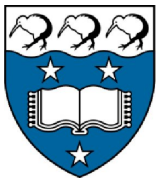
Graphical applications for large-scale conservation projects

Oliver Stevenson



Department of Statistics
University of Auckland
November 9, 2015

Acknowledgements

I would like to thank my supervisor, Rachel Fewster, for her guidance, encouragement and always positive attitude throughout this project.

Also many thanks to my family, Chris and Steph for your support during the year.

Abstract

At a glance, data is more meaningful when presented in graphical form. This project explored innovative methods of automating the display of catch data for large-scale conservation projects. High priority was given to developing methods that allow users to interact with their data, affording them some control over the graphics that are produced. Two interactive applications were developed that allow conservation volunteers to select the data they want to view and how to view it. After a day in the field, volunteers are able to use these applications to see their day's work summarised on a map or graphic. These graphics highlight the positive impact their efforts are having on the local environment, keeping volunteers motivated and engaged in their work. Various methods of improving the automation of these graphics are outlined, as well as other practical uses of these statistical applications.

Contents

1 Introduction

This honours project concerns the display and analysis of catch data from various large-scale animal trapping programmes in New Zealand. The primary aim of these programmes is to eradicate common pests found in New Zealand environments, such as rats, weasels, stoats and possums. The work conservation volunteers contribute to the country's environment is invaluable, helping protect our native bush and native species, such as the kiwi. Exploring ways and means of keeping these volunteers motivated in their work, as well as encouraging new volunteers to get involved, is crucial to encouraging community involvement in these programmes.

Recording conservation data allows communities to quantify the positive impact of pest eradication programmes on their local environments. However, for volunteers affiliated with such programmes, it can be difficult to ascertain the impact of their work using just facts and figures. This same issue arises when trying to inform an interested member of the public inquiring about a project. Is trapping 100 rats in the last year a great success? Is trapping 20 fewer stoats this month compared to last a noticeable decline?

A more effective method of presenting conservation data is to use attractive and informative graphics. Finding innovative, visual means of presenting data does the job of both informing the public of the impact of local projects, and providing motivation for conservation volunteers to continue their work. However, interpreting and displaying this data requires computing and statistical knowledge that the everyday conservation volunteer might not possess.

As statisticians we are able to provide solutions to these issues, but is time consuming to produce graphics for every potentially interesting pattern present in the vast quantities of data being recorded. The main focus of this project was therefore to explore automated methods of providing conservation volunteers with the ability to produce informative graphics themselves, without coming into direct contact with computer code or statistical jargon.

The solution was to develop two interactive applications to produce automated graphics. The first produces a variety of customisable bar charts and the second a selection of heat maps. Both applications allow individual users to choose how they wish to view their data and require only web browsers to access.

1.1 Report structure

Chapter 2 outlines the format and structure of the data and the benefits of automating graphic production. Chapter 3 details the software used to create graphics, as well as several of the computational challenges to overcome.

Chapter 4 provides an overview of the bar chart application and describes some of the complexities, while Chapter 5 illustrates a practical use of this application in the form of a bait comparison trial.

Chapter 6 provides an overview of the heat map application and describes several of the viewing options available. Finally, Chapter 7 details the methodology behind the production of the heat maps, and the difficulties encountered when automating this process.

2 Data and CatchIT

This project is concerned with animal catch data from various conservation programmes around New Zealand. The data are obtained from CatchIT, an online data management program for conservation pest-control projects. Catch data are stored in easy-to-read tables that can be downloaded anytime (CatchIT (2011), https://www.stat.auckland.ac.nz/~fewster/CatchIT/). A variety of information about each catch is available, such as the date, species caught, the specific trap type and bait used, and the person who recorded the catch.

2.1 Format of data

Catch data are recorded by park rangers and volunteers who then upload their records to CatchIT. Every conservation project registered with CatchIT has a master CSV file which is added to each time an upload for that specific project is made. Thus, up-to-date records of any project's all-time data are always available in the format of a CSV file (*Figure 1*).

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	Area	Line	Date	TrapName	TrapChecked	Triggered	Species	Sex	Age	PestID	Bait	Comments	SampleID	SampleConditi	Latitude	Longitude	TrapType	TimeTaken	FirstName
2	Te Henga	Wainamu	20/02/2014	L1	Y	N					Egg Lure				-36.877833	174.489671	DOC200		Matt
3	Te Henga	Wainamu	20/02/2014	L2	Y	N					Egg Lure				-36.879501	174.491083	DOC200		Matt
4	Te Henga	Wainamu	20/02/2014	L3	Y	N					Egg Lure				-36.88095	174.49202	DOC200		Matt
5	Te Henga	Wainamu	20/02/2014	L4	Y	N					Egg Lure				-36.882583	174.491839	DOC200		Matt
6	Te Henga	Wainamu	20/02/2014	L5	Y	N					Egg Lure				-36.884312	174.492018	DOC200		Matt
7	Te Henga	Wainamu	20/02/2014	L6	Y	N					Egg Lure				-36.886715	174.49514	DOC200		Matt
8	Te Henga	Wainamu	20/02/2014	L7	Y	N					Egg Lure				-36.886927	174.493326	DOC200		Matt
9	Te Henga	Wainamu	20/02/2014	L8	Y	N					Egg Lure				-36.887246	174.491626	DOC200		Matt
10	Te Henga	Wainamu	20/02/2014	L9	Y	N					Egg Lure				-36.887483	174.489982	DOC200		Matt
11	Te Henga	Wainamu	20/02/2014	L10	Y	N					Egg Lure				-36.887836	174.487744	DOC200		Matt
12	Te Henga	Wainamu	20/02/2014	L11	Y	N					Egg Lure				-36.88654	174.486832	DOC200		Matt
13	Te Henga	Wainamu	20/02/2014	L12	Y	N					Egg Lure				-36.885616	174.485008	DOC200		Matt
14	Te Henga	Wainamu	20/02/2014	L13	Y	N					Egg Lure				-36.885394	174.484072	DOC200		Matt
15	Te Henga	Wainamu	20/02/2014	L14	Y	N					Egg Lure				-36.886062	174.482503	DOC200		Matt
16	Te Henga	Wainamu	20/02/2014	L15	Y	N					Egg Lure				-36.887231	174.481314	DOC200		Matt
17	Te Henga	Wainamu	20/02/2014	L16	Y	Y	Weasel				Egg Lure				-36.88792	174.47952	DOC200		Matt
18	Te Henga	Wainamu	20/02/2014	L17	Y	N					Egg Lure				-36.887528	174.477751	DOC200		Matt
19	Te Henga	Wainamu	20/02/2014	L18	Y	N					Egg Lure				-36.88721	174.477173	DOC200		Matt
20	Te Henga	Wainamu	20/02/2014	L19	Y	Y	Ship Rat				Egg Lure				-36.888473	174.476411	DOC200		Matt
21	Te Henga	Wainamu	20/02/2014	L20	Y	N					Egg Lure				-36.8887	174.475552	DOC200		Matt
22	Te Henga	Wainamu	20/02/2014	L21	Y	N					Egg Lure				-36.889444	174.473715	DOC200		Matt
23	Te Henga	Wainamu	20/02/2014	L22	Y	N					Egg Lure				-36.890634	174.474377	DOC200		Matt
24	Te Henga	Wainamu	20/02/2014	L23	Y	N					Egg Lure				-36.890569	174.476721	DOC200		Matt
25	Te Henga	Wainamu	20/02/2014	L24	Y	N					Egg Lure				-36.891312	174.475703	DOC200		Matt
26	Te Henga	Wainamu	20/02/2014	L25	Y	N					Egg Lure				-36.891514	174.473923	DOC200		Matt
27	Te Henga	Wainamu	20/02/2014	L26	Y	N					Egg Lure				-36.893119	174.473931	DOC200		Matt
28	Te Henga	Wainamu	20/02/2014	L27	Y	Y	Ship Rat				Egg Lure				-36.892288	174.47258	DOC200		Matt
29	Te Henga	Wainamu	20/02/2014	L28	Y	N					Egg Lure				-36.89154	174.47114	DOC200		Matt
30	Te Henga	Wainamu	20/02/2014	L29	Y	N					Egg Lure				-36.89231	174.470078	DOC200		Matt
31	Te Henga	Wainamu	20/02/2014	L30	Y	N					Egg Lure				-36.891745	174.46909	DOC200		Matt
32	Te Henga	Wainamu	20/02/2014	L31	Y	N					Egg Lure				-36.890813	174.467939	DOC200		Matt
33	Te Henga	Wainamu	20/02/2014	L32	Y	N					Egg Lure				-36.890014	174.466263	DOC200		Matt
34	Te Henga	Wainamu	20/02/2014	L33	Y	N					Egg Lure				-36.889463	174.464872	DOC200		Matt
35	Te Henga	Wainamu	20/02/2014	L34	Y	N					Egg Lure				-36.888095	174.463982	DOC200		Matt
36	Te Henga	Wainamu	20/02/2014	L35	Y	N					Egg Lure				-36.887089	174.462908	DOC200		Matt
37	Te Henga	Wainamu	20/02/2014	L36	Y	N					Egg Lure				-36.887202	174.46107	DOC200		Matt
38	Te Henga	Wainamu	20/02/2014	L37	Y	N					Egg Lure				-36.886867	174.459717	DOC200		Matt
39	Te Henga	Wainamu	20/02/2014	L38	Y	N					Egg Lure				-36.886926	174.45716	DOC200		Matt
40	Te Henga	Wainamu	20/02/2014	L39	Y	N					Egg Lure				-36.887523	174.454758	DOC200		Matt
41	Te Henga	Wainamu	20/02/2014	L40	Y	N					Egg Lure				-36.887956	174.4525	DOC200		Matt
42	Te Henga	Wainamu	20/02/2014	L41	Y	N					Egg Lure				-36.881438	174.456303	DOC200		Matt
43	Te Henga	Wainamu	20/02/2014	L42	Y	N					Egg Lure				-36.880325	174.457382	DOC200		Matt
44	Te Henga	Wainamu	20/02/2014	L43	Y	N					Egg Lure				-36.878963	174.458893	DOC200		Matt
45	Te Henga	Wainamu	20/02/2014	L44	Y	N					Egg Lure				-36.877588	174.461481	DOC200		Matt
46	Te Henga	Wainamu	20/02/2014	L45	Y	N					Egg Lure				-36.876957	174.464443	DOC200		Matt
47	Te Henga	Wainamu	20/02/2014	L46	Y	N					Egg Lure				-36.876739	174.466043	DOC200		Matt
48	Te Henga	Wainamu	20/02/2014	L47	Y	N					Egg Lure				-36.876279	174.468996	DOC200		Matt
49	Te Henga	Wainamu	20/02/2014	L48	Y	N					Egg Lure				-36.876289	174.471061	DOC200		Matt
50	Te Henga	Wainamu	20/02/2014	L49	Y	Y	Ship Rat				Egg Lure				-36.876331	174.47353	DOC200		Matt
51	Te Henga	Wainamu	20/02/2014	L50	Y	N					Egg Lure				-36.875853	174.475686	DOC200		Matt
52	Te Henga	Wainamu	20/02/2014	L51	Y	N					Egg Lure				-36.876053	174.477799	DOC200		Matt

Figure 1. Example of a CatchIT.csv file

Having data in this format is advantageous as it is structured to have one catch record per row and uses columns for each of the variables such as date, species, bait, et cetera. This makes it highly compatible with statistical software such as R, allowing for easy data handling and analysis.

An example of the benefits of storing data in CatchIT can be seen with the Te Henga project in the Waitakere Ranges, West Auckland. Previous to joining CatchIT, the Te Henga project stored data in text documents such as the one shown in *Figure 2*.

Trap catch data. Wainamu circuit [Wai] and Wetland circuit [Wet]
Species: Weasel [We], Stoat [St], Ship rat [Sr], Norvegian rat [Nr], Ferret [Fe], Hedgehog [He]
Possum [Po]
Steve Allen cat traps SA
From 20/02/14 -26/04/14 Egg lure
20/02/14 Wai We 16, Sr 19, 27, 48, 55.
07/03 Wai We 11, Sr 50. Time 3h15
21/03 Wai We 6, 11, 24, 27, Sr 29, 47. Time 3h30
02/04 Wai Sr 13, 15, 16, 25, 34, We 5. Time 3h 20
12/04 Wet Placing eggs
18/04 Wai Sr 2, 20, He 5, We 19, Sr 4, 14, 26, 41, 49, 54. Time 3h 45
26/04 Wet He MC10, MC11, St KJ3, KJ8, We MC15, KJ5, Sr KJ2, KJ9, KJ10, KJ12. Time 3h 30
30/04 Wai He 48, Sr 11, 14, 21, 24, 43, 45, 52, 54, 55. Time 3h 40
12/05 Wai Sr 17, 22, 25, 27, 28, 43, He 54
12/05 Wet Sr MC14, KJ2, KJ10, St KJ11. Time 6h 55
26/05 Wai We 30, 32, Sr 13, 16, 17, 27, 48.
26/05 Wet He MC14, St KJ3, Sr KJ8 Time 6h 55
14/06 Wai Sr 15, 17, 29, 34, 43, 44, 50, 51, We 30
14/06 Wet We MC14. Time 7h 9
28/06 Wet Sr MC10, We MC14, MC15,
28/06 Wai Sr 2, 14, 29, 51, We 27, 40, 48, St 8, 16, 22, Po SA@16 Time 8h 23
11/07 Wet Sr DR5, MC4, KJ2, We MC7, Po SA@MC17. Time 3h 50
13/07 Wai Sr 33, 55, St 22, We 30. Time 4h 7
26/07 Wai Po SA@11, Sr 14,22,23, 48, 50, 51, 52, 53, 54. Time 4h 42
28/07 Wet Sr MC15, KJ10, Po SA@MC4, SA@MC17. Time 4h25
From 26/07/14 alternating Erayze and salted rabbit lure
08/08 Wai Sr 9, 20, 21, 28, 29, 32, 44, 45, 47, 52, 53 Po SA@16, St 22, 30, We 21 Time 4h24
09/08 Wet Sr DR6, MC4, MC10, KJ3, KJ10, KJ11, Po SA@MC4, We MC7, MC15, He MC9, 4h
23/08 Wet Po SA@MC4, SA@MC17, He MC7, We MC15, KJ5 Sr KJ2, St KJ6, Time 3h40
24/08 WAi Po SA@16,We 31, Sr 16, 17, 19, 41, 44, 49. Time 3h 45
05/09 Wai Sr 14, 17, 20, 44, 45. Time 4h 26
06/09 Wet Sr KJ10, KJ12 Time 4h26 **Spotless crake heard@MC10**
20/09 Wet Po SA@MC4, Sr KJ5, We MC7, KJ3, KJ10, Time 3h 52
22/09 Wai Sr 12, 20, 27, 36, We 31 Time 4h6
03/10 Wai Sr 3, 15, SA@16, 19, 27, 48, Nr 43 Time 3h 50

Figure 2. Example of Te Henga data prior to storage in CatchIT

Data in this format is clearly difficult to interpret, let alone analyse using statistical software. To summarise the data, manual counts are required, which are time consuming and often inaccurate. Therefore, a benefit of CatchIT storing data in a consistent format is that any R code used to produce graphics and analyses for one project can ideally be applied to other projects, without any change to the code. This greatly increases our efficiency as statisticians, giving us more time to focus on other areas of a project.

While at face value this sounds relatively straightforward, a great deal of effort is required at the programming stage to ensure that the selected data set will produce relevant and attractive graphics. The code has to account for any combination of missing or extreme values to ensure that the resulting plots are attractive and display the appropriate information. This might require computations to be designed and applied to the data set to ensure that the default parameters produce sensible graphics and analyses for each CatchIT project.

