, we can apply statistical operations across them.

```
1 density(dist, at = 1)
```

```
[1] 0.07326256 0.27067057 0.01487251 0.13298076 0.12098536
0.08801633
```





Better distributions for R

The **distributional** package makes it simpler to create and use distributions in R.



Creating distributions

All distributions in the package start with `dist_*()`.

```
1 dist <- c(  
2   dist_poisson(c(4, 2, 6)),  
3   dist_normal(mu = c(1, 3, -1), sigma = c(3, 2, 4))  
4 )
```

The package currently provides:

- 25 continuous distributions,
- 9 discrete distributions,
- p/d/q/r distributions via `dist_wrap()`,
- sample, degenerate and percentile distributions.



Vectorised distribution statistics

The p/d/q/r functions have more descriptive alternatives:

- `p->cdf()`: The CDF
- `d->density()`: The density (PDF)
- `q->quantile()`: The quantile
- `r->generate()`: Random samples



Distributional operations

These functions are the same for any distribution.



Vectorised distribution statistics

💡 Other operations

There are many more statistics than p/d/q/r.

- `log_likelihood()/likelihood()`
- `hilo()`
- `hdr()`
- `support()`
- `mean()`
- `variance()/covariance()`
- `skewness()`
- `kurtosis()`



Vectorised distribution statistics

Use distribution vectors in data frames!

```
1 library(dplyr)
2 tibble(
3   dist = c(
4     dist_poisson(c(4, 2, 6)),
5     dist_normal(mu = c(1, 3, -1), sigma = c(3, 2, 4))
6   )
7 )
```

```
# A tibble: 6 × 1
```

```
  dist
```

```
  <dist>
```

```
1  Pois(4)
```

```
2  Pois(2)
```

```
3  Pois(6)
```

```
4  N(1, 9)
```

```
5  N(3, 4)
```

```
6 N(-1, 16)
```



Vectorised distribution statistics

Perform statistics alongside distributions.

```
1 library(dplyr)
2 tibble(
3   dist = c(
4     dist_poisson(c(4, 2, 6)),
5     dist_normal(mu = c(1, 3, -1), sigma = c(3, 2, 4))
6   )
7 ) |>
8 mutate(
9   mean = mean(dist), var = variance(dist),
10  pdf = density(dist, 1), cdf = cdf(dist, 1)
11 )
```

A tibble: 6 × 5

	dist	mean	var	pdf	cdf
	<dist>	<dbl>	<dbl>	<dbl>	<dbl>
1	Pois(4)	4	4	0.0733	0.0916
2	Pois(2)	2	2	0.271	0.406
3	Pois(6)	6	6	0.0149	0.0174
4	N(1, 9)	1	9	0.133	0.5
5	N(3, 4)	3	4	0.121	0.159
6	N(-1, 16)	-1	16	0.0880	0.691

Mixed-type semantic vectors



Probabilistic



```
1 library(distributional)
2 c(dist_normal(1, 2), dist_poisson(3))
```

```
<distribution[2]>
[1] N(1, 4) Pois(3)
```

Mixed-type semantic vectors



Probabilistic



Temporal



```
1 library(mixtime)
2 today <- Sys.Date()
3 c(year(today), yearquarter(today), yearmonth(today))
```

```
<mixtime[3]>
```

```
[1] 2026      2026 Q3   2026 Jul
```


Mixed-type semantic vectors



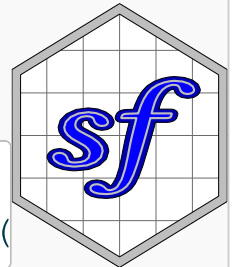
Probabilistic



Temporal



Spatial



```
1 library(sf)
2 c(st_point(c(1, 1)), st_linestring(rbind(c(
```

Geometry set for 3 features

Geometry type: GEOMETRY

Dimension: XY

Bounding box: xmin: 0 ymin: 0 xmax: 4 ymax: 4

Geodetic CRS: WGS 84

Mixed-type semantic vectors



Probabilistic



Temporal



Spatial



Preferential



```
1 library(prefio)
2 read_preflib("00004 - netflix/00004-0000013")
```

```
[1] [Beverly Hills Cop > Mean Girls > Mission:
Impossible II > The Mummy Returns]
[2] [Mean Girls > Beverly Hills Cop > Mission:
Impossible II > The Mummy Returns]
[3] [Beverly Hills Cop > Mean Girls > The
Mummy Returns > Mission: Impossible II]
```


Mixed-type semantic vectors



Probabilistic



Temporal



Spatial



Preferential

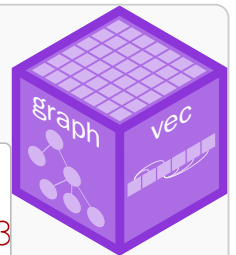


Graph

```
1 library(graphvec)
2 edge_vec(from = c(1L, 2L, 1L), to = c(2L, 3L, 1L))
```

```
<edge_vec[3]>
```

```
[1] [1:A]->[2:B] [2:B]->[3:C] [1:A]->[3:C]
```



Combining semantic vectors



Combining semantic vectors

-   Forecasting:

`{mixtime} + {distributional}`



Combining semantic vectors

-   Forecasting:

`{mixtime} + {distributional}`

-   Spatio-temporal data:

`{sf} + {mixtime}`

Combining semantic vectors

-   Forecasting:

`{mixtime} + {distributional}`

-   Spatio-temporal data:

`{sf} + {mixtime}`

-   Network modelling:

`{graphvec} + {distributional}`

Combining semantic vectors

-   Forecasting:

`{mixtime} + {distributional}`

-   Spatio-temporal data:

`{sf} + {mixtime}`

-   Network modelling:

`{graphvec} + {distributional}`

-    

Probabilistic-spatio-temporal-graph:

`{distributional} + {sf} +
{mixtime} + {graphvec}`

Combining semantic vectors

Forecasts from `{fable}` combines
`{mixtime}` and `{distributional}` vectors.



```
1 library(fable)
2 sunspots_tsbl |>
3   model(ARIMA(value)) |>
4   forecast(h = "10 years")
```

```
# A fable: 10 x 4 [1Y]
```

```
# Key:      .model [1]
```

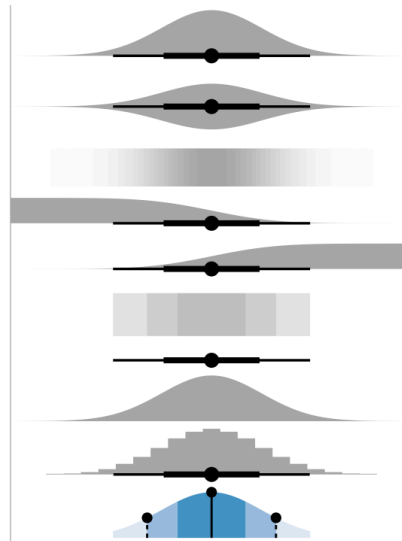
	.model <chr>	index <mixtime>	value <dist>	.mean <dbl>
1	ARIMA(value)	1989	N(145, 240)	145.
2	ARIMA(value)	1990	N(165, 582)	165.
3	ARIMA(value)	1991	N(161, 819)	161.
4	ARIMA(value)	1992	N(136, 915)	136.
5	ARIMA(value)	1993	N(98, 928)	98.3
6	ARIMA(value)	1994	N(62, 929)	61.8
7	ARIMA(value)	1995	N(38, 938)	37.9
8	ARIMA(value)	1996	N(33, 941)	33.4
9	ARIMA(value)	1997	N(48, 947)	48.5
10	ARIMA(value)	1998	N(77, 1006)	77.0



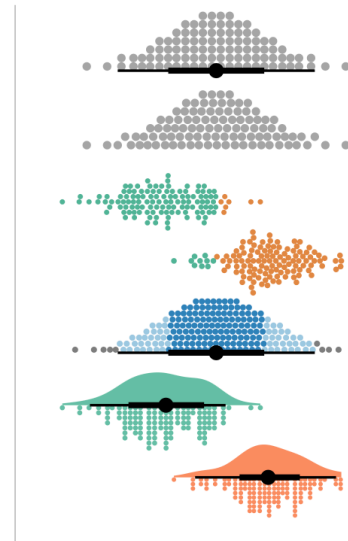
Visualising distributions

The `{ggdist}` and `{ggdibbler}` packages extend `{ggplot2}` with distributional graphics.

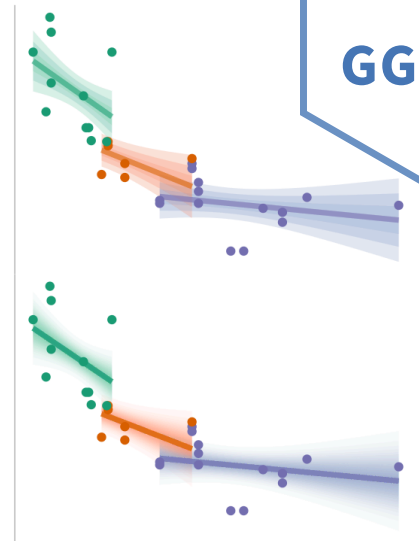
slabinterval



dotsinterval



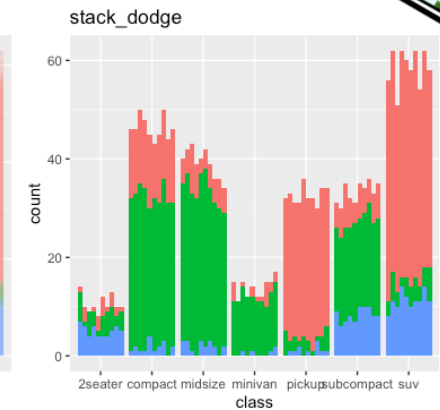
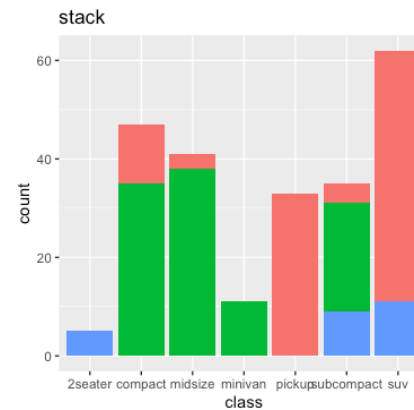
linerribbon



Some examples from the three main families of ggdist geometries

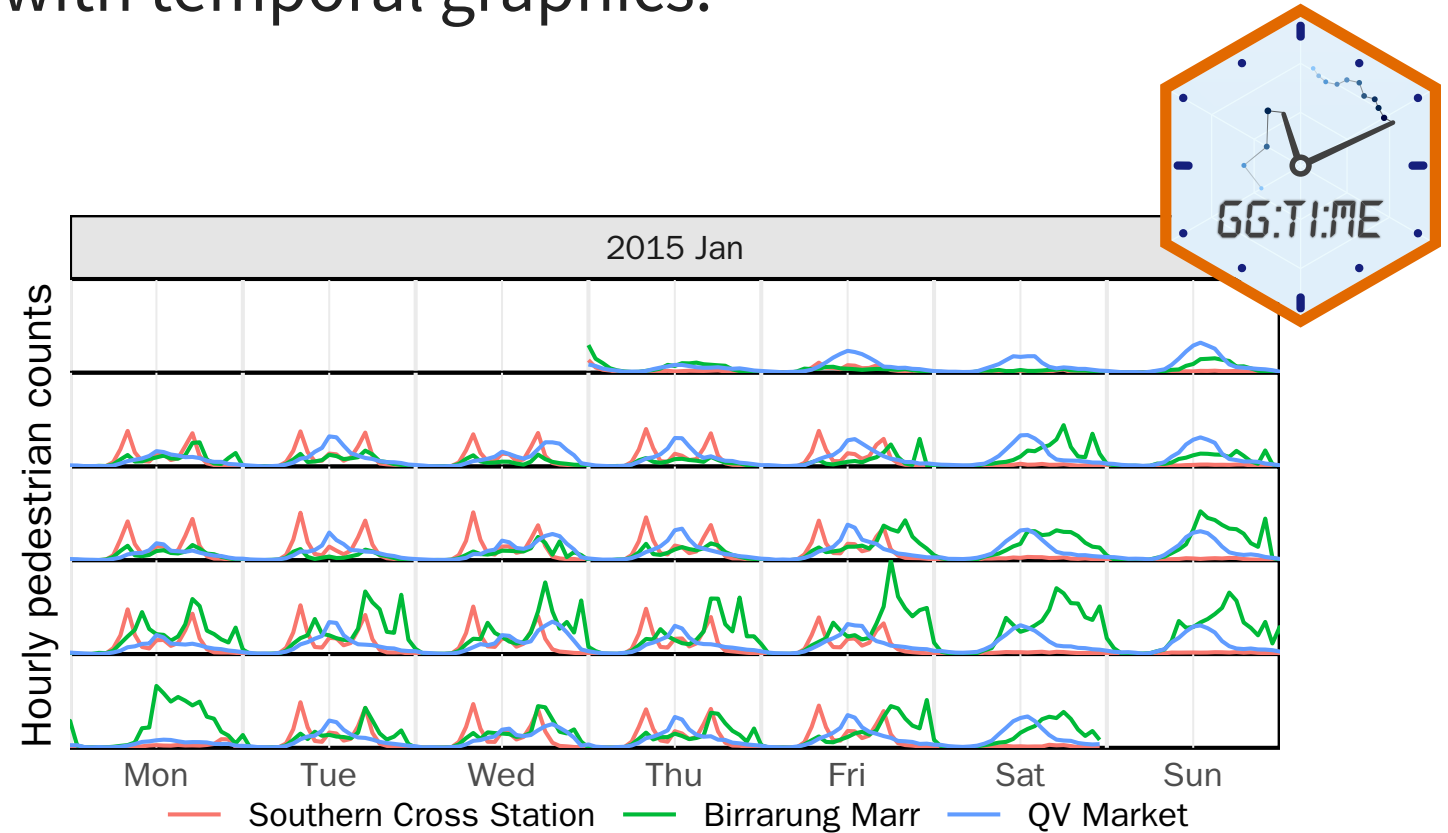
Visualising distributions

The `{ggdist}` and `{ggdibbler}` packages extend `{ggplot2}` with distributional graphics.



