LABORATORY DEVELOPMENT OPERATIONS

A CASE (STUDY) FOR BESPOKE SOFTWARE

by

Adam “Kai” Aragaki

A dissertation submitted to Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy

Baltimore, Maryland

November, 2024

Contents

Figures

Science has a reproducibility problem

Science is incremental in nature, and relies on tuning, revising, and refuting previous models. Because of this, the reliability of the information being built upon is central to this endeavor. Needing to assume published information to be correct without replication is sometimes unavoidable — particularly in the case of large studies or studies with very rare patients, where it would be impractical or impossible to replicate. Even in cases where replication is possible, it is impossible to tell if failure to replicate previously published results is the true reality (that is, the published results were in error) or technical differences preclude reproducibility. In recent years, a variety of replication studies have been performed, with the vast majority of studies failing to be replicated. Investigators at Bayer and Amgen were unable to reproduce results in 65% and 89% of cases, respectively[1,2], with other studies presenting similar results (one review indicating a range of 75–90% of preclinical studies failed to replicate[3]). If we refuse to accept this as an unavoidable aspect of science, we must accept that this is a source of immense waste of time, effort, material, and funding.

Science has an *efficiency* problem

Replication and research efficiency are linked by their nature: Poor research produces poorly reproducible or erroneous results, which in turn leads investigators to waste time and effort either attempting to reproduce or build off of previously determined results. An estimated 85% of research effort is 'wasted' through various points in the scientific process[4], and a series of articles in the *Lancet* reports on the various stages in which research efficiency can be improved, including study planning, publishing, regulation, and reporting[5–9]. From this series, it is clear that the sources of waste are from all aspects of the scientific process. While a single 'silver bullet' solution is not possible, it also means that solutions can and must come from a variety of sources, including from within our own labs.

In the series presented by the *Lancet*, Ioannidis et al.[6] mention inefficiencies brought by reporting and analysis. In particular, they mention both the shocking frequency of incorrect p–value reporting (at least one incongruent p–value in 38% of papers in *Nature*, vol 409–412[10]), as well as the poor availability of protocols. While basic science protocols tend to be more readily available in materials and methods sections of papers, these abbreviated formats seem to be insufficient to easily reproduce the experiment — otherwise, we may expect journals like *Nature Protocols* to be much shorter. Other fields have noted the insufficiency of this format as well. In his address, former president of the Royal Statistical Society Peter James Green asserted that the current format of articles did not allow for sufficient detail to reproduce the results shown in the paper[11]. Since his address in 2003, methods have only become more complex, yet method reporting has largely remained the same.

Access to information also defines an area for improvement. There are an assumed vast amount of data that remain unpublished due to negative results[12], despite being informative for other investigators in the field or — in the case of clinical trials — patients. This is compounded by the barrier–to–entry to many journals of the expense of both

publishing and access to papers. While pre–print servers like bioRxiv exist, research implies that authors may be self–filtering their manuscript submissions with null results, rather than journals explicitly selecting manuscripts with significant results more regularly than non-significant results[8]. This may imply that the issue must be addressed at a psychological and social level first.

Avenues to overcome these challenges have been offered for clinical research, but little infrastructure exists for basic research. This may be in part due to the heterogeneity of basic research, where 'endpoints' collected are varied and growing as new experiments and assays are performed and phenomena are discovered. This is an 'on-target' effect of basic research, and trying to mold it to fit the structures created for clinical research would be a detriment to the process. To enhance reproducibility and efficiency, our solutions must be able to mold to our research, not vice versa.

Someone needs to write code. It doesn't have to be you.

Code is a powerful tool for reproducibility. Code can define how data becomes a plot, and can perform analyses with consistency in a way that hand calculation can fail. When shared, it provides an exact window into what was done, and provides an easy way for others to repeat the same analysis (or perhaps a similar analysis with different data). However, writing code is not easy. While myriad high–quality sources for learning to code exist, learning to code takes time and effort, and learning best practices takes longer still. Depending on the situation, it may not be efficient for all members of the lab to become proficient in learning how to code. Rather, it may be more beneficial for there to be a specialist that is capable of producing easy–to–use interfaces that help facilitate best practices at no hassle to the user, via abstraction.

A car salesman is the daily subject to the effects of quantum mechanics, as the entire universe is dictated by it. However, to be a successful car salesman, he needs to know no quantum physics. This is an (extreme) example of (unintentional) abstraction: being able to perform high level tasks without having to understand the particulars of how they happen. It is currently not possible — and likely never will be — to understand everything down to the fundamental level. Abstraction is a necessity, and we should treat it as a small gift rather than a nuisance.

Through abstractions, bench scientists can successfully and easily use applications to help automate their tasks, create reports, and generate reproducible scripts, all without needing to understand how the underlying code works. For this to be fruitful, there must be some overlap in domain knowledge between the bench scientists and developer. A developer who does not understand what a bench scientists wants and what their workflows will produce unnecessary or irritating software that will — at best — not be used.

In the realm of software development, DevOps (a portmanteau of "software development" and "IT operations") is a loosely defined term referring — broadly — to the practice of decreasing development time of robust software via the creation of tooling particular for the team or company in which they work. I believe a similar role should exist for laboratories.

At the McConkey Lab, I had the opportunity to work simultaneously in a wet–lab and dry–lab setting, in a lab that contained largely short term members (1 year) that were primarily focused on wet–lab duties with little to no coding experience. This created a unique opportunity for reproducibility to be enhanced, but with a team that did not have the time to learn how to code. During this time, I developed software infrastructure and learned what things tended to work, and what things were simply lofty ideas. The following sections are descriptions and case studies of some of the software and infrastructure I have developed in my time at the McConkey lab.

Overview

I developed packages on an as-needed basis, typically 'incubating' as a small set of functions in a generic lab package (`bladdr`) before growing them out into a standalone package. This was the case for packages like `qp`, `amplify`, and `ezmtt` Sometimes, patterns would appear in these packages that would warrant a separate 'backend' package, which created packages like `gplate` and `mop`. Still some other packages were created without any direct connection to the network of packages, and were motivated by the needs of current research - like `tidyestimate`, `reclanc`, and `classifyBLCA`.

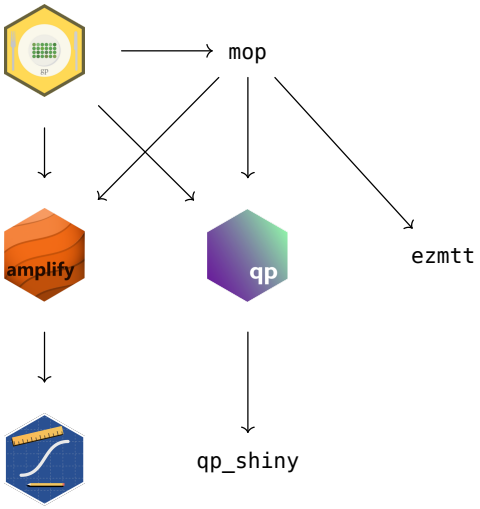


Figure 1: Dependency graph of selected packages

All of these packages exist on a continuum of audience specificity. Some packages or apps are useful only to the members of the lab, as it is tailored to a specific protocol; others are more broadly applicable. One side of the gradient is not necessarily better than the other: large frameworks provide a solid foundation for more rapid future development, while not doing anything on their own. Specific packages and apps may not impact wide swaths of the scientific endeavor, but the users they do help - particularly those who are less familiar with code - are able to get things done.

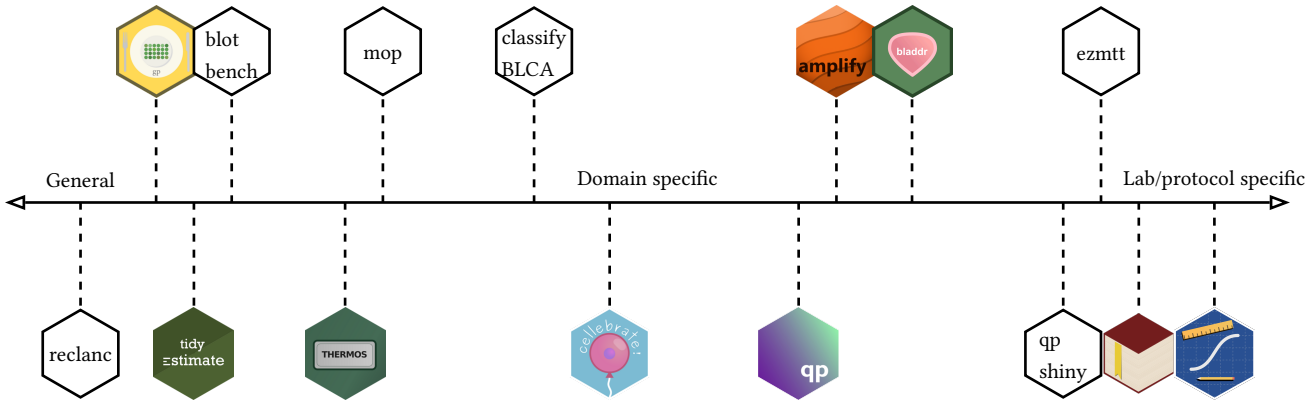


Figure 2: My packages placed pseudo-quantitatively on a gradient of specificity

Documentation

Figure 3: The hex logo for our documentation

Problem

When I first joined the McConkey lab, protocols existed either as printed papers or in a large binder. This has advantages: being able to write on protocols, as well as being able to take protocols into places where an electronic device might not fit — for instance, taped to a fume hood. However, it comes with disadvantages. Paper can become damaged or lost, and in several cases there were 'good' copies with notes, amendments, or differences from the other copies. Additionally, many things were not documented as they might have not made sense as a full–fledged protocol, but were still important knowledge that became 'implicit'. Finally, making the protocols we use in the lab open access allows for greater reproducibility and accountability.

Solution

I created a publicly available repository of protocols at https://kai.quarto.pub/bok. Of the solutions I will present, creating documentation has had the highest impact within our lab, indicating that computational solutions need not be cutting–edge to be helpful.

Some advantages of this approach include the ability to cross–reference and search information, so protocols could be kept brief but still provide prerequisite information a link away, if needed. This also reduces redundancy. For example, the mycoplasma testing protocol need not recount how to split cells, but references it with a link for those who need to refresh their memory. Additionally, this allows like–information to be listed alongside the sought information, allowing users to stumble upon pertinent protocols and information.

This also encouraged the writing of 'protocol adjacent' information that was needed for the lab. For instance, diluting liquids is a frequent pain point for new members in the lab. I wrote a chapter on how to calculate dilutions, as well as the practical aspects of it (eg pipetting less than 1µL of volume is tricky — best scale up). This chapter can serve as a reference for both mentees within and outside of our lab. Maintaining a stable repository of institutional knowledge is important in cases where new students are frequent and/or mentor time is limited.

I wrote the documentation in a Markdown–like language called Quarto, with the end goal of creating something that many students could contribute to. Markdown is an incredibly simple syntax that most users have already come across, though they may not know it by that name. Commonmark — a type (also known as 'flavor') of Markdown — has an incredibly gentle introduction that can be completed in the span of ten minutes. As most of the document is prose, rather than code, most contributors would not have to interact with the more technical aspects of Quarto.

In concert, these aspects have made thorough electronic documentation an incredibly useful piece of infrastructure for our lab, and continues to be my highest–yield endeavor.

Limitations

Contributing to the documentation is non–trivial. While the language that produces the documents (Quarto, a markdown–like language) is fairly simple, there is still a significant barrier to entry for a lab member not familiar with code. An investigator who has no code experience must — at the very least — create a GitHub account and figure out how to modify the script to their liking. Pressed for time and with the lack of incentive and motivation to learn how, it is not particularly mysterious as to why contributions were rare.

Some attempts have been made to combat this. The first is by accepting contributions in a variety of formats. I told users — both in person as well as in the contributions section — that if they were able to email me or otherwise send me some version of their protocol or section, I could transcribe it and add it to the documentation myself. This allowed for a few contributions that would have not otherwise been made.

Another attempted solution was by created a wiki. One of the advantages of a wiki is an editor that allows rapid, visual, *in situ* editing. However, this experiment was met with very little adoption. Data showing if and when users have attempted to log in to the wiki show that only 2/7 users have ever logged in once, and none have edited. This implies that it was not the content or structure of the wiki, but possibly due to lack of interest, motivation, or need.

Conclusions

Despite low levels of contribution, documentation has appeared to be a large source of benefit for members of the lab, with high levels of adoption.

`blotbench`

Problem

A previous report has found that roughly 4% of 20,621 paper analyzed across 40 journals contained inappropriate image duplication[13], particularly with western blots. When preparing western blots for presentation, typical workflows usually involve cropping the blots in something like PhotoShop, GIMP, FIJI, etc. Because this manipulation was done in a separate program, usually the best we can do in terms of reproducibility is by providing the original, unmodified images along with the cropped version (Figure 4).

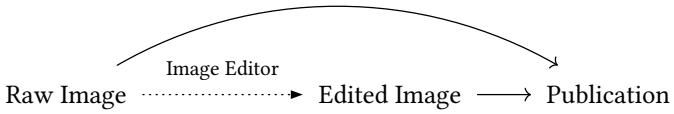


Figure 4: A traditional western blot editing workflow

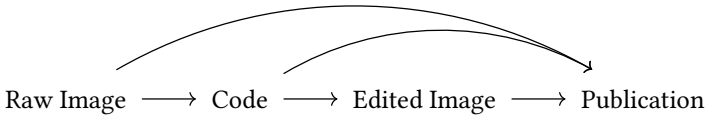


Figure 5: Code-based western blot editing workflow

There are methods for reproducibly manipulating images in R (such as `magick` and `EBImage`) (Figure 5), but they aren't nearly as convenient as the real–time visual feedback of typical photo–editing software — the guess–then–render–repeat loop (Figure 6) of trying to find the perfect cropping geometry for an image without this visual feedback is tedious at best, and convenience and ease are important to promote adoption.

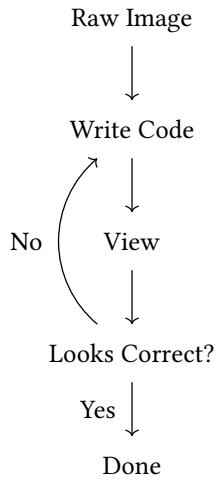


Figure 6: Standard code–based guess–then–render–repeat loop

Solution

`blotbench` attempts to solve this by providing a Shiny app within the package to perform rudimentary image manipulations with visual feedback. This app outputs code that should be written to create these transformations

(rather than the image itself) so the declarative and reproducible benefits of a script can be reaped while still leveraging the convenience of a graphical interface.

In addition, `blotbench` introduces a new object (a `wb` object) that can store row and column annotation much like a `SummarizedExperiment` (Figure 7). This provides additional benefits such as intuitive indexing (which allows for treating a blot image almost like a `data.frame`) and automatic annotation.

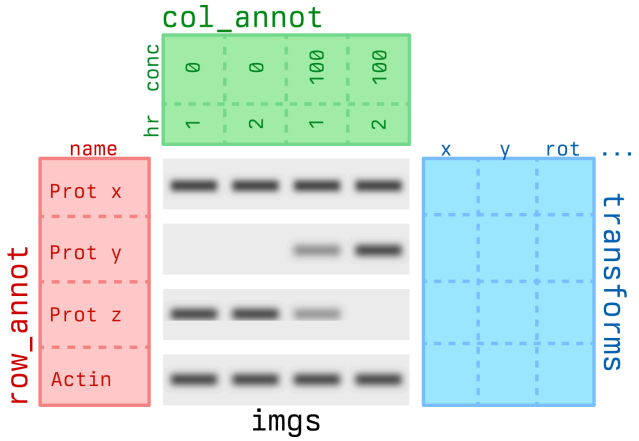


Figure 7: An example `wb` object

Creating a `wb` object

A `wb` object is composed of 4 components:

1. `imgs`: A vector of `image-magick` images
2. `col_annot`: A `data.frame` containing lane annotation, one line for each column, with the top row referring to the left-most lane
3. `row_annot`: A `data.frame` containing names of the protein blotted for in each image.
4. `transforms`: A `data.frame` containing information detailing what transformations should be performed on the image for presentation.

When creating a `wb` object, you typically will not specify the transforms at the outset, and both `col_annot` and `row_annot` are optional. At bare minimum, you need to supply a vector of `image-magick` images. To demonstrate the full capabilities of `blotbench`, I will show an example with both row and column annotation.

Our experiment consisted of cells exposed to a drug (erdafitinib — an FGFR inhibitor) at several timepoints. We blotted for three proteins — TRAIL, PARP, and actin.