2.2 Applications for CatchIT data

The initial aim of the project was to explore interesting methods of presenting and analysing catch data from large-scale animal trapping programmes in New Zealand. Given the vast quantities of data these conservation projects are recording, it can be difficult to effectively present data in an interesting manner that is still easily interpretable to the everyday conservation volunteer. Conventional, simple methods of displaying data (for example static plots), while informative, might fail to grasp the attention of a viewer. Therefore a main focus was to develop methods of viewing data that are stimulating, while being easy to interpret, as the users of these graphics and analyses will vary greatly in their statistical and computing knowledge.

The idea behind this aim is that after a volunteer comes home after a day in the field, they are able to go online and see their day's work presented on a map or graphic, highlighting their own individual contribution to their project. Most people would find data that is presented graphically not only more interesting than staring at a table of numbers, but also more informative. Ideally, seeing customised personal data will keep volunteers motivated, and it illustrates exactly how their own work is impacting their local community.

In the past, when writing reports for local councils and funding bodies, project leaders would either produce graphics themselves, or not at all. Therefore, by automating the production of graphics, we as statisticians bring a lot to the table in terms of value to these projects. Enabling easy access to summary graphics and analyses eliminates the need for volunteers to have any statistical programming knowledge as well as giving peace of mind that their data have been handled correctly.

Therefore, the graphics and analyses must strike a balance between: interesting but complex and simple but boring. The solution was to develop interactive applications which display simple, informative graphics, but are entirely user controlled. These are interactive in the sense that the user has the ability to decide exactly which elements of the data they wish to view, and how they want to view them. Given the many different categories of data stored in CatchIT, this allows users to choose the graphics to display the aspects of the project they are particularly interested in.

Using R (version 3.2.2) and R Shiny, I have developed two interactive applications. The first creates bar charts that display catch data for any selected combination of categories. This may range from simply viewing total catch numbers for each species, to the number of stoats caught using peanut butter in 2015 by a specific volunteer. The second application generates heat maps

that show 'hotspots' of catches on a Google map of a selected conservation project. These maps can be customised to only show the catches for a specific species or specific volunteer. Essentially, the applications bring the data to life, giving the user complete freedom to explore their project by providing numerous permutations of viewing options.

3 R Shiny

R Shiny is a web application framework for R that allows users to turn analyses into applications with interactive interfaces that can be accessed via the internet (RStudio, Inc, 2013). Such applications can be created to bring data and analyses to life by allowing individual users to select what they want to view, and how they want to view it. An application will usually consist of a variety of selection options, with different combinations of selections resulting in different outputs. Entire applications can be coded using the R language, though HTML, JavaScript and CSS languages are supported.

3.1 Overview

An R Shiny application consists of two components; a user interface (UI) and server. The UI is what the user sees on screen, and consists of 'inputs' such as the various selection options, and 'outputs' that are R-generated analyses and graphics. The server is what links the UI with R. This is done by running specified R code when a certain input is selected in the UI and returning the result as an 'output', viewable in the UI.

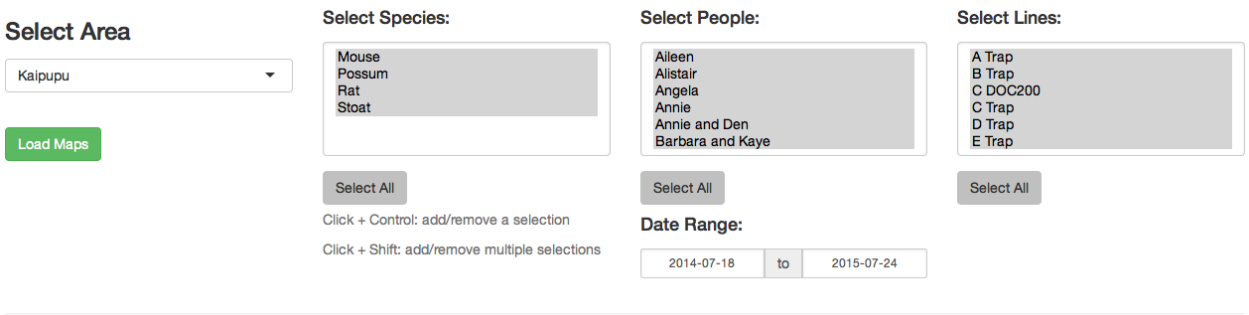


Figure 3. Example R Shiny user interface

Figure 3 gives an example UI with numerous inputs in the forms of the various selection options. Users are able to select which area, species, people, lines and dates they wish to view catch data for. The application will then take the selected inputs and return the appropriate analysis based on these selections.

3.2 The user interface

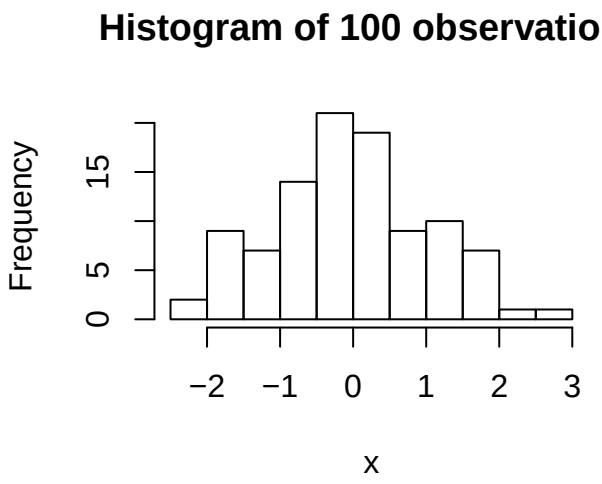
The UI is one of R Shiny's most unique features. It allows an interactive interface to send R syntax to the console via the server code, rather than running code from a script file. Thus, assuming the

server code has been correctly set up, a user has access to R code with complex options, without ever coming into direct contact with R syntax. This is particularly useful in the present case, as many volunteers linked with CatchIT projects will have little to no familiarity with R, yet will be able to produce R-graphics.

One of our main goals is to allow users to directly interact with the data, without having to do any computing themselves. The UI is our first step towards achieving this. Coding an interface that is tailored to the graphics we wish to produce provides users with an unintimidating computing environment to explore the elements of the data they are most interested in. A typical UI contains a number of 'inputs' that can be chosen from a variety of selection tools, such as dropdown menus, checkboxes and action buttons.

The following is a simple example of the difference between running code directly through the R console, and through a UI. Say we wish to simulate and plot a number of observations from a standard normal distribution. In R, this is as simple as:

```
## Simulate and plot 100 observations from a standard normal
hist(rnorm(100), main = paste("Histogram of 100 observations"), xlab = "x")
```



However, we can produce the same result by building a user interface in R Shiny whereby the user can generate the plot themselves through a web browser window. The user is presented with a slider bar to select the number of observations they wish to plot, and the application automatically produces the appropriate plot:

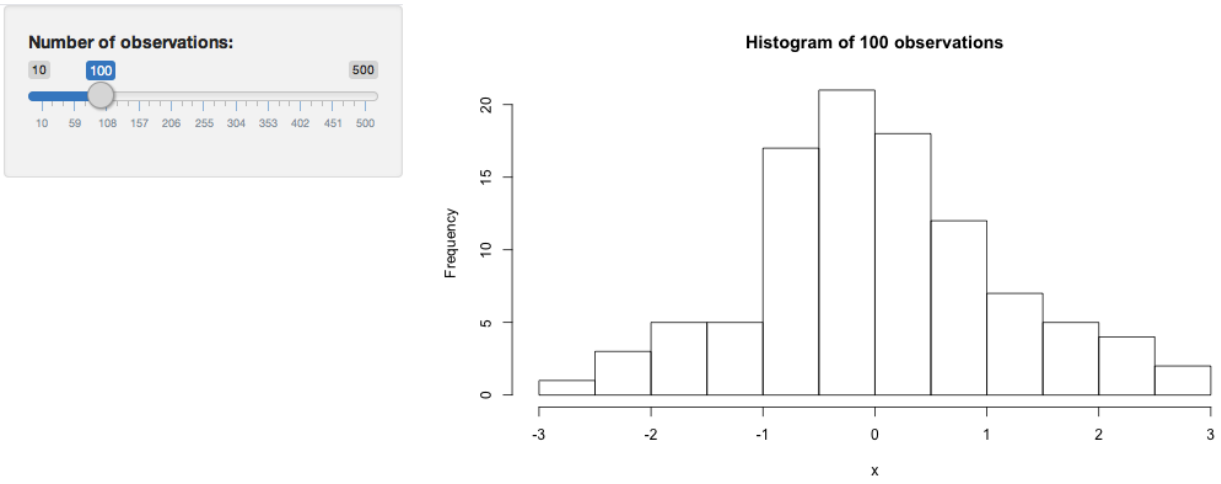


Figure 4. R Shiny histogram interface

Therefore, we have created a simple application where we have effected all the coding for the user, but all they see is a slider and a plot. This gives the user the ability to run R code without having to do any coding themselves.

3.3 Reactive environment

While the UI allows R code to be executed by clicking buttons, what makes programming in R Shiny discernibly different from R is that it allows for coding in a reactive environment. This implies that an application will undergo some sort of reaction in response to an 'event'. An event relates to the changing of an input and can take various forms, such as clicking a specific button, or changing a selection. Thus, as a user changes a particular input, the output will adapt accordingly, giving the user greatly increased flexibility in how they wish to analyse and view data.

Reactivity is primarily coded within the server function of the R Shiny code. Any output that is displayed within the UI can take a dependency on a reactive input value by making the appropriate call to `input$`. Therefore, changing a particular input causes the output to react and adapt accordingly.

In the example in *Figure 4* above, the object `input$number` is a reactive input in the sense that it reacts to match the value selected by the slider. Consequently, the histogram is a reactive output as it takes a dependency on the value of `input$number`. Once the user has made a selection, R reacts accordingly by updating the value of `input$number`, which in turn causes the output to react and produce the appropriate histogram.

```
## Figure 4 code
## Function creating user interface
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      ## Creates the slider selecting the number of observations
      ## The value specifies the default value when the app first loads
      sliderInput("number", label = "Number of observations:", min = 10, max
                  = 500, value = 100)
    ),
    mainPanel(plotOutput("distPlot"))
  )
)
## Server function linking R with the user interface
server <- function(input, output) {
  ## Plots a histogram where input£number = the selection of the slider
  output$distPlot <- renderPlot({
    hist(rnorm(input$number), main = paste("Histogram of", input$number,
                                           "observations"), xlab = "x")
  })
}
shinyApp(ui = ui, server = server)
```

This reiterates the earlier point that a user is able to run R code without realising, and without being overwhelmed by confusing computing jargon. We can extend this idea of using a reactive environment to producing graphics for the catch data stored in CatchIT. If we code our server function carefully, we can be confident that the various inputs will work regardless of which project's data we are viewing, as all projects are stored in an identical format.

3.4 Useful functions for programming in R Shiny

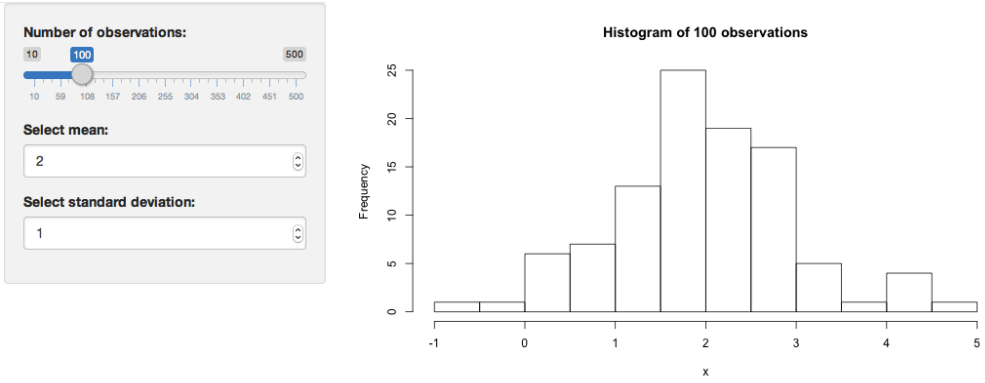
3.4.1 The `renderUI` function

While allowing for improved interactivity and flexibility, programming in R Shiny's reactive environment presents some challenges to overcome. The application in *Figure 4* illustrates an example of how a reactive value can change an output. However, should we want the interface itself to change dependent on the user's selections, R Shiny allows for the creation of dynamic UIs through the use of the **`renderUI`** function. That is, as a user changes various selections, this alters how inputs are presented within the UI.

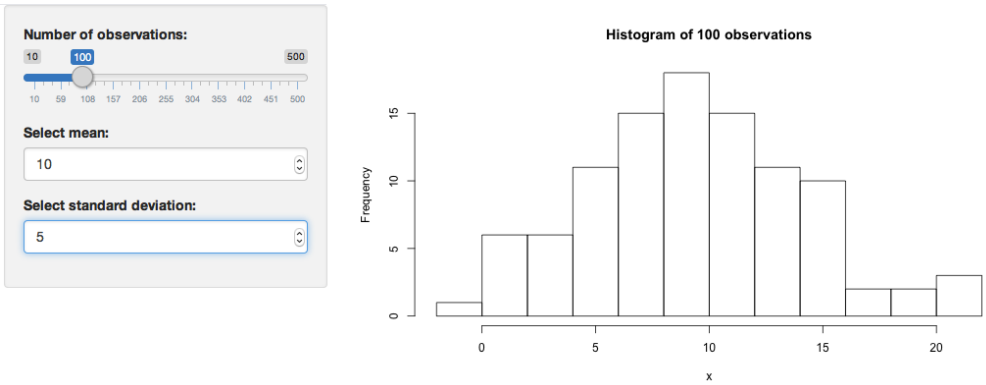
To build on the simple application in *Figure 4*, we might want to include options for the user to select the mean and standard deviation for the distribution of the observed values. Further, we might wish to constrain the distribution such that the default mean is twice the standard deviation. Using the **`renderUI`** function, we can ensure that whenever the input for standard deviation is changed, the value for the mean seen on the UI changes accordingly to fulfil the constraint.

```
## UI code
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      sliderInput("number", label = "Number of observations:", min = 10, max
                  = 500, value = 100),
      ## The selection for mean as a dynamic input
      uiOutput("mean"),
      ## The selection for standard deviation as a static input
      numericInput("sdev", label = "Select standard deviation:", value = 1)
    ),
    mainPanel(plotOutput("distPlot"))
  )
)
## Server code
server <- function(input, output){
  output$distPlot <- renderPlot({
```

```
    hist(rnorm(input$number, input$mean, input$sdev), main = paste("Histogram
         of", input$number, "observations"), xlab = "x")
  })
  ## The mean as a dynamically changing input, relying on value of std.dev
  output$mean <- renderUI({
    numericInput("mean", label = "Select mean:", value = 2*input$sdev)
  })
}
shinyApp(ui = ui, server = server)
```



(a) Mean = 1, Standard Deviation = 2



(b) Mean = 5, Standard Deviation = 10

Figure 5. The mean automatically defaults to twice the selected standard deviation

The example in *Figure 5* also demonstrates another interesting feature that distinguishes R Shiny from R. Unlike R, code in R Shiny is not strictly run in a hierarchical order from top to bottom. Due to the reactive environment, there may be objects defined near the beginning of the

code that depend on values of objects defined later in the code, and vice versa. Applications can quickly develop such that objects have a complex web of dependencies on one another.