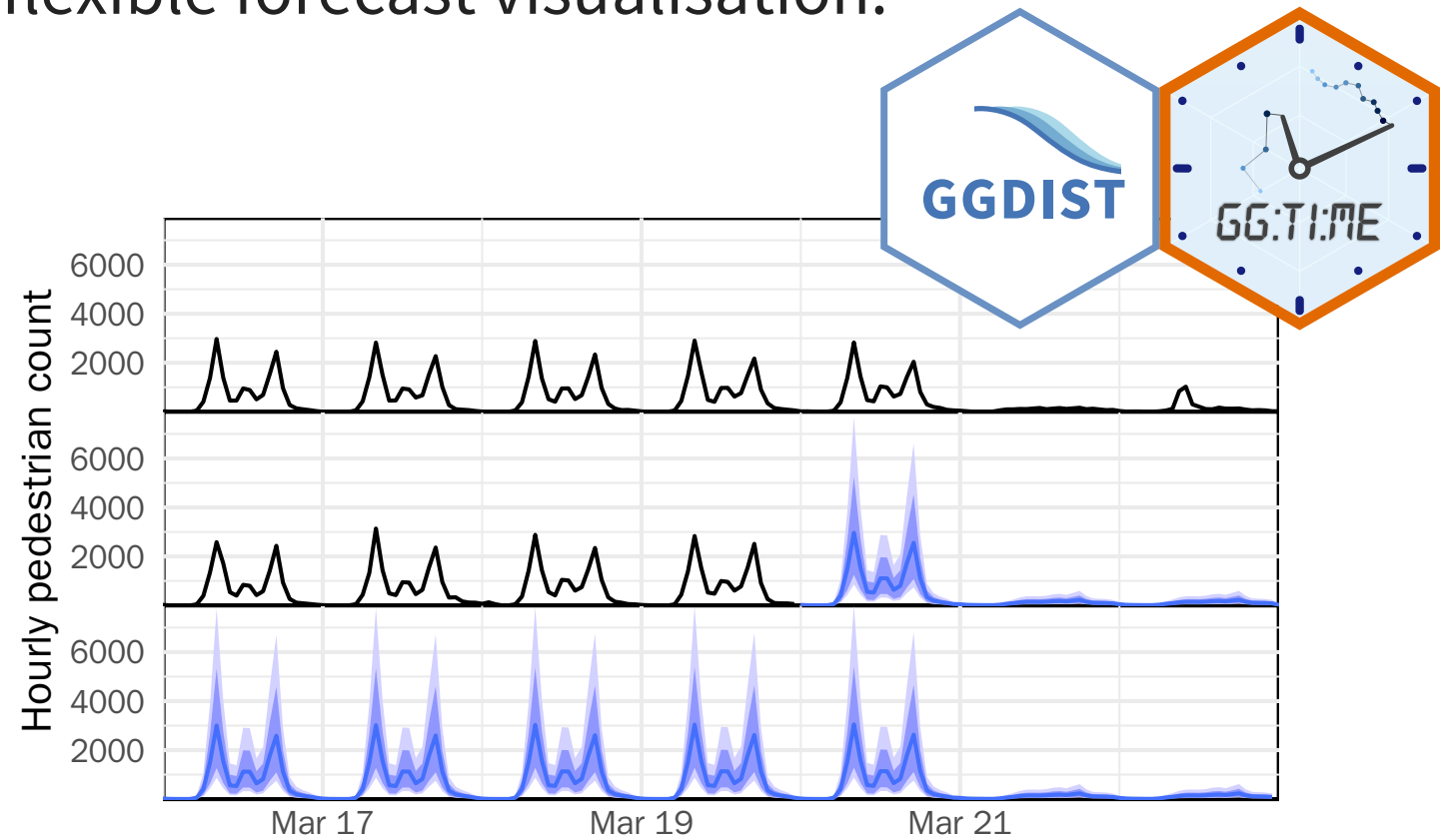
Visualising time

The `{ggtime}` package extends `{ggplot2}` with temporal graphics.



Visualising forecasts

Combining `{ggtime}` and `{ggdist}` enables flexible forecast visualisation.



Thanks for your time!

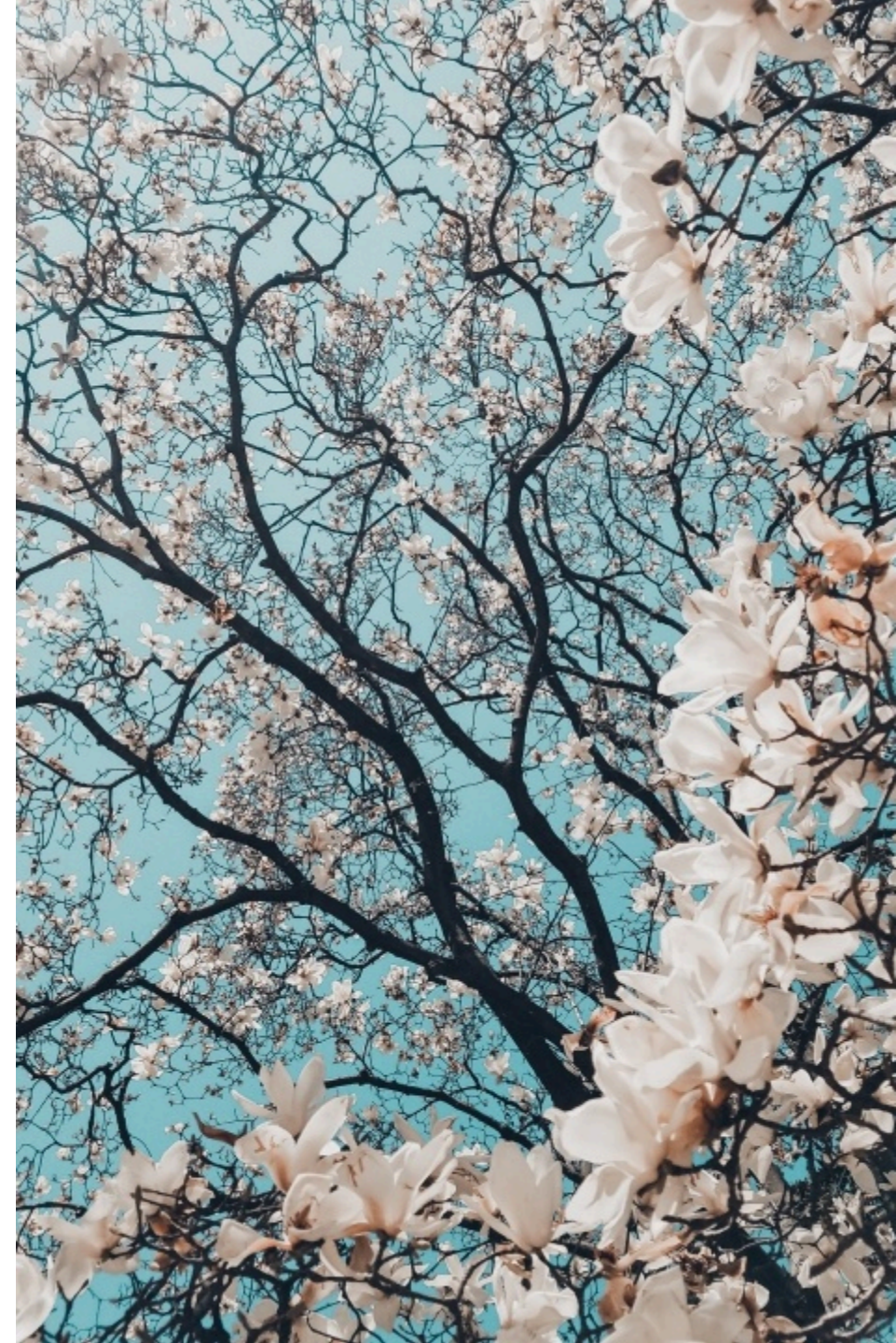


Final remarks



Useful links

- 📄 social.mitchelloharawild.com
- 📄 slides.mitchelloharawild.com/user2026
- 🔗 [mitchelloharawild/talk-vecvec-user2026](https://mitchelloharawild.com/talk-vecvec-user2026)



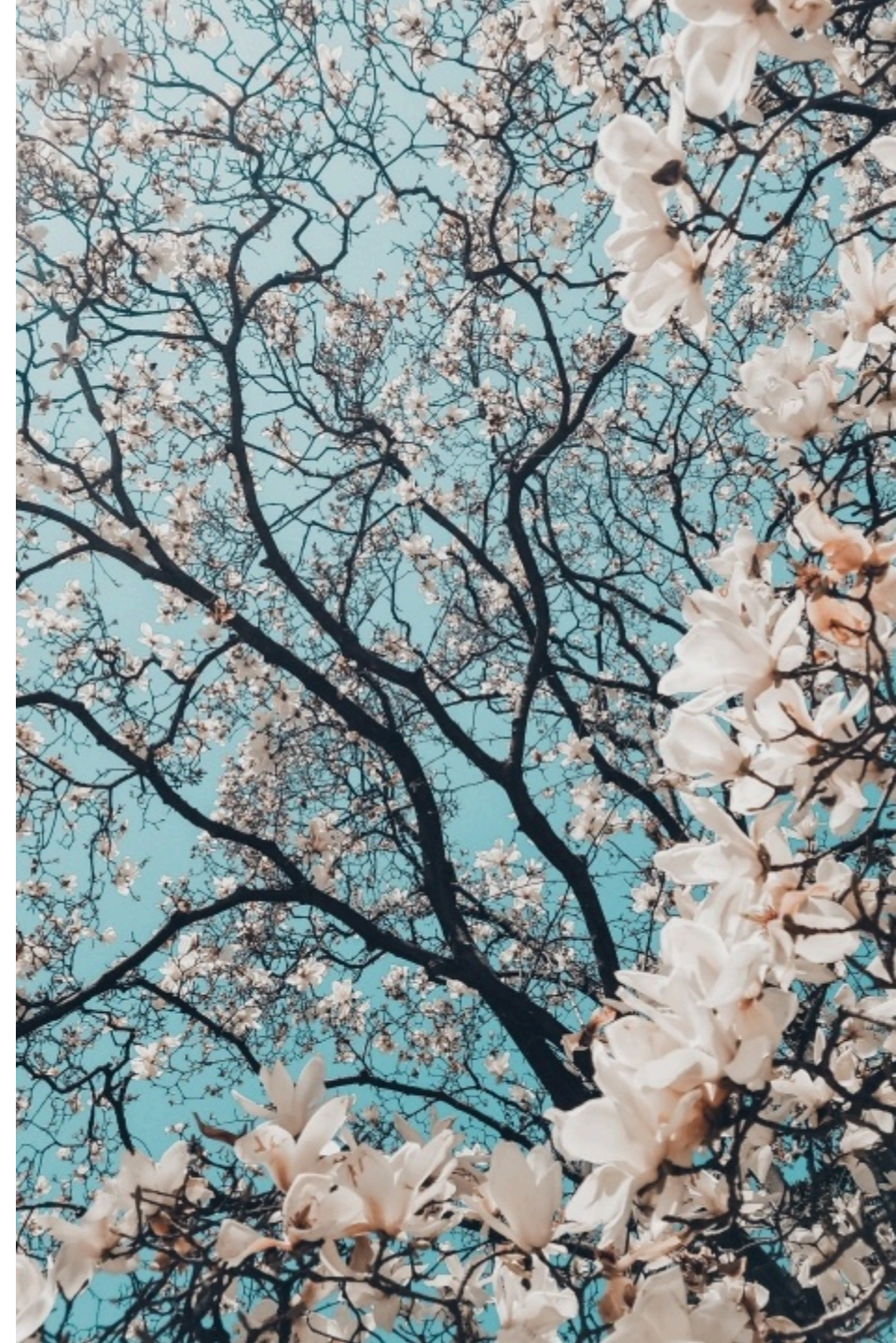
Thanks for your time!