Here is our PARP blot:

```
library(blotbench)
library(magick)
parp <- image_read(
  system.file("extdata", "parp.tif", package = "blotbench")
)
plot(parp)
```

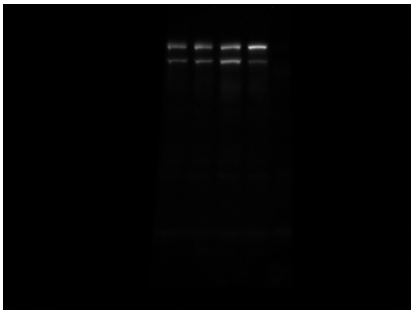


Figure 8: A raw image of a western blot of PARP

After blotting for PARP, we probed the same blot again for TRAIL:

```
trail <- image_read(
  system.file("extdata", "trail.tif", package = "blotbench")
)
plot(trail)
```

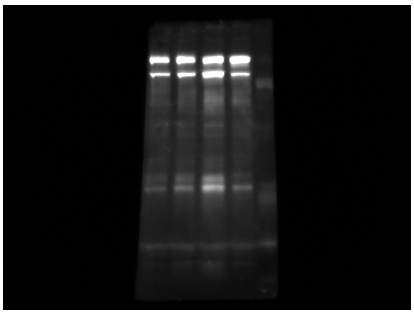


Figure 9: A raw image of a western blot of PARP and TRAIL

And finally for actin:

```
actin <- image_read(
  system.file("extdata", "actin.tif", package = "blotbench")
)
plot(actin)
```

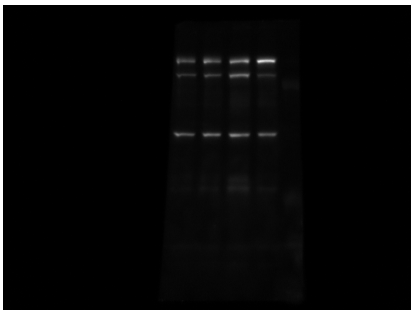


Figure 10: A raw image of a western blot of PARP, TRAIL, and actin

To make the column annotation, create a `data.frame` that has one row per lane in the blot. The columns should represent experimental conditions. The order of the rows should be the order of the columns *after image manipulation*. This is important, as these images are mirrored — we'll flip them the right way once we get on to image manipulation.

```
ca <- data.frame(
  drug = c("DMSO", "Erdafitinib", "Erdafitinib", "Erdafitinib"),
  time_hr = c(0, 24, 48, 72)
)
```

Row annotation can be supplied as a `data.frame` with just one column — `name` — or, much more simply, as a character vector, which is what we'll do here. The order should match the order of images.

With that, we have everything we need:

```
wb <- wb(
  imgs = c(parp, trail, actin),
  col_annot = ca,
  row_annot = c("PARP", "TRAIL", "Actin")
)
```

Editing blots

Now that we have a blot object, we can call `wb_visual_edit` on it to help us generate code to transform out blots:

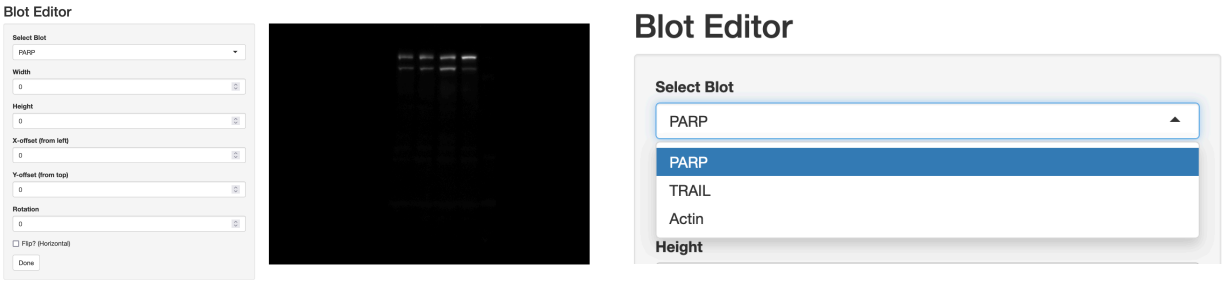


Figure 11: Some screenshots of the visual editor

After editing your individual blots and clicking 'Done', the app will quit and the code to write the transformations will appear in your console:

```
Paste in your script to crop the images as seen in the app:
transforms(wb) <- tibble::tribble(
  ~width, ~height, ~xpos, ~ypos, ~rotate, ~flip,
    190L,     60L,   269,    51,    -0.5,  TRUE,
    190L,     50L,   238,   276,       0,  TRUE,
    190L,     30L,   283,   206,       0,  TRUE
  )
```

Doing so, we get:

```
transforms(wb) <- tibble::tribble(
  ~width, ~height, ~xpos, ~ypos, ~rotate, ~flip,
    190L,     60L,   269,    51,    -0.5,  TRUE,
    190L,     50L,   238,   276,       0,  TRUE,
    190L,     30L,   283,   206,       0,  TRUE
```

```
  )
wb
```

```
imgs
# A tibble: 3 × 7
  format width height colorspace matte filesize density
  <chr>  <int>  <int> <chr>      <lgl>    <int> <chr>
1 TIFF     696    520 Gray       FALSE   728306 83x83
2 TIFF     696    520 Gray       FALSE   728306 83x83
3 TIFF     696    520 Gray       FALSE   728306 83x83

$col_annot
         drug time_hr
1        DMSO       0
2 Erdafitinib      24
3 Erdafitinib      48
4 Erdafitinib      72

$row_annot
   name
1  PARP
2 TRAIL
3 Actin

$transforms
# A tibble: 3 × 6
  width height  xpos  ypos rotate flip
  <int>  <int> <dbl> <dbl>  <dbl> <lgl>
1   190     60   269    51   -0.5 TRUE
2   190     50   238   276    0   TRUE
3   190     30   283   206    0   TRUE

attr(,"class")
[1] "wb"
```

Note that the transforms have not been **applied**: the `imgs` are still the width and height that they were before updating the transformations. This allows you to re–edit the blots if you so desire. The transformations can manually be applied using `apply_transforms`, but they are also automatically applied upon `wb_present`:

```
wb_present(wb)
```

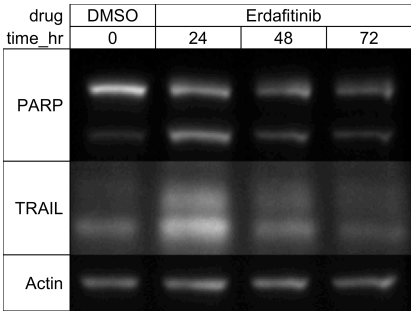


Figure 12: Presented western blot, with transformations automatically applied

If you want to exclude certain proteins, you can index by row just like a `data.frame`:

```
wb_present(wb[-2, ])
```

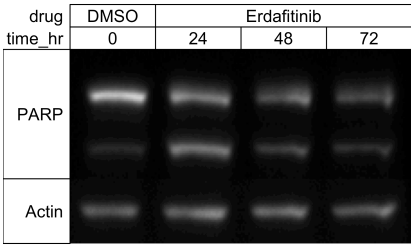


Figure 13: Western blot with TRAIL removed through row indexing

You can additionally select lanes as though they were columns:

```
wb_present(wb[, 2:4])
```

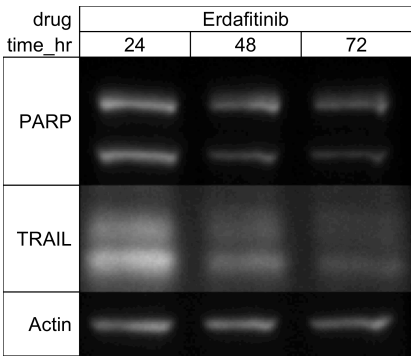


Figure 14: Western blot with 0hr timepoint removed through column indexing

This workflow allows for a relatively painless way to link raw data to output figures. This also makes auditing simple: an outside investigator could be given the raw image and the code used to create the image and compare the output to the provided final blot.

Conclusion

While this will not prevent fraud for those who wish to commit it, hopefully `blotbench` provides a simple mechanism for investigators to transparently show their editing processes. In future version of this application, an improved 'cropping' interface as well as the simultaneous display of all blots can be introduced. Hopefully these features can improve usability and — in turn — increase adoption.

`amplify`, `plan-pcr`

Figure 15: The `amplify` and `plan-pcr` logos

Problem

qPCR assays are routine in the McConkey lab. Both $\Delta\Delta$ C_t qPCR assays as well as standard curve assays are frequently used in our workflows. Despite their regularity and consistency, these calculations were often performed manually on paper or in Excel. This is not only tedious, but leaves room for human error. Furthermore, these calculations were largely seen as intermediate and were usually unsaved.

Downstream visualization is possible using exported data, but challenging if recalculations are necessary. While it is possible to go back to the original software to perform recalculations, the software is not available for all operating systems, and it is not immediately obvious how to recalculate manually and consistently to ensure parity with what the default software would provide.

Solution

`plan-pcr`

`plan-pcr` (https://kai-a.shinyapps.io/plan-pcr/) is a web–based `Shiny` application front–end for the `amplify` package (Figure 16), designed to be used with the in–house $\Delta\Delta$ C_t RT–qPCR assay protocol (https://kai.quarto.pub/bok/dd-ct-pcr.html). With `plan-pcr`, users can log into and upload a raw NanoDrop file or other tabular data describing the concentrations (and optionally sample names).

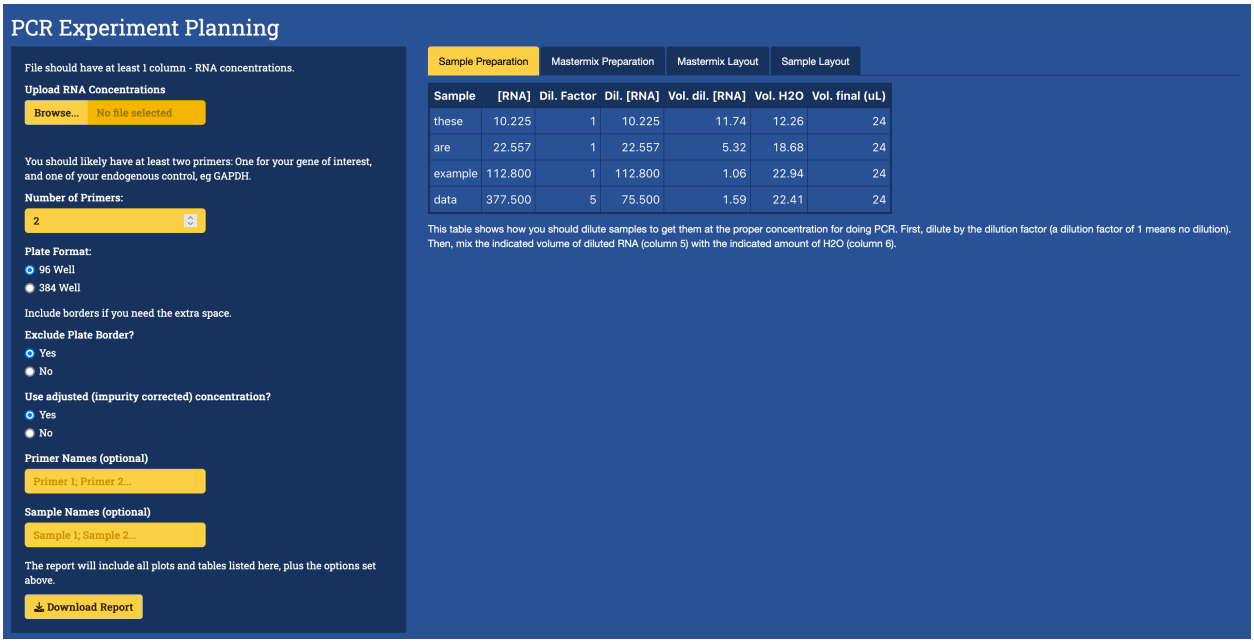


Figure 16: The default landing page for `plan-pcr`

From there, dilutions are calculated with volumes adjusted to the number of selected primers, and mastermix volumes are calculated based on the number of samples (Figure 17).

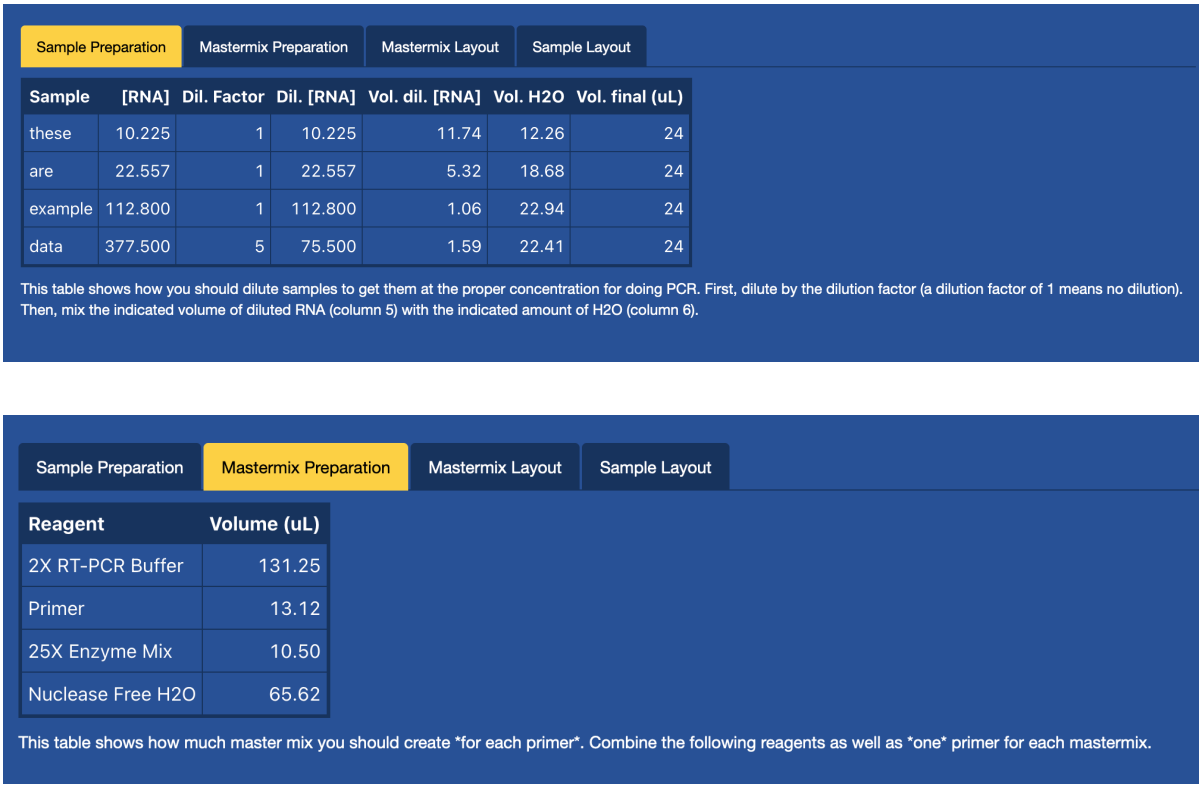


Figure 17: Sample and mastermix preparation tables that react dynamically to arguments set in sidebar

Additional features include the ability to error if a user is trying to plate an experiment that is larger than the selected plate has space for, or warn if the user may have forgotten a control probe (such as GAPDH). A no–template control (NTC) is included by default, and mastermix volumes increased to account for it. Finally, suggested plate layouts are created, as well as the ability to download a report that specifies all details provided and calculated for the experiment (Figure 18).

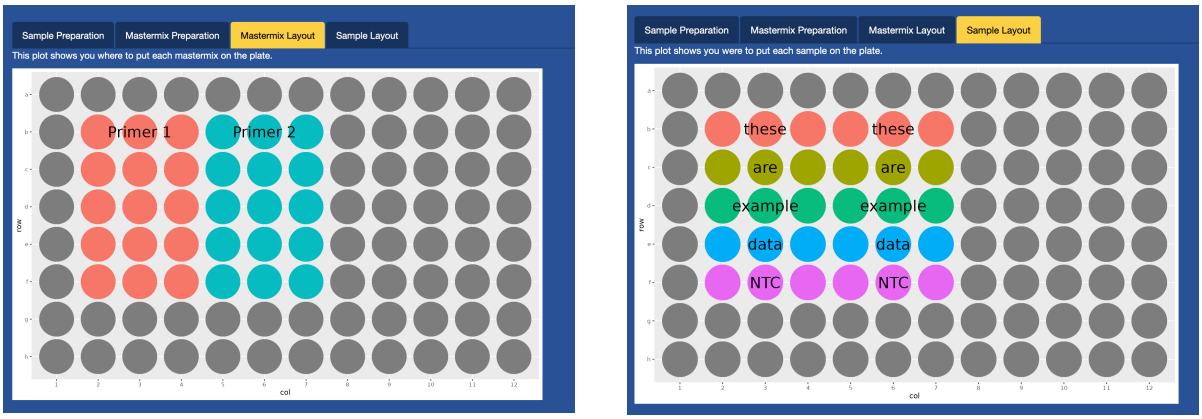


Figure 18: Suggested mastermix and sample layouts

amplify

`amplify` is the backend library that drives `plan-pcr`, and thus anything that can be done in `plan-pcr` can also be done in `amplify`. This is useful for those that prefer a code–based workflow (especially useful in the case of automation).

Consider a toy dataset included with `amplify`:

```
library(amplify)
dummy_rna_conc
```

```
       sample   conc
1 24hr_DMSO_1 126.80
2 48hr_DMSO_1  93.00
3  24hr_1uM_1 143.07
4  48hr_1uM_1  67.09
5 24hr_DMSO_2  88.32
6 48hr_DMSO_2 123.53
7  24hr_1uM_2  94.24
8  48hr_1uM_2  80.18
```

All arguments that can be set in the sidebar of the web app can be set in arguments to the function `pcr_plan`:

```
planned <- pcr_plan(
  data = dummy_rna_conc,
  n_primers = 2,
  format = 384,
  exclude_border = TRUE,
  primer_names = c("GENEX", "GENEY")
)

planned
```

```
$mm_prep
# A tibble: 4 × 2
  reagent             vol
  <chr>             <dbl>
1 2X RT-PCR Buffer  206.
2 Primer             20.6
3 25X RT-PCR Enzyme  16.5
4 Nuclease Free H2O 103.

$sample_prep
# A tibble: 8 × 7
  sample       conc dilution_factor diluted_concentration diluted_rna_to_add
  <chr>       <dbl>           <int>                 <dbl>              <dbl>
1 24hr_DMSO_1 127.                5                  25.4               4.73
2 48hr_DMSO_1  93                 1                  93                 1.29
3 24hr_1uM_1  143.                5                  28.6               4.19
4 48hr_1uM_1   67.1               1                  67.1               1.79
5 24hr_DMSO_2  88.3               1                  88.3               1.36
6 48hr_DMSO_2 124.                5                  24.7               4.86
7 24hr_1uM_2   94.2               1                  94.2               1.27
8 48hr_1uM_2   80.2               1                  80.2               1.50
# ℹ 2 more variables: water_to_add <dbl>, final_vol <dbl>

$plate

      3
   ______
1 | ○ ○ ○
```

```
Start corner: tl
Plate dimensions: 16 x 24

$n_primers
[1] 2

$format
[1] "384"

$exclude_border
[1] TRUE

$primer_names
[1] "GENEX" "GENEY"
```

A report can be generated (identical to the web version) using `pcr_plan_report`

```
pcr_plan_report(planned, "~/path/to/report.html")
```

Analysis

`amplify` is also capable of downstream analysis, after PCR is performed and data is exported.