In *Figure 5*, the plot is created within the `output$distPlot` code chunk, taking the arguments `input$number`, `input$mean` and `input$sdev`. However, the value for `inupt$mean` is defined in `output$mean`, which is defined after the plot function is called. In R, this would normally give an `'object not found'` error, as it would attempt to run `hist(rnorm(input$number, input$mean, input$sdev))`, before `input$mean` is defined. However, R Shiny recognises that in order to generate the plot, `input$mean` is required, thus the chunk `output$mean` is executed first.

These considerations mean that R Shiny code does not always execute in the order in which it is written, and at times the execution order can be difficult to anticipate. Code chunks updating out of order, while often not fatal to the application, can cause some fairly unusual problems. An example is outlined in the bar chart application in Section 4.2.2, where bar chart titles update before the graphic itself, briefly showing the wrong data below the correct title.

3.4.2 The `isolate` and `observeEvent` functions

Suppose we now have an output that depends on multiple reactive values. Due to the reactivity of our inputs, the output constantly updates itself whenever we change any of the inputs. Though not seen in these static images, whenever an application updates itself in R Shiny, a brief flash can be seen as a new plot is generated. This continual flashing is quite unpleasant, and may become confusing as a user scrolls through input values. Instead, we might prefer the application to update itself only when the user is satisfied with the input selections. This can be achieved using the `isolate` and `observeEvent` functions.

When the `isolate` function encloses an expression, this tells R Shiny that the expression should not take a dependency on any reactive objects inside of it. As a result, the expression will not automatically update itself every time the value of a reactive object it depends on changes. In the case of the application in *Figure 5*, we do not want the histogram to continually update itself until the user is happy with their selections for `input$number`, `input$mean` and `input$sdev`.

```
## UI code same as for Figure 5: omitted here
## Server code
server <- function(input, output) {
  output$mean <- renderUI({
```

```
    numericInput("mean", label = "Select mean:", value = 2*input$sdev)
  })
  ## The plot with all reactive dependency removed
  output$distPlot <- renderPlot({
    isolate(hist(rnorm(input$number, input$mean, input$sdev), main = paste
                 ("Histogram of", input$number, "observations"), xlab = "x"))
  })
}
```

Enclosing the histogram expression within `output$distPlot` with `isolate` prevents the histogram from updating itself each time the user changes the value of one of the three reactive objects. However, now the histogram will never update as we have removed all dependency on the reactive objects it takes as arguments. Using the `observeEvent` function, we can tell the isolated expression to update itself when, and only when, a defined 'event' occurs.

An 'event' can take one of many forms. In this example, an action button has been added for users to click when they are satisfied with their selections. Clicking this button will trigger the event `input$simulate`, and tells the histogram to update itself to match the present selections.

```
## UI code
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      sliderInput("number", label = "Number of observations:", min = 10, max =
                    500, value = 100),
      uiOutput("mean"),
      numericInput("sdev", label = "Select standard deviation:", value = 1),
      ## The action button in the user interface
      actionButton("simulate", label = "Simulate Observations!")
    ),
    mainPanel(plotOutput("distPlot"))
  )
)
```

```
## Server code
server <- function(input, output) {
  output$mean <- renderUI({
    numericInput("mean", label = "Select mean:", value = 2*input$sdev)
  })
  ## The plot only updates when input£simulate is observed
  ## (i.e. clicking the action button)
  observeEvent(input$simulate, {
    output$distPlot <- renderPlot({
      isolate(hist(rnorm(input$number, input$mean, input$sdev), main = paste
              ("Histogram of", input$number, "observations"), xlab = "x"))
    })
  })
}
shinyApp(ui = ui, server = server)
```

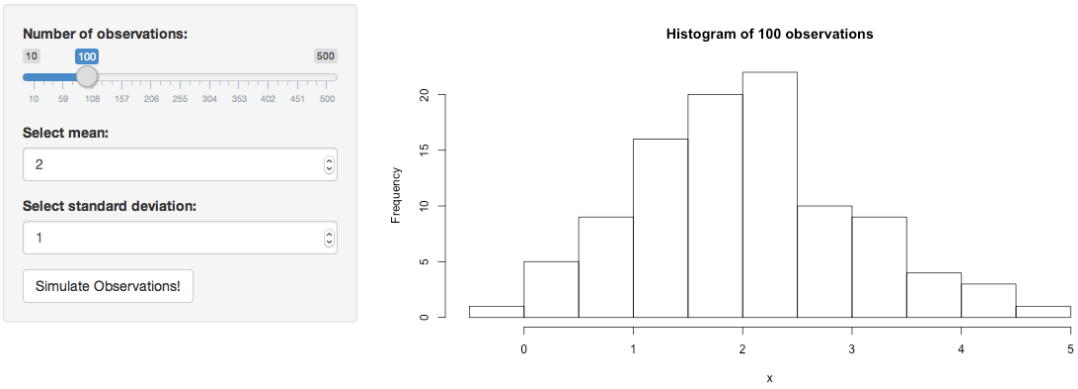


Figure 6. Clicking 'Simulate Observations!' updates the histogram

Because the graphics and analyses for larger projects can take up to a few seconds to load and display, use of the `isolate` and `observeEvent` functions is very important to prevent applications from automatically updating when a selection is changed. Allowing an application to continually re-compute graphics quickly becomes computationally expensive, greatly reducing the user experience which is one of the major benefits of using R Shiny in the first place.

This simple application demonstrates several important R Shiny functions that are used frequently in the applications detailed in Chapters 5 and 7.

4 Bar chart application

The bar chart application gives users the ability to view catch numbers for any project, which can be colour coded by a range of variables. The user can also choose to display catch numbers for individual selections of a variable, such as a plot for each volunteer, or a plot for each species. The default setting displays an 'overall' bar chart depicting the total number of catches for each species in the current year, as well as individual bar charts for each volunteer's catch data broken down by species for that year.

Figure 7. Example bar chart application interface. The top plot depicts the overall information for the area, while the following plots are the individual plots

The interface in *Figure 7* gives an example of the output a user would be presented with using the default settings for the Okura Bush project. Section 4.1 below gives an overview of the application. Code excerpts are included to illustrate the complexity of coding in such general terms, while also outlining the web of interdependancies between inputs within the interface. Also included are examples of code excerpts that are are dedicated to exhibiting the level of attention to detail required—such as, how to deal with users who have identical names, ensuring legends are of reasonable size, and automating the colour selections to ensure graphics are attractive and easy to read.

4.1 Overview and walk-through

The application presents users with a number of selection options, allowing them to customise the bar charts to cater to their personal preferences. Firstly, a user must select (i) an area (project), (ii) how they wish to colour the bar charts, (iii) whether they want to produce individual plots for any variable, and (iv) which year(s) they wish to view data from. Continuing with the example in *Figure 7*, the user has selected to view data for (i) Okura Bush, (ii) to colour their plots by species, (iii) to display plots for individual people, and (iv) to only plot catches from 2015. Under the 'Colour Options' tab, they have chosen to view data for all available species and under 'Specific plot options' they have chosen to only view catch data for volunteers 'Jonathan' and 'Rob M'. Individual plots for the items selected under 'Specific plot options' are also produced. Each of these selections is an 'input' and changing any will result in a different set of output graphics.

Although the graphics the application produces are fairly simple, the interactions and dependencies between the inputs are reasonably complex. Since each project has unique data, the options available within each input will vary dependent on the currently selected project. As such, the interface must be very flexible in order to accommodate these differences. Therefore, a dynamic UI using `renderUI` has been implemented to allow the various inputs to change according to the selected area. Upon selecting an area, the application loads the corresponding master CSV file for that area and reacts accordingly, only allowing feasible selections for the various inputs to be available for selection by the user. The application begins by running a function that computes all the key information that is required for determining the various inputs (for example, unique species; unique volunteers; unique years). Due to privacy reasons, only volunteers' first names are recorded. Therefore this function is designed to recognise if there are any duplicates in volunteer

names and, if so, attaches their username so each person can be uniquely identified.

```
## Server code excerpt:
## Reactive function of the data - returns a list containing the key
## information for the selected area
area.data <- reactive({

  ## Update only when a new area is selected - otherwise remain isolated
  input$selectArea
  isolate({

    ## Area name.csv
    csvname <- paste(input$selectArea, ".csv", sep = "")

    ## Read the all time data
    dat <- read.csv(csvname, as.is = TRUE)
    ...
    ...
    ...
    < Ommitted 25 lines of code assigning each trapcheck: an index, a       >
    < calender year, a calender month, a julian day, the previous bait used >
    < and the previous person to have checked the trap                       >

    ## Extract the project's unique species, volunteer names, lines, traps,
    ## baits and years
    unique.species <- unique(dat$Species[dat$Species != ""])
    unique.names <- unique(dat$FirstName[dat$FirstName != ""])
    unique.lines <- unique(dat$Line[dat$Line != ""])
    unique.traps <- unique(dat$TrapName[dat$TrapName != ""])
    unique.bait <- unique(dat$Bait[dat$Bait != ""])
    unique.years <- unique(dat$calyear[dat$calyear != ""])
    ...
```

```
...
...
< Ommitted 25 lines of code assigning each trapcheck a previous trap  >
< index referring to the index of the last trapcheck for that trap    >
< Also creates an object 'datcatch', which only contains trapchecks   >
< where a species was caught                                          >

## Determining duplicates - trim.func ensures accidental leading and
## trailing spaces are removed from names
## Assign alias to be each person's 'trimmed' name
alias <- trim.func(unique.names)

## Determine which (if any) are duplicate First Names
which.dups <- which(duplicated(alias) | rev(duplicated(rev(alias))))

## Add username to any duplicate names so volunteers are distinguishable
## (e.g. Ollie, oste1)
if(length(which.dups) != 0){
  alias[which.dups] <- paste(alias[which.dups],
                             unique(dat$Username)[which.dups], sep= ", ")

  ## Assign each duplicate person their FirstName + Username
  for(i in 1:length(dat[, 1])){
    person.counter <- match(dat$FirstName[i], unique.names)
    dat$FirstName[i] <- alias[person.counter]
  }
}

## List of key information to be extracted when determining selection
## options to be available on the UI
list(dat = dat, datcatch = datcatch, unique.species = unique.species,
```

```
          unique.names = unique.names, unique.lines = unique.lines,
          unique.traps = unique.traps, unique.bait = unique.bait,
          unique.years = unique.years)
  })
})
```

Using the values from this function, the application determines the unique years the area has available data, and allows only these years to be selectable options under the 'Select Year' tab. The selected project (Okura Bush) only has recorded data for the years 2014 and 2015, hence these are the only available options under 'Select Year'. When viewing data by 'Month over all years', or simply by 'Year', the application uses the entire dataset. Therefore to keep things simple, the dropdown menu is only available when the user has selected to view data by 'Month in a selected year'. Having the menu on screen when the user wishes to view all time data serves no purpose, other than to confuse users into thinking they must select a year when it will have no impact on the graphics produced.

```
## UI code excerpt:
uiOutput("selectYear")


## Server code excerpt:
## Reactive dropdown menu for year selections
output$selectYear <- renderUI({

  ## Define the data as checks where a species was caught
  datcatch <- area.data()$datcatch

  ## Extract the unique years available for the area and sort into decreasing
  ## order
  unique.years <- unique(datcatch$calyear)
  unique.years <- sort(unique.years, decreasing = TRUE)

  ## Conditional dropdown menu - only appears when the user selects to view
```

```
  ## data in a selected year
  conditionalPanel(
    condition = "input.selectAggregate == 'Month in a selected year'",
    selectInput("selectYear",
                label = "Select Year",
                choices = c(unique.years)
    )
  )
})
```

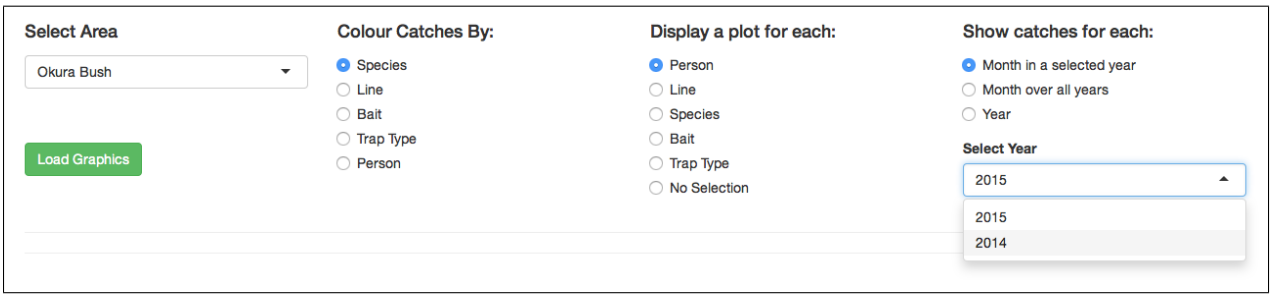


Figure 8. Select year options for Okura Bush

The application then determines the options that should be available under the 'Colour Options' tab. Because the user has selected to colour catches by 'Species', these options are the unique species that have been caught in the selected area, Okura Bush. However, because the user has selected to only view data from 2015, the application has filtered out any species that were not caught in the selected year. For example, if the user had selected to view data exclusively from 2014, there is no 'Possum' option available, as no possums were caught in Okura Bush during 2014.

```
## Ui code excerpt:
uiOutput("subject")


## Server code excerpt:
## Reactive list of 'subject' options ('colour by' selection options)
output$subject <- renderUI({


  ## Define 'subject' so R looks in the correct column of csv file
```

```
subject <- input$selectSubject
if(input$selectSubject == "Person"){
  subject <- "FirstName"
}
if(input$selectSubject == "Trap Type"){
  subject <- "TrapType"
}
## This is important - when we want to plot catches by bait type remember
## the bait the animal ate was the previously set bait
if(input$selectSubject == "Bait"){
  subject <- "prevbait"
}

## Define the data as trapchecks where a species was caught
datcatch <- area.data()$datcatch

## Create a column in the data that refers to the currently set 'subject'
## for each trapcheck
datcatch$subject <- datcatch[[subject]]



## Select trapchecks only from the selected year (if necessary)
if(input$selectAggregate == "Month in a selected year"){
  datcatch <- datcatch[datcatch$calyear == input$selectYear, ]

  ## Return an error to the user if no catches made for the selected year
  if(is.na(datcatch[1, 1]) == TRUE){
    return("No data for the area in selected year")
  }
}
```

```
  ## Sort 'subject' in descending order of total catches
  sort.table <- sort(table(datcatch$subject), decreasing = TRUE)

  ## Find unique 'subjects' to list
  unique.subject <- names(sort.table)

  ## Scrolling list of 'subjects' - default option selects all
  ## selectize = FALSE, multiple = TRUE creates a scrolling list that
  ## allows multiple selections
  selectInput('subject',
              label = h5(paste("Select ", input$selectSubject, ":", sep = "")),
              choices = unique.subject,
              selected = unique.subject,
              multiple = TRUE,
              selectize = FALSE,
              size = 5)
})
```