Final remarks

-  Strict semantics supports safer statistics.

Useful links

- 📄 social.mitchelloharawild.com
- 📄 slides.mitchelloharawild.com/user2026
- 🔗 [mitchelloharawild/talk-vecvec-user2026](https://mitchelloharawild.com/talk-vecvec-user2026)



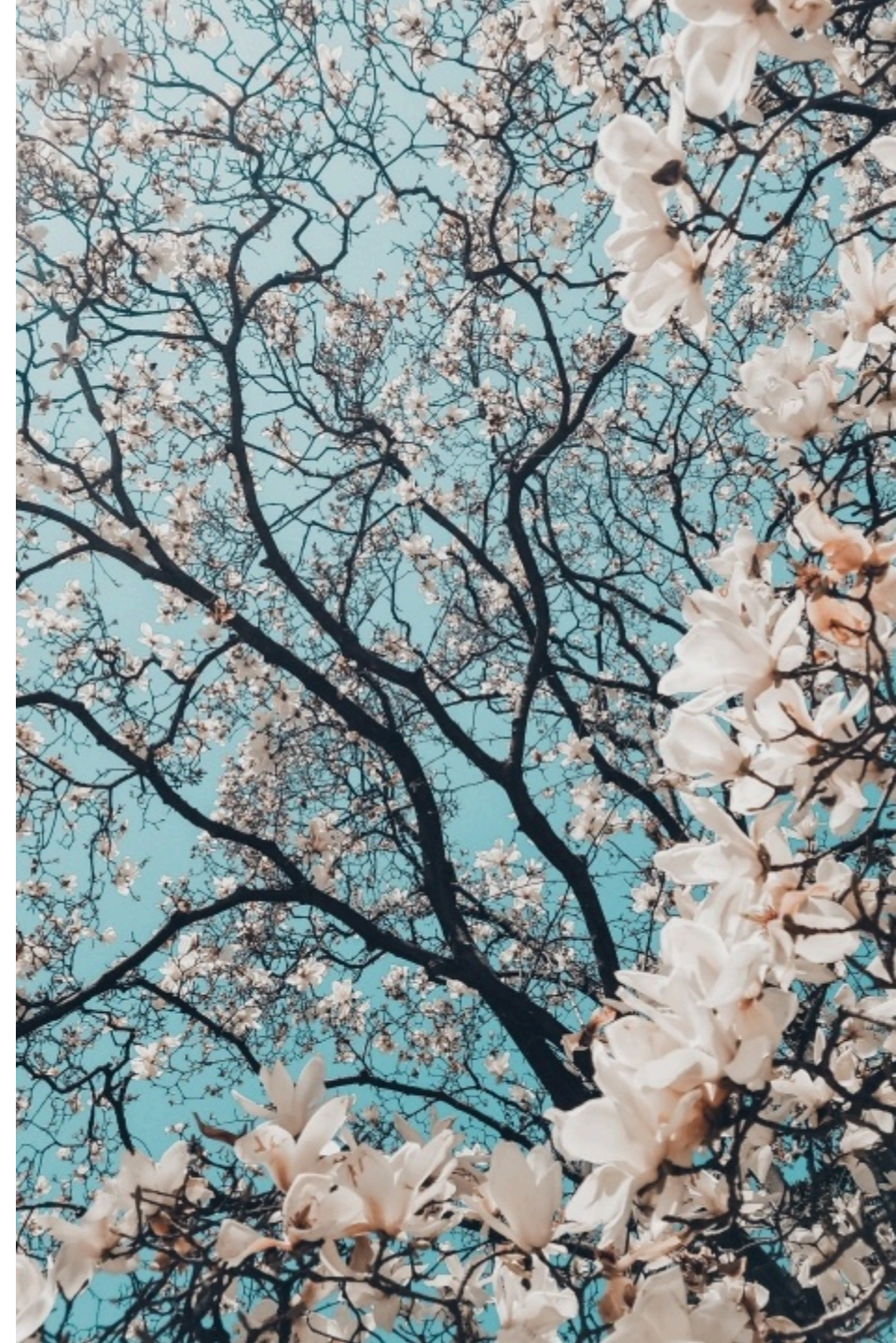
Thanks for your time!

Final remarks

-  Strict semantics supports safer statistics.
-  Endless combinations for stats and data viz.

Useful links

-  social.mitchelloharawild.com
-  slides.mitchelloharawild.com/user2026
-  mitchelloharawild.com/talk-vecvec-user2026



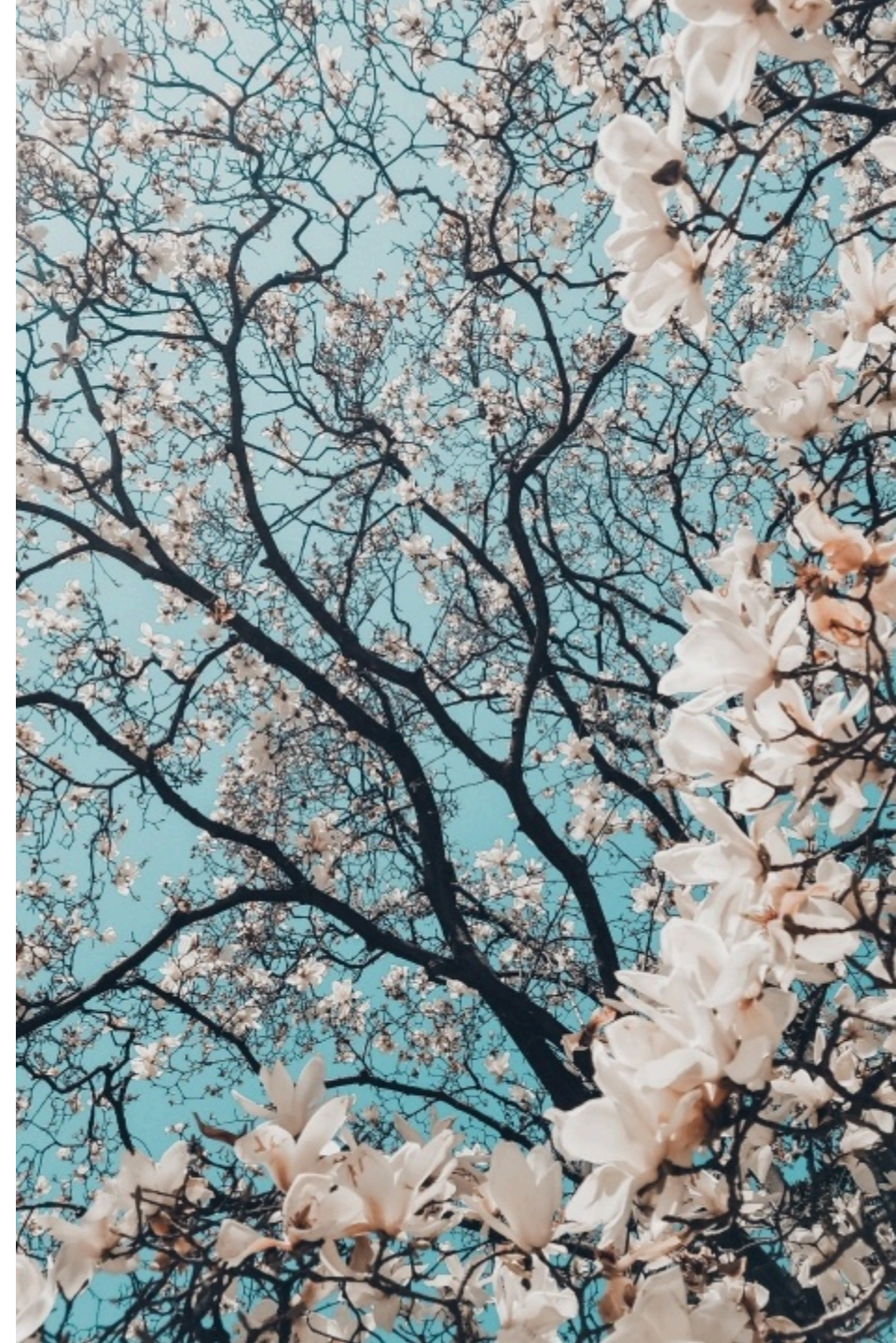
Thanks for your time!

💡 Final remarks

- 📊 Strict semantics supports safer statistics.
- ∞ Endless combinations for stats and data viz.
- 🧑‍💻 Vectorised code 🤝 statistical analysis.

🔗 Useful links

- 🖥️ social.mitchelloharawild.com
- 📄 slides.mitchelloharawild.com/user2026
- 🔔 mitchelloharawild.com/talk-vecvec-user2026



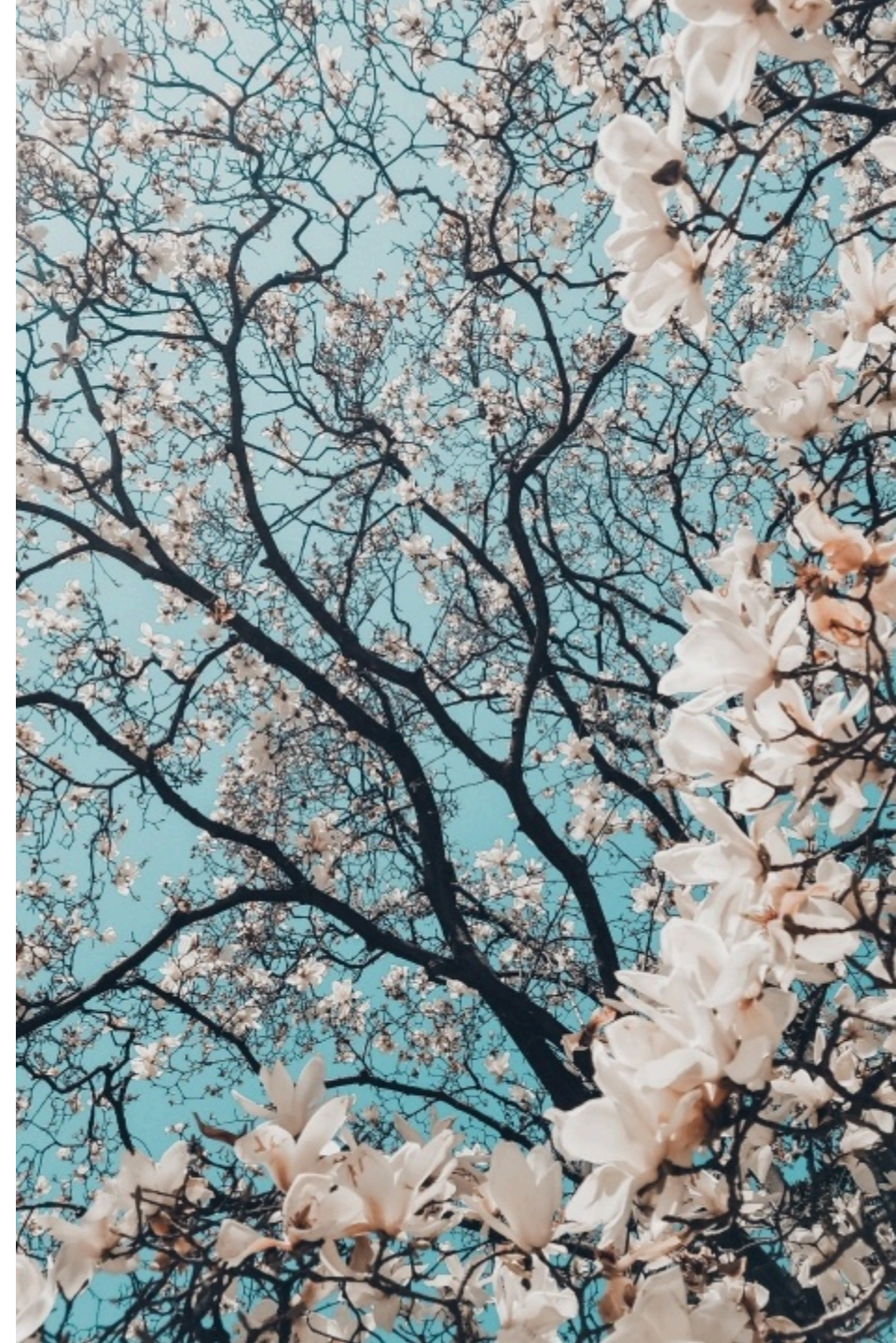
Thanks for your time!

💡 Final remarks

- 📊 Strict semantics supports safer statistics.
- ∞ Endless combinations for stats and data viz.
- 🧑🏫 Vectorised code 🤝 statistical analysis.
- 💭 Can 'vecvec' style solutions solve other problems?