ΔΔ C_t qPCR

`amplify` contains example data from a ΔΔ C_t qPCR experiment. Since these data are in a non–standard format, `amplify` also includes functions to convert them into a workable `data.frame`. These functions (`read_pcr`, `scrub`) are from the package `mop`, which will be covered in detail later sections of this chapter.

```
dat <- system.file("extdata", "untidy-pcr-example-2.xlsx", package = "amplify") |>
  read_pcr() |>
  scrub()
dat
```

```
# A tibble: 384 × 41
    .row  .col  well well_position omit  sample_name target_name task   reporter
   <dbl> <dbl> <dbl> <chr>         <lgl> <chr>       <chr>       <chr>  <chr>
 1     1     1     1    NA NA            NA    NA          NA          NA     NA
 2     1     2     2     2 A2            FALSE UC3 PBS     CDH1        UNKNO… FAM
 3     1     3     3     3 A3            FALSE UC3 PBS     CDH1        UNKNO… FAM
 4     1     4     4     4 A4            FALSE UC3 Drug    CDH1        UNKNO… FAM
```

```
 5     1     5     5 A5            FALSE UC3 Drug    CDH1        UNKNO… FAM
 6     1     6     6 A6            FALSE UC3 Drug    CDH1        UNKNO… FAM
 7     1     7     7 A7            FALSE T24 PBS     CDH1        UNKNO… FAM
 8     1     8     8 A8            FALSE T24 PBS     CDH1        UNKNO… FAM
 9     1     9     9 A9            FALSE T24 PBS     CDH1        UNKNO… FAM
10     1    10    10 A10           FALSE T24 Drug    CDH1        UNKNO… FAM
# ℹ 374 more rows
# ℹ 32 more variables: quencher <chr>, quantity <lgl>, quantity_mean <lgl>,
#   quantity_sd <lgl>, rq <dbl>, rq_min <dbl>, rq_max <dbl>, ct <dbl>,
#   ct_mean <dbl>, ct_sd <dbl>, delta_ct <lgl>, delta_ct_mean <dbl>,
#   delta_ct_sd <dbl>, delta_ct_se <dbl>, delta_delta_ct <dbl>,
#   automatic_ct_threshold <lgl>, ct_threshold <dbl>, automatic_baseline <lgl>,
#   baseline_start <dbl>, baseline_end <dbl>, comments <lgl>, expfail <chr>, …
# ℹ Use `print(n = ...)` to see more rows
```

Typically, it's useful to get a 'bird's eye view' of the data — particularly if return to the data after a long time. `amplify` includes two functions to do this. `pcr_plate_view` allows users to look at the data as though they were looking at the original plate, while `pcr_plot` plots the relative quantities of the primers in a traditional bar–plot format, stratified by primer.

By default, `pcr_plate_view` uses `target_name` as the variable to color on, showing the layout of the primers:

```
pcr_plate_view(dat)
```

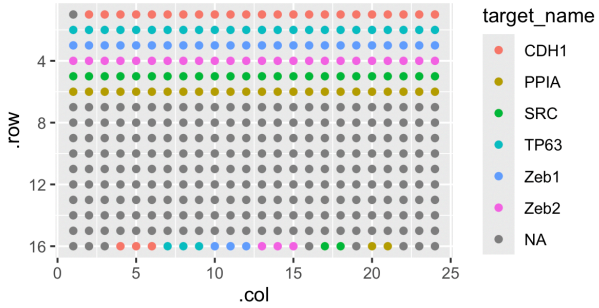


Figure 19: Plate view of primer layout

However, any column of `dat` can be used:

```
pcr_plate_view(dat, sample_name)
```

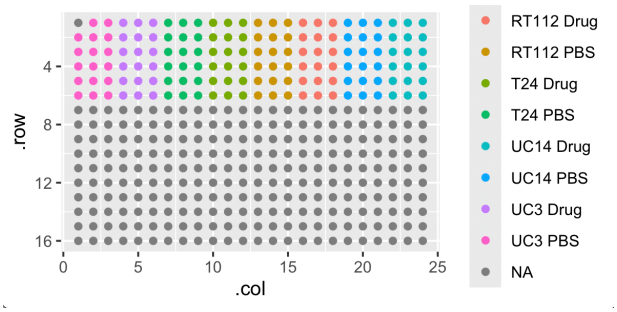


Figure 20: Plate view of sample layout

By plotting `ct`, we can see that some samples didn't seem to amplify at all:

```
library(ggplot2)
pcr_plate_view(dat, ct) +
  scale_color_viridis_c(end = 0.9)
```

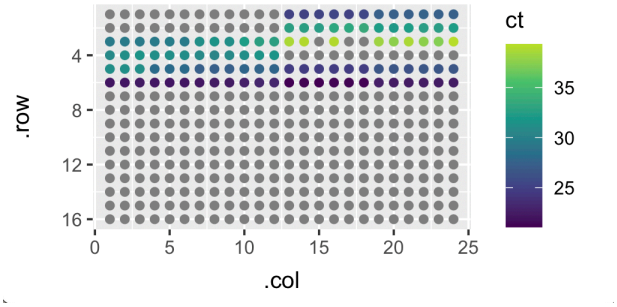


Figure 21: Plate view of C_t values

`pcr_plot` can quickly show RQ values:

```
pcr_plot(dat)
```

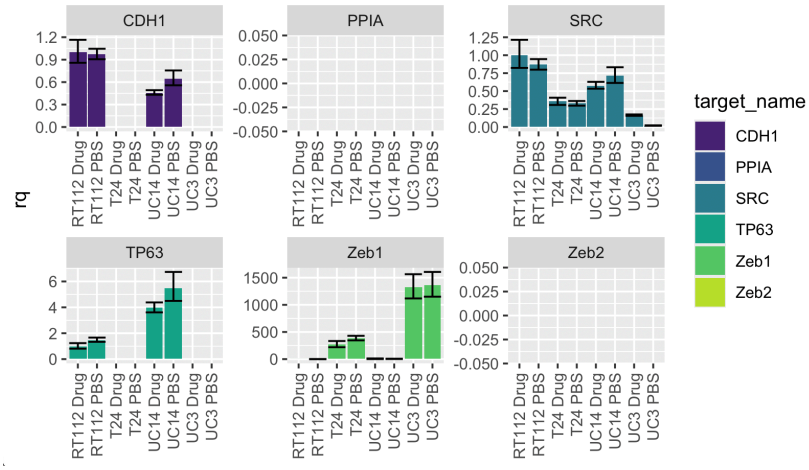


Figure 22: Default plot of relative quantities

One thing you'll notice is that the control probe (here PPIA) appears to have no expression. This is the default value exported by QuantStudio, and due to how $\Delta\Delta$ C_t values are calculated, they should all be 1. However, they do have a certain amount of spread, which can be useful to visualize. We can force the RQ values to be calculated for all probes by using `pcr_rq`, which renormalizes the expression to a given sample name. If we supply the current relative sample (RT112 Drug), the only thing that will change is the addition of RQ values for PPIA:

```
pcr_rq(dat, "RT112 Drug") |>
  pcr_plot()
```

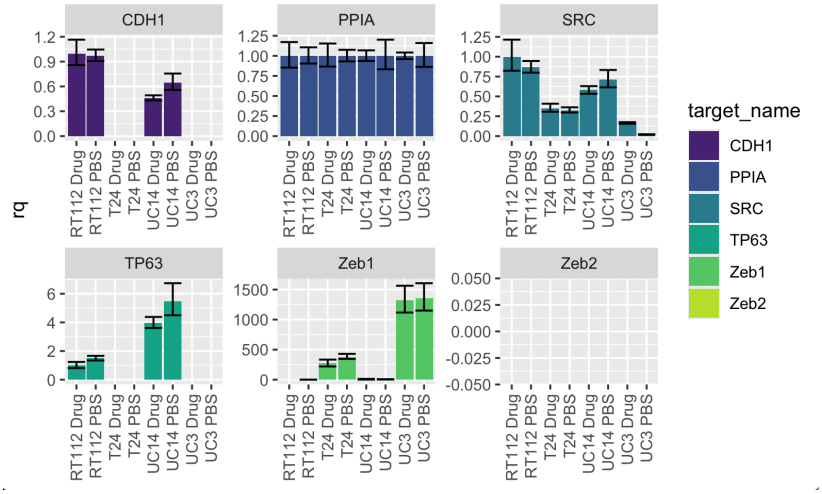


Figure 23: Plot of relative quantities with PPIA relative quantities included

We can use the `group` parameter to normalize within a given subgroup. This is useful for our case, since it might make more sense to normalize within each cell line, rather than globally.

```
dat |>
  tidyr::separate(sample_name, c("cell_line", "sample_name"), remove = FALSE) |>
  pcr_rq("PBS", group = "cell_line") |>
  pcr_plot() +
  ggplot2::facet_grid(cell_line~target_name) +
  ggplot2::scale_y_log10()
```

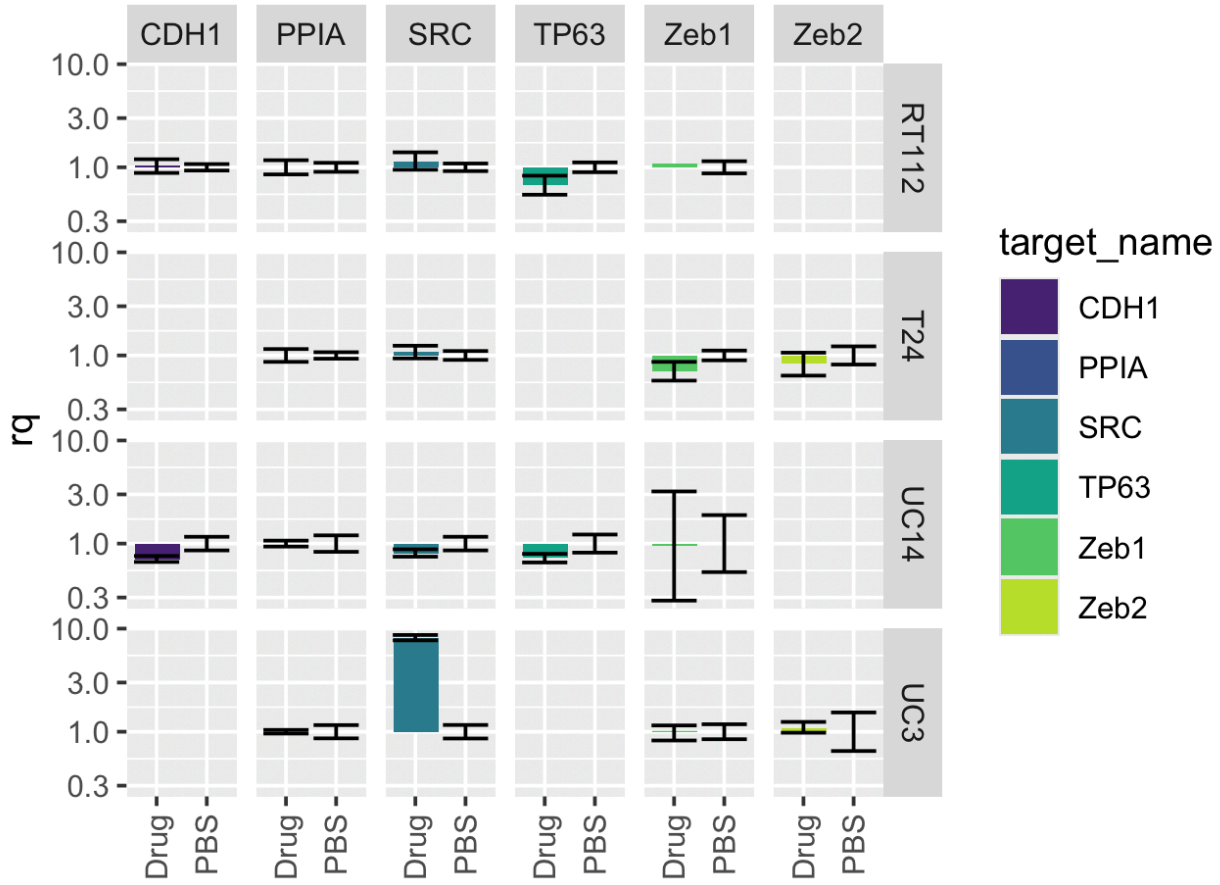


Figure 24: RQ values normalized to their respective cell line controls

We can see from these data that TP63 tends to be down-regulated in our luminal (RT112, UC14) lines upon exposure to our drug (but not our basal lines, T24 and UC3, which appear not to express TP63).

Standard Curves PCR

Another routine PCR task in the McConkey lab is PCR to quantify library concentration. This is to ensure the correct absolute amount of library is loaded into the chip, to ensure equal balancing and thus equal depth of coverage for both samples within the same chip (multiplexing via barcoding), as well as additional samples across other chips for a given study.

`amplify` comes with toy data from one of these experiments:

```
tidy_lib <- system.file("extdata", "untidy-standard-curve.xlsx", package = "amplify") |>
  read_pcr() |>
  tidy_lab(pad_zero = TRUE)
```

(The `pad_zero` argument converts sample names from "Sample 1" to "Sample 01" to ensure the proper order upon lexicographic ordering)

By default, `pcr_tidy` assumes a standards serial dilution starting at 6.8, diluted by a factor of 10, going all the way down to 0.00068, and that all of them should be included. There are a couple instances in which this might not be the case:

1. Different serial dilutions were used
2. A particularly bad standard is making slope calculations inaccurate

In that instance, supply a numeric vector to the `usr_standards` argument. If you wish to omit a given set of standards, simply do not include them in this vector:

```
custom_lib <- system.file("extdata", "untidy-standard-curve.xlsx", package = "amplify") |>
  read_pcr() |>
  tidy_lab(pad_zero = TRUE, usr_standards = c(6.8, .68, .068, .0068))

scrub(custom_lib) |>
  dplyr::filter(task == "STANDARD") |>
  dplyr::select(sample_name, quantity)
```

```
   sample_name quantity
   <chr>          <dbl>
 1 Standard 01  6.80
 2 Standard 01  6.80
 3 Standard 01  6.80
 4 Standard 02  0.680
 5 Standard 02  0.680
 6 Standard 03  0.0680
 7 Standard 03  0.0680
 8 Standard 03  0.0680
 9 Standard 04  0.00680
10 Standard 04  0.00680
11 Standard 04  0.00680
```

This will automatically update the slope column of the dataframe as well. This can be called standalone (say, after manually removing a few standards replicates from your dataset) by running `pcr_calc_slope`.

Library concentrations can easily be calculated with `pcr_lib_calc`, specifying the sample dilution factor with `dil_factor` (here 1:1000):

```
lib_conc <- tidy_lib |>
  pcr_lib_calc(dil_factor = 1000)

lib_conc |>
  scrub() |>
  dplyr::filter(task == "UNKNOWN") |>
  dplyr::select(sample_name, concentration)
```

```
# A tibble: 42 × 2
   sample_name concentration
   <chr>               <dbl>
 1 Sample 06           2039.
 2 Sample 06           2039.
 3 Sample 06           2039.
 4 Sample 12           1893.
 5 Sample 12           1893.
 6 Sample 12           1893.
 7 Sample 04           1694.
 8 Sample 04           1694.
 9 Sample 04           1694.
10 Sample 16           1493.
# ℹ 32 more rows
# ℹ Use `print(n = ...)` to see more rows
```

Routine quality control plots can also be generated quickly. First, the data is generated using `pcr_lib_qc`:

```
qc <- pcr_lib_qc(lib_conc)
```

This output is generally not useful by itself. Using `pcr_lib_qc_plot_*` functions on it, however, generates plots that display valuable visual QC summaries

Making standard curve for libraries requires making a serial dilution of standards. It is important we determine that this serial dilution was diluted properly, or the results calculated from it will be unreliable. The standard dilution plot helps with this:

```
pcr_lib_qc_plot_dil(qc)
```

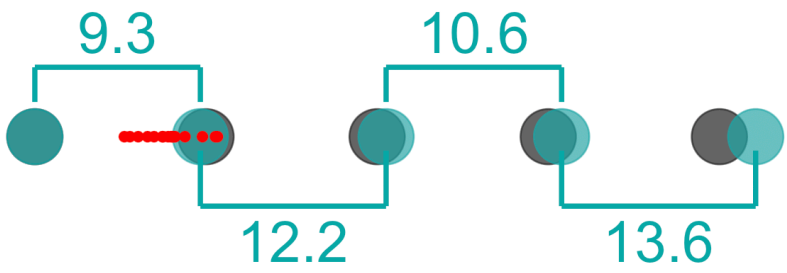


Figure 25: A standard curve quality control plot

From this plot, we can see the relative dilution factors between samples. From this example, we can see a 9.3x dilution between the first and second, 12.2x dilution between second and third, etc. They gray dots represent where our blue dots should land if all the dilutions are perfect AND if efficiency is 100%. The red dots represent where the samples lie.