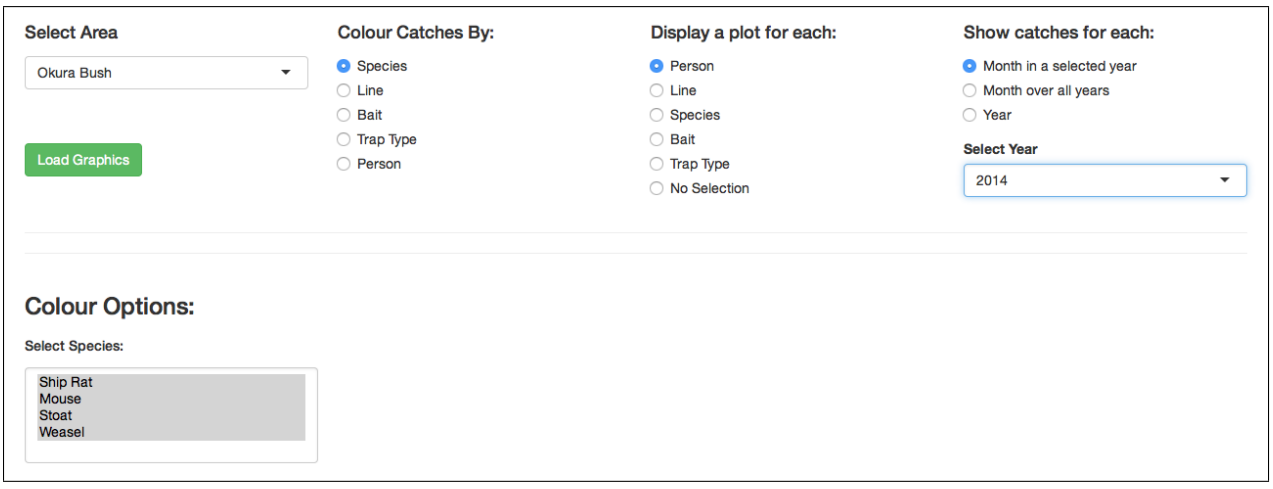


Figure 9. Colour options available when viewing data for Okura Bush from 2014

The available 'Specific plot options' are determined by a number of inputs. Because the 'Display a plot for each' input is set to 'Person', this means we wish to produce individual plots for each volunteer's catch data. Therefore, the application firstly determines each unique person who has

worked on the selected project, then filters out those who have not made any catches in the selected year. Then, based on the selected species under 'Colour Options', the application filters out any volunteers who have not caught any of these species. For example, when viewing data from 2015 for weasels only, we only have two available options, 'Jonathan' and 'Steele' as they were the only volunteers to have a recorded weasel catch.

```
## Ui code exerpt:
uiOutput("subby")


## Server code excerpt:
## Reactive list of 'Display a plot for each' options
output$subby <- renderUI({

  ## Define 'subject' and 'by'
  ## ubject is the 'Colour catches by' selection
  ## by.vec is the 'Display a plot for each' selection
  subject <- input$selectSubject
  by.vec <- input$selectBy

  ## Correct for if 'subject' == 'Person', 'Trap Type' or 'Bait'
  if(input$selectSubject == "Person"){
    subject <- "FirstName"
  }
  if(input$selectSubject == "Trap Type"){
    subject <- "TrapType"
  }
  if(input$selectSubject == "Bait"){
    subject <- "prevbait"
  }

  ## Define the data as trapchecks where a species was caught
  datcatch <- area.data()$datcatch
```

```
datcatch$subject <- datcatch[[subject]]

## Select trapchecks only from the selected year (if necessary)
if(input$selectAggregate == "Month in a selected year"){
  datcatch <- datcatch[datcatch$calyear == input$selectYear, ]

  ## Return an error to the user if no catches made for the selected year
  if(is.na(datcatch[1, 1]) == TRUE){
    return(NULL)
  }
}

## Extract input$selectBy and define in a new column in the data
datcatch$by <- datcatch[[by.vec]]
if(by.vec == "Person"){
  datcatch$by <- datcatch$alias
}
if(by.vec == "Trap Type"){
  datcatch$by <- datcatch$TrapType
}
if(by.vec == "Bait"){
  datcatch$by <- datcatch$prevbait
}

## Remove cases where the respective 'by' column has been left
## blank (e.g. no line or person entered in the data)
datcatch <- datcatch[datcatch$by != "", ]

## Create a dataframe only for the trapchecks specified in input$subject
unique.subject <- input$subject
store.vec <- data.frame()
```

```
for(i in 1:length(unique.subject)){
  datvec <- datcatch[datcatch$subject == unique.subject[i], ]
  store.vec <- rbind(store.vec, datvec)
}
datcatch <- store.vec

## Sort input$selectBy in descending order of catches
## This is done to ensure bar charts will plot selections in descending
## order of number of catches
sort.table <- table(datcatch$by)
sort.table <- sort(sort.table, decreasing = TRUE)

## List of unique 'by' options in descending order of catches
unique.by <- names(sort.table)

## Adopt plural form if input$selectBy == 'Person'
if(input$selectBy == "Person"){
  by.vec <- "People"
}

## Conditional scrolling list selection - only viewable when user selects
## 'List input$selectBy'
conditionalPanel(
  condition = "input.selectBy != 'No Selection' && input.by == 'List People'
    || input.by == 'List Lines' || input.by == 'List Species' ||
    input.by == 'List Baits' || input.by == 'List Trap Types'",
    selectInput("subby",
                label = h5(paste("Select subset of ", by.vec, ":",
                                 sep = "")),
                choices = unique.by,
                selected = unique.by,
```

```
                    multiple = TRUE,
                    selectize = FALSE,
                    size = 10)
    )
})
```

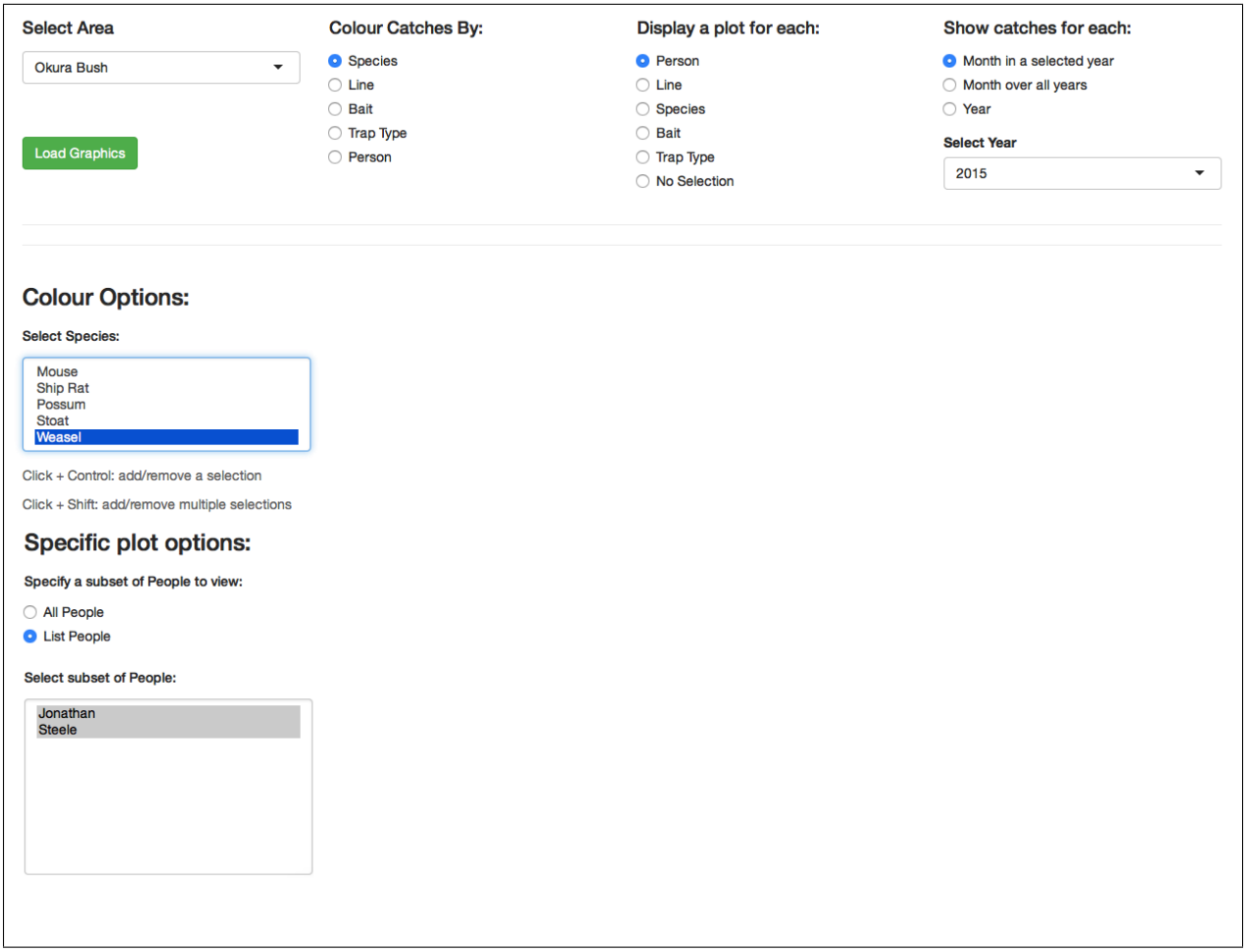


Figure 10. Specific plot options for volunteers who have caught weasels in 2015

Figures 9 and 10 give two clear examples of how complex coding in R Shiny can be. Both code chunks produce scrolling lists of selections that are dependent on previous inputs. The lists themselves are fairly uncomplicated, yet it has taken over 100 lines of R code to account for all of the possible permutations and hiccups we might run into. Admittedly, after programming in R Shiny for close to a year, I believe if the code were to be re-written I could produce the same results using far fewer lines. However, the point still remains that something as simple as a list still requires a fair amount of attention when it takes multiple dependencies.

Finally, once a user is satisfied with their selections, they can produce their chosen graphics by clicking the 'Load Graphics' button. Dependent on the size of the selected project, graphics can take up to 5 seconds to compute and display on screen. It is at this final stage of producing graphics that the functions `isolate` and `observeEvent`, outlined in Section 3.4.2, are of crucial importance. Without removing dependencies using `isolate`, the application would be continually producing and updating bar charts as we alter a single input. For example, the user might make five changes to the default options; without `isolate` this would result in four sets of bar charts being needlessly computed before the final portfolio of graphics the user wanted in the first place is produced. Instead, the user clicks the 'Load Graphics' button to tell the application when it should take the current selections and submit them to the graphics-producing code.

The final graphics displayed here show yearly catch numbers by volunteer for the Okura Bush project, with no individual plots to be displayed.

```
## Ui code excerpt:
plotOutput("overall"),
uiOutput("stackedplot")


## Server code excerpt:
## Upon clicking the 'Load Graphics' button the code that plots
## the overall graphic (1st plot) runs
observeEvent(input$submit, {
  output$overall <- renderPlot({
    ...
    ...
    ...
    < Ommitted 150 lines of code that takes the currently selected options >
    < and extracts the appropriate data to be plotted                      >

    ## Plot the barplot of selected options
    barplot(datTable, beside = TRUE,
            col = colour.diff[1:length(rownames(datTable))])
    ...
```

```
    ...
    ...
    < Omitted 20 lines of code that produce the appropriate titles and     >
    < legends for the current selections                                   >
  })
})

## Upon clicking the 'Load Graphics' button the code that plots the
## individual graphics run
observeEvent(input$submit, {
  output$stackedplot <- renderUI({
    ...
    ...
    ...
    < Omitted 240 lines of code that takes the currently selected options  >
    < and extracts the appropriate data to be plotted.                     >
    < The code also determines the number of individual bar charts to be   >
    < plotted, dependent on the number of 'Display a plot for each'        >
    < selections and begins a loop to plot the graphics                    >

    ## Plot the individual barplot for the ith 'Display a plot for each'
    ## selection option
    barplot(namesTables[[my_i]], col = colour.diff[1:length(input$subject)])
    ...
    ...
    ...
    < Omitted 70 lines of code that produce the appropriate titles and     >
    < legends for the current selections                                   >

  })
})
```

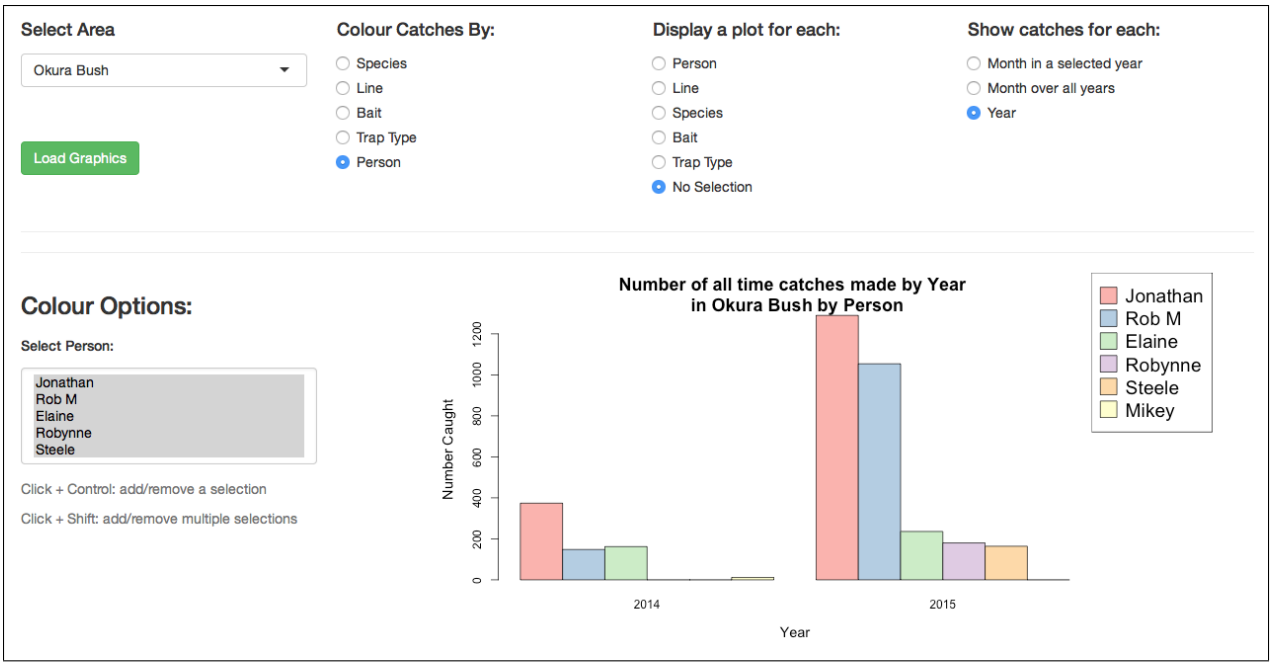


Figure 11. Plot of catch numbers for each year by volunteer for Okura Bush

The walk-through example gives a brief overview of how each of the selection options are interrelated, and it is evident that many inputs are dependent on selections of previous inputs. This goes to show that the code must be carefully structured such that any given input can take many different values and still produce the appropriate graphics. It is here that we can begin to appreciate the flexibility we are afforded by having all data in the same CatchIT format.

4.2 Solutions to technical challenges

Section 5.1 outlines the basic structure of how the various inputs of the bar chart application fit together. However, there are many small details that go unnoticed, but if left unattended, would result in clear instances of discontinuity in the application.

4.2.1 Producing the individual bar charts

A keen reader may have noticed in the code chunk relating to *Figure 11*, the first plot is produced in the UI output using `renderPlot`, whereas the individual bar charts use `uiOutput`. Regardless of the users' selections, the first plot that details the project's overall catch data is always plotted. The `renderPlot` function is used to create a plotting space for this one bar chart.