🔗 Useful links

- 🖥️ social.mitchelloharawild.com
- 📺 slides.mitchelloharawild.com/user2026
- 🔊 [mitchelloharawild/talk-vecvec-user2026](https://mitchelloharawild.com/talk-vecvec-user2026)





Bonus Slides



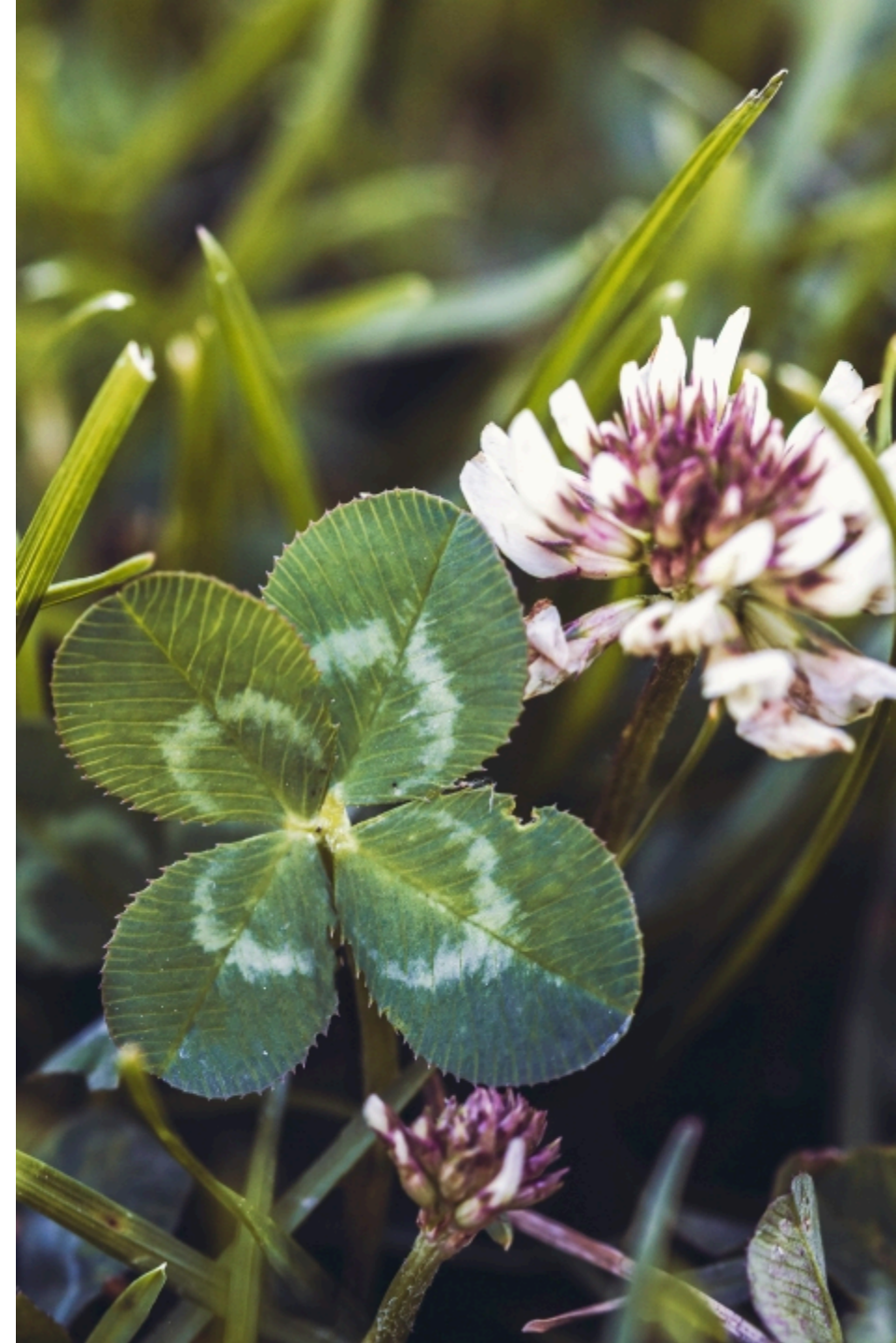
Modifying distributions

Distributions can be transformed using mathematical operations.

```
1 dist_normal(1, 3)
```

```
<distribution[1]>
```

```
[1] N(1, 9)
```



Modifying distributions

Distributions can be transformed using mathematical operations.

```
1 dist_normal(1, 3)
```

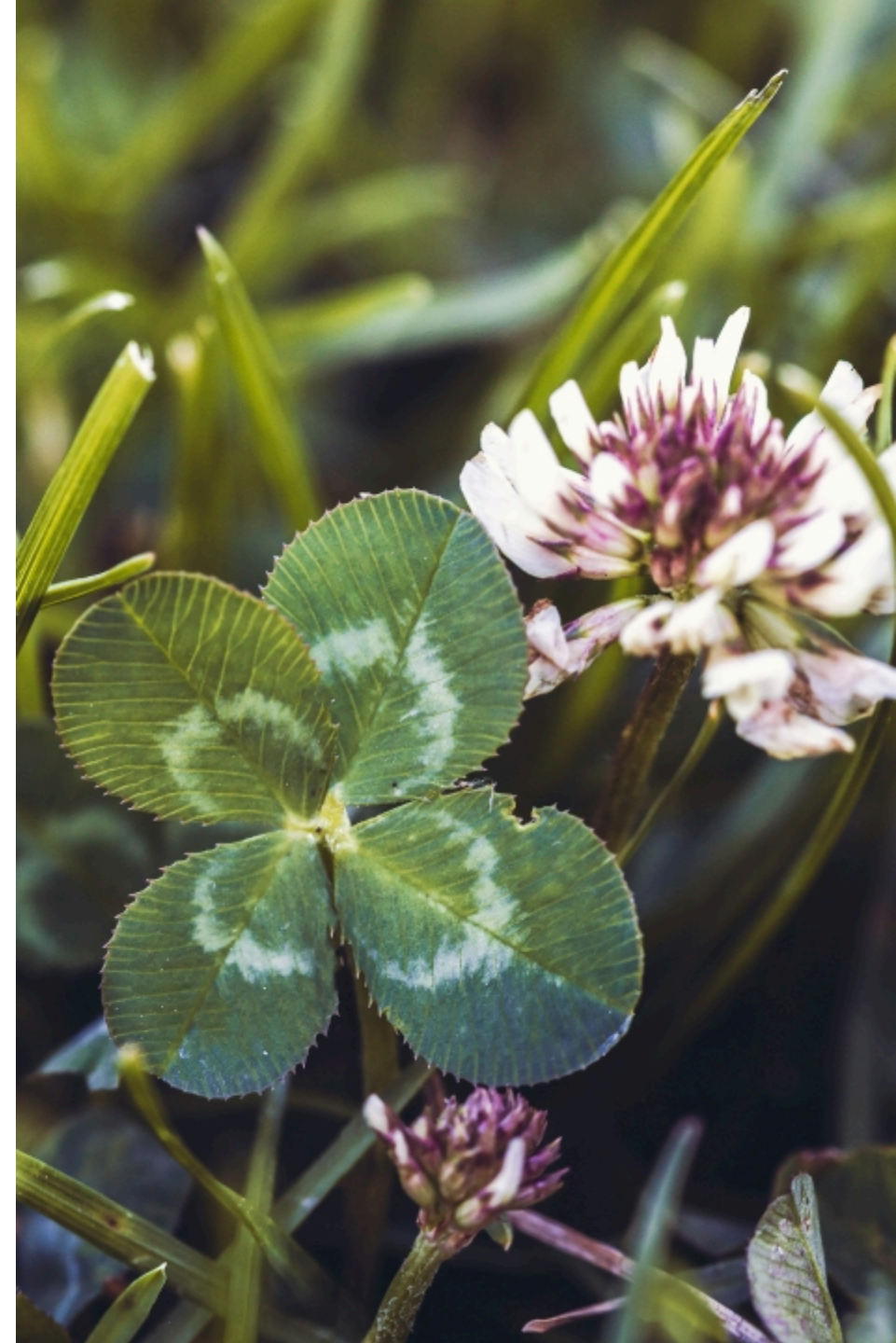
```
<distribution[1]>
```

```
[1] N(1, 9)
```

```
1 2 + dist_normal(1, 3)
```

```
<distribution[1]>
```

```
[1] N(3, 9)
```



Modifying distributions

Distributions can be transformed using mathematical operations.

```
1 dist_normal(1, 3)
```

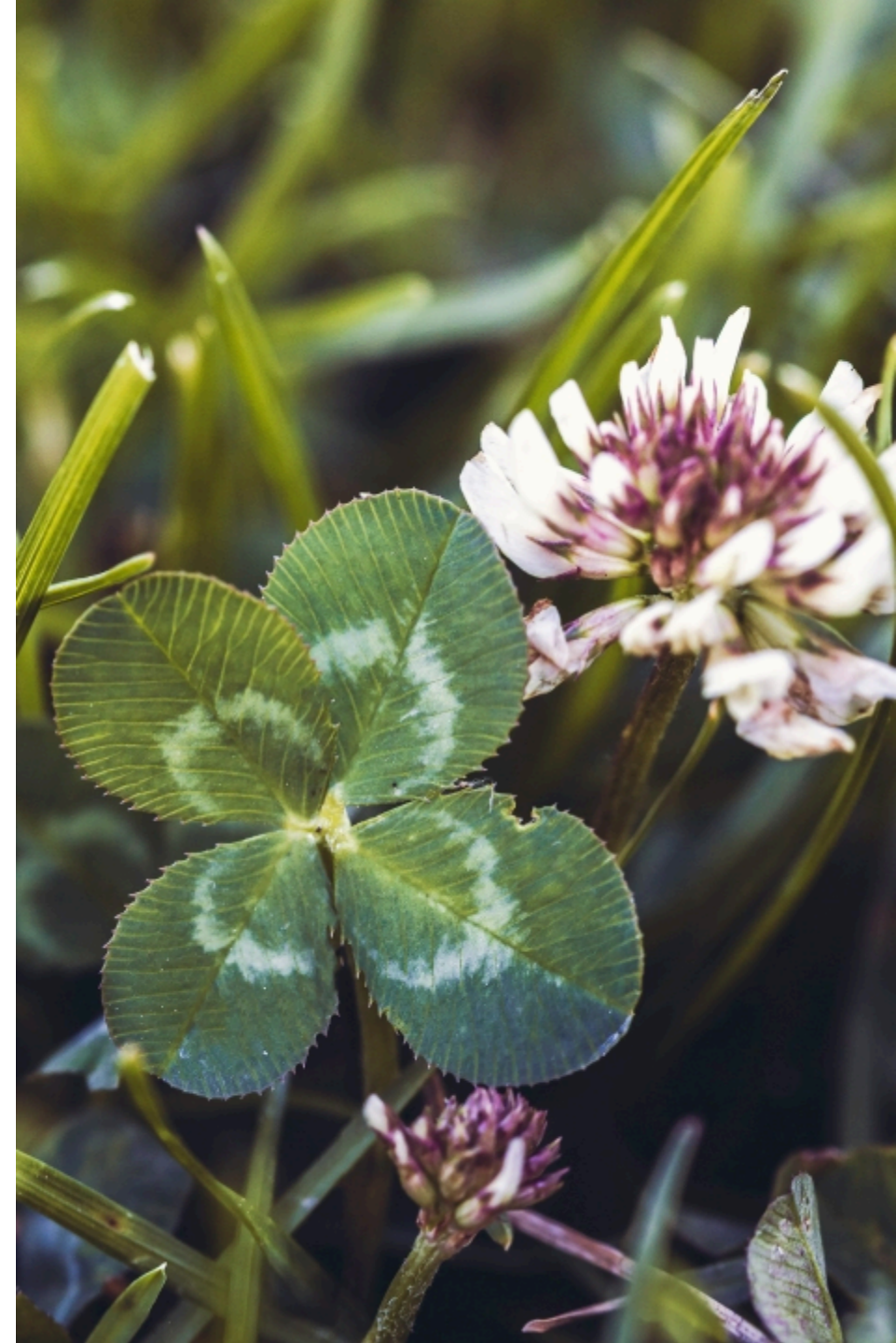
```
<distribution[1]>  
[1] N(1, 9)
```

```
1 2 + dist_normal(1, 3)
```

```
<distribution[1]>  
[1] N(3, 9)
```

```
1 3 * (2 + dist_normal(1, 3))
```

```
<distribution[1]>  
[1] N(9, 81)
```



Modifying distributions

Distributions can be transformed using mathematical operations.

```
1 dist_normal(1, 3)
```

```
<distribution[1]>  
[1] N(1, 9)
```

```
1 2 + dist_normal(1, 3)
```