This plot can catch three sources of issues:

- Inconsistent pipetting, which would show dilution factors widely varying from 10x or
- Systematically incorrect pipetting, which would show dilutions consistently below or above 10x or
- Poor efficiency of the enzyme, which would appear to show dilutions consistently above 10x.

Determining efficiency issues vs consistently under-pipetting, however, is impossible to determine with the data alone.

Another useful diagnostic plot involves plotting the logarithm of the known concentrations of standards against their observed C_t values:

```
pcr_lib_qc_plot_slope(qc)
```

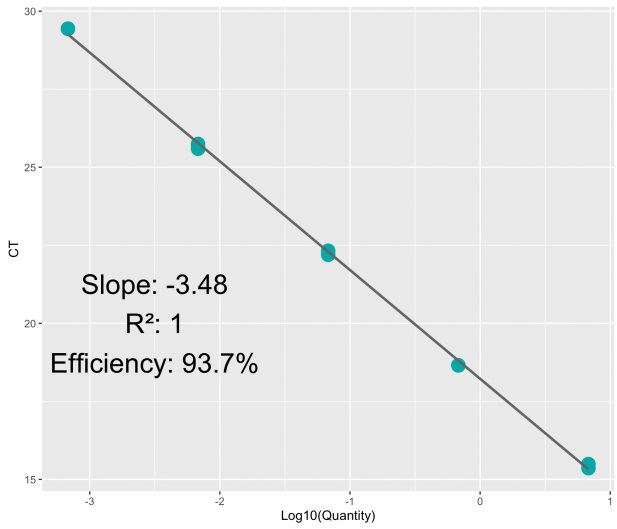


Figure 26: A diagnostic plot of log(quantity) versus log(C_t)

In this plot, the log10 of the theoretical values of the standards is plotted against the C_t values of the standards. In a perfect world, where the enzyme perfectly doubles the amount of product each cycle, we would expect that a standard 1/10th of the concentration would reach the same level of amplification in around 3.3 cycles ($2^{3.3}$ is approximately 10). Thus, in a perfect world we expect to see a slope of −3.3, an R^2 of 1, and an efficiency of 100%.

Additional diagnostic plots, such as outlier detection and removal, can be generated along with all the other plots and combined in a report using the `pcr_lib_qc_report` function. This report includes annotations for each plot to help users interpret their results.

Conclusion

`amplify` (which powers `plan-pcr`) provides straightforward and consistent means for both experimental setup and analysis for ΔΔ C_t and standard curve PCR experiments. Importantly, it does not attempt to be an all purpose tool, but rather to be tailored specifically to routine tasks in the McConkey lab.

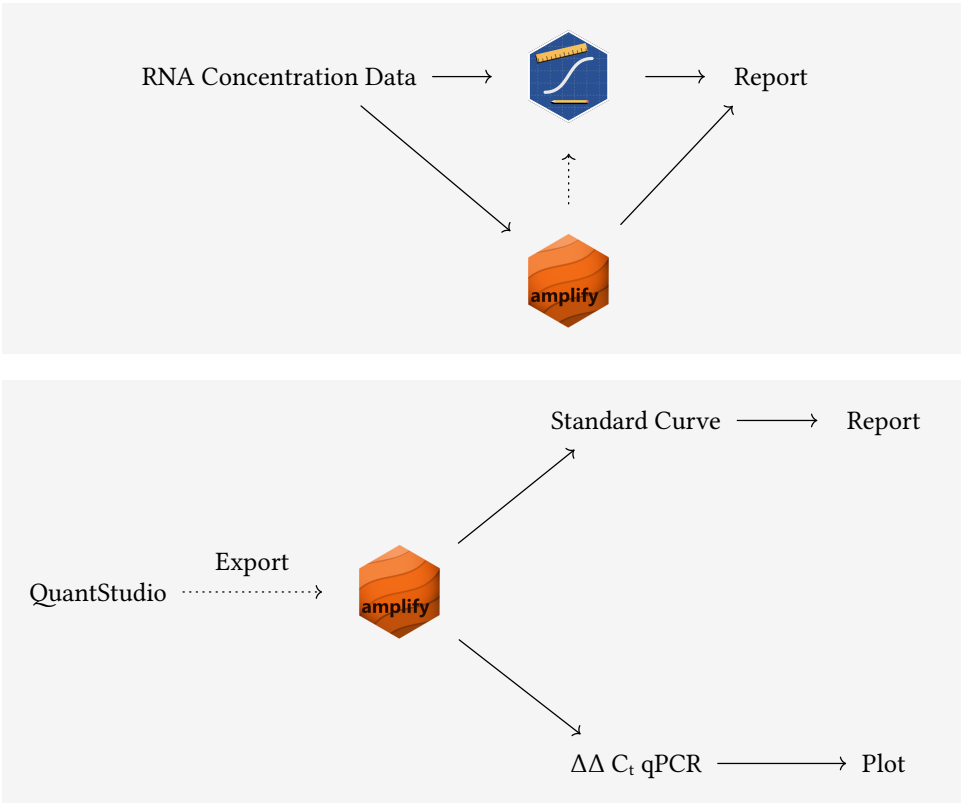


Figure 27: A broad overview of workflows. Top: Workflows for experiment setup. Bottom: Workflows for result analysis

`gplate`

Figure 28: The `gplate` package hex logo

Problem

Microwell plates are usually arranged in visually meaningful ways but are not tidy data ('tidy' is a format in which each column of data is a variable, each row is a 'case', and each cell/box is a value), and their manipulation to and from a tidy form is cumbersome. While packages like `plater` exist, they require creation of a template file, which limits its usefulness in a programmatic setting and reduces reproducibility.

Solution

`gplate` provides a grammar of plates. The goal is to provide a succinct yet flexible language for specifying a variety of common plate layouts. This allows for both rapid tidying of data as well as plotting plate layouts, useful for instances like writing documentation (for protocols specifying plate layouts), user interfaces, and quality control (such as to check for spatial patterns of variables).

`gplate` has three main verbs:

- `gp`, which creates the plate
- `gp_sec`, which adds sections to the plate
- `gp_plot`, which plots the plate with a variable overlaid

Plotting

`gplate` can be used to plot plate layouts. To begin, we first need to create a `gp` object:

```
gp <- gp(rows = 8, cols = 12)
```

Then we can plot it with `gp_plot`:

```
gp_plot(gp)
```

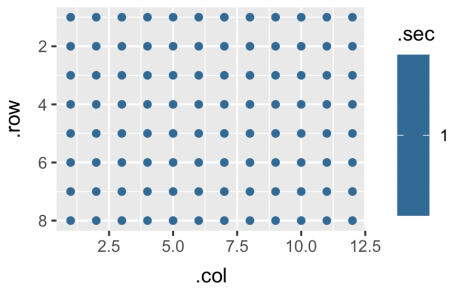


Figure 29: A `gp`, plotted

By default, this is not particularly useful. By adding 'sections', we can begin to specify common plate layouts. At its simplest, consider a plate divided into four quadrants:

```
gp_quads <- gp_sec(
  gp, name = "quadrant", nrow = 4, ncol = 6
)
gp_plot(gp_quads)
```

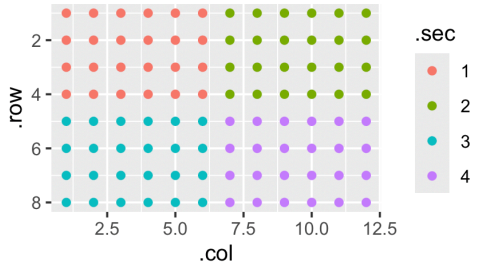


Figure 30: A gp plotted with quadrant sections

Sections are, by default, 'nested' — meaning sections can themselves have sections:

```
gp_split <- gp_sec(
  gp_quads, name = "split", nrow = 2, ncol = 3
)
gp_plot(gp_split)
```

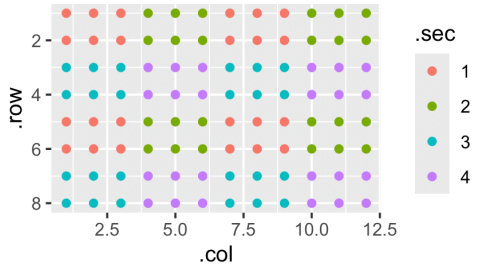


Figure 31: A gp where the sections have sections

By default, `gp_plot` will plot the most recently added section — but lower sections can still be plotted by their section `name`:

```
gp_plot(gp_split, quadrant)
```

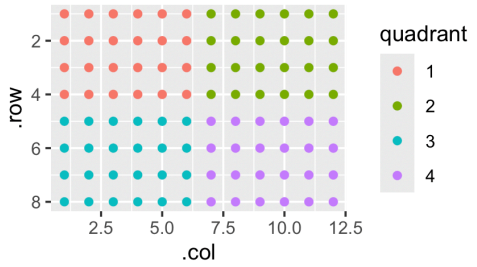


Figure 32: A gp with multiple layers, with a previous layer plotted

Various arguments to `gp_sec` allow for more flexible specifications:

```
with_margin <- gp_sec(
  gp, name = "with_margin", nrow = 3, ncol = 3, margin
= 1
)
gp_plot(with_margin)
```

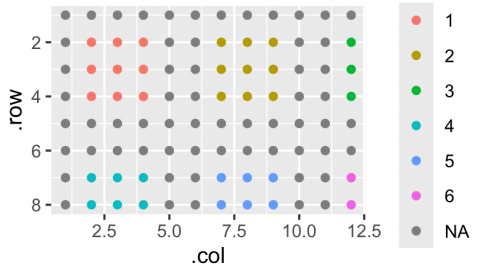


Figure 33: A gp where each section has a margin

```
with_wrapping <- gp_sec(
  gp, name = "with_wrap", nrow = 3, ncol = 7, wrap
= TRUE
)
gp_plot(with_wrapping)
```

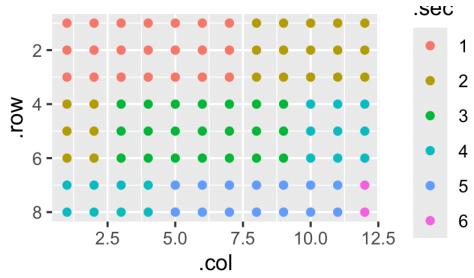


Figure 34: A gp where sections can wrap to the next section below

```
no_breaking <- gp_sec(
  gp, name = "no_break", nrow = 3, ncol = 7,
  break_sections = FALSE
)
gp_plot(no_breaking)
```

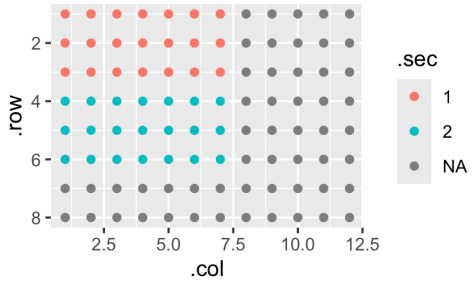


Figure 35: A gp where only whole sections are allowed

```
flow_by_col <- gp_sec(
  gp, name = "flow_col", nrow = 3, ncol = 7,
  wrap = TRUE, flow = "col"
)
gp_plot(flow_by_col)
```

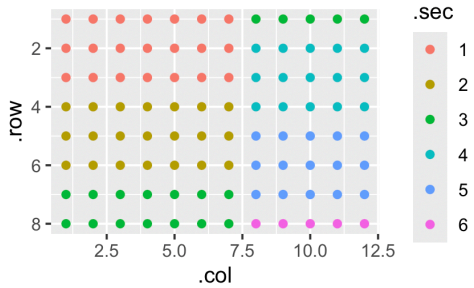


Figure 36: A gp where the next section is in the same column, rather than the same row

These can be combined for highly elaborate designs:

```
with_wrapping <- gp_sec(
  gp, name = "with_wrap", nrow = 4, ncol = 4,
  margin = c(1, 1, 0, 0), wrap = TRUE, flow = "col"
)
gp_plot(with_wrapping)
```

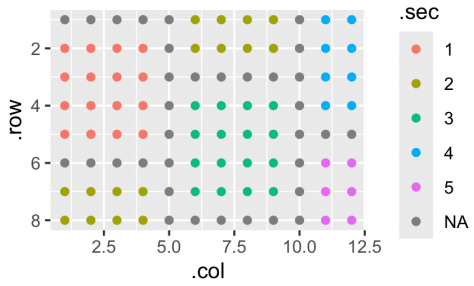


Figure 37: A highly elaborate gp

Tidying

With gplate, plotting and tidying go hand–in–hand. As a motivating example, gplate comes with absorbance data from a 96 well plate from a protein quantification (BCA) assay.

```
protein_quant
```

```
      [,1]   [,2]   [,3]   [,4]   [,5]   [,6]   [,7]   [,8]   [,9]  [,10]  [,11]  [,12]
[1,] 0.0691 0.0801 0.0978 0.1212 0.1731 0.2395 0.3812 0.2402 0.2593 0.2525 0.2371 0.2572
[2,] 0.0693 0.0810 0.0966 0.1247 0.1732 0.2454 0.3988 0.2527 0.2636 0.2636 0.2419 0.2616
[3,] 0.0711 0.0827 0.1011 0.1256 0.1855 0.2466 0.3967 0.2515 0.2580 0.2602 0.2422 0.2608
[4,] 0.2735 0.2725 0.2583 0.2708 0.2693 0.2749 0.2610 0.0739 0.0718 0.0715 0.0682 0.0651
[5,] 0.2501 0.2634 0.2559 0.2630 0.2650 0.2629 0.2548 0.0696 0.0667 0.0646 0.0621 0.0622
[6,] 0.2549 0.2699 0.2513 0.2578 0.2588 0.2624 0.2463 0.0726 0.0727 0.0725 0.0710 0.0708
[7,] 0.0799 0.0951 0.0805 0.0796 0.0768 0.0792 0.0774 0.0762 0.0766 0.0767 0.0760 0.0784
[8,] 0.0456 0.0456 0.0505 0.0469 0.0469 0.0476 0.0474 0.0457 0.0456 0.0474 0.0467 0.0457
```

Coincidentally, the process of describing the layout of the plate through plotting and the process of tidying the data are one and the same:

Each sample is in triplicate, and each triplicate stands next to one another moving from left to right, wrapping around to the next 'band' of rows when it hits an edge. Or, more simply:

```
gp(8, 12) |>
  gp_sec("samples", nrow = 3, ncol = 1) |>
  gp_plot(samples) +
  ggplot2::theme(legend.position = "none")
```

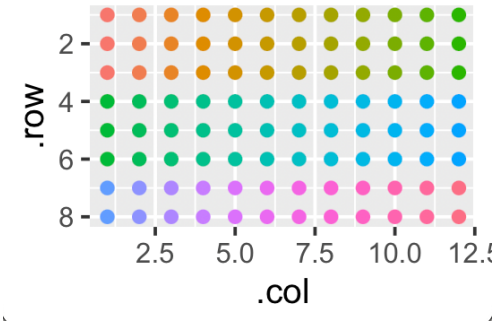


Figure 38: Protein quantification sample placement strategy

However, there are some wells that have sample in them, and some that are empty. To specify the difference between the two:

```
gp(8, 12) |>
  gp_sec(
    "has_sample", nrow = 3, ncol = 19,
    wrap = TRUE, labels = "sample"
  ) |>
  gp_plot(has_sample)
```

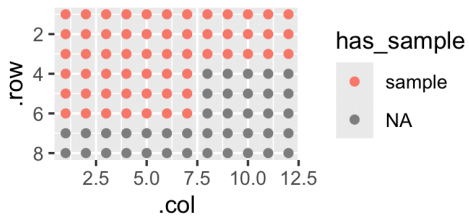


Figure 39: Wells that have samples

Notice the `wrap = TRUE` — this allows for sections that are bigger than the 'parent section' (here the plate) by wrapping them around to the next 'band'.

To label each replicate as a number of a triplicate — the top sample is 1, the middle is 2, and the bottom is 3 — we do:

```
gp(8, 12) |>
  gp_sec(
    "has_sample", nrow = 3, ncol = 19,
    wrap = TRUE, labels = "sample"
  ) |>
  gp_sec("replicate", nrow = 1) |>
  gp_plot(replicate)
```

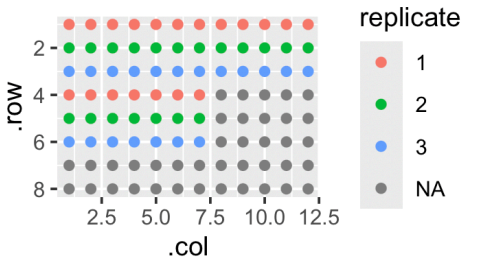


Figure 40: Technical replicate for each sample

Here, specifying `ncol` is not necessary. This is because by default, a section will take up the maximum space possible (here 19).

Some of these samples make up a standard curve, while others make up 'unknowns'. Note how I specify a vector `c(7, 12)` to denote two differently sized sections, labeled as shown in the `labels` argument

```
gp(8, 12) |>
  gp_sec(
    "has_sample", nrow = 3, ncol = 19,
    wrap = TRUE, labels = "sample"
  ) |>
  gp_sec("replicate", nrow = 1, advance = F) |>
  gp_sec(
    "type", nrow = 3, ncol = c(7, 12),
    labels = c("standard", "sample")
  ) |>
  gp_plot(type)
```

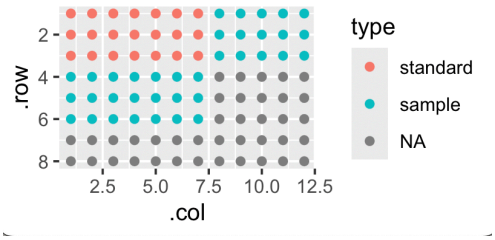


Figure 41: Annotating sample type

Note the addition of the argument `advance = F` in the previous section. This ensures that the next section — type — will be a sibling of replicate, rather than its child. That is, we continue to annotate relative to `has_sample` rather than annotating relative to replicate.