However, the number of individual bar charts to be plotted varies according to the number of selections made under 'Specific plot options'. Therefore the function `renderPlot` cannot be

used, as this only creates a plotting space in the output for a single bar chart. Piecing together code found on the internet (http://stackoverflow.com/questions/15875786/dynamically-add-plots-to-web-page-using-shiny) with my own code, the best solution was to use the more flexible `renderUI` function. This allows the application to automatically determine the number of individual bar charts to be plotted and then, in turn, to create the correct number of plotting spaces using `renderUI`.

```
## Server code excerpt:
## Upon clicking the 'Load Graphics' button the code that plots the
## individual graphics runs
observeEvent(input$submit, {
  output$stackedplot <- renderUI({
    ...
    ...
    ...
    < Omitted 140 lines of code that takes the currently selected options  >
    < and extracts the appropriate data to be plotted.                     >

    ## Set plotlength for right number of outputs
    ## (# of items selected under 'Specific plot options')
    plotlength <- length(input$subby)

    ## Creates a list of outputs to be filled with plots
    output$stackedplot <- renderUI({

      ## Give each plot a unique name to refer to in the output
      plot_output_list <- lapply(1:plotlength, function(i) {
        plotname <- paste("plot", i, sep="")
        plotOutput(plotname)
      })

      ## Convert the list to a tagList - this is necessary for the list of
```

```
  ## items to display properly (StackOverflow)
  do.call(tagList, plot_output_list)
})

## Call renderPlot for each individual plot and fill each slot in the
## tagList with the appropriate plot

for (i in 1:plotlength){
## Need a local environment so that each item gets its own number.
## Without it, the value of i in the renderPlot() will be the same
## across all instances, because of how the expression is evaluated
## (StackOverflow)
local({
  my_i <- i

  ## Fill the appropriate output in output£stackedplot
  plotname <- paste("plot", my_i, sep = "")

  ## Call each plot
  output[[plotname]] <- renderPlot({
    ...
    ...
    ...
    < Omitted 20 lines of code automating the plot colours and legend >
    < font size                                                      >

    ## Plot the individual barplot for the ith 'Display a plot for each'
    ## selection option
    barplot(namesTables[[my_i]], col = colour.diff[1:length(input$subject)])
    ...
    ...
```

```
        ...
        < Omitted 70 lines of code that produce the appropriate titles and     >
        < legends for the current selections                                   >
      })
    })
  }
})
```

4.2.2 Clarity and continuity

In order to maintain clarity with regard to what is currently plotted on a user's screen, the application clears all graphics as soon any input is changed. Again, using the `observeEvent` function, whenever the application registers the event of any input change, the plotting area is re-written to be blank. A loading widget will appear and the screen will grey for the duration it takes the application to recompute the selection options that should be available due to the input change.

```
## Ui code excerpt:
## Loading widget
busyIndicator("Please Wait", wait = 0)


## Server code excerpt:
## Load blank plots when area is changed
observeEvent(input$selectArea, {
  ## Clears the first (overall) plot
  output$overall <- renderPlot({
    return(NULL)
  })
  ## Clears the 'Display a plot for each' plots
  output$stackedplot <- renderPlot({
    return(NULL)
  })
})
```

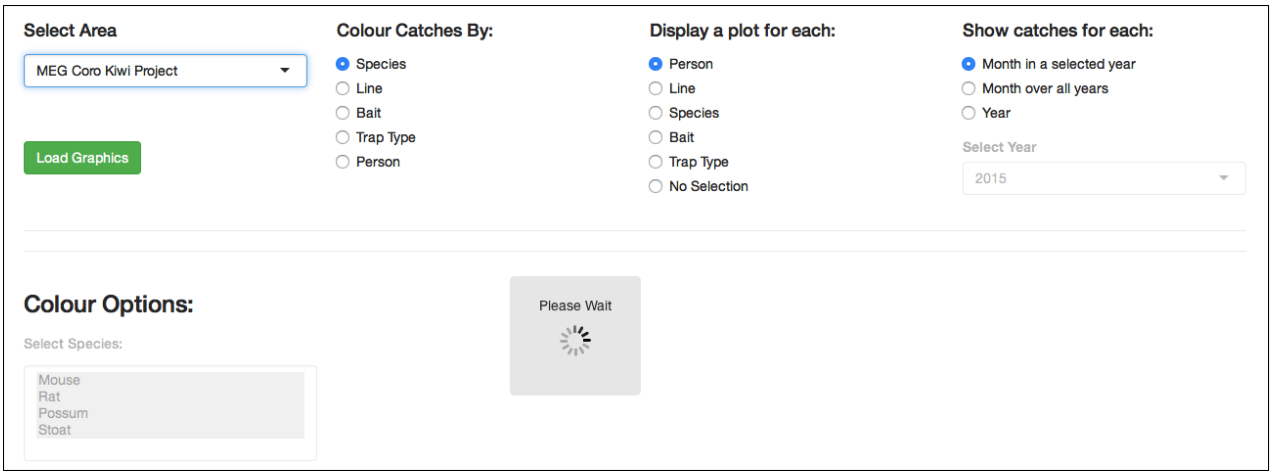


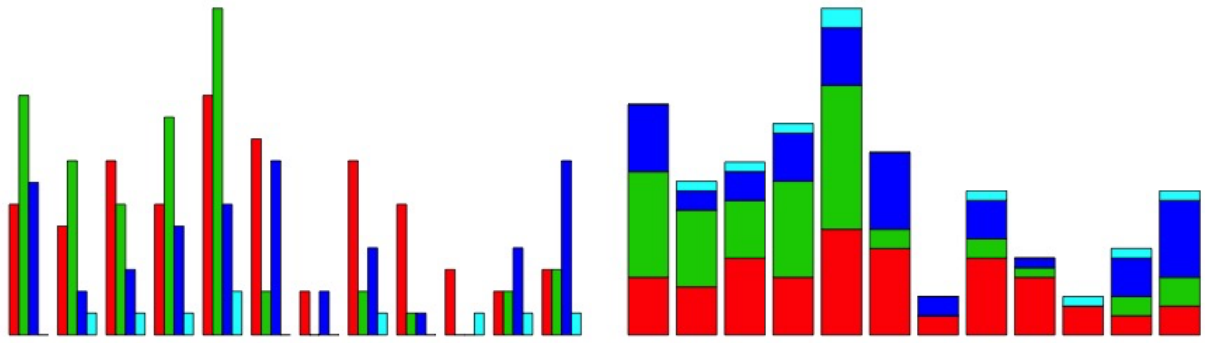
Figure 12. Example of loading widget and plots being cleared as area changes from Kaipupu to MEG Coro Kiwi Project

This ensures that if any graphics are on the screen, they will match the currently selected options exactly, limiting any confusion users might have regarding what they are viewing.

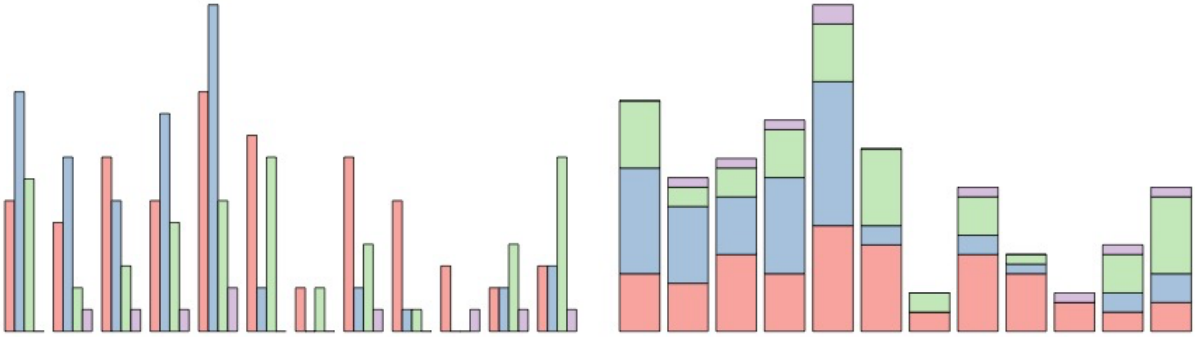
Another continuity issue, which was never entirely fixed, was the problem outlined in Section 3.4.1 where the non-linear structure of R Shiny code can cause the title of a plot to update before the data. The cause of this was never fully resolved given the time constraints and it appears to be a very minor issue, with the incorrect title lingering no more than a split second.

4.2.3 Automating colour selection and legends

The colouring of the plots is another area that required more thought than than one might think. Ihaka (2003) outlines sensible guidelines with regards to colour selection in presentation graphics. These recommendations include (i) avoiding large areas of high-chroma (bright) colours, as these can produce distracting after-image effects; and (ii) using colours that are easy to distinguish when colours are used to indicate group membership. Brewer (1999) recommends that, when each category is equally important, a qualitative colour scheme should be used, such that each colour should have a similar contrast. Using the R package `RColorBrewer`, which provides colour schemes designed by Brewer (1999), a selection of nine pastel colours was chosen to best distinguish between groups.



(a) Default R colours



(b) RColourBrewer colours

Figure 13. Comparison of default R colours and RBrewerPal colours

In cases where the number of items to be plotted exceeds nine, the application automatically interpolates between the nine base colours to produce the required number of colours:

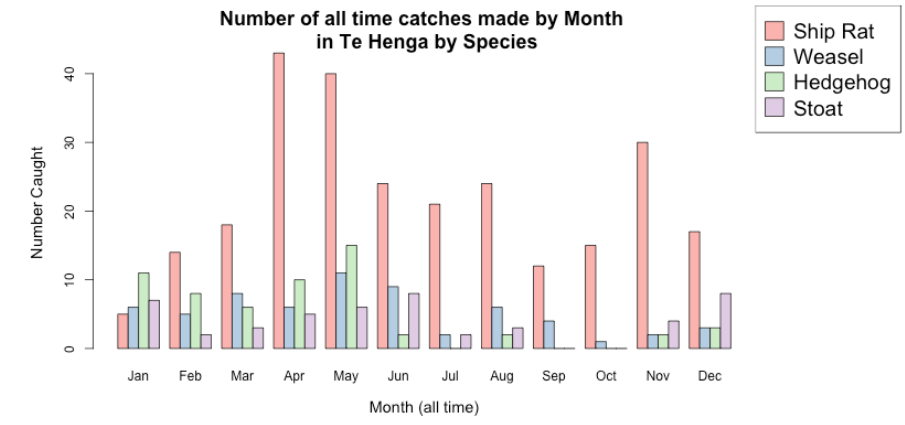
```
## Define the 9 base colours
colour.diff <- brewer.pal(9, "Pastel1")


## Add colours if need more than 9
if(length(legend.string) > 9){
  colour.diff <- colorRampPalette(brewer.pal(brewer.pal.info["Pastel1",
                                             "maxcolors"], "Pastel1"))
}
```

These colours are still easily distinguishable from one another, although in cases where the number of items to be plotted is very high, it is unavoidable to have some colours that are not

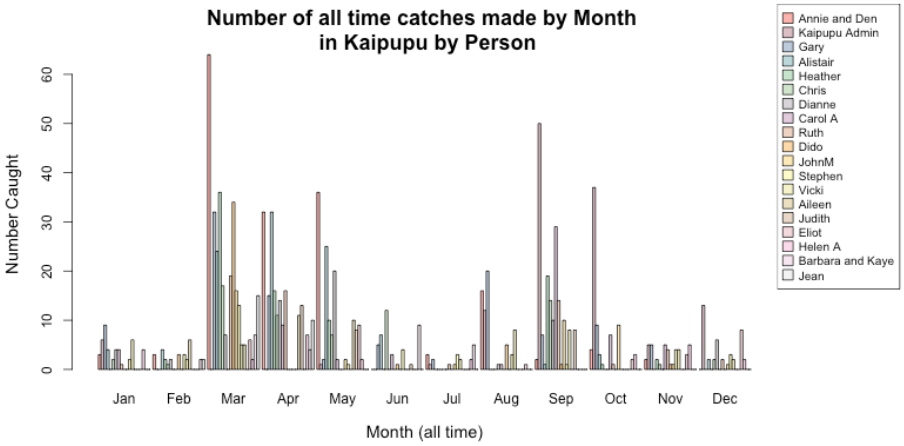
dissimilar.

Another important, though seemingly innocuous piece of programming lies within the coding of the legends. While coding a legend in R is usually a straightforward exercise, it is slightly more difficult when automating the process. Firstly, it is important to determine the font size of the legend based on the number of items to be listed. This ensures when a large number of items or a particularly long item are to be plotted, the legend does not cover the entire graphic. Conversely, when only a few items are plotted, this ensures the legend is a reasonable size, and not too small to read.

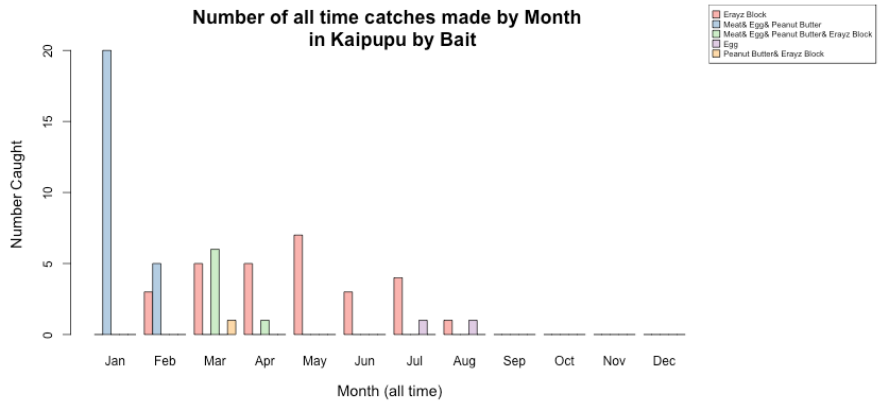
```
## Server code excerpt:
## Legend size automation
min.cex <- 0.8
max.cex <- 2

## Number of items to be plotted
legend.string <- c(input$subject)

## Determine the font size dependent on the number and length of the items
## to be plotted
auto.cex <- max(min.cex, min(c(20/length(legend.string)),
                             20/nchar(legend.string)))
auto.cex <- min(auto.cex, max.cex)

## Plot the legend
legend("topleft", legend = legend.string, col = 1, pt.bg =
       colour.diff[1:length(input$subject)], pch = 22, cex = auto.cex*0.8,
       pt.cex = auto.cex*1.75, xpd = TRUE)
```



(a) A legend containing a small number of items



(b) A legend containing a large number of items



(c) A legend containing a long item

Figure 14. Plots illustrating the automatic selection of colour and font size

Figure 14 clearly illustrates how the application automatically determines appropriate colours and font sizes for any selection of inputs. This goes to show how two usually simple exercises in plotting colours and legends, become much more complex when automating the process.

5 Te Henga bait trial

In February 2015, the co-coordinators of the Te Henga conservation project approached the University of Auckland for advice on conducting a bait comparison trial. Their question of interest was whether the bait currently funded by the Department of Conservation, Salted Rabbit, was any more or less alluring than an alternative bait, Erayze - a commercial dried rabbit product. This trial provided an opportunity to explore how the bar chart application might be used in a more traditionally statistical manner.

The Te Henga project is situated on the west coast of Auckland, New Zealand, and consists of two traplines that are checked fortnightly. The first line, 'Wainamu', consists of 54 traps around Lake Wainamu, while the second line, 'Wetlands', consists of 35 traps in the Te Henga wetlands, alongside Bethell's Beach. As of late July 2014, traps have been baited each fortnight with either Salted Rabbit or Erayze. Traps with an even numbered suffix were baited with Salted Rabbit and those with an odd numbered suffix with Erayze. This gave a total of 44 traps baited with Salted Rabbit and 45 traps baited with Erayze. It must also be noted there was a week in August 2015 where all traps were baited with Erayze due to a late delivery of Salted Rabbit.