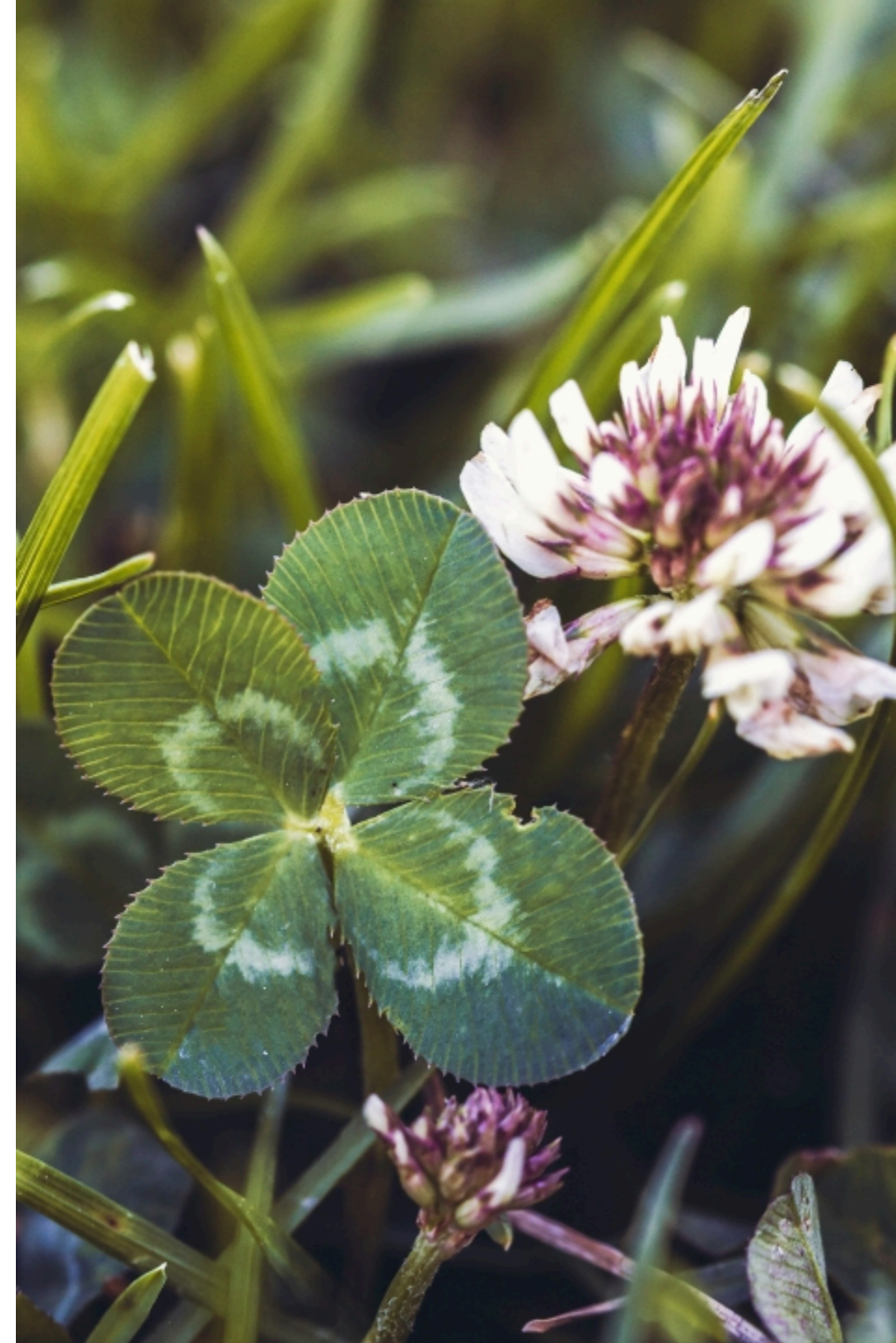
```
<distribution[1]>  
[1] N(3, 9)
```

```
1 3 * (2 + dist_normal(1, 3))
```

```
<distribution[1]>  
[1] N(9, 81)
```

```
1 exp(3 * (2 + dist_normal(1, 3)))
```

```
<distribution[1]>  
[1] lN(9, 81)
```



Modifying distributions

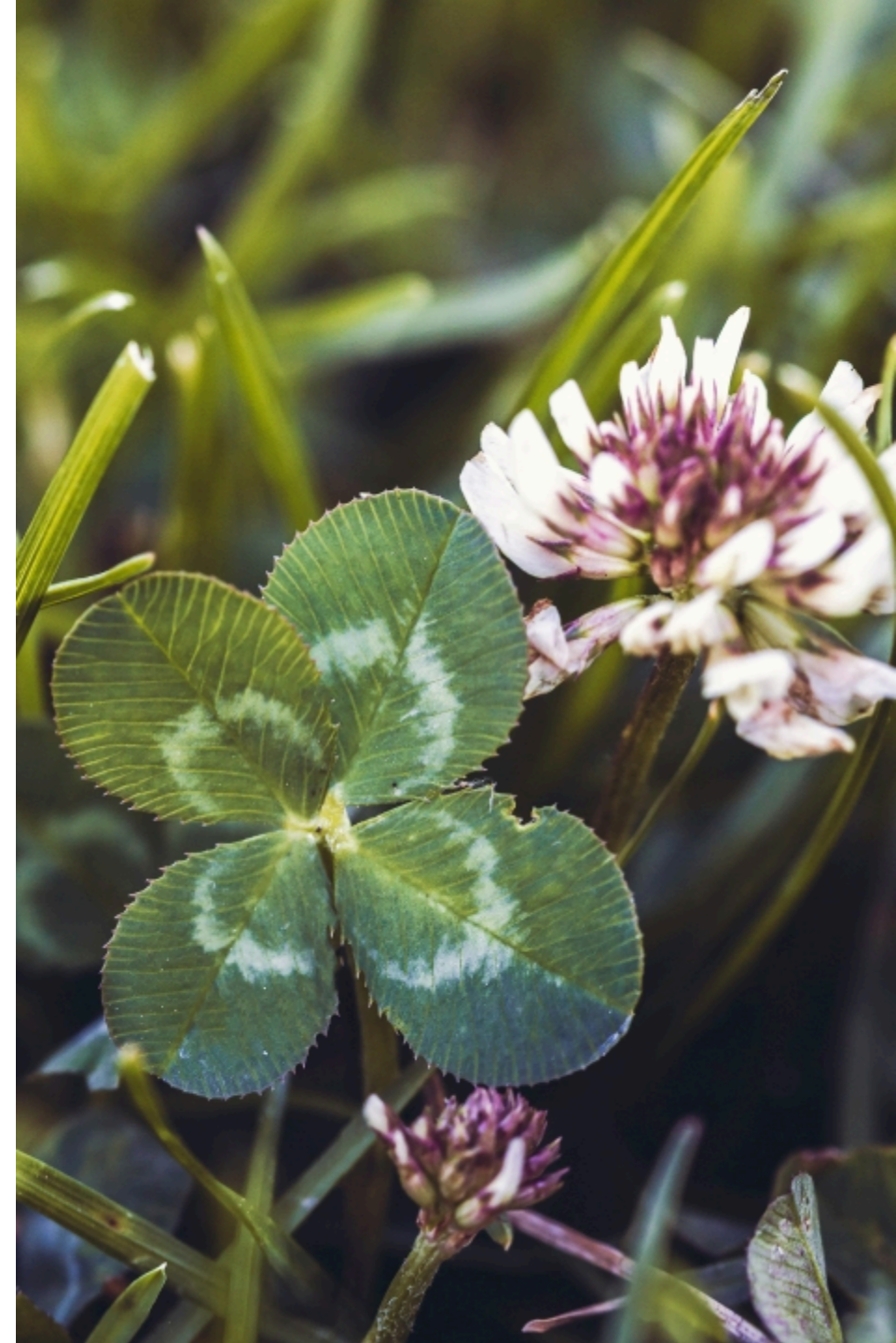
i Transformed distribution

`dist_transformed()` enables arbitrary transformations.

```
1 (3 * (2 + dist_normal(1, 3)))^2
```

```
<distribution[1]>
```

```
[1] t(N(9, 81))
```



Modifying distributions

Transformed distribution

`dist_transformed()` enables arbitrary transformations.

```
1 (3 * (2 + dist_normal(1, 3)))^2
```

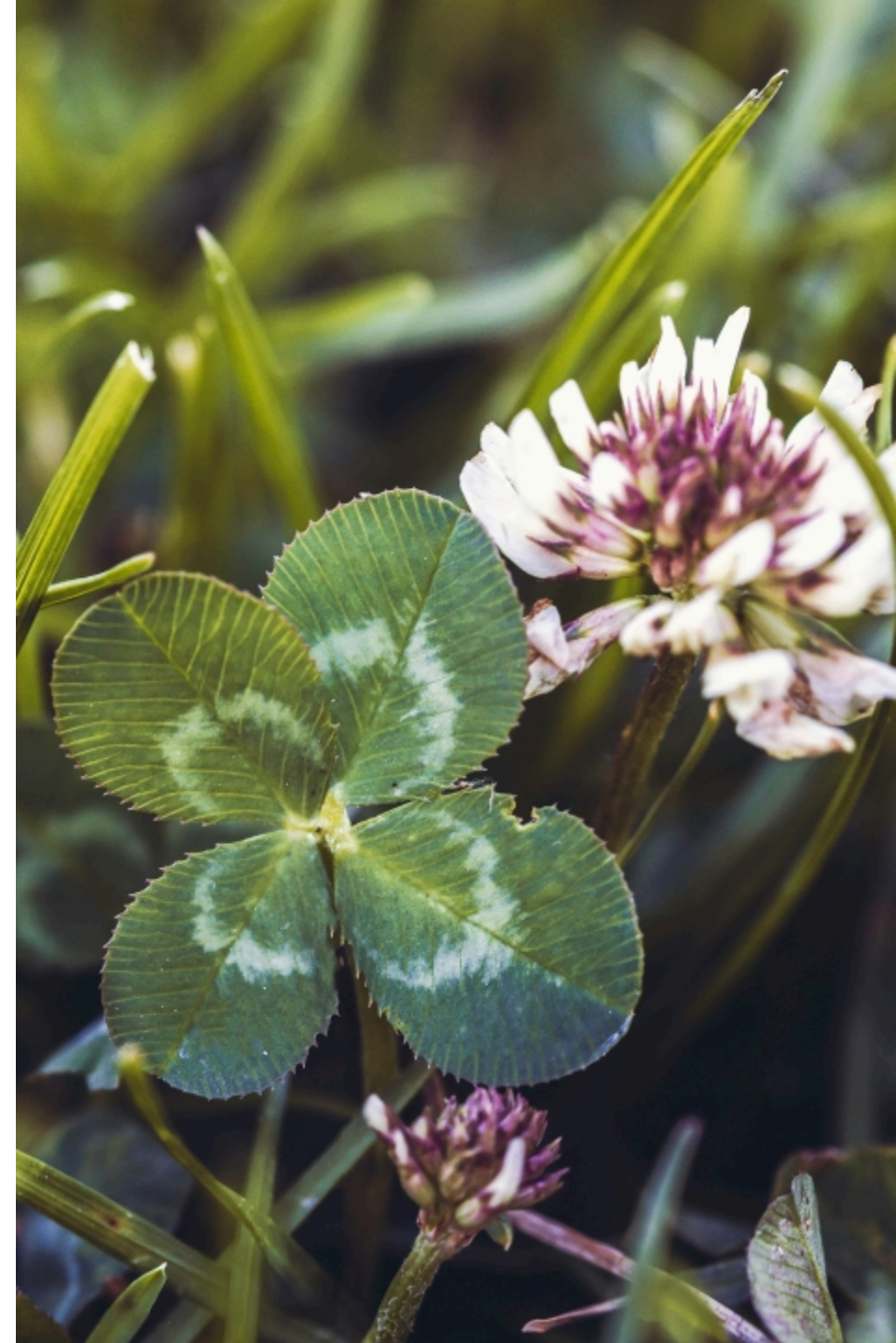
```
<distribution[1]>  
[1] t(N(9, 81))
```

Other distribution modifiers

 `dist_inflated()`

 `dist_truncated()`

 `dist_mixture()`



Vectorised operations

Vectorised operations in distributional are safer than the p/d/q/r equivalents.

i Vectorising in two ways

There are two types of operation arguments:

- `vector/matrix` inputs

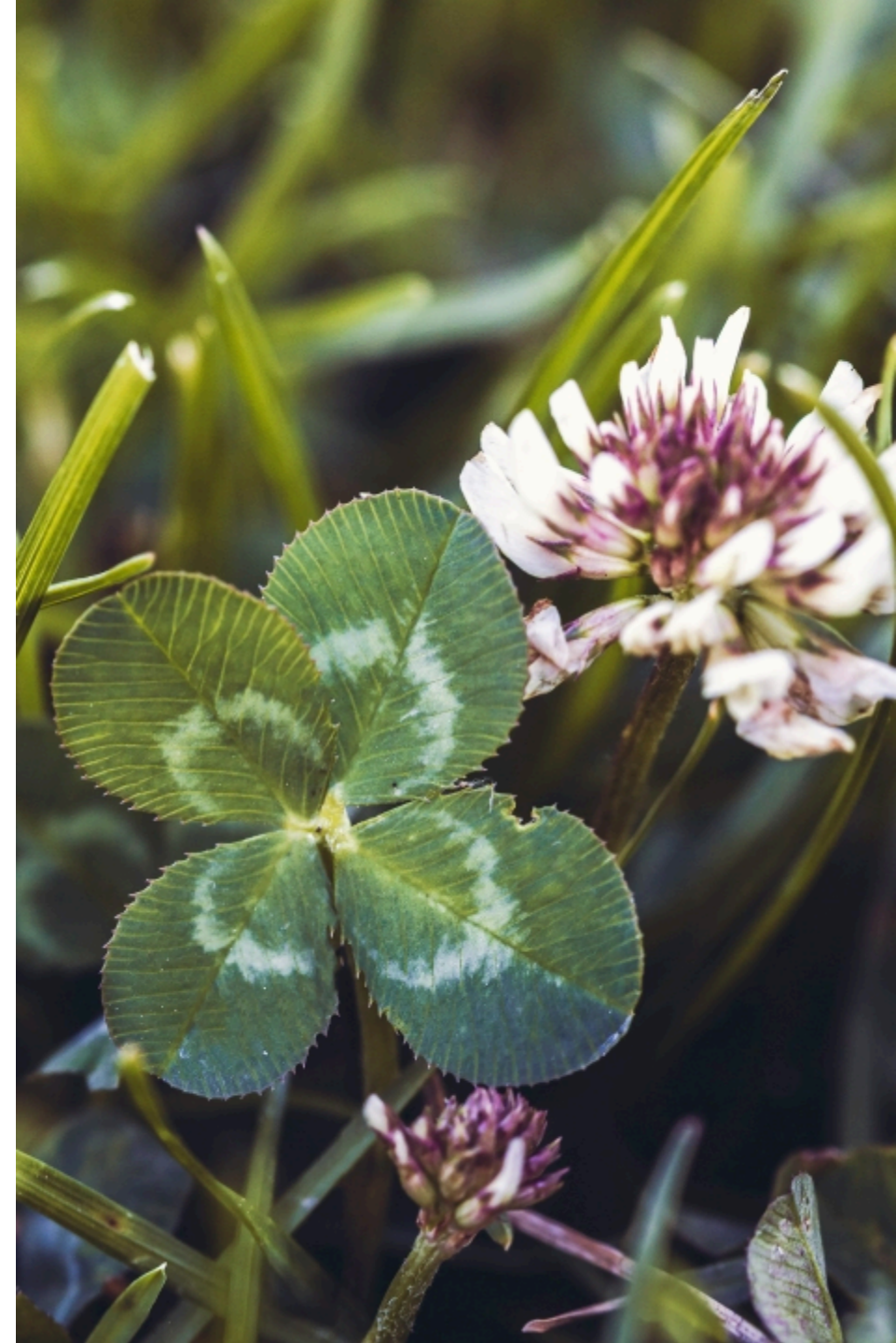
Vectorises across distributions, then arguments.

This approach is **simpler**, especially single arguments.

- `list/data.frame` inputs

Vectorises across arguments, then distributions.

This approach is **more flexible and powerful**.



Vectorised operations (vectors)

```
1 (y <- c(dist_normal(0, 1), dist_beta(5, 1), dist_gamm  
<distribution[3]>  
[1] N(0, 1)      Beta(5, 1)  Γ(2, 1)
```

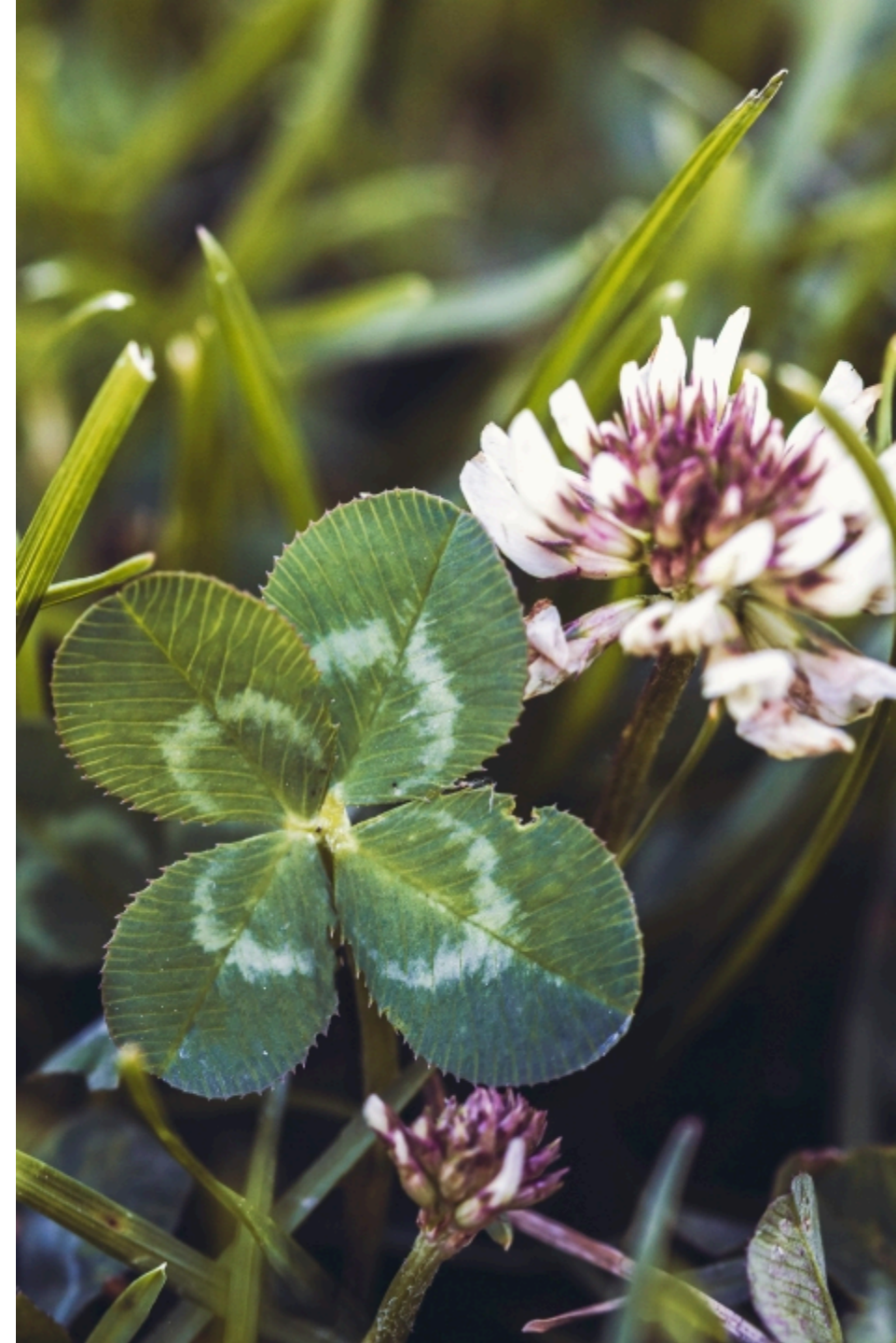
Vectors/matrices apply the same operation
inputs to each distribution.

```
1 density(y, at = 0.65)  
[1] 0.3229724 0.8925312 0.3393298
```

```
1 density(y, at = c(0.65, 0.9))  
[[1]]  
[1] 0.3229724 0.2660852
```

```
[[2]]  
[1] 0.8925312 3.2805000
```

```
[[3]]  
[1] 0.3393298 0.3659127
```



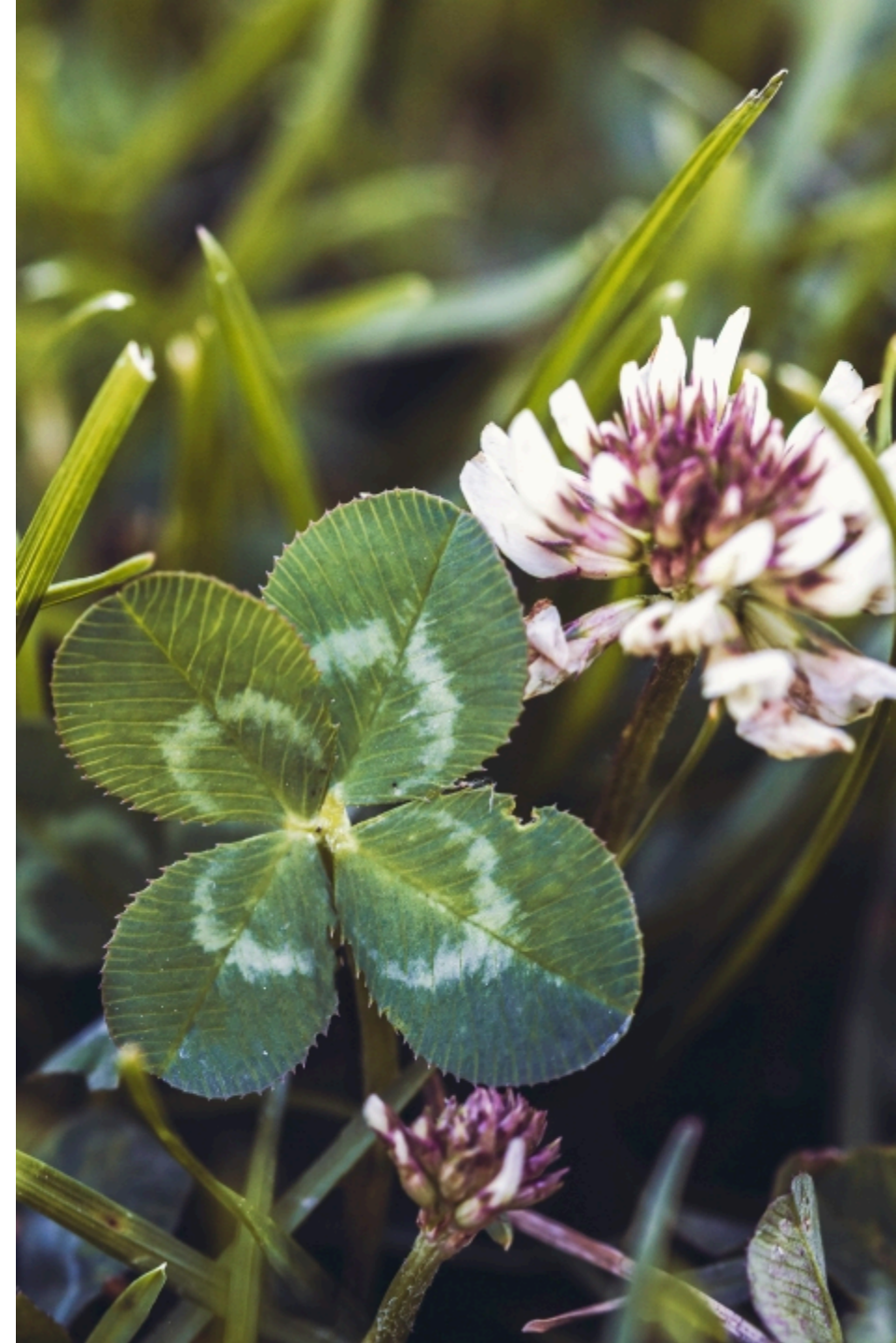
Vectorised operations (vectors)

i Distributions in data analysis

This also works well with data frames.

```
1 tibble::tibble(y) |>  
2   dplyr::mutate(density(y, at = 0.65))
```

```
# A tibble: 3 × 2  
      y `density(y, at = 0.65)`  
  <dist>          <dbl>  
1   N(0, 1)        0.323  
2 Beta(5, 1)       0.893  
3   Γ(2, 1)        0.339
```



Vectorised operations (vectors)

i Distributions in data analysis

This also works well with data frames.