Finally, I'm going to give an index for each sample:

```
gp(8, 12) |>
  gp_sec(
    "has_sample", nrow = 3, ncol = 19,
    wrap = TRUE, labels = "sample"
  ) |>
  gp_sec("replicate", nrow = 1, advance = F) |>
  gp_sec(
    "type", nrow = 3, ncol = c(7, 12),
    labels = c("standard", "sample")
  ) |>
  gp_sec("sample", ncol = 1) |>
  gp_plot(sample) +
  theme(
    # Too many samples - clutters the plot
    legend.position = "none"
  )
```

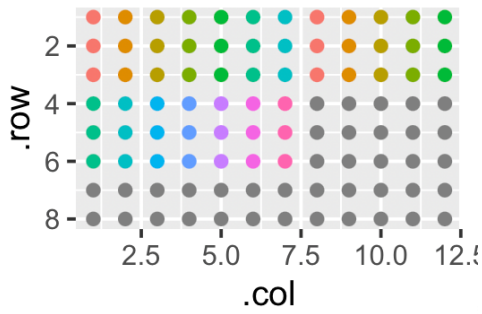


Figure 42: Finished annotated `gp` with sample indices

Now, the fun part: since we described our data so well, tidying it is very easy. First, we supply our data as the third argument of `gp`:

```
my_plate <- gp(8, 12, protein_quant) |>
  gp_sec(
    "has_sample", nrow = 3, ncol = 19,
    wrap = TRUE, labels = "sample"
  ) |>
  gp_sec("replicate", nrow = 1, advance = F) |>
  gp_sec(
    "type", nrow = 3, ncol = c(7, 12),
    labels = c("standard", "sample")
  ) |>
  gp_sec("sample", ncol = 1)
```

And now we use `gp_serve`:

```
gp_serve(my_plate) |>
  dplyr::arrange(.row, .col) |>
  head(20)
```

```
# A tibble: 20 × 7
    .row  .col  value has_sample replicate type     sample
   <int> <int>  <dbl> <fct>      <fct>     <fct>    <fct>
 1     1     1 0.0691 sample     1         standard 1
 2     1     2 0.0801 sample     1         standard 2
 3     1     3 0.0978 sample     1         standard 3
 4     1     4 0.121  sample     1         standard 4
 5     1     5 0.173  sample     1         standard 5
 6     1     6 0.240  sample     1         standard 6
 7     1     7 0.381  sample     1         standard 7
 8     1     8 0.240  sample     1         sample   1
 9     1     9 0.259  sample     1         sample   2
10     1    10 0.252  sample     1         sample   3
11     1    11 0.237  sample     1         sample   4
12     1    12 0.257  sample     1         sample   5
13     2     1 0.0693 sample     2         standard 1
14     2     2 0.081  sample     2         standard 2
15     2     3 0.0966 sample     2         standard 3
16     2     4 0.125  sample     2         standard 4
17     2     5 0.173  sample     2         standard 5
18     2     6 0.245  sample     2         standard 6
19     2     7 0.399  sample     2         standard 7
20     2     8 0.253  sample     2         sample   1
```

Note that each well is properly annotated with its initial absorbance (in the `value` column), as well as all other details pertaining to each sample.

Conclusion

`gplate` provides a succinct yet flexible grammar to allow for both plotting and tidying of microwell plate data. This package has been used to create the plate interfaces for `plan-pcr`, as well as for the `pcr_plate_view` function in `amplify`.

`mop`

Problem

Bench science typically involves receiving data from various machines, all with their own data output. Unfortunately, this format of data is rarely — if ever — conducive for data science. As Hadley Wickham (referencing Tolstoy) said in his seminal 'Tidy Data' paper[14],

> *Tidy datasets are all alike but every messy dataset is messy in its own way*

Standard workflows typically involve manual extraction of the data from the output file into a spreadsheet. While this is straightforward, it is neither reproducible, nor easily automated.

Solution

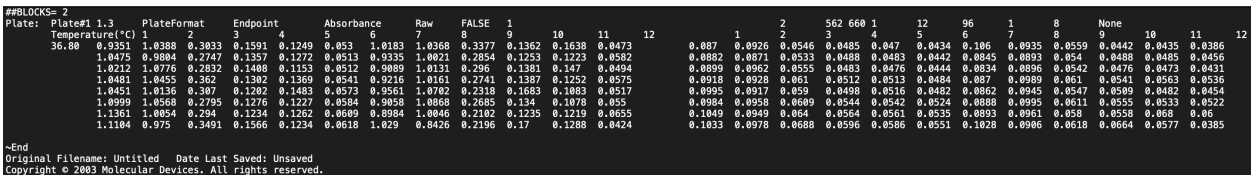
In a similar vein, the package `broom` has been developed by Robinson et al.[15] to turn the heterogeneous outputs from various statistical tests into a normalized format. `mop` seeks to be the wet–lab analogue to `broom`, by providing a library of tidying methods for lab equipment.

As an example, consider the output from the SpectraMax Plus 384 microplate reader:

```
\u{23}\u{23}BLOCKS= 2
Plate:  Plate\u{23}1  1.3 PlateFormat Endpoint  Absorbance  Raw FALSE 1          2 562 660 1
12  96  1 8 None
  Temperature(°C) 1 2 3 4 5 6 7 8 9 10  11  12    1 2 3 4 5 6 7 8 9 10  11  12
  36.80 0.9351  1.0388  0.3033  0.1591  0.1249  0.053 1.0183  1.0368  0.3377  0.1362  0.1638
0.0473    0.087 0.0926  0.0546  0.0485  0.047 0.0434  0.106 0.0935  0.0559  0.0442  0.0435  0.0386
   1.0475  0.9804  0.2747  0.1357  0.1272  0.0513  0.9335  1.0021  0.2854  0.1253  0.1223  0.0582
0.0882  0.0871  0.0533  0.0488  0.0483  0.0442  0.0845  0.0893  0.054 0.0488  0.0485  0.0456
    1.0212  1.0776  0.2832  0.1408  0.1153  0.0512  0.9089  1.0131  0.296 0.1381  0.147 0.0494
0.0899  0.0962  0.0555  0.0483  0.0476  0.0444  0.0834  0.0896  0.0542  0.0476  0.0473  0.0431
    1.0481  1.0455  0.362 0.1302  0.1369  0.0541  0.9216  1.0161  0.2741  0.1387  0.1252  0.0575
0.0918  0.0928  0.061 0.0512  0.0513  0.0484  0.087 0.0989  0.061 0.0541  0.0563  0.0536
    1.0451  1.0136  0.307 0.1202  0.1483  0.0573  0.9561  1.0702  0.2318  0.1683  0.1083  0.0517
0.0995  0.0917  0.059 0.0498  0.0516  0.0482  0.0862  0.0945  0.0547  0.0509  0.0482  0.0454
    1.0999  1.0568  0.2795  0.1276  0.1227  0.0584  0.9058  1.0868  0.2685  0.134 0.1078  0.055
0.0984  0.0958  0.0609  0.0544  0.0542  0.0524  0.0888  0.0995  0.0611  0.0555  0.0533  0.0522
    1.1361  1.0054  0.294 0.1234  0.1262  0.0609  0.8984  1.0046  0.2102  0.1235  0.1219  0.0655
0.1049  0.0949  0.064 0.0564  0.0561  0.0535  0.0893  0.0961  0.058 0.0558  0.068 0.06
    1.1104  0.975 0.3491  0.1566  0.1234  0.0618  1.029 0.8426  0.2196  0.17  0.1288  0.0424
0.1033  0.0978  0.0688  0.0596  0.0586  0.0551  0.1028  0.0906  0.0618  0.0664  0.0577  0.0385
```

```
~End
Original Filename: Untitled   Date Last Saved: Unsaved
Copyright © 2003 Molecular Devices. All rights reserved.
```

It is difficult to tell with line–wrapping, but these data show absorbances of a 96 well plate at two wavelengths (562 and 660nm). A screenshot on a sufficiently wide editor will show this pattern:



This format may be useful for directly copy–pasting in to a spreadsheet, but is tremendously difficult to work with programatically in its current state.

Unlike `broom`, which can detect the kind of data given its `class`, there is no simple and robust way to automatically detect the kind of data provided. `mop` provides several 'reader' functions to wrangle data from a particular source:

```
library(bladdr)
library(mop)

file <- bladdr::get_gbci(
  "Raw Data/SPECTRAmax/aragaki-kai/MTT/2024-07-21_upfl1r-309-erda-dr.txt"
)

tidy <- mop::read_spectramax(file, date = as.Date("2024-07-21"))
tidy
```

```
<spectramax[4]>
[[1]]
[[1]]$data

                12

   ____________________________
  | O O O O O O O O O O O O
  | O O O O O O O O O O O O
  | O O O O O O O O O O O O
  | O O O O O O O O O O O O
8 | O O O O O O O O O O O O
  | O O O O O O O O O O O O
```

```
  | ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
  | ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○

Start corner: tl
Plate dimensions: 8 x 12

[[1]]$type
[1] Plate

[[1]]$wavelengths
[1] 562 660



# Date: 2024-07-21
```

Each reader exports its own class (here a `spectramax`) that developers can use to form common interfaces for like–data on different machines. However, as an end user it can often be easier to work with the data as a flat `data.frame`. Each class also has its own `scrub` method to convert the object into a `data.frame`:

```
scrub(tidy)
```

```
# A tibble: 96 × 6
    .row  .col nm562  nm660 exp_date   is_tidy
   <int> <dbl> <dbl>  <dbl> <date>     <lgl>
 1     1     1     1 0.935 0.087  2024-07-21 TRUE
 2     1     1     2 1.04  0.0926 2024-07-21 TRUE
 3     1     1     3 0.303 0.0546 2024-07-21 TRUE
 4     1     1     4 0.159 0.0485 2024-07-21 TRUE
 5     1     1     5 0.125 0.047  2024-07-21 TRUE
 6     1     1     6 0.053 0.0434 2024-07-21 TRUE
 7     1     1     7 1.02  0.106  2024-07-21 TRUE
 8     1     1     8 1.04  0.0935 2024-07-21 TRUE
 9     1     1     9 0.338 0.0559 2024-07-21 TRUE
10     1     1    10 0.136 0.0442 2024-07-21 TRUE
# ℹ 86 more rows
# ℹ Use `print(n = ...)` to see more rows
```

Conclusion

This package provides a starting point for a library of tidying functions. Each function has the following:

- A reading function
- A tidying function (often inextricably linked to the reading function)
- A `scrub` method

Current supported types include data exported from the Incucyte, QuantStudio, SpectraMax, and Nanodrop. Like `broom`, the straightforward and modular structure of this package makes contribution for additional machine types and data formats easy.

Increasing Ergonomics and Accessibility of Existing Packages

Problem

An excellent data product can be hampered by its ability to be used. In some instances, it is the interface of the product that makes it challenging to use: if a given application does not meld with the current workflows of the people using it, it may very well not be used at all. Other times, simply obtaining the data product is difficult — sometimes impossible.

These parts of software have relatively little to do with their core functionality, but are *required* for their use and therefore demand as careful consideration as developing core functionality.

Solution

In this section, I present two case studies (`tidyestimate` and `reclanc`) in which I have made pre–existing software (`ESTIMATE` and `ClaNC`) both more usable and available.

`tidyestimate`

`ESTIMATE` is an excellent R package used to estimate stromal and immune infiltration within a tumor on a single–sample basis, which can be used to infer tumor purity[16]. However, its usefulness is hampered by its relatively poor ergonomics and discoverability. The most common sources of R packages are from the Comprehensive R Archive Network (CRAN), Bioconductor, or GitHub; ESTIMATE does not exist on any of these and instead exists on R–forge https://r-forge.r-project.org/ and thus suffered from low visibility. `ESTIMATE` also relies on a workflow that both ingested and returned external .GCT files. While these are used by some external programs, it is uncommon and often unnecessary for them to exist in a standard R workflow, where it is often more convenient to keep data as R objects in the environment. Because of this, in a typical workflow, the user would need to turn their output into a .GCT file exclusively for `ESTIMATE`, which would produce a .GCT file, which the user would then need to read back in for further analysis. Additionally, since a .GCT file has a header, care would need to be taken to skip the two header lines to read in properly. Further, `ESTIMATE` lacked documentation for its functions, making it difficult to know what inputs and outputs are expected. Finally, `ESTIMATE` is over 10 years old, and many of its identifiers used for its gene signatures go by different names or no longer exist.

Solution

Each issue listed is by no means insurmountable, but as a sum they create enough friction to prevent usage. I maintained the algorithmic core of `ESTIMATE` and wrapped it in a new package, `tidyestimate`.

The stage at which single-sample gene set enrichment analysis comes is typically after normalization of gene expression, usually performed by packages like `limma`, `DESeq2`, or `edgeR`, and existing as some form of matrix of expression or something easily coercible to this form. Therefore, `tidyestimate` takes in `data.frame`, `matrix`, or similar objects opposed to external files that `ESTIMATE` accepts. A file is a responsibility: it must have a reasonable

place and a reasonable name, and it requires special consideration in regards to inter–user portability. If we can do without creating an external file, we should. For `ESTIMATE`, this responsibility felt unnecessarily self–inflicted and was circumvented in `tidyestimate`.

`tidyestimate` also creates functions that are 'pipe–able' — that is, the output of previous functions can serve as the input of the next function, allowing them to be chained in a natural and common pattern.

`ESTIMATE` also had little to no documentation for its functions. `tidyestimate` adds documentation to make clear what the expected input and output is, as well as what each function is doing. In addition, `tidyestimate` adds a vignette to demonstrate how to use the package.

Gene symbols have changed over time, in part due to our updated understanding of the purpose of the genes, and other times to avoid coercion to dates by Excel. The `ESTIMATE` algorithm is sensitive to the presence and expression not only the genes within each of its stromal and immune signatures, but to a large list of roughly 10,000 genes that were common to a variety of array–based expression platforms. To ensure maximal compatibility between old and new datasets, an optional, conservative alias matching algorithm is supplied. In the original dataset supplied with `ESTIMATE`, 488 of the 10364 (5%) of the genes used for `ESTIMATE` were out of date; alias matching allowed 461 out of 488 (94%) to be updated and therefore recovered.

Finally, and most importantly, this package has been made available on CRAN (Comprehensive R Archive Network), the de–facto repository for R packages. This, along with its inclusion in the R Task View for Omics https://cran.r-project.org/web/views/Omics.html, has greatly increased its visibility and discoverability.

The impact of these small changes has been surprising. The package has been downloaded over 8,000 times, and has been cited in journals such as Nature Medicine[17], Cancer Cell International[18], Cell Reports[19], BMC Cancer[20], JCI Insight[21], and others.

Conclusion

It can not be overstated that the vast majority of the work to create this package was done by the original authors. However, the additional benefit provided by `tidyestimate` has shown to provide a clear additional benefit. This underscores the importance of the 'softer' aspects of software ergonomics for increasing adoption.

`reclanc`

Despite lacking physical form, the internet is prone to 'rot'. Known as 'link–rot', in a study performed by Pew, vast swaths — nearly 40% — of content on the internet from as little as 10 years ago is no longer available[22]. `ClaNC` (Classification of microarrays to nearest centroids) was a victim of such rot. Although the paper describing ClaNC remains[23], the original source code is no longer available.

`ClaNC` was an R package that took expression data from pre–classified samples and generated centroids (collections of genes and their expression levels that are particular to a given class). These centroids could be used in turn to

predict the class of additional samples. ClaNC differed from a similar centroid–based classifier PAM[24] (prediction analysis of microarrays) in ways that made it more sensitive and accurate, the details of which will be covered later in this section.

`ClaNC` was used by others in our field. Wanting to perform a similar but distinct analysis, I attempted to find the `ClaNC` software, but to no avail. The only place in which it could still incidentally be found was implemented in part of a larger pipeline known as `sake` https://github.com/naikai/sake. To revitalize `ClaNC`, I extracted it from `sake` to once again make it into stand–alone software, modernized it to fit within the current ecosystems of machine learning and bioinformatics in R, and created documentation.

Solution

`ClaNC`'s source was removed from `sake`. Dead code — which referenced a now defunct graphical user interface (GUI) — was removed, and code was rewritten to be more idiomatic. Significant attention was given to writing documentation. While previously there was no documentation for any of the functions, extensive documentation has been given for user facing functions, along with examples, a usage vignette, and a case study vignette. Additionally, an introductory 'theory of `ClaNC`' blog post was created to describe how `ClaNC` worked, rather than just how to use it, at an introductory level.

Beyond documentation, `ClaNC` was modernized to leverage the current machine learning ecosystem tooling. `ClaNC` used an internal resampling and hyper–parameter tuning algorithm, which were removed. This allows the user to use whatever method or package they would like to perform these tasks, and reduces the amount of code that need to be maintained within `reclanc`.

Finally, `reclanc` provides an additional prediction method that is correlation based, rather than distance based. This is particularly useful in cases where the scales of expression may be different from one another, such as in the case of cross–sequencing–platform classification and prediction.

Documentation

The paper associated with `ClaNC` was well written, but it was written with an expert audience in mind, particularly those with a stronger understanding of statistics. To assist users who might have less of a background in statistics in learning how `ClaNC` works, I wrote a less technical blog post helping to visually explain the process by which `ClaNC` both classified and predicted samples. This following description is an adaptation of that blog post.

What is classification, and why do it?