Under the null hypothesis we assume that a trap baited with Salted Rabbit has equal probability of making a catch compared with a trap baited with Erayze. A two-tailed t-test between two different independent samples was used to compare the proportion of capture occasions that had made successful catches between baits (Wild and Seber, 2000).

p_{sr} = the probability of making a catch per capture occasion using a trap baited with Salted Rabbit

$\hat{p}_{sr}$ = the observed proportion of traps baited with Salted Rabbit that did make a catch

p_e = the probability of making a catch per capture occasion using a trap baited with Erayze

$\hat{p}_e$ = the observed proportion of traps baited with Erayze that did make a catch

n_{sr} = the number of traps baited with Salted Rabbit

n_e = the number of traps baited with Erayze

$H_0 : p_{sr} - p_e = 0$

Under $H_0 : \frac{\hat{p}_{sr} - \hat{p}_e}{S.E.(\hat{p}_{sr} - \hat{p}_e)} \sim t_{df}$

Where: df = min(n_{sr} - 1, n_e - 1); $S.E.(\hat{p}_{sr} - \hat{p}_e) = \sqrt{\frac{\hat{p}_{sr}(1-\hat{p}_{sr})}{n_{sr}} + \frac{\hat{p}_e(1-\hat{p}_e)}{n_e}}$

This test allows us to calculate a p-value to determine whether there is any evidence to suggest a bait has a significantly higher or lower probability of making a catch, compared with the other

bait. Our assumptions of the trial outcomes being approximately normally distributed is fulfilled as the number of trials was large ($n_{sr} = 1355$ traps baited with Salted Rabbit; $n_e = 1442$ baited with Erazye).

The following report has been sent to the Te Henga project and summarises the findings of the data obtained in the last 15 months. Bearing in mind that the co-coordinators of conservation projects affiliated with CatchIT vary in statistical knowledge, reports are written using easy to understand language, clearly explaining any statistical terms used.

Reports typically begin with easy-to-understand bar charts that give a clear picture of the data. The more traditional statistical analyses involving p-values are left to the technical analysis section

5.1 Te Henga report

Explanatory bar charts (Data as of 3rd October 2015)

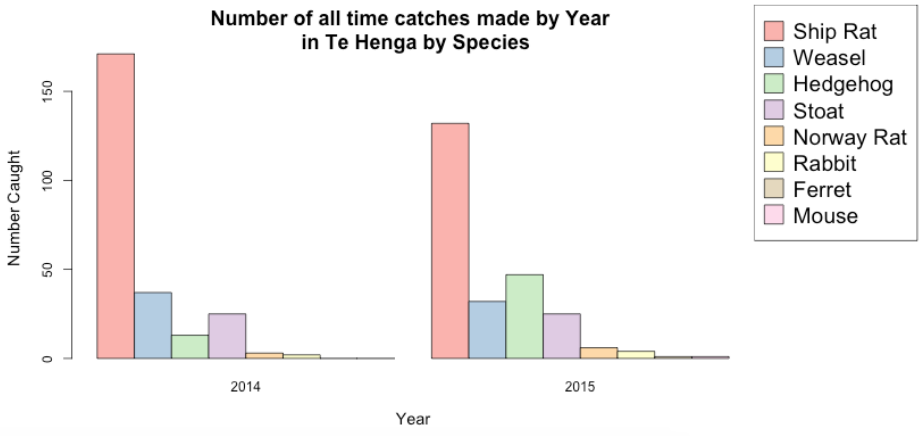


Figure 15. Bar chart of all catches for all species displayed by year.

Figure 15 shows species displayed by number of catches in descending order from left to right within each year, i.e. ship rats are the most commonly caught species and Mice are the least commonly caught species.

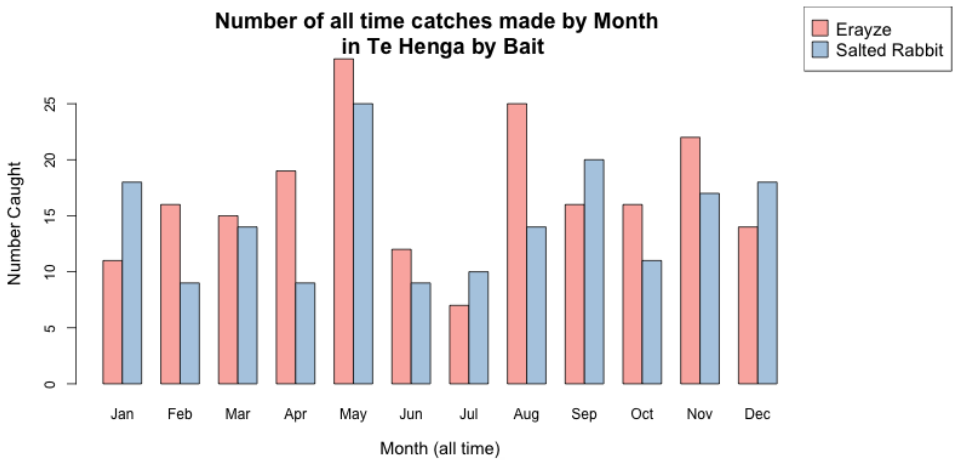


Figure 16. Bar chart of all catches using Erayze and Salted Rabbit.

Figure 16 displays the number of catches made each month using both Erayze and Salted Rabbit. Traps baited with Erayze have made slightly more catches overall than traps baited with Salted Rabbit since the start of the trial.

Individuals of all species caught using Erayze = 209

Individuals of all species caught using Salted Rabbit = 182

Rats

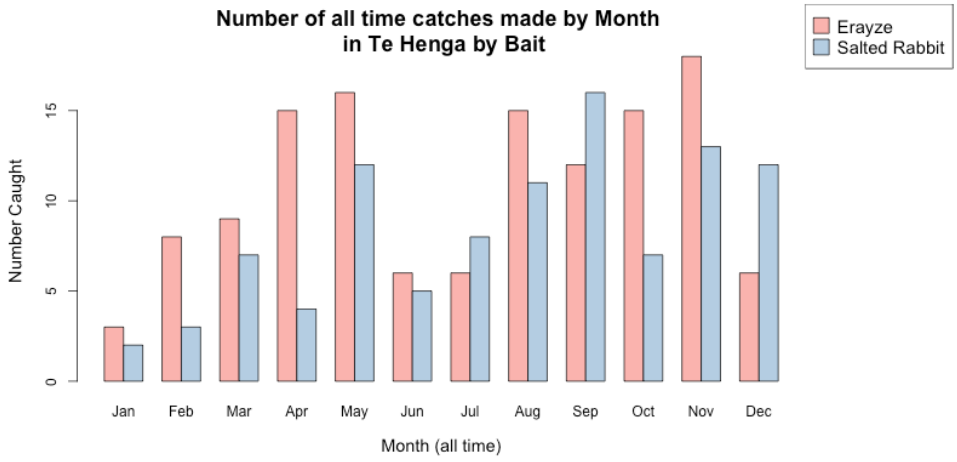


Figure 17. Bar chart of overall bait catch numbers for rats (ship rats & Norway rats).

Figure 17 displays the number of rats caught each month using Erayze and Salted Rabbit. Most months appear to have more catches made using Erayze (red) than Salted Rabbit (blue).

Rats caught using Erayze = 136

Rats caught using Salted Rabbit = 105

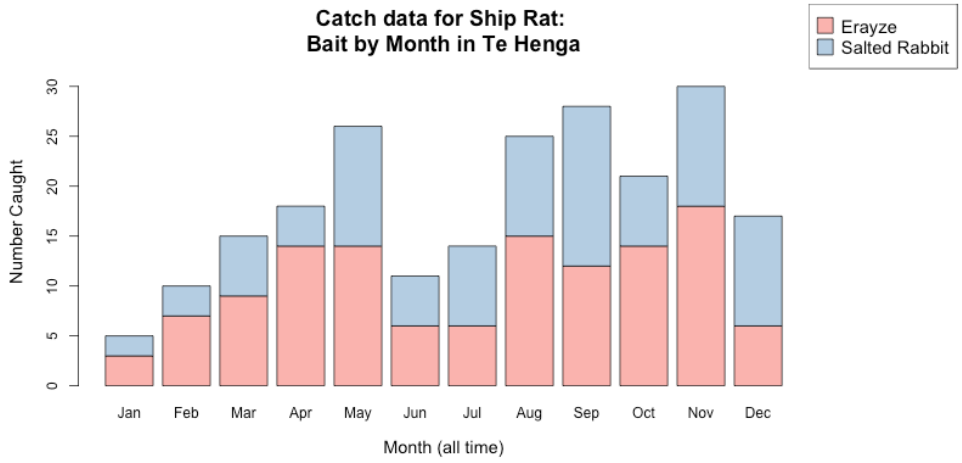


Figure 18. Bar chart of bait catch numbers for ship rats.

Figure 18 displaying the number of ship rats caught each month using Erayze and Salted Rabbit. Most months appear to have more catches made using Erayze than Salted Rabbit.

Ship rats caught using Erayze = 131

Ship rats caught using Salted Rabbit = 101

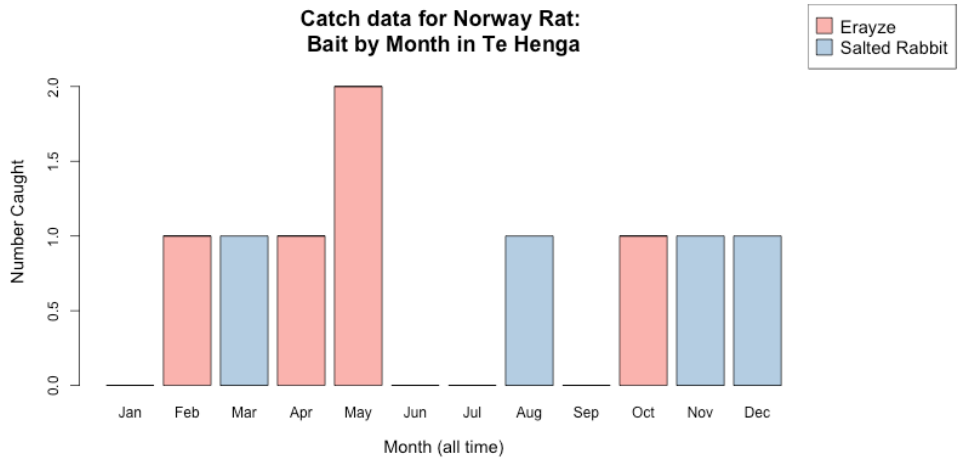


Figure 19. Bar chart of bait catch numbers for Norway rats.

Figure 19 displays the number of Norway rats caught each month using Erayze and Salted Rabbit.

There appears to be similar numbers of catches made using each type of bait, given not many Norway rats have been caught.

Norway rats caught using Erayze = 5
Norway rats caught using Salted Rabbit = 4

Mustelids

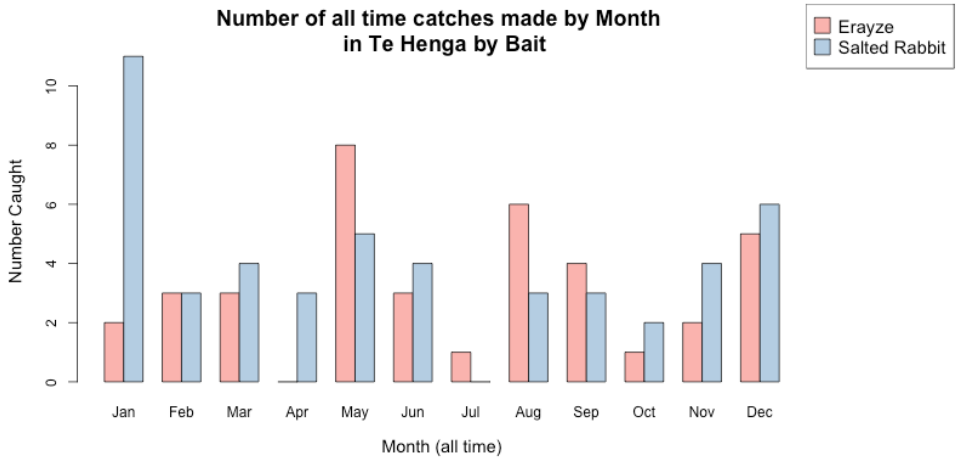


Figure 20. Bar chart of bait catch numbers for mustelids (stoats, weasels and ferrets).

Figure 20 displays the number of mustelids caught each month using Erayze and Salted Rabbit. There appears to be similar numbers of mustelids caught by Erayze and Salted Rabbit in most months, although far more have been caught using Salted Rabbit in January.

Mustelids caught using Erayze = 39
Mustelids caught using Salted Rabbit = 50

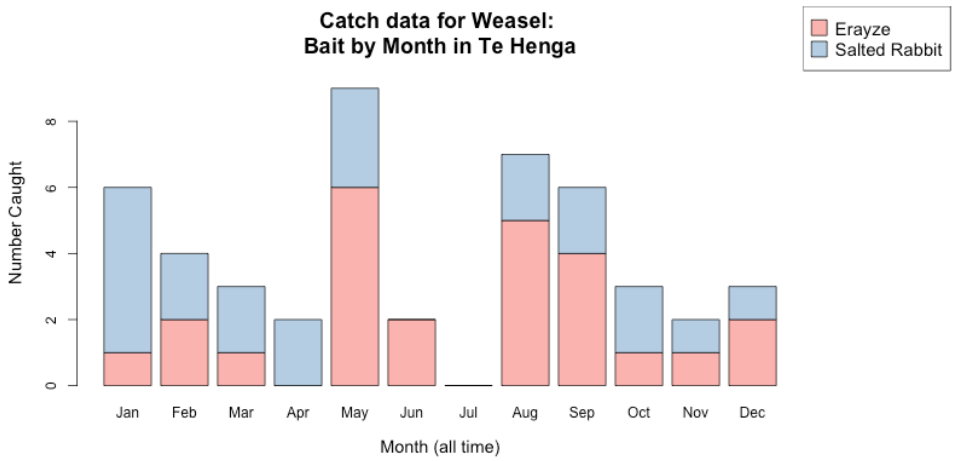


Figure 21. Bar chart of bait catch numbers for weasels.

Figure 21 displays the number of weasels caught each month using Erayze and Salted Rabbit. There appears to be slightly more catches made using Erayze than Salted Rabbit.

Weasels caught using Erayze = 26

Weasels caught using Salted Rabbit = 22

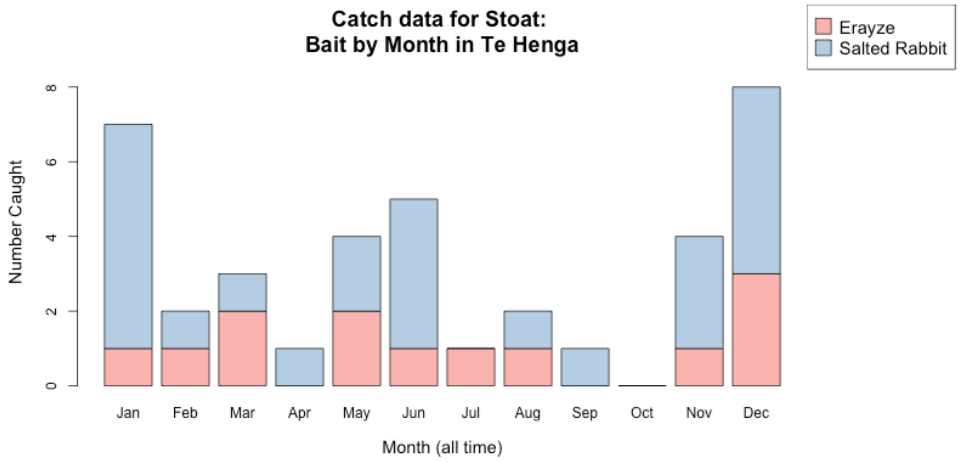


Figure 22. Bar chart of bait catch numbers for stoats.