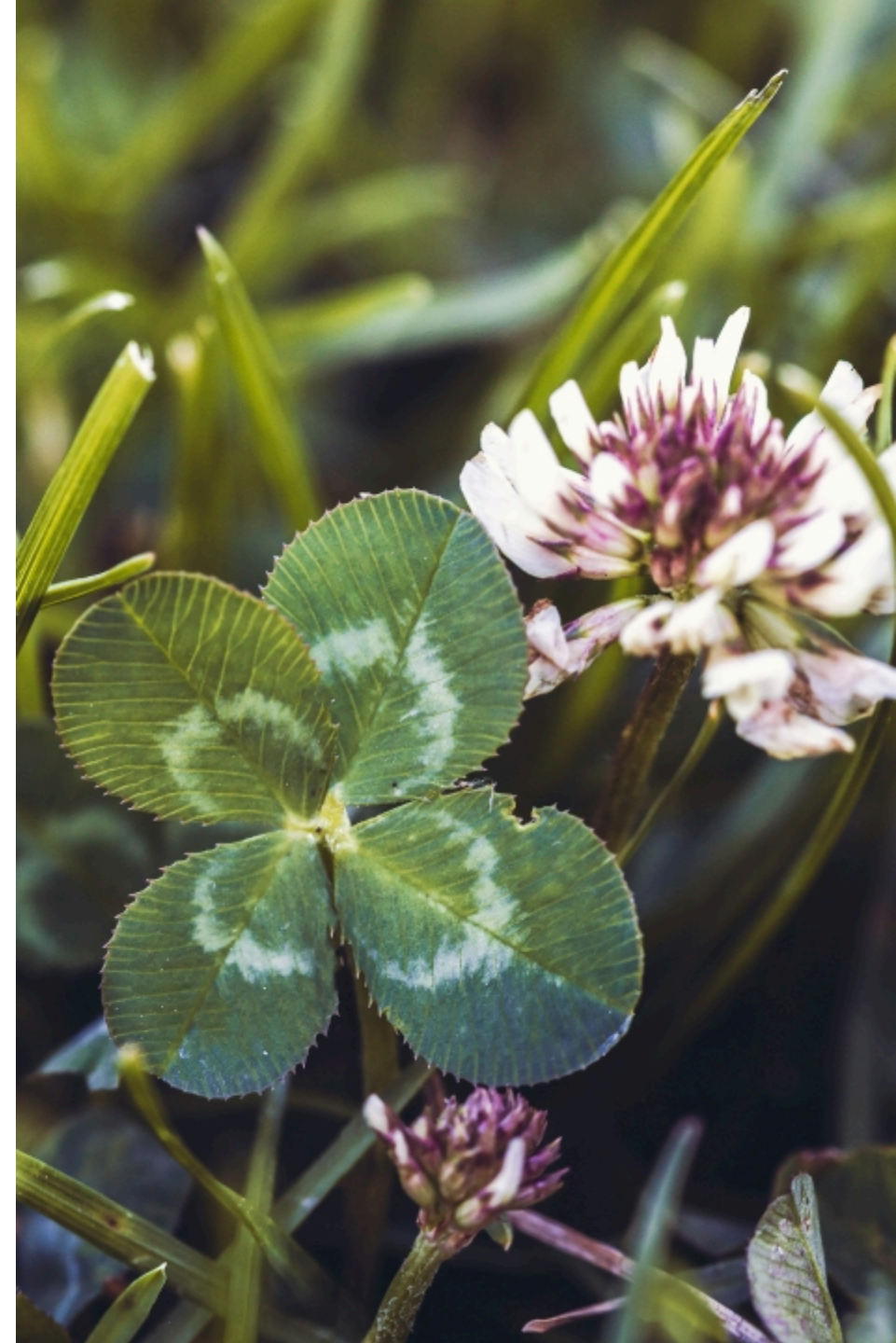
```
1 tibble::tibble(y) |>
2   dplyr::mutate(density(y, at = 0.65))
```

```
# A tibble: 3 × 2
  y `density(y, at = 0.65)`
  <dbl>                   <dbl>
1 N(0, 1)                 0.323
2 Beta(5, 1)              0.893
3  $\Gamma(2, 1)$          0.339
```

Although its a bit tricky for more than one input.

```
1 tibble::tibble(y) |>
2   dplyr::mutate(density(y, at = c(0.65, 0.9)))
```

```
# A tibble: 3 × 2
  y `density(y, at = c(0.65, 0.9))`
  <dist> <list>
1 N(0, 1) <dbl [2]>
2 Beta(5, 1) <dbl [2]>
3  $\Gamma(2, 1)$  <dbl [2]>
```



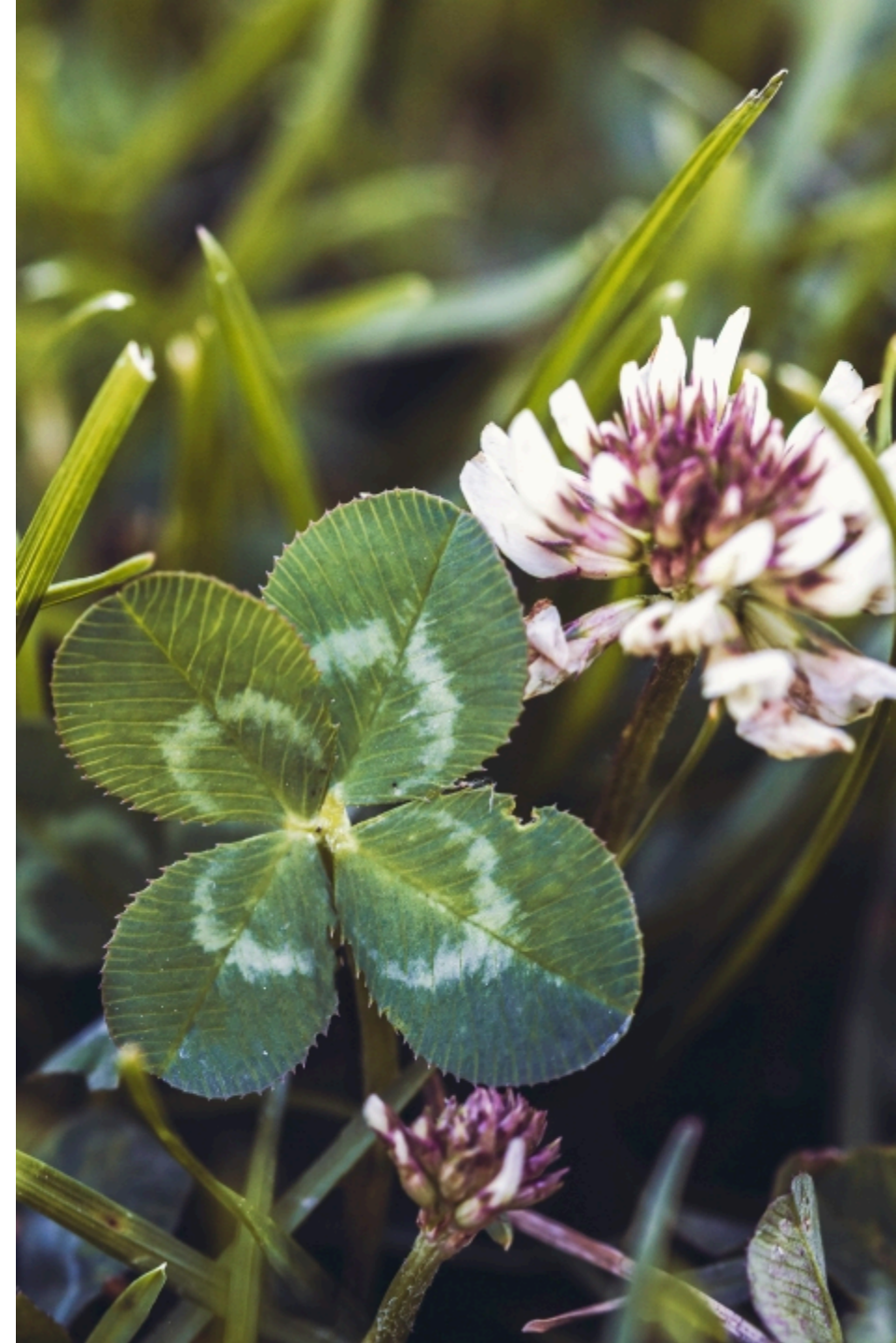
Vectorised operations (lists)

```
1 (y <- c(dist_normal(0, 1), dist_beta(5, 1), dist_gamm  
<distribution[3]>  
[1] N(0, 1)      Beta(5, 1)  Γ(2, 1)
```

Lists/data.frames recycle each input argument to the length of distributions.

```
1 density(y, at = list(d1 = 0.65))  
  
          d1  
1 0.3229724  
2 0.8925312  
3 0.3393298
```

```
1 density(y, at = list(d1 = 0.65, d2 = 0.9))  
  
          d1          d2  
1 0.3229724 0.2660852  
2 0.8925312 3.2805000  
3 0.3393298 0.3659127
```

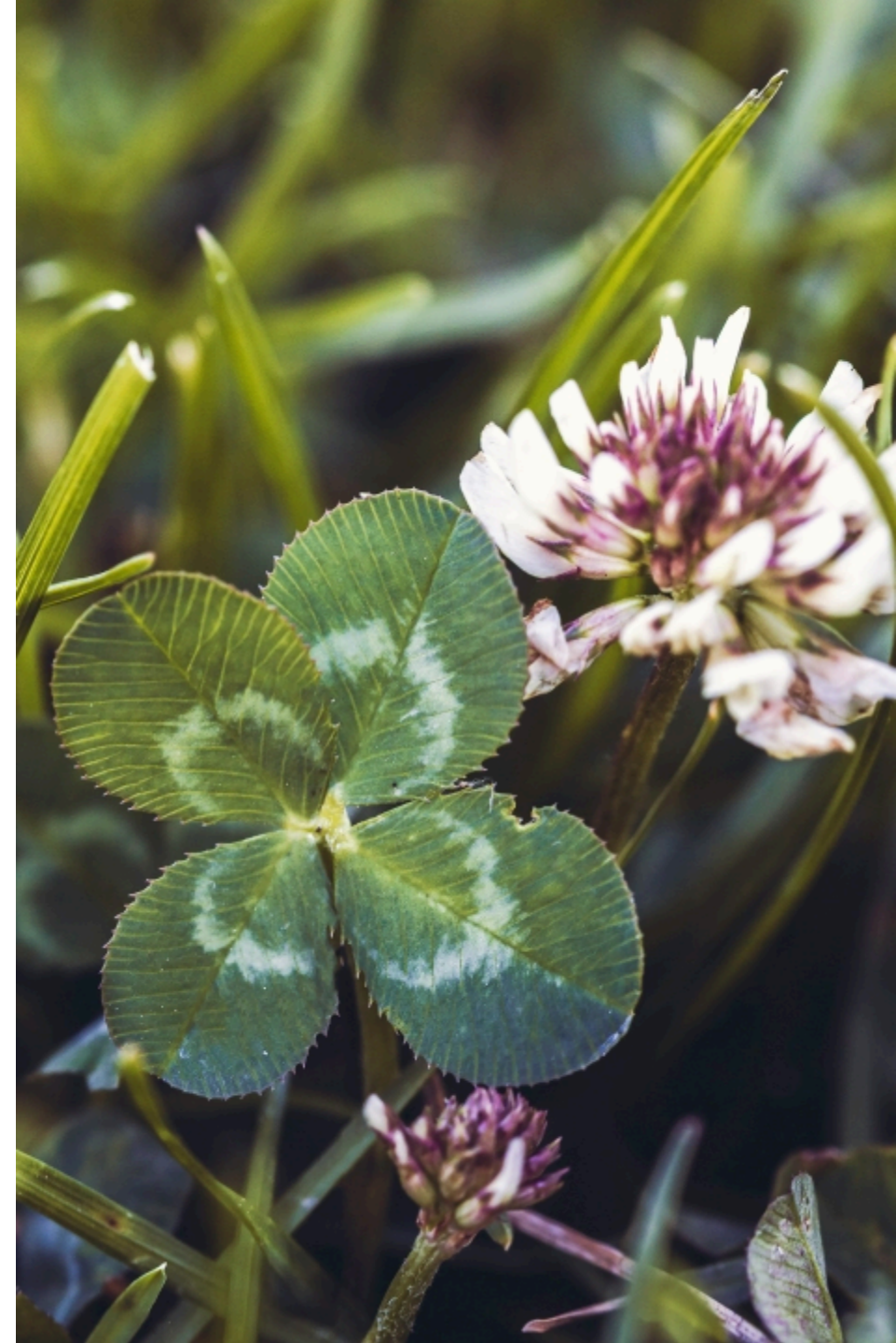


Vectorised operations (lists)

```
1 (y <- c(dist_normal(0, 1), dist_beta(5, 1), dist_gamm  
<distribution[3]>  
[1] N(0, 1)      Beta(5, 1)  Γ(2, 1)
```

This also allows vectorisation across both inputs and distributions.

```
1 density(y, at = list(dens = c(0.65, 0.9, 0.3)))  
  
      dens  
1 0.3229724  
2 3.2805000  
3 0.2222455
```



Vectorised operations (lists)

```
1 (y <- c(dist_normal(0, 1), dist_beta(5, 1), dist_gamm  
<distribution[3]>  
[1] N(0, 1)      Beta(5, 1)  Γ(2, 1)
```

This also allows vectorisation across both inputs and distributions.