Classification, in essence, requires two steps:

1. Find the distinguishing features of each of the classes in your pre–labeled/clustered data ('fitting')
2. Use these distinguishing features to classify samples from other datasets that do not have these labels ('predicting')

Both of these steps provide utility.

For the first step, let's imagine you have tumors from cancer patients that responded to a drug and those that did not respond to a drug. We might use those as our 'labels' for each class — responders and non–responders. We could use a classifier to extract distinguishing features about each one of our classes. In the case of this package, these features refer to genes, and what distinguishes them from class to class is their expression levels. We can look at what features the classifier took and gain insight into the biology of these responders — maybe even forming a hypothesis for the mechanism by which these responders respond. The extent which the features extracted represent anything useful depends on the intepretability of the model, of which `ClaNC` is highly interpretable.

The utility of the second step is more straightforward. If you want to apply the knowledge of a given subtype to a new set of data, classification is incredibly useful here. For instance, imagine you have developed a classifier that can help you predict whether a cancer patient will respond to a drug based on expression from their tumor. Being able to classify which class the new patients tumor falls into provides you with actionable information as to how to treat the patient.

(Aside: not all classifiers can classify new, single samples — some require context of additional samples around them. However, `ClaNC` doesn't need this and can classify a single new sample (sometimes called a 'single sample classifier').)

What is ClaNC?

`ClaNC` both creates classifiers (fits) as well as uses the classifiers to assign new samples to a class (predicts). It is a nearest–centroid classifier. We'll get into the details later, but as a brief summary, it means it tries to find the average, distinguishing features of a given class (step 1), and then uses that average as a landmark to compare new samples to (step 2). Other nearest–centroid classifiers have existed (like, for instance, PAM), but `ClaNC` distinguishes itself by tending to be more accurate and sensitive than PAM.

How does it work?

Fitting

Our first step provides the algorithm with examples of what each class looks like so it can extract the features that distinguish one class from another. These 'examples' can come from some external phenotype (such as our responder/non–responder example from above) or from the data themselves (such as clusters from after doing, say, k–means clustering). Regardless, the input should be expression data that has been labeled with some kind of class (Figure 43, left).

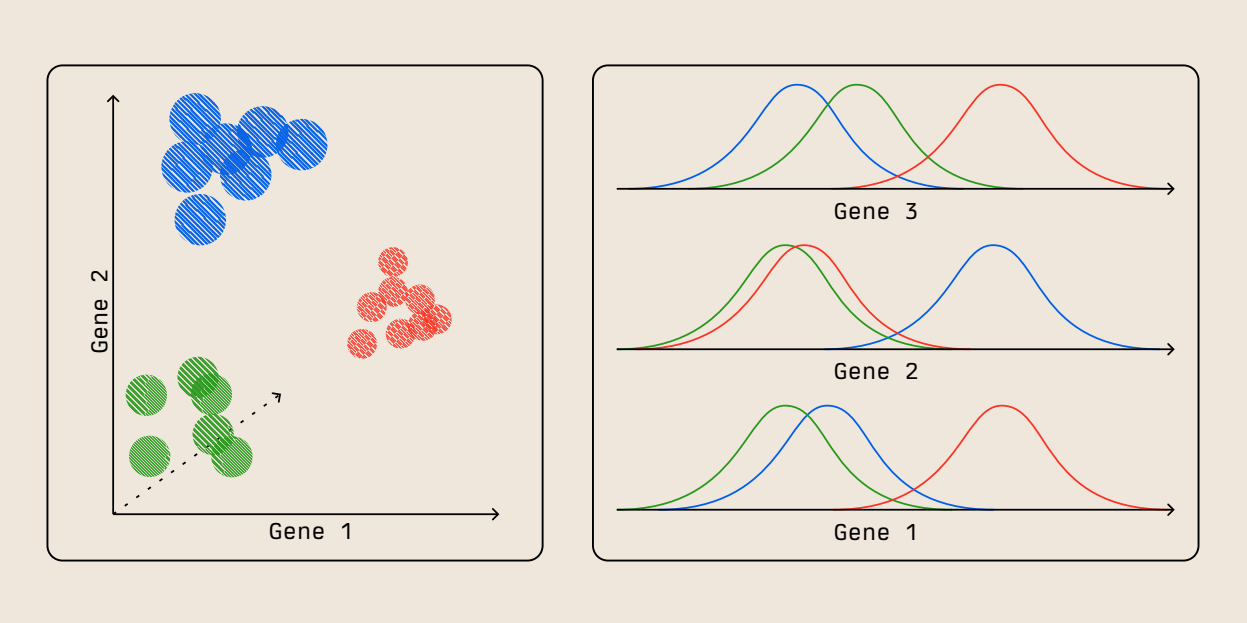


Figure 43: **Left:** Our samples, colored by class, floating in N–dimensional space. **Right:** Each dimension separated from one another

The first assumption we make is that we can treat each gene independently. While this might not be exactly true in reality, it greatly simplifies the problem by allowing us to deal with each gene one at a time (Figure 43, right). Despite this simplification, it also works pretty well.

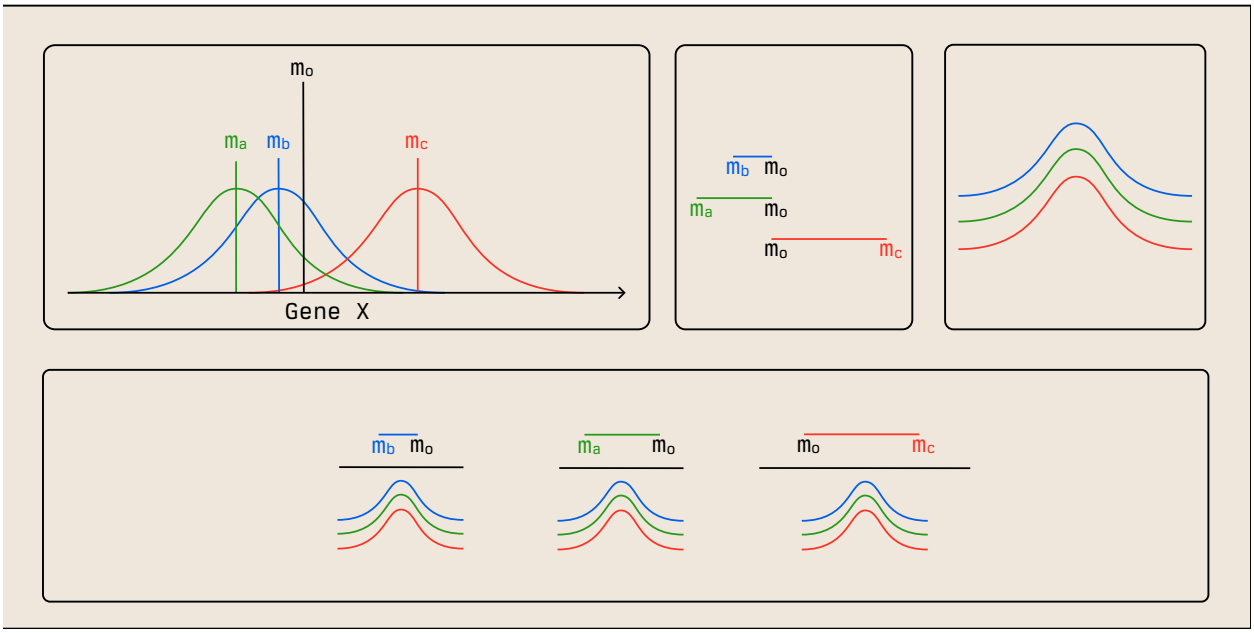


Figure 44: **Top left:** The expression of a gene for each given class, with class means denoted by m_a, m_b, and m_c, and the overall mean as m_o. **Top middle:** distances between the class and overall means. **Top right:** Pooled standard deviations are calculated for each gene. **Bottom:** dividing each class distance by the pooled standard deviation

For each gene, we calculate the overall mean, as well as the mean within each class (Figure 44, top left). We then find the distance between each class mean and the overall mean (Figure 44, top middle) and the pooled standard deviation for the gene (Figure 44, top right). Dividing the distance by the pooled standard deviation, we get, essentially, a t–statistic.

We repeat this calculation for every gene (Figure 45, top), then take the absolute value of each statistic (Figure 45, bottom left) and rank them per–class (that is, each class has a 1, 2, etc) (Figure 45, bottom right).

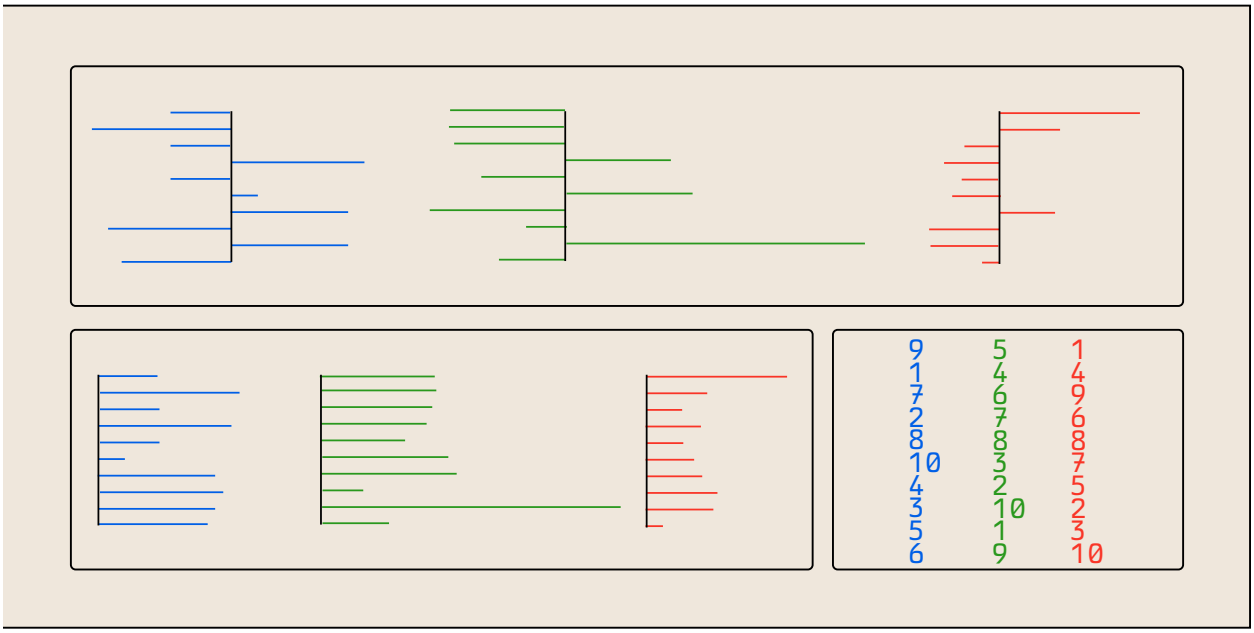


Figure 45: **Top:** t–statistics for each class (color) and each gene (row). **Bottom left:** absolute value of the t–statistics. **Bottom right:** class–wise rank of absolute value t–statistics

One thing to note is what Dabney calls 'active genes'. An 'active gene' is a gene that has been selected to be a distinguishing feature for a given class. At the outset, you can select how many active genes you want per class (and it needn't be the same number of genes per class).

One thing that sets `ClaNC` apart from other nearest–centroid classifiers is that it only lets each gene be 'used' as a distinguishing feature once. That is, it cannot be used in multiple classes. Because of this, the classes 'compete' with one another to see who gets what gene. It's based on the class–rank of the gene (using the underlying absolute value t–statistic as a tie–breaker) as well as if a class needs more 'active genes' or if it already has all that it needs.

The game of gene selection goes like this:

1. Each class tries to select its highest rated gene. In the case of Figure 46, panel 1, every class gets its desired gene. These genes are then taken out of future rounds (since each gene can only be in one class, a restriction we mentioned above).
2. Classes continue to select their next highest rated gene, so long as there isn't and conflict (panel 2).

3. If a class can't get its next highest rated gene (in this case, blue can't get 3 because it's been taken by red in a previous round), then it chooses its next best available choice (all the way at 6 for blue) (panel 3)
4. If there's a tie, it is typically resolved by looking at the underlying t–statistics. Whichever class has the larger absolute t–statistic wins the gene. However, in this case, suppose we set each class to only want 3 active genes. In that case, both blue and green have met their quota, and red wins by default (panel 4).

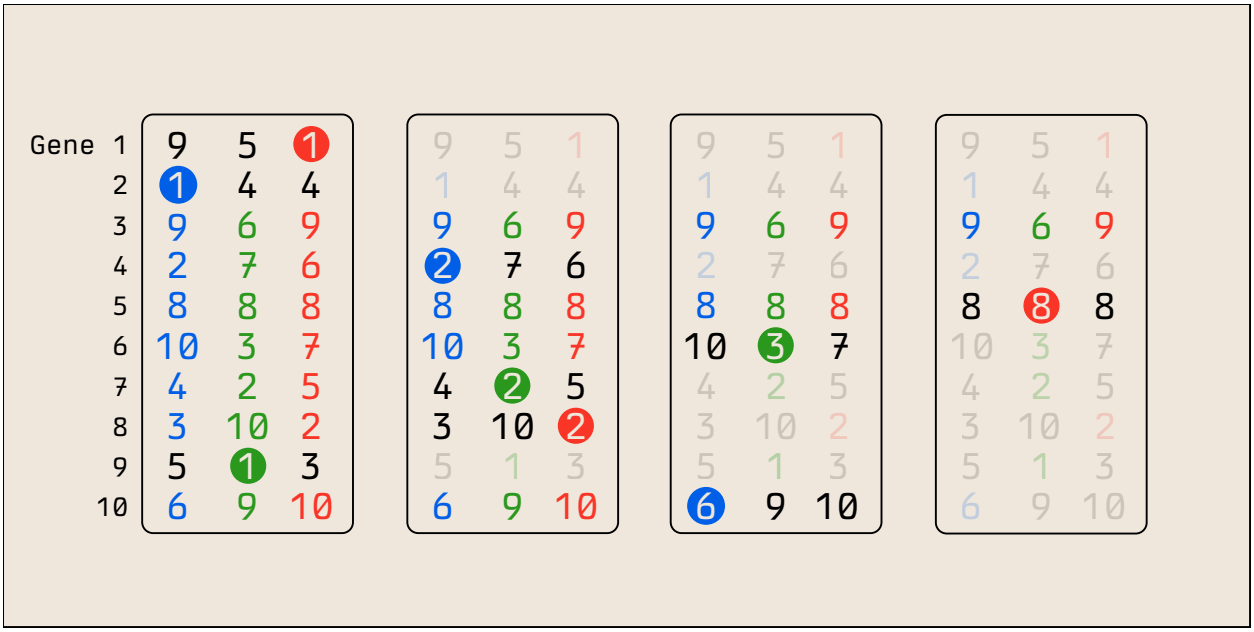


Figure 46: **1:** All classes select their top rated gene. **2:** Classes continue to select their next highest rated gene. **3:** Blue selects its next best rank since previous ranks were taken. **4:** Despite a tie, since number of active genes = 3, red wins.

Once all genes have been selected, the rest are tossed — they're unneeded for defining the centroid (Figure 46, gene 3). If a given class 'won' a gene, it uses its class mean as a value for that gene (Figure 46, colored means). Otherwise, it uses the overall mean (Figure 46, black means). The pooled standard deviations are also brought along. These are our centroids!

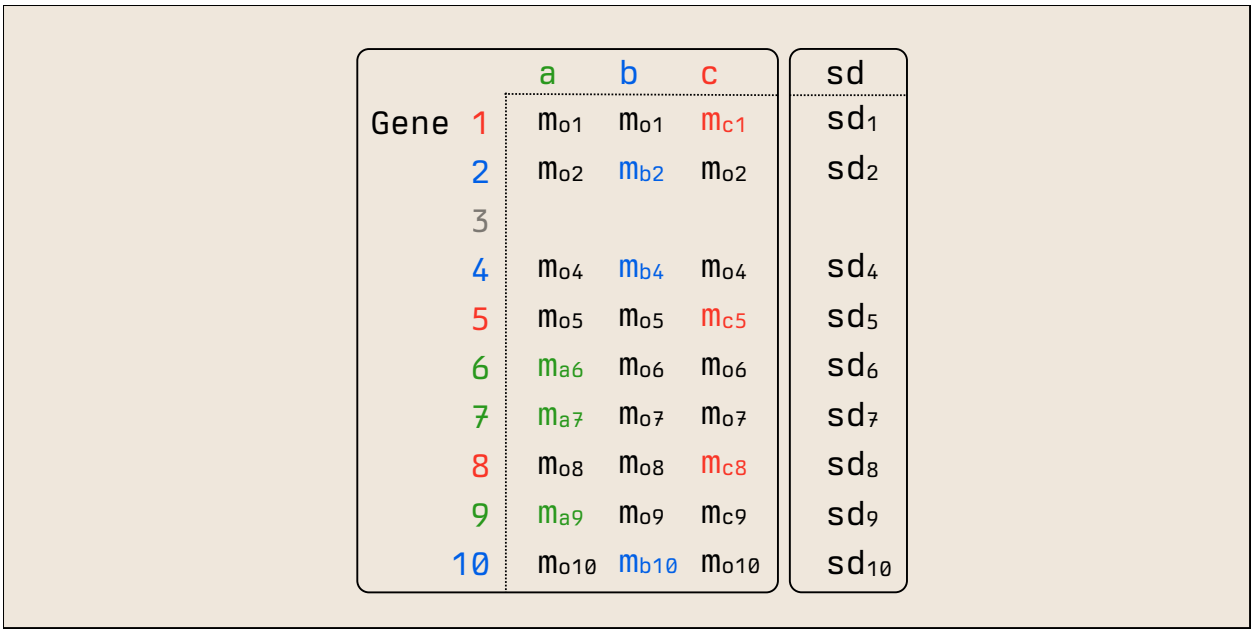


Figure 47: Centroids, at long last.