Figure 22 displays the number of stoats caught each month using Erayze and Salted Rabbit. Unlike weasels, more stoats appear to have been caught using Salted Rabbit than Erazye.

Stoats caught using Erayze = 13

Stoats caught using Salted Rabbit = 27

No individual bar chart for ferrets has been provided, as only one ferret has been caught (using Salted Rabbit).

Hedgehogs

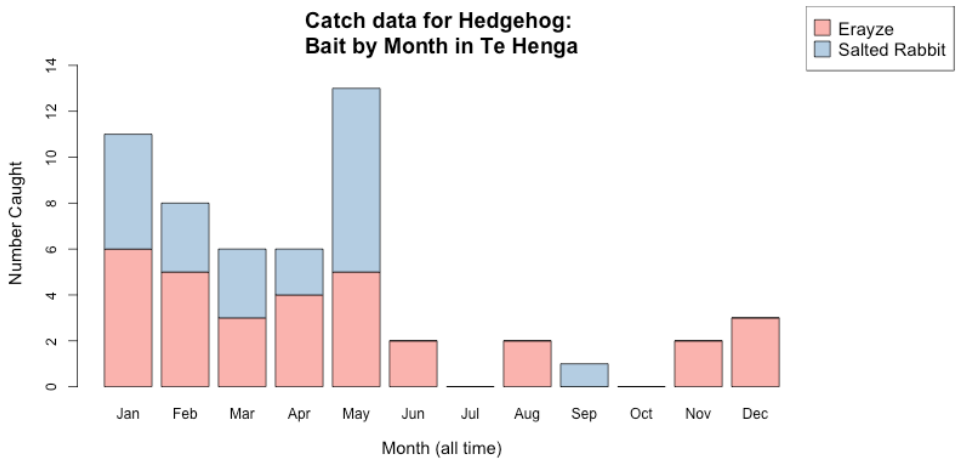


Figure 23. Bar chart of bait catch numbers for hedgehogs.

Figure 23 displays displaying the number of hedgehogs caught each month using Erayze and Salted Rabbit. There appears to be more catches made using Erayze than Salted Rabbit.

Hedgehogs caught using Erayze = 32

Hedgehogs caught using Salted Rabbit = 22

Technical Analysis

The above graphics give a rough indication of which bait (if any) each species might prefer, however some simple statistical tests will give a better idea of whether these preferences are statistically significant or have occurred due to random chance. A significant result is a statistician's way of saying we have some concrete evidence beyond ordinary chance.

In this experiment we have begun by assuming Erayze and Salted Rabbit should catch close to equal numbers of individuals. For each species we can perform a test on the difference between the observed number of catches made using Erayze and Salted Rabbit. This test will tell us whether the difference is plausibly simply due to chance, or if the difference is large enough to be considered such an unlikely observation that we might suspect a species prefers a particular bait type. In such

a case we have evidence against the original assumption that catches are equally apportioned to the two baits.

To determine how unlikely an observation is, we calculate a probability value (p-value) for each species. This value represents the probability of a discrepancy between baits as large as the one that we have observed under the the hypothesis of equal bait preference. Traditionally, a p-value of 0.05 (5%) is considered to be an unlikely enough observation to give us evidence to suspect that the baits differ in their attractiveness for that species.

Overall

Individuals of all species caught using Erayze = 209

Individuals of all species caught using Salted Rabbit = 182

Total individuals of all species caught = 391

p-value = 0.418

While more individuals of all species were caught using Erayze, we must take into account the fact that slightly more traps were set with Erayze compared with Salted Rabbit (1442 vs. 1355). The proportion of traps baited with Erayze that made a catch was $\frac{209}{1442} = 0.145$, or 14.5%. For Salted Rabbit the proportion was $\frac{182}{1355} = 0.134$, or 13.4%, which is fairly similar to the Erayze proportion.

The p-value associated with the overall data indicates that the probability of observing a bait-wise discrepancy as large as this for individuals of all species is 41.8% under the hypothesis of equal bait preference. This is a reasonably likely event and gives us no evidence to suggest Salted Rabbit catches fewer individuals of all species than Erayze. However, we can look at the data more closely to determine if there are any species-specific differences.

Rats

Rats caught using Erayze = 136

Rats caught using Salted Rabbit = 105

Total rats caught = 241

p-value = 0.112

The p-value indicates that the probability of observing a bait-wise discrepancy as large as this

for rats is only 11.2% under the hypothesis of equal bait preference. This is not a particularly likely observation, however it is not unlikely enough to provide evidence that Salted Rabbit does not catch as many rats as Erayze. If more data are collected, it may become more clear as to whether there is a real difference between bait catch effectiveness for rats.

Mustelids

Mustelids caught using Erayze = 39

Mustelids caught using Salted Rabbit = 50

Total mustelids caught = 89

p-value = 0.140

The p-value signifies that the probability of observing a bait-wise discrepancy as large as this for mustelids is only 14.0% under the hypothesis of equal bait preference. Again, this is not a particularly likely observation, however it does not provide any conclusive evidence to suggest Salted Rabbit catches more mustelids than Erayze. While there does not appear to be a significant difference in catch rates between baits for mustelids as a whole, we can look more closely at the individual catch data.

Weasels

Weasels caught using Erayze = 26

Weasels caught using Salted Rabbit = 22

Total Weasels caught = 48

p-value = 0.715

Unlike the overall mustelid data, which had more Salted Rabbit catches, weasels have more Erayze catches. However, the p-value of 0.715 means that the probability of observing a bait-wise discrepancy as large as this for weasels is 71.5%. This is a likely event that gives no evidence to suggest a difference in the weasel catch rates between the two baits.

Stoats

Stoats caught using Erayze = 13

Stoats caught using Salted Rabbit = 27

Total Stoats caught = 40

p-value = 0.016

Over twice the number of stoats have been caught using Salted Rabbit, despite more traps being set with Erayze. The p-value of 0.016 means that the probability of observing a bait-wise discrepancy as large as this for stoats is 1.6% under the hypothesis of equal bait preference. This is an unlikely event and does provide strong evidence against our assumption of equal catch rates. Therefore, there is evidence to suggest Erayze might not catch as many stoats as Salted Rabbit. Whether this difference is due to an avoidance of Erayze or an attraction to Salted Rabbit cannot be determined, but suggests Salted Rabbit is more effective nonetheless.

Ferrets

As only one ferret has been caught thus far, we are unable to perform any meaningful tests on this species.

Hedgehogs

Hedgehogs caught using Erayze = 32

Hedgehogs caught using Salted Rabbit = 22

Total hedgehogs caught = 54

p-value = 0.251

More hedgehogs have been caught using Erayze, however the p-value indicates that the probability of observing 22 or fewer catches (of the 54) using Salted Rabbit is 25.1%. This is a somewhat likely event and provides us with no evidence to suggest Salted Rabbit does not catch as many hedgehogs as Erayze.

5.2 Summary of findings

The conclusions are somewhat interesting, as there is only statistically significant evidence at the 5% level to conclude that Salted Rabbit might catch more stoats compared with Erayze. However, overall more individual pests have been caught using Erayze, although this may be due to the fact there were slightly more traps baited with Erayze. The results suggest that any advantage of using Erayze over Salted Rabbit is as yet undemonstrated, although the observed distribution of catches for rats are approaching statistical significance. It appears as though if there is any advantage, it is species specific and as such it would be wrong to declare that one bait is better than the other in all facets. Instead, baits could be used together to target particular species, such as using Erayze for rats, and Salted Rabbit for stoats.

It is also worth noting that due to time constraints, the calculated p-values have not accounted for the fact we have conducted multiple hypothesis tests. The more hypothesis tests we conduct, the more likely we are to find a statistically significant result somewhere. However, the issues that arise with multiple comparisons are lessened as we continue to collect data over time. This is because we are able to record independent future observations, to see if a previously observed effect has persisted. Clearly we do not yet have enough data to do this for the Te Henga trial but the point remains that within the wider scope of CatchIT, multiple comparisons might not be such a problem.

Using the bar chart application we are able to quickly and easily produce visual summaries of the data. Even to an untrained statistical eye, the bar charts at a glance give a relatively good idea as to whether an overwhelming number of a species was caught by one particular bait. While in this case it was explicitly asked that the relationship between Salted Rabbit and Erayze be investigated, this report goes to show that visually inspecting data can quickly give a reasonably good idea of whether a true statistical trend is present. As statisticians, we neither have the time, nor frankly the interest, to spend all day looking for statistical patterns in bar charts for every different permutation of data available to us. However, by affording conservation volunteers the opportunity to explore their data via the bar chart application, they themselves may come across particularly intriguing patterns in the data that might warrant further investigation.

6 Heat map application

The heat map application is the second application I developed using R Shiny. It allows users to see on a Google map where high densities of animal catches are being made for a selected conservation project. This can be useful for gaining an understanding of the geographical layout of a project and where the traps are located. Additionally, it helps provide a rough idea of which areas of traps are making lots of catches, such as: traps found near water, or traps found at higher altitudes. The user is able to choose to view data for specific species, specific volunteers and specific traplines, as well as choose from a selection of viewing options.

The application produces two maps (i) a 'heat map' on the left, which represents high density catch areas using a smooth overlay and (ii) a 'catch map' which has a bubble representing each trap. The heat map displays the number of catches made in an area using a heat scale, ranging from a transparent light yellow (very few catches) to dark red (many catches). For the catch map, the size of the bubble is related to the number of catches made by that trap. These maps work well in combination to inform the user where high numbers of species are being caught. The default setting will produce maps for all species, all volunteers and all traplines.

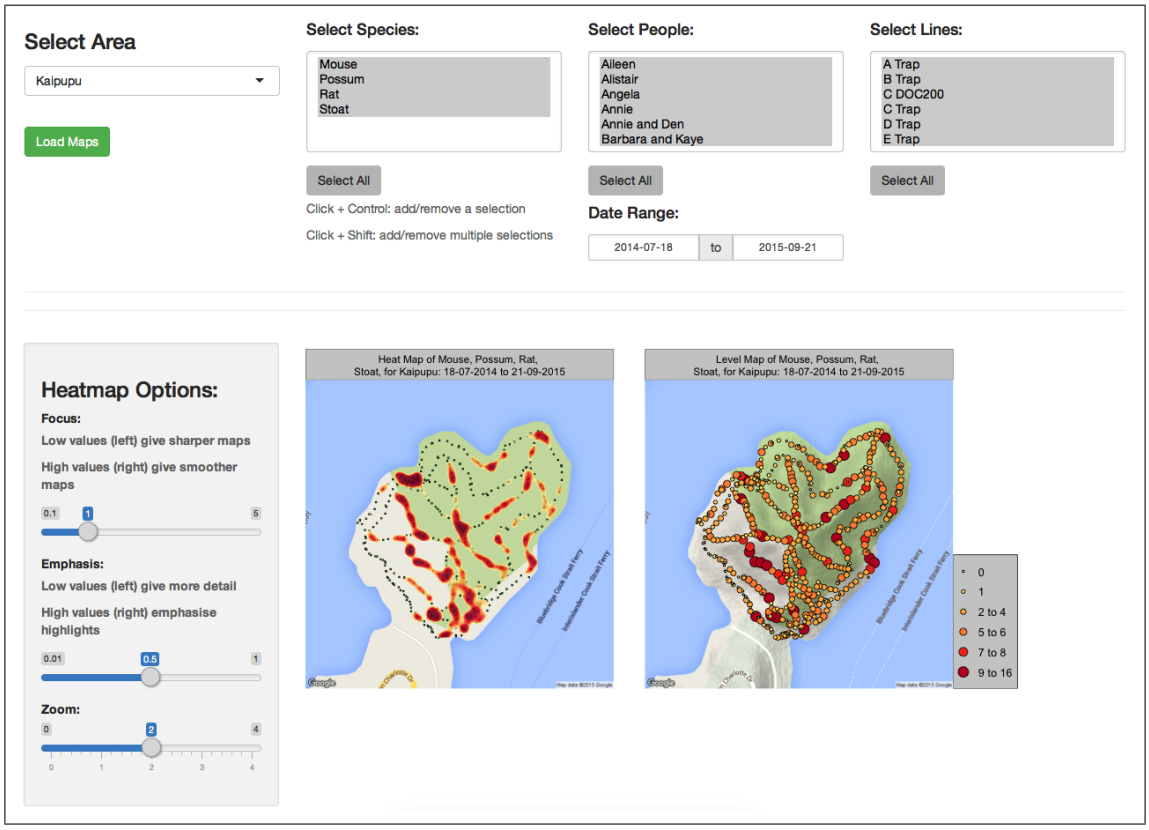


Figure 24. Example heat map application interface

The interface in *Figure 24* gives an example of the output a user would be presented with using the default settings for the Kaipupu project. Like the bar chart application, an overview of the application is provided below in Section 6.1. Code excerpts are provided to detail the interdependencies between inputs within the interface.

6.1 Overview and walk-through

The application provides a user with six main selection options, namely the:

1. Area (project)
2. Species
3. Individual volunteers
4. Traplines
5. Date range
6. Various viewing options (addressed in Section 6.2)

Like the bar chart application, the interdependencies between inputs are fairly complex. A dynamic UI has again been implemented using `renderUI` to allow the inputs to react according to the selected area. The user begins by selecting the area they wish to view data for, and the application will upload the relevant master CSV file. A reactive function then runs, determining what the appropriate selection options should be, and returns these options within the UI in the form of scrolling lists.

```
## Server code excerpt:
## Reactive function of the data
ui.data <- reactive({

  ## Update when area is changed
  input$selectArea
  isolate({
    ## Area name.csv
    csvname <- paste(input$selectArea, ".csv", sep = "")
```

```
    ## Read the all-time data
    dat <- read.csv(csvname, as.is = TRUE)

    ## Select cases where a species was caught
    datcatch <- dat[dat$Species != "", ]

    ## Extract unique species, lines and traps
    unique.species <- unique(datcatch$Species)
    unique.lines <- unique(datcatch$Line)
    unique.traps <- unique(datcatch$TrapName)

    ## Determining duplicate volunteer names
    ## (only display first names to protect privacy)
    alias <- trim.func(unique.names)

    ## Determine which (if any) are duplicate First Names
    which.dups <- which(duplicated(alias) | rev(duplicated(rev(alias))))
    ...
    ...
    ...
    < Omitted 10 lines of code determining whether any volunteers have duplicate >
    < first names, and if so, distinguishing them by adding their usernames      >

    ## List of key information to be used when determining selection options
    ## within the UI
    list(dat = dat, datcatch = datcatch, unique.species = unique.species,
         unique.names = unique.names, unique.lines = unique.lines, unique.traps
         = unique.traps)
  })
})
```

Using the information stored in the reactive list of this function, the application identifies the appropriate species, volunteers and traplines specific to the selected project and displays them as scrolling lists. The dates of the first and most recent observations are also extracted, allowing the user to view data for a particular range of dates.