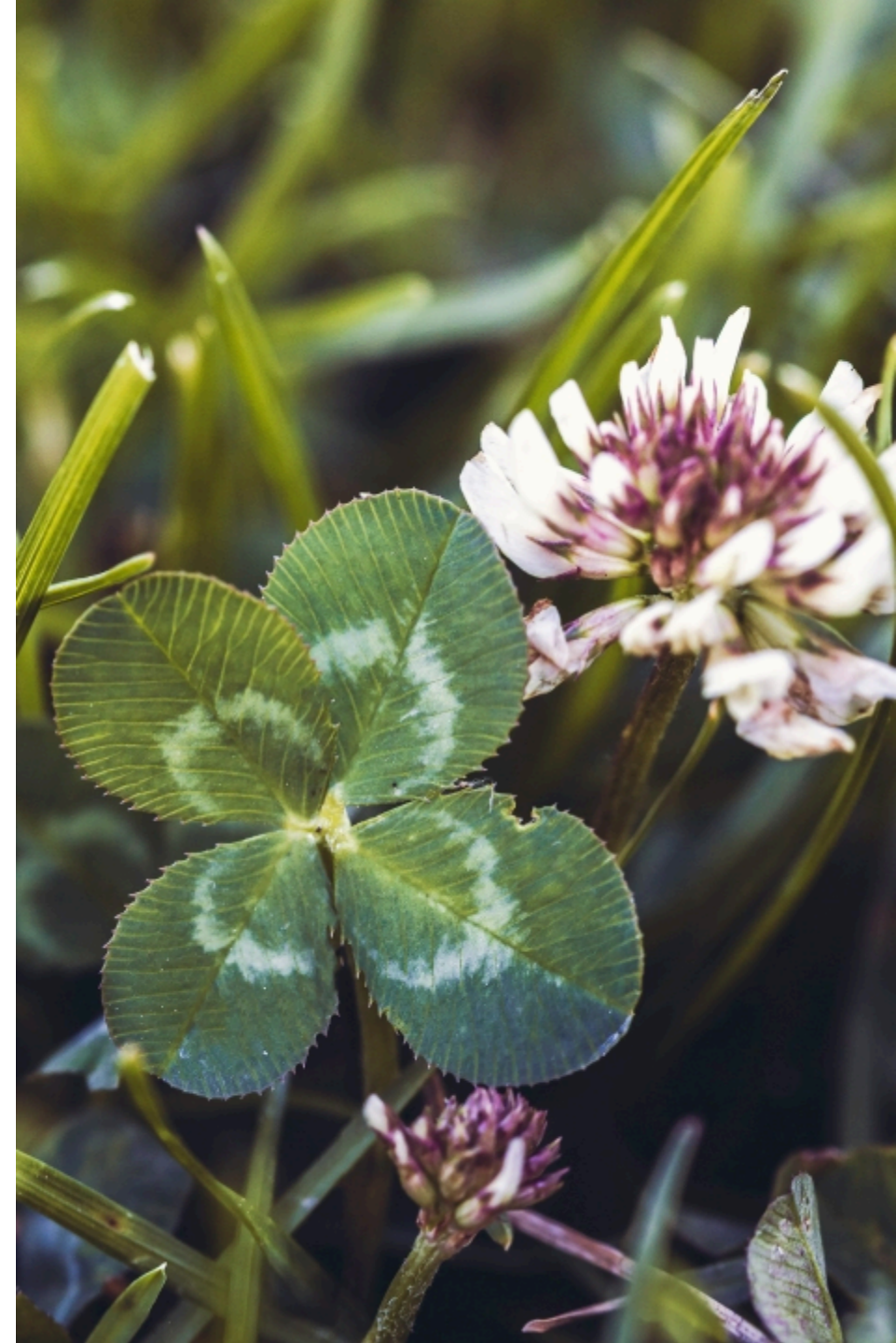
```
1 density(y, at = list(dens = c(0.65, 0.9, 0.3)))  
  
      dens  
1 0.3229724  
2 3.2805000  
3 0.2222455
```



Reliable recycling

```
1 density(y, list(dens = c(0.65, 0.9)))
```

```
Error in `FUN()`:  
! Cannot recycle input of size 2 to match the  
distributions (size 3).
```



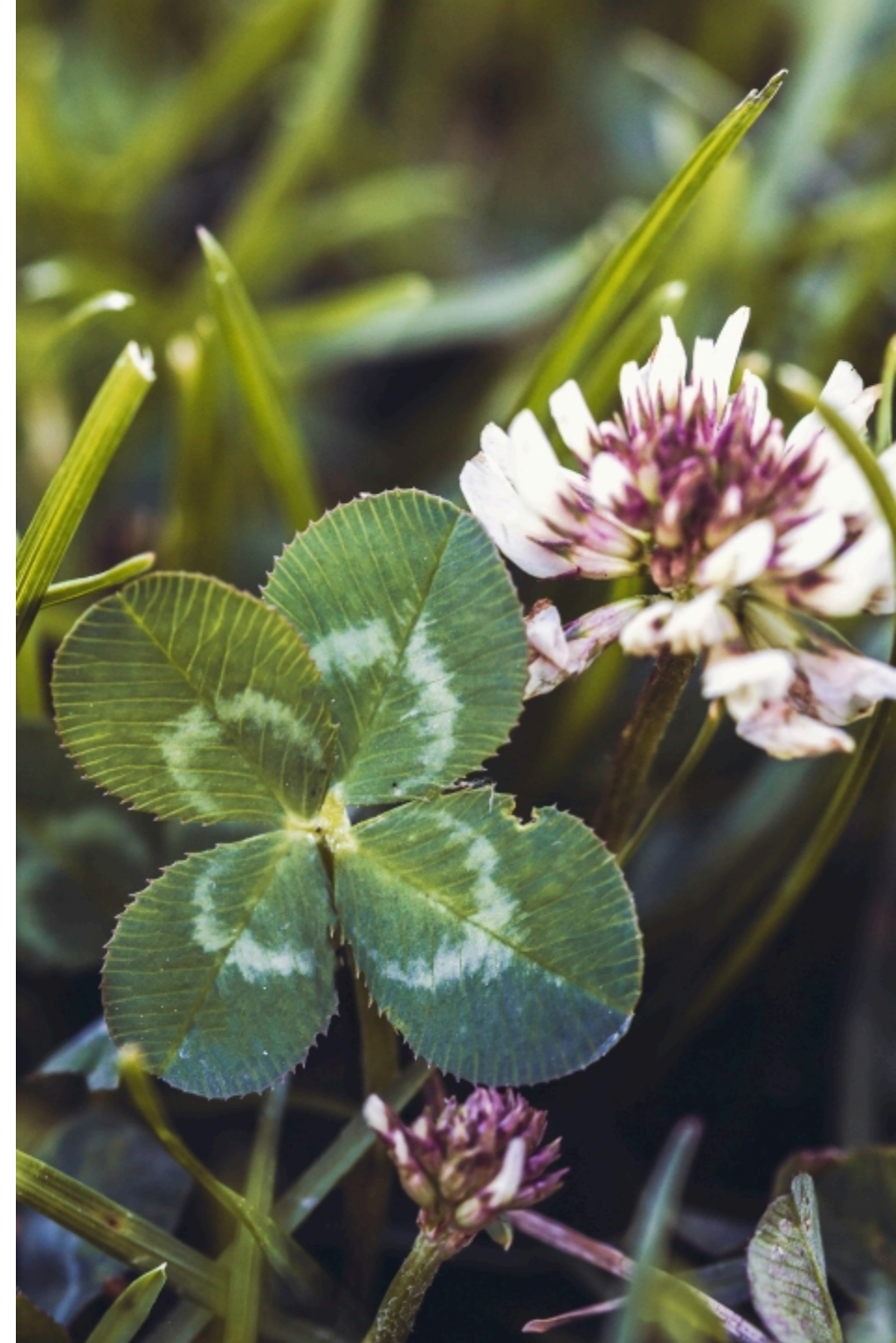
Vectorised operations (lists)

i Distributions in data analysis

This also works *really* well with data frames.

```
1 tibble::tibble(y) |>  
2   dplyr::mutate(density(y, at = list(d1 = 0.65)))
```

```
# A tibble: 3 × 2  
      y      d1  
  <dist> <dbl>  
1  N(0, 1) 0.323  
2 Beta(5, 1) 0.893  
3   Γ(2, 1) 0.339
```



Vectorised operations (lists)

i Distributions in data analysis

This also works *really* well with data frames.

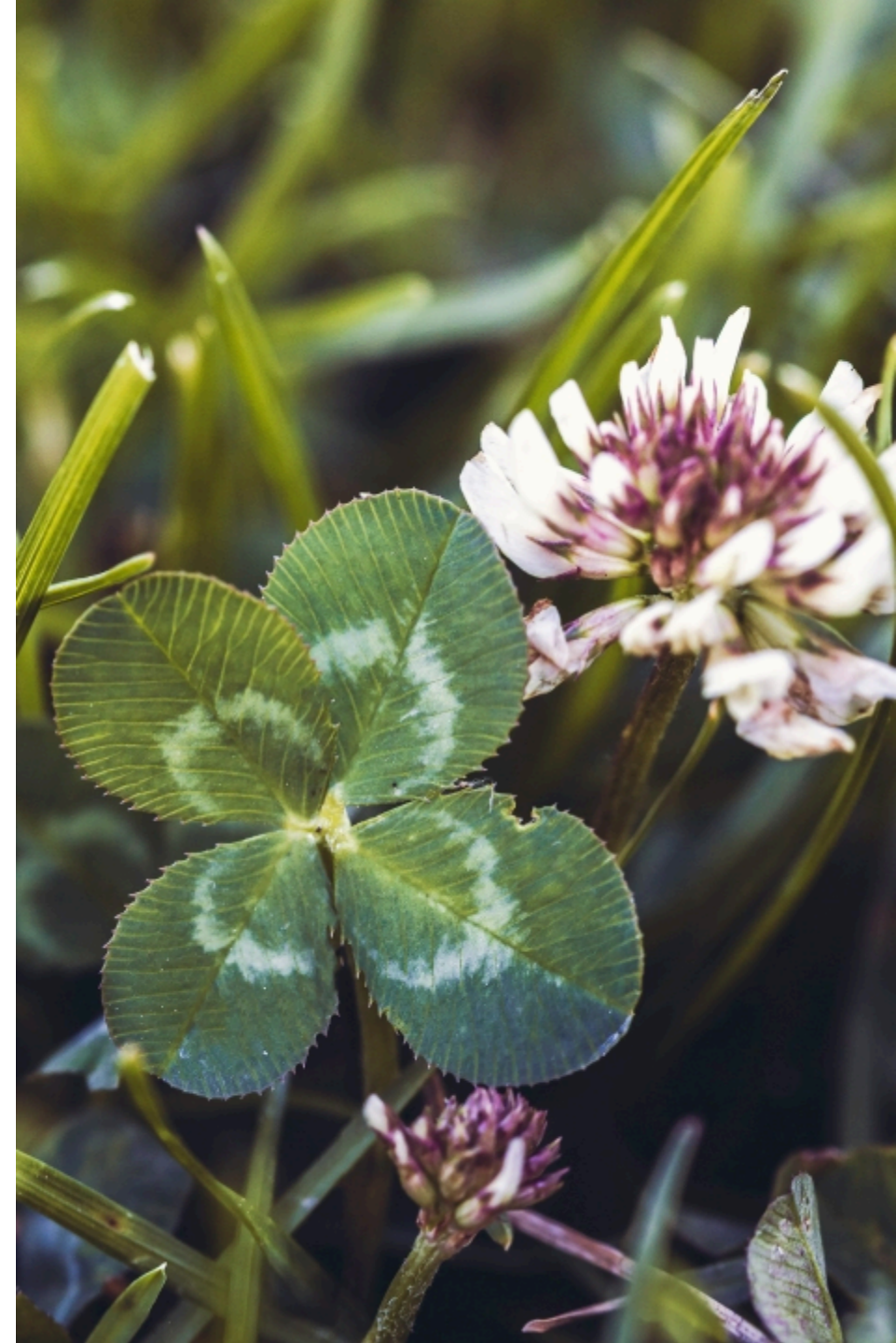
```
1 tibble::tibble(y) |>  
2   dplyr::mutate(density(y, at = list(d1 = 0.65)))
```

```
# A tibble: 3 × 2  
      y      d1  
  <dist> <dbl>  
1  N(0, 1) 0.323  
2 Beta(5, 1) 0.893  
3   Γ(2, 1) 0.339
```

`mutate()` automatically unpacks the results if unnamed.

```
1 tibble::tibble(y) |>  
2   dplyr::mutate(  
3     density(y, at = list(d1 = 0.65, d2 = 0.9))  
4   )
```

```
# A tibble: 3 × 3  
      y      d1      d2  
  <dist> <dbl> <dbl>  
1  N(0, 1) 0.323 0.266  
2 Beta(5, 1) 0.893 3.28  
3   Γ(2, 1) 0.339 0.366
```



Visualising distributions

```
1 library(ggdist)
2 library(ggplot2)
3 df <- tibble::tibble(
4   name = c("Gamma(2,1)", "Normal(5,1)", "Mixture"),
5   dist = c(dist_gamma(2,1), dist_normal(5,1),
6            dist_mixture(dist_gamma(2,1), dist_normal(
7   )
8   ggplot(df, aes(y = factor(name, levels = rev(name))))
9     stat_dist_halfeye(aes(dist = dist)) +
10     labs(title = "Density function for a mixture of dis
```

Density function for a mixture of distributions