Predicting

Now we have created our centroids, we might be interested in applying them to classify future samples of unknown class.

Suppose we have a new sample that we have expression data of (Figure 48, panel 1). The genes that are not included in the centroids will have no bearing on the classification, so we can remove them (this might be a feature: perhaps an inexpensive assay is developed that only measures the expression of the centroid genes) (panel 2).

Distance-based metric

For every class, and every gene in that class, find the distance between the class centroid's mean and the new sample's mean and square it (Figure 48 panels 3, 4, 5; numerator) and divide by the pooled standard deviation we kept in our centroids (Figure 48 panels 3, 4, 5; denominator).

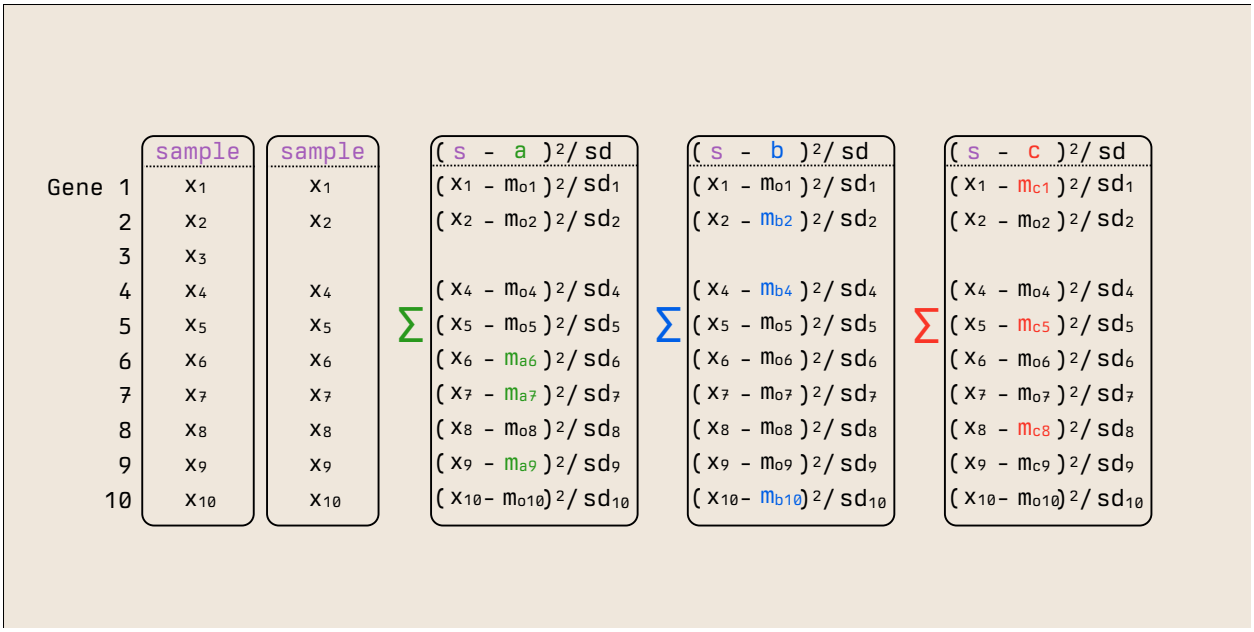


Figure 48: Calculating the distance between centroids and a new sample

Let's think about these new statistics and what they mean. If the sample gene's expression is very close to a class's expression, the statistic will be very small (scaled by how much it tends to deviate — we shouldn't punish expression from being far from the mean if it's fairly typical). If a sample is very similar to a given class, we expect all of these scores to be quite small. To this end, we take the sum of all the scores for a given class and compare the sums across all classes. The one with the smallest score is the most similar class, to which the sample gets assigned. Note that it makes sense to square the distance between the sample's expression and the centroid's expression, because we don't want a sample that is 'equally wrong on both sides' to average out and get flagged as very similar.

Correlation–based metric

The distance–based metric classification can fail, such as when samples are scaled differently or the expression comes from a different sequencing platform. Consider a particularly pathological example shown in Figure 49, where the colored dots represent the training samples used to create our centroids, and the black dots represent new samples we want to classify. Despite our new samples showing three distinct clusters that appear to have a similar pattern to our training data clusters, they will all be called 'red' because it is the closest cluster.

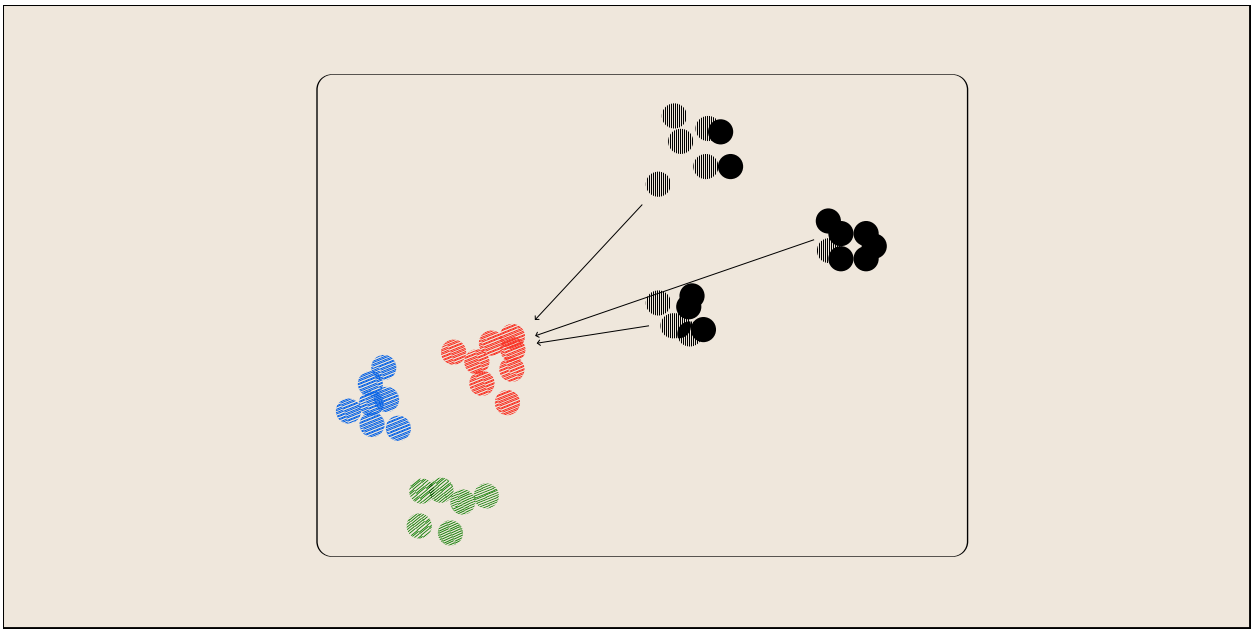


Figure 49: When distance metrics fail

One way around this is to look at the correlation between centroid expressions and the new sample's expression. If the centroid is related with the sample but for a difference in scaling, we expect a positive correlation between the two, such as shown between the unknown sample and the green centroid in Figure 50.

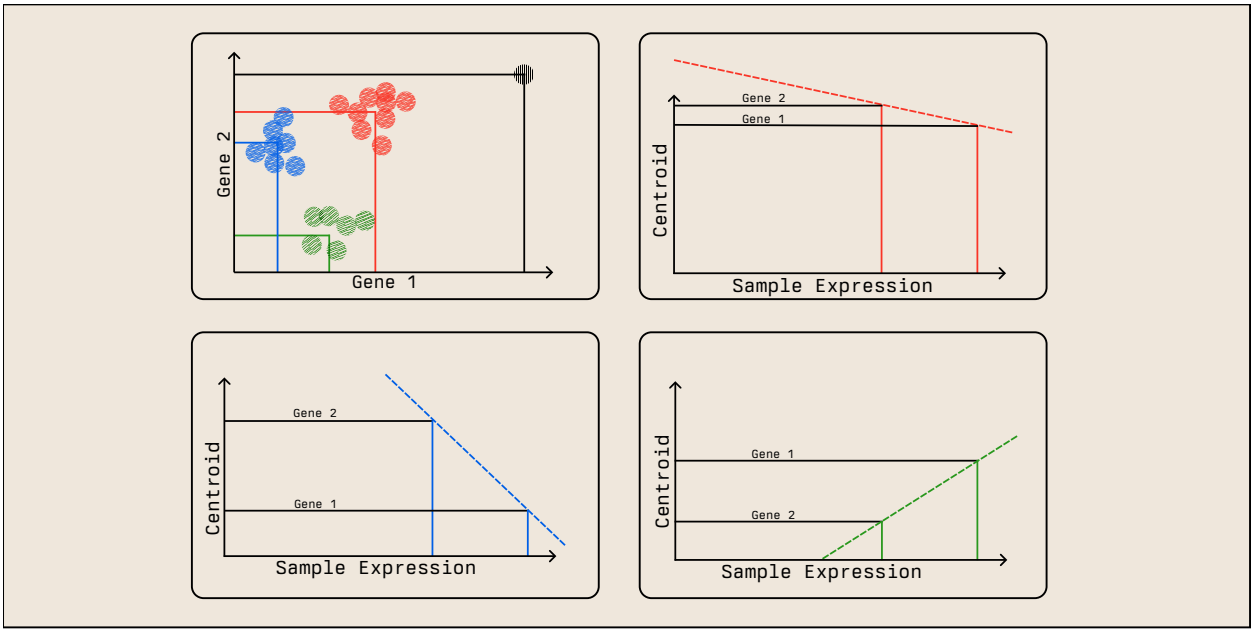


Figure 50: Classification through correlation

(For the sake of illustration, I've simplified the centroids to only use their class means for all genes, and don't consider anything about 'winning' genes — the principle is the same though.)

Case Study

To provide a tangible and full–fledged exploration of the features of reclanc, I created a vignette showcasing how this package would be used on data 'in the wild'. This case study happened to be the analysis I set out to perform before learning of `ClaNC`'s disappearance.

Introduction

```
library(reclanc)
library(aws.s3)
library(Biobase)
```

Let's consider a relatively full–featured, practical use case for `reclanc`. In this vignette, we'll go over the basics of fitting models, as well as how to leverage tidymodels to do more elaborate things like resampling and tuning hyperparameters. We'll fit a final model, then use that to predict subtypes of an entirely new dataset.

This vignette tries to assume very little knowledge about machine learning or `tidymodels`.

Fitting

A simple fit

Let's start with the fitting procedure. We first need gene expression data.

The data I'm using is from Sjödahl et al. (2012)[25]. It contains RNA expression from 308 bladder cancer tumors.

```
lund <- s3readRDS("lund.rds", "reclanc-lund", region = "us-east-2")
lund
```

```
ExpressionSet (storageMode: lockedEnvironment)
assayData: 16940 features, 308 samples
  element names: exprs
protocolData: none
phenoData
  sampleNames: UC_0001_1 UC_0002_1 ... UC_0785_1 (308 total)
  varLabels: title source ... sample (16 total)
  varMetadata: labelDescription
featureData: none
experimentData: use 'experimentData(object)'
Annotation:
```

In their paper, Sjödahl et al. used the transcriptional data to classify the tumors into seven molecular subtypes (MS):

```
table(lund$molecular_subtype)
```

```
MS1a    MS1b  MS2a.1  MS2a.2  MS2b.1 MS2b2.1 MS2b2.2
  53      78      30      55      43      20      29
```

We'd like to apply this subtype framework to other datasets. To do this, we first need to generate centroids. Before we can begin, though, we need to convert our outcomes to factors. In this case, our outcomes are the molecular subtypes:

```
lund$molecular_subtype <- factor(lund$molecular_subtype)
```

In its simplest form, since `clanc` accepts `ExpressionSet` objects, we could do the following and be done with it:

```
simple_centroids <- clanc(lund, classes = "molecular_subtype", active = 5)
head(simple_centroids$centroids)
```

```
  class    gene expression pooled_sd active     prior
1  MS1a   CXCL1   6.534490 0.8749133      5 0.1428571
2  MS1a     MMD   7.922508 0.6429620      5 0.1428571
3  MS1a C9orf19   8.378910 0.7510552      5 0.1428571
4  MS1a    BNC1   5.297095 0.2106762      5 0.1428571
5  MS1a  SLFN11   7.362887 0.6824663      5 0.1428571
6  MS1a    CRAT   6.004517 0.3425669      5 0.1428571
```

The problem with this method, though, is we have no idea if this is a good fit or not. active is an argument that specifies the number of genes that are used as distinguishing features for a given class. In this case, each class will find 5 genes that have expression patterns peculiar to that given molecular subtype, and each subtype will have 7 (the total number of subtypes) x 5 (number of active genes) = 35 genes in it (see my blog post or — better yet — the original paper for more details). Could we have gotten a better fit with more genes? Are we selecting more genes than we need? How would we know?

Setting the stage for more elaborate analyses

Before we can get started on tackling these larger questions, let's take a brief detour to the land of `tidymodels`. `tidymodels` is a collection of packages that make running and tuning algorithms like this much less painful and much more standardized.

In order to leverage `tidymodels`, we need to buy–in to their data structures.

(Aside: I don't mean to make the buy–in sound begrudging. When I say need, I really mean it: we're going to be specifying very long formulas, which for some reason R really, really hates. Emil Hvitfeldt recently (at time of

writing) has allowed `tidymodels` to handle long formulas gracefully, so using `tidymodels` infrastructure is a gift, not a chore.)

```
library(tidymodels)
```

Many `tidymodels` workflows begin with a model specification. The rationale behind this is to separate the model specification step from the model fitting step (whereas in base R, they generally all happen at once). `reclanc` makes it easy to specify a model by adding a custom engine to `parsnip::discrim_linear`, so specifying a model looks like this:

```
mod <- discrim_linear() |>
  set_engine(
    engine = "clanc", # Note: "clanc", not "reclanc"
    active = 5
  )
```

This `mod` doesn't do anything — and that's kind of the point: it only specifies the model we will later fit with, but doesn't do any fitting itself. This allows us to reuse the specification across our code.

The next step is to wrangle our data a bit to be in a 'wide' format, where all columns are outcomes (classes) and predictors (genes), and all rows are observations (samples):

```
wrangled <- data.frame(class = lund$molecular_subtype, t(exprs(lund)))
head(wrangled[1:5])
```

```
           class LOC23117   FCGR2B    TRIM44 C15orf39
UC_0001_1   MS1b 5.565262 5.306654  9.305053 6.430063
UC_0002_1 MS2b.1 5.505854 5.731128  9.242790 7.265748
UC_0003_1 MS2a.2 5.336140 5.540470  9.888668 7.244976
UC_0006_2 MS2b.1 5.576748 5.847743  9.408895 7.377358
UC_0007_1 MS2a.2 5.414919 5.510507 10.482469 6.435552
UC_0008_1 MS2b.1 5.279174 5.633093  9.112754 7.057977
```

Finally, we specify a formula for fitting the model. This uses the recipes package from `tidymodels`. While this is a delightful package that can help you preprocess your data, it's out of the scope of this vignette. Instead, just think of it as a way to specify a formula that keeps R from blowing up:

```
# Note that the recipe requires 'template data'
recipe <- recipe(class ~ ., wrangled)
```

We can bundle our model specification (mod) and our preprocessing steps (recipe, which is just a formula) into a workflow:

```
wf <- workflow() |>
  add_recipe(recipe) |>
  add_model(mod)
wf
```

```
══ Workflow ════════════════════════════════════════════════════════════════════
Preprocessor: Recipe
Model: discrim_linear()

── Preprocessor ────────────────────────────────────────────────────────────────
0 Recipe Steps

── Model ───────────────────────────────────────────────────────────────────────
Linear Discriminant Model Specification (classification)

Engine-Specific Arguments:
  active = 5

Computational engine: clanc
```

Now we can fit our model:

```
tidymodels_fit <- fit(wf, data = wrangled)
head(extract_fit_parsnip(tidymodels_fit)$fit$centroids)
```

```
  class    gene expression pooled_sd active     prior
1  MS1a   CXCL1   6.534490 0.8749133      5 0.1428571
2  MS1a     MMD   7.922508 0.6429620      5 0.1428571
3  MS1a C9orf19   8.378910 0.7510552      5 0.1428571
4  MS1a    BNC1   5.297095 0.2106762      5 0.1428571
5  MS1a  SLFN11   7.362887 0.6824663      5 0.1428571
6  MS1a    CRAT   6.004517 0.3425669      5 0.1428571
```

You'll notice that our results are the same as what we saw previously, demonstrating that while we're using tidymodels rather than base R, we're still doing the same thing.

Measuring fit accuracy with cross–validation

Now that we've dialed in to the `tidymodels` framework, we can do a lot of elaborate things with ease. One of our concerns is whether 5 active genes was a good choice (active = 5). A somewhat simple way to determine how good our choice of 5 genes is to use cross–validation. Cross–validation allows us to test how good our fit is by training our model on, say, 80% of our data, and testing it on the rest (see the Wikipedia diagram of a k–fold cross validation). This allows us to get a measure of how good our fit is, without having to break out our actual test data — which in general should only be used when we're ready to finalize our model.

Speaking of test data, let's go ahead and split that off now. We'll lock our test data away and only use it once we've fit our final model. Until then, we'll use cross validation to assess how good the fit is, essentially using our training data as its own testing data.

Of course, `tidymodels` makes this easy too, by using `rsample::initial_split`:

```
set.seed(123)
splits <- initial_split(wrangled, prop = 0.8, strata = class)
train <- training(splits)
test <- testing(splits)
```

`train` and `test` are just subsets of the original data, containing 80% and 20% of the original data (respectively). It also tries to maintain the relative proportions of each of the classes within each of the datasets (because we set `strata = class`):

```
round(prop.table(table(train$class)), 2)
```

```
MS1a    MS1b  MS2a.1  MS2a.2  MS2b.1 MS2b2.1 MS2b2.2
0.17    0.25    0.10    0.18    0.15    0.07    0.08
```

```
round(prop.table(table(test$class)), 2)
```

```
MS1a    MS1b  MS2a.1  MS2a.2  MS2b.1 MS2b2.1 MS2b2.2
0.19    0.27    0.08    0.16    0.11    0.05    0.16
```

Creating folds for cross validation is nearly the same as `initial_split`:

```
folds <- vfold_cv(train, v = 5, strata = class)
folds
```

```
# 5-fold cross-validation using stratification
# A tibble: 5 × 2
  splits           id
  <list>           <chr>
1 <split [193/51]> Fold1
2 <split [193/51]> Fold2
3 <split [195/49]> Fold3
4 <split [197/47]> Fold4
5 <split [198/46]> Fold5
```

We can reuse our workflow `wf`, which contains our model and formula. The only difference is that we use `fit_resamples`, and we specify a metric we want to use to measure how good our fit is (remember that every fold has a chunk of data it uses to test the fit). For simplicity, let's use accuracy:

```
fits <- fit_resamples(
  wf,
  folds,
  metrics = metric_set(accuracy)
)
fits
```

```
35/35 (100%) genes in centroids found in data
35/35 (100%) genes in centroids found in data
35/35 (100%) genes in centroids found in data
35/35 (100%) genes in centroids found in data
35/35 (100%) genes in centroids found in data
# Resampling results
# 5-fold cross-validation using stratification
# A tibble: 5 × 4
  splits           id    .metrics         .notes
  <list>           <chr> <list>           <list>
1 <split [193/51]> Fold1 <tibble [1 × 4]> <tibble [0 × 3]>
2 <split [193/51]> Fold2 <tibble [1 × 4]> <tibble [0 × 3]>
3 <split [195/49]> Fold3 <tibble [1 × 4]> <tibble [0 × 3]>
4 <split [197/47]> Fold4 <tibble [1 × 4]> <tibble [0 × 3]>
5 <split [198/46]> Fold5 <tibble [1 × 4]> <tibble [0 × 3]>
```

We can then extract our accuracy metrics by using `collect_metrics`, which roots around in each of our fits and helpfully extracts the metrics, aggregates them, and calculated the standard error:

```
metrics <- collect_metrics(fits)
metrics
```

```
# A tibble: 1 × 6
  .metric  .estimator  mean     n std_err .config
  <chr>    <chr>      <dbl> <int>   <dbl> <chr>
1 accuracy multiclass 0.737     5  0.0289 Preprocessor1_Model1
```

Our model has an accuracy of about 74%. Applying this model to our testing data:

```
# Fit a model using *all* of our training data
final_fit <- clanc(class ~ ., train, active = 5)

# Use it to predict the (known) classes of our test data
preds <- predict(final_fit, new_data = test, type = "class")
w_preds <- cbind(preds, test)
# Compare known class vs predicted class
metric <- accuracy(w_preds, class, .pred_class)
metric
```

```
35/35 (100%) genes in centroids found in data
# A tibble: 1 × 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy multiclass     0.734
```

Note that our testing data accuracy (%) approximates the training data accuracy (74%).

Tuning hyperparameters with tune

Now we at least have some measure of how good our model fits, but could it be better with more genes? Could we get away with fewer? Running the same command over and over again with different numbers is a drag — fortunately, there's yet another beautiful package to help us: `tune`.

To use tune, we need to re–specify our model to let tune know what parameters we want to tune:

```
tune_mod <- discrim_linear() |>
  set_engine(
    engine = "clanc",
```

```
    active = tune()
  )
```

We could update our previous workflow using `update_model`, but let's just declare a new one:

```
tune_wf <- workflow() |>
  add_recipe(recipe) |>
  add_model(tune_mod)
```

We then have to specify a range of values of active to try:

```
values <- data.frame(active = seq(from = 1, to = 50, by = 4))
values
```

```
   active
1       1
2       5
3       9
4      13
5      17
6      21
7      25
8      29
9      33
10     37
11     41
12     45
13     49
```

We can then fit our folds using the spread of values we chose:

```
# This is going to take some time, since we're fitting 5 folds 13 times each.
tuned <- tune_grid(
  tune_wf,
  folds,
  metrics = metric_set(accuracy),
  grid = values
```

```
)
tuned
```

```
7/7 (100%) genes in centroids found in data
# Tuning results
# 5-fold cross-validation using stratification
# A tibble: 5 × 4
  splits           id    .metrics          .notes
  <list>           <chr> <list>            <list>
1 <split [193/51]> Fold1 <tibble [13 × 5]> <tibble [0 × 3]>
2 <split [193/51]> Fold2 <tibble [13 × 5]> <tibble [0 × 3]>
3 <split [195/49]> Fold3 <tibble [13 × 5]> <tibble [0 × 3]>
4 <split [197/47]> Fold4 <tibble [13 × 5]> <tibble [0 × 3]>
5 <split [198/46]> Fold5 <tibble [13 × 5]> <tibble [0 × 3]>
```

As before, we can collect our metrics — this time, however, we have a summary of metrics for each of values for active:

```
tuned_metrics <- collect_metrics(tuned)
tuned_metrics
```

```
# A tibble: 13 × 7
   active .metric  .estimator  mean     n std_err .config
    <dbl> <chr>    <chr>      <dbl> <int>   <dbl> <chr>
 1      1 accuracy multiclass 0.585     5  0.0368 Preprocessor1_Model01
 2      5 accuracy multiclass 0.737     5  0.0289 Preprocessor1_Model02
 3      9 accuracy multiclass 0.748     5  0.0496 Preprocessor1_Model03
 4     13 accuracy multiclass 0.781     5  0.0403 Preprocessor1_Model04
 5     17 accuracy multiclass 0.770     5  0.0280 Preprocessor1_Model05
 6     21 accuracy multiclass 0.774     5  0.0335 Preprocessor1_Model06
 7     25 accuracy multiclass 0.785     5  0.0378 Preprocessor1_Model07
 8     29 accuracy multiclass 0.794     5  0.0319 Preprocessor1_Model08
 9     33 accuracy multiclass 0.773     5  0.0281 Preprocessor1_Model09
10     37 accuracy multiclass 0.790     5  0.0295 Preprocessor1_Model10
11     41 accuracy multiclass 0.794     5  0.0339 Preprocessor1_Model11
12     45 accuracy multiclass 0.815     5  0.0267 Preprocessor1_Model12
13     49 accuracy multiclass 0.815     5  0.0277 Preprocessor1_Model13
```

Or graphically:

```
ggplot(tuned_metrics, aes(active, mean)) +
  geom_line() +
  coord_cartesian(ylim = c(0, 1)) +
  labs(x = "Number Active Genes", y = "Accuracy")
```

It looks like we read maximal accuracy at around 21 genes — let's choose 20 genes for a nice round number:

```


final_fit_tuned <- clanc(class ~ ., data = train, active = 20)
# Use it to predict the (known) classes of our test data:
preds <- predict(final_fit_tuned, new_data = test, type = "class")
w_preds <- cbind(preds, test)
# Compare known class vs predicted class:
metric <- accuracy(w_preds, class, .pred_class)
metric
```

```
140/140 (100%) genes in centroids found in data
# A tibble: 1 × 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy multiclass     0.812
```

It looks like our accuracy is a little better now that we've chosen an optimal number of active genes.

Predicting

Now we want to apply our classifier to new data. Our second dataset is RNAseq data from 30 bladder cancer cell lines:

```
library(cellebrate)
cell_rna
```

```
class: DESeqDataSet
dim: 18548 30
metadata(1): version
assays(2): counts rlog_norm_counts
rownames(18548): TSPAN6 TNMD ... MT-ND5 MT-ND6
rowData names(0):
colnames(30): 1A6 253JP ... UC7 UC9
colData names(5): cell bsl lum call clade
```

Predicting is incredibly simple. Since we're using a different sequencing method (RNAseq vs array–based sequencing), it probably makes sense to use a correlation based classification rather than the original distance–based metric used in the original ClaNC package. We can do that by specifying `type = "numeric"` and then whatever correlation method we prefer.

```
cell_preds <- predict(
  final_fit_tuned,
  cell_rna,
  assay = 2,
  type = "numeric",
  method = "spearman"
)

out <- cbind(colData(cell_rna), cell_preds) |>
  as_tibble()

out
```

```
118/140 (84%) genes in centroids found in data
 # A tibble: 30 × 12
   cell     bsl    lum call  clade            .pred_MS1a .pred_MS1b .pred_MS2a.1
   <chr>  <dbl>  <dbl> <chr> <fct>                 <dbl>      <dbl>        <dbl>
 1 1A6     99.0   1.02 BSL   Epithelial Other     0.0600      0.224        0.149
 2 253JP   76.6  23.4  BSL   Unknown              0.0574      0.240        0.219
 3 5637    98.5   1.46 BSL   Epithelial Other     0.0958      0.243        0.160
 4 BV      49.9  50.1  LUM   Unknown              0.0758      0.262        0.238
 5 HT1197  56.0  44.0  BSL   Epithelial Other     0.119       0.288        0.224
 6 HT1376  10.9  89.1  LUM   Epithelial Other     0.100       0.277        0.238
 7 J82     98.1   1.91 BSL   Mesenchymal          0.127       0.292        0.219
 8 RT112    0   100    LUM   Luminal Papilla…     0.173       0.380        0.294
 9 RT4      0   100    LUM   Luminal Papilla…     0.134       0.317        0.257
10 RT4V6    0   100    LUM   Luminal Papilla…     0.143       0.207        0.165
# ℹ 20 more rows
# ℹ 4 more variables: .pred_MS2a.2 <dbl>, .pred_MS2b.1 <dbl>,
#   .pred_MS2b2.1 <dbl>, .pred_MS2b2.2 <dbl>
# ℹ Use `print(n = ...)` to see more rows
```

```
plotting_data <- out |>
  pivot_longer(cols = starts_with(".pred"))

plotting_data |>
  ggplot(aes(cell, value, color = name)) +
  geom_point() +
facet_grid(~clade, scales = "free_x", space = "free_x")
```

In the Sjödahl paper, the seven subtypes were simplified into five subtypes by merging some of the two that had similar biological pathways activated. To ease interpretation, we can do that too:

```
table <- plotting_data |>
  summarize(winner = name[which.max(value)], .by = c(cell, clade)) |>
  mutate(
    five = case_when(
      winner %in% c(".pred_MS1a", ".pred_MS1b") ~ "Urobasal A",
      winner %in% c(".pred_MS2a.1", ".pred_MS2a.2") ~ "Genomically unstable",
      winner == ".pred_MS2b.1" ~ "Infiltrated",
      winner == ".pred_MS2b2.1" ~ "Uro-B",
      winner == ".pred_MS2b2.2" ~ "SCC-like"
    )
  ) |>
  relocate(cell, five, clade)

print(table, n = 30)
```

```
 # A tibble: 30 × 4
   cell   five                clade             winner
   <chr>  <chr>               <fct>             <chr>
 1 1A6    SCC-like            Epithelial Other  .pred_MS2b2.2
 2 253JP  SCC-like            Unknown           .pred_MS2b2.2
 3 5637   SCC-like            Epithelial Other  .pred_MS2b2.2
 4 BV     Urobasal A          Unknown           .pred_MS1b
 5 HT1197 SCC-like            Epithelial Other  .pred_MS2b2.2
 6 HT1376 SCC-like            Epithelial Other  .pred_MS2b2.2
 7 J82    Urobasal A          Mesenchymal       .pred_MS1b
 8 RT112  Urobasal A          Luminal Papillary .pred_MS1b
```

```
 9 RT4    Urobasal A           Luminal Papillary .pred_MS1b
10 RT4V6  Urobasal A           Luminal Papillary .pred_MS1b
11 SCaBER SCC-like             Epithelial Other  .pred_MS2b2.2
12 SW780  Urobasal A           Luminal Papillary .pred_MS1b
13 T24    SCC-like             Mesenchymal       .pred_MS2b2.2
14 TCCSup SCC-like             Mesenchymal       .pred_MS2b2.2
15 UC10   SCC-like             Epithelial Other  .pred_MS2b2.2
16 UC11   SCC-like             Mesenchymal       .pred_MS2b2.2
17 UC12   Urobasal A           Mesenchymal       .pred_MS1b
18 UC13   SCC-like             Mesenchymal       .pred_MS2b2.2
19 UC14   Urobasal A           Luminal Papillary .pred_MS1b
20 UC15   SCC-like             Epithelial Other  .pred_MS2b2.2
21 UC16   SCC-like             Epithelial Other  .pred_MS2b2.2
22 UC17   SCC-like             Luminal Papillary .pred_MS2b2.2
23 UC18   SCC-like             Mesenchymal       .pred_MS2b2.2
24 UC1    Urobasal A           Luminal Papillary .pred_MS1b
25 UC3    SCC-like             Mesenchymal       .pred_MS2b2.2
26 UC4    Urobasal A           Unknown           .pred_MS1b
27 UC5    Urobasal A           Luminal Papillary .pred_MS1b
28 UC6    Urobasal A           Luminal Papillary .pred_MS1b
29 UC7    Urobasal A           Epithelial Other  .pred_MS1b
30 UC9    Genomically unstable Epithelial Other  .pred_MS2a.1
```

Bibliography

1. Prinz, F., Schlange, T. & Asadullah, K. Believe it or not: how much can we rely on published data on potential drug targets?. *Nature Reviews Drug Discovery* **10,** 712 (2011)
2. Begley, C. G. & Ellis, L. M. Raise standards for preclinical cancer research. *Nature* **483,** 531–533 (2012)
3. Begley, C. G. & Ioannidis, J. P. Reproducibility in Science: Improving the Standard for Basic and Preclinical Research. *Circulation Research* **116,** 116–126 (2015)
4. Chalmers, I. & Glasziou, P. Avoidable waste in the production and reporting of research evidence. *The Lancet* **374,** 86–89 (2009)
5. Chalmers, I. *et al.* How to increase value and reduce waste when research priorities are set. *The Lancet* **383,** 156–165 (2014)
6. Ioannidis, J. P. A. *et al.* Increasing value and reducing waste in research design, conduct, and analysis. *The Lancet* **383,** 166–175 (2014)

7. Salman, R. A.-S. *et al.* Increasing value and reducing waste in biomedical research regulation and management. *The Lancet* **383,** 176–185 (2014)

8. Chan, A.-W. *et al.* Increasing value and reducing waste: addressing inaccessible research. *The Lancet* **383,** 257–266 (2014)

9. Glasziou, P. *et al.* Reducing waste from incomplete or unusable reports of biomedical research. *The Lancet* **383,** 267–276 (2014)

10. García-Berthou, E. & Alcaraz, C. Incongruence between test statistics and P values in medical papers. *BMC Medical Research Methodology* **4,** (2004)

11. Green, P. J. Diversities of gifts, but the same spirit. *Journal of the Royal Statistical Society: Series D (The Statistician)* **52,** 423–438 (2003)

12. Rosenthal, R. The file drawer problem and tolerance for null results. *Psychological Bulletin* **86,** 638–641 (1979)

13. Bik, E. M., Casadevall, A. & Fang, F. C. The Prevalence of Inappropriate Image Duplication in Biomedical Research Publications. *mBio* **7,** (2016)

14. Wickham, H. Tidy Data. *Journal of Statistical Software* **59,** (2014)

15. Robinson, D., Hayes, A. & Couch, S. broom: Convert Statistical Objects into Tidy Tibbles. http://dx.doi.org/10.32614/CRAN.package.broom (2014) doi:10.32614/cran.package.broom

16. Yoshihara, K. *et al.* Inferring tumour purity and stromal and immune cell admixture from expression data. *Nature Communications* **4,** (2013)

17. Acanda De La Rocha, A. M. *et al.* Feasibility of functional precision medicine for guiding treatment of relapsed or refractory pediatric cancers. *Nature Medicine* **30,** 990–1000 (2024)

18. Zhang, X. *et al.* Signature construction and molecular subtype identification based on liver-specific genes for prediction of prognosis, immune activity, and anti-cancer drug sensitivity in hepatocellular carcinoma. *Cancer Cell International* **24,** (2024)

19. Ghoshdastider, U. & Sendoel, A. Exploring the pan-cancer landscape of posttranscriptional regulation. *Cell Reports* **42,** 113172 (2023)

20. Hao, S., Yang, Z., Wang, G., Cai, G. & Qin, Y. Development of prognostic model incorporating a ferroptosis/cuproptosis-related signature and mutational landscape analysis in muscle-invasive bladder cancer. *BMC Cancer* **24,** (2024)

21. Sharifi, M. N. *et al.* Clinical cell-surface targets in metastatic and primary solid cancers. *JCI Insight* **9,** (2024)

22. Chapekis, A., Bestvater, S., Remy, E. & Rivero, G. When Online Content Disappears. https://www.pewresearch.org/data-labs/2024/05/17/when-online-content-disappears/ (2024)

23. Dabney, A. R. Classification of microarrays to nearest centroids. *Bioinformatics* **21,** 4148–4154 (2005)

24. Tibshirani, R., Hastie, T., Narasimhan, B. & Chu, G. Diagnosis of multiple cancer types by shrunken centroids of gene expression. *Proceedings of the National Academy of Sciences* **99,** 6567–6572 (2002)

25. Sjödahl, G. *et al.* A Molecular Taxonomy for Urothelial Carcinoma. *Clinical Cancer Research* **18,** 3377–3386 (2012)