It might be noted in the following code excerpt that fewer lines of code are used to produce a list, compared with code that produces lists for the bar chart application. This was my second attempt at building an R Shiny application and as a result I had a much better idea of how reactive objects interact with each other and was able to make the code more clear and concise. However, the scrolling lists in the heat map application are less complex than those in the bar chart application as they only take a single dependency—the selected area. The lists in the bar chart application take multiple dependencies on the selections of other lists, which explains some of the additional code required.

```
## Ui code excerpt:
column(3,
  ## Select area dropdown menu
  selectInput("selectArea",
    label = h3("Select Area"),
    choices = c("Kaipupu", "Mataia", "MEG Coro Kiwi Project", "Okura Bush",
                "Te Henga"),
    selected = "Kaipupu"
  ),
  br(),
  ## Load Maps button
  actionButton("submit", label = "Load Maps", styleclass = "success")
),
column(3,
       ## Species scrolling list
       uiOutput("selectSpecies"),
       ## 'Select All' action button
       actionButton("allSpecies", label = "Select All"),
       helpText("Click + Control: add/remove a selection"),
```

```
        helpText("Click + Shift: add/remove multiple selections")
),
column(3,
        ## People scrolling list
        uiOutput("names"),
        ## 'Select All' action button
        actionButton("allNames", label = "Select All"),
        ## Dates selection box
        uiOutput("dates")
    ),
column(3,
        ## Traplines scrolling list
        uiOutput("subLines"),
        ## 'Select All' action button
        actionButton("allLines", label = "Select All")
)



## Server code excerpt:
##### Selection Options #####
## Date range box
output$dates <- renderUI({
  ## Load the data by extracting it from the ui.data list
  dat <- ui.data()$dat

  ## Ensure "/" are "-" in .csv file
  dat.ft <- ifelse(length(grep("/", dat$Date[1])) > 0, "%d/%m/%Y", "%d-%m-%Y")

  ## Make dat£Date into "Date" class
  dat$Date <- as.Date(dat$Date, format = dat.ft)
```

```
  ## Define first and last dates
  first <- min(dat$Date)
  last <- max(dat$Date)

  dateRangeInput("dates",
                 label = h4("Date Range:"),
                 start = first,
                 end = last)
})

## Select species scrolling list
output$selectSpecies <- renderUI({
  selectInput("selectSpecies",
              label = h4("Select Species:"),
              choices = sort(c(ui.data()$unique.species)),
              selected = c(ui.data()$unique.species),
              multiple = TRUE,
              selectize = FALSE,
              size = 6
            )
})

## Select people scrolling list
output$names <- renderUI({
  selectInput("names",
              label = h4("Select People:"),
              choices = sort(c(ui.data()$unique.names)),
              selected = c(ui.data()$unique.names),
              multiple = TRUE,
              selectize = FALSE,
              size = 6
```

```
            )
})


## Select traplines scrolling list
output$subLines <- renderUI({
  selectInput("subLines",
              label = h4("Select Lines:"),
              choices = sort(c(ui.data()$unique.lines)),
              selected = c(ui.data()$unique.lines),
              multiple = TRUE,
              selectize = FALSE,
              size = 6
             )
})
```

Once the user is satisfied with their selections, it is as simple as clicking the 'Load Maps' button to produce the graphics. The application then runs a second reactive function that extracts the 'working data': the data that are relevant to the user's selections. The benefit of this is that the relevant data to be displayed graphically are extracted just once and can be simply plugged into the plotting code. This is more efficient than the bar chart application which extracts the relevant data twice, once for the overall plot and once for the individual plots.

```
## Server code excerpt:
## Reactive function of the working data
working.data <- reactive({

  ## Update when load map is clicked
  input$submit

  ## Isolate so only updates when submit is clicked
  isolate({
```

```
## Load the data
dat <- ui.data()$dat

## Select appropriate date range
dat.ft <- ifelse(length(grep("/", dat$Date[1])) > 0, "%d/%m/%Y", "%d-%m-%Y")
dat$Date <- as.Date(dat$Date, format = dat.ft)
dat <- dat[dat$Date >= input$dates[1] & dat$Date <= input$dates[2], ]

## The data records for the selected species are automatically extracted within
## the levelMap.plot function (which is external code for creating catch maps)

## Extract data records for selected people
dat.vec <- c()
names.vec <- data.frame()
for(i in 1:length(input$names)){
  dat.vec <- dat[dat$FirstName == input$names[i], ]
  names.vec <- rbind(names.vec, dat.vec)
}
working.dat <- names.vec

## Extract cases for selected lines for selected people
dat.vec <- c()
lines.vec <- data.frame()
for(i in 1:length(input$subLines)){
  dat.vec <- working.dat[working.dat$Line == input$subLines[i], ]
  lines.vec <- rbind(lines.vec, dat.vec)
}
working.dat <- lines.vec

## Extract unique species, names, lines and traps
unique.species <- input$selectSpecies
```

```
    unique.names <- input$names
    unique.lines <- input$lines

    ## Bandwidth (Focus), Quantile Threshold (Emphasis), Dates and Zoom values
    ## These are explained in more detail in Section 6.2
    project <- input$selectArea
    band <- input$range

    ## Quant is adjusted to prevent the 'bleeding' effect
    quant <- log(exp(3)*input$quantile - input$quantile + 1)/3
    lat <- range(working.dat$Latitude)
    long <- range(working.dat$Longitude)
    zoom <- MaxZoom(lat, long)
    zoomer <- input$zoomer
    dates <- input$dates
    ## Return a list of working data objects to plug into the map outputs
    list(working.dat = working.dat, unique.species = unique.species, unique.names
         = unique.names, unique.lines = unique.lines, project = project, dates =
         dates, band = band, quant = quant, zoom = zoom, zoomer = zoomer)
  })
})
```

The final graphics displayed in *Figure 25* show where catches have been made in the Te Henga project for all species in 2015. Within the plotting code below, the data to be plotted is simply plugged in from the reactive `working.data` list.

```
## Ui code excerpt:
plotOutput("maps2")

## Server code excerpt:
##### Map Outputs #####
## Update maps when Load Maps is clicked
```

```
observeEvent(input$submit, {
  output$maps2 <- renderPlot({

    ## Set the appropriate layout for the graphics - a 2x4 matrix of plots
    ## The top row of plots is used for the titles, the bottom for the maps
    layout(matrix(c(1, 2, 3, 4, 5, 6, 7, 8), byrow=T, ncol=4), widths=c(1, 0.1,
           1, 0.25), heights=c(0.1, 1), respect=T)

    ## First plot: title of heat map
    ## Remove the margins so the plots are juxtaposed
    ## Plot a grey rectangle for the title to be superimposed on
    par(mar = c(0, 0, 0, 0), mgp = c(0, 0, 0))
    plot(0, type = "n", bty = "n", xlim = c(0, 1), ylim=c(0, 1), xaxt = "n",
         yaxt = "n")
    rect(-0.04, -0.04, 1.04, 1.04, col=rgb(0.8, 0.8, 0.8, 1), lwd=1)

    < Ommitted 30 lines of code that automates the title for the heat map  >
    < and determines over how many lines the title should be spread        >

    ## Second plot: blank space between titles
    frame()

    ## Third plot: title of catch map
    par(mar=c(0, 0, 0, 0), mgp=c(0, 0, 0))
    plot(0, type="n", bty="n", xlim=c(0, 1), ylim=c(0, 1), xaxt="n",
         yaxt="n")
    rect(-0.04, -0.04, 1.04, 1.04, col=rgb(0.8, 0.8, 0.8, 1), lwd=1)

    < Ommitted 10 lines of code that automates the title for the catch map >
    < and determines over how many lines the title should be spread        >
```

```
    ## Fourth plot is blank space - above the catch map legend
    frame()

    ## Fifth plot is the heat map
    intensity.wrap(dat = working.data()$working.dat, plotWhat =
                   working.data()$unique.species, bandw = working.data()$band,
                   quant.threshhold = working.data()$quant, trans.lo = 0.7,
                   trans.hi = 0.9, nsmooth = 500, zoom = working.data()$zoom +
                   working.data()$zoomer - 2, saveMap = )

    ## Sixth plot: blank space between maps
    frame()

    ## Seventh plot is the catch map
    ## The legend creation is masked within the levelMap.plot function
    levelMap.plot(dat = working.data()$working.dat, plotWhat =
                  working.data()$unique.species, zoom = working.data()$zoom +
                  working.data()$zoomer - 2)
  })
})
```

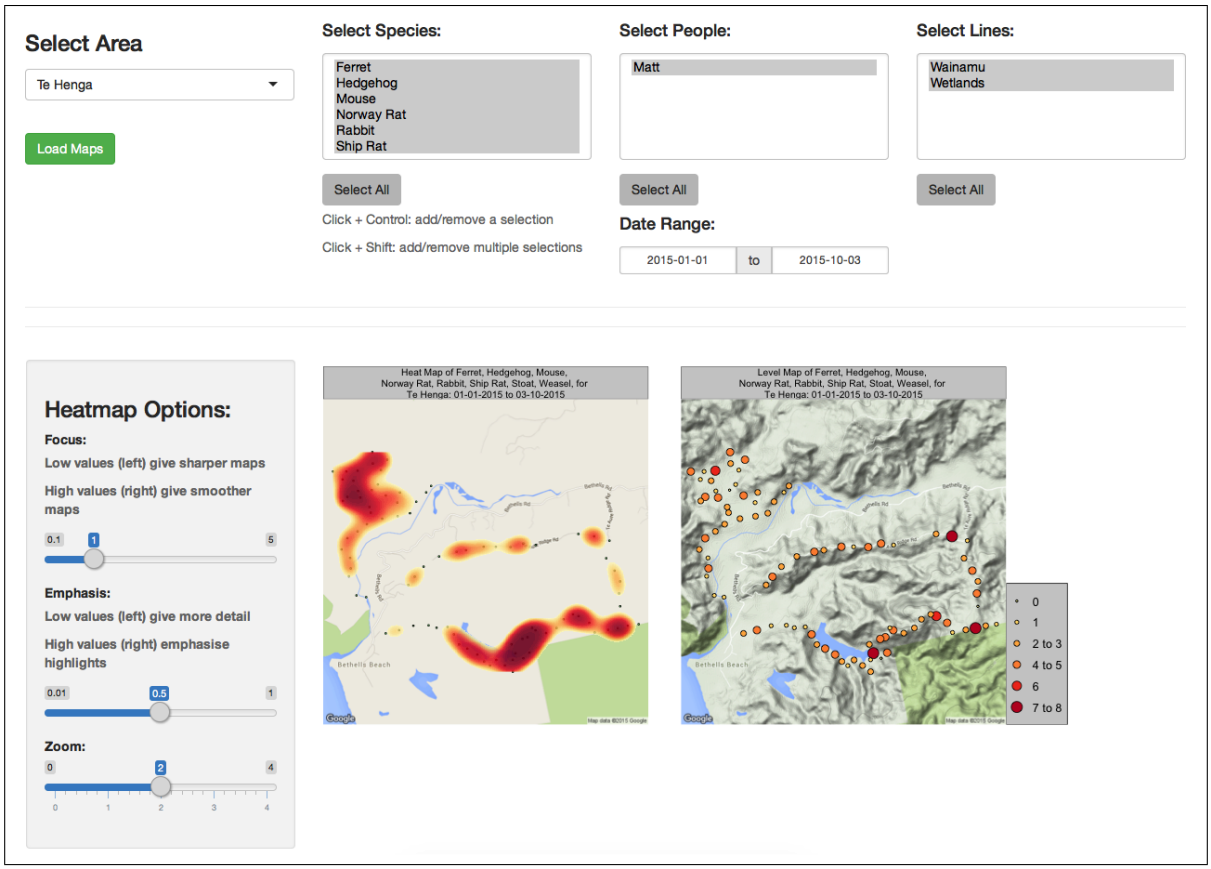


Figure 25. Heat map and catch map for all species caught in Te Henga in 2015

Any set of heat maps and catch maps can be produced for projects affiliated with CatchIT, which again goes to show the flexibility in graphics creation we are afforded by having all data in an identical format. The development of this application is another example of how we as statisticians can give those with less statistical knowledge the ability to create personalised graphics and analyses.

6.2 Heat map viewing options

The application also offers several options to customise the heat map and how it displays catch data. These options will only affect how the heat map is presented; the catch map remains constant as a depiction of the raw data.

6.2.1 Focus

The first of these options is the 'focus', which essentially determines how much smoothing is applied to the heat map. Statistically speaking the focus refers to the bandwidth, however, one of the challenges of this project was to decide on non-technical terms for technical concepts. Focus was

chosen as a word that suitably conveys this concept. A low value for focus gives a sharp heat map, while a high value will give a smooth heat map. The default value for focus is 1 which gives a map with a moderate amount of smoothing. The automation of producing the default heat map to have a reasonably balanced appearance is a concept that has demanded a lot of attention, and is addressed in more detail in Chapter 7.

A low value for focus means each trap's influence on the map does not extend very far from where it is placed, resulting in a heat map with numerous small areas of concentrated catches. Increasing the value for focus increases the range of a trap's influence, resulting in a smoother heat map with only a few large areas of concentration

(a) Focus = 0.4 - a sharp heat map

(b) Focus = 4 - a smooth heat map

Figure 26. Heat maps with low and high levels of focus

6.2.2 Emphasis

The emphasis will affect the amount of detail displayed on the heat map. Areas with the lowest catch-rates are displayed transparently on the heat map; the emphasis setting selects what level of catch-rate is considered low enough to be displayed transparently. A high value will display information only for the regions which have made a large number of catches, emphasising the areas that make the highest number of catches. A low value for emphasis will display information for almost all regions. This option is useful for distinguishing areas of traps that make large numbers of catches from areas which make fewer catches. *Figure 27* below gives a clear example of this. The heat map on the left appears to suggest that a lot of areas have made a high number of catches, as shown by the many dark red ('hot') areas. However, when the emphasis is increased, only the traps that are making the highest number of catches are shown on the map.

(a) Emphasis = 0.1 *(b)* Emphasis = 0.9

Figure 27. Heat maps with low and high levels of emphasis

6.2.3 Zoom

Given a currently selected set of options, the application determines the appropriate level of zoom for the heat map. However, if a user wants to zoom in or out, they can do so by using the zoom slider. Zooming out is useful if a user wants to get a better idea of the surrounding landscape, while zooming in can be useful when closely examining neighbouring traplines. Because the zoom feature zooms in and out on the centre of the selected traplines, it is not particularly useful to zoom in when many traplines are selected, as a good chunk of the map will get cut off. The example in *Figure 28* below illustrates how a user might zoom out to get a better view of the surrounding area, or zoom in on a specific set of traplines.

(a) Heat map zoomed out showing surrounding area *(b)* Heat map zoomed in on a few specific traplines

Figure 28. Heat maps with different levels of zoom

7 Automating the default heat maps

As noted in Chapter 6, a considerable amount of effort was placed into ensuring that the heat maps produced using the default options are both attractive and informative. The challenge here is that each conservation project has a unique shape that requires a unique overlay. Therefore, our chosen method must be very flexible in adapting to new projects that might join CatchIT with any sort of abnormal shape.

These shapes can range from the layouts of the Kaipupu and Tahi projects, which cover a fairly square area and have fairly equally spaced traplines, to the more linear layouts of the Okura Bush and D'Urville Island projects, which have traplines placed in a line with few loops, covering an L-shaped or T-shaped area.

(a) Layout for the Kaipupu project *(b)* Layout for the Tahi project

Figure 29. Equally spaced traplines covering a square area

(a) Layout for the Okura Bush project *(b)* Layout for the D'Urville Island project

Figure 30. Linear traplines

7.1 Creating the heat maps

Creating the heat map overlay in R first requires the data in the CSV file to be converted to a point process object. This is essentially a dataset that has an entry for each trap, containing the x and y-coordinates of the trap in the plotting region and the total number of catches made by that trap.

```
## Example point process object for the Kaipupu project
## data.ppp is a planer point process object that belongs to the 'spatstat' package
head(data.ppp)
           x          y m
1 -109.51794   60.45919 1
2  -94.60486   15.81249 5
3  -29.82616 -209.28879 1
4  -21.43755  -11.47185 7
5  -29.36013  -13.33215 5
6  -43.34114  -11.47185 4
```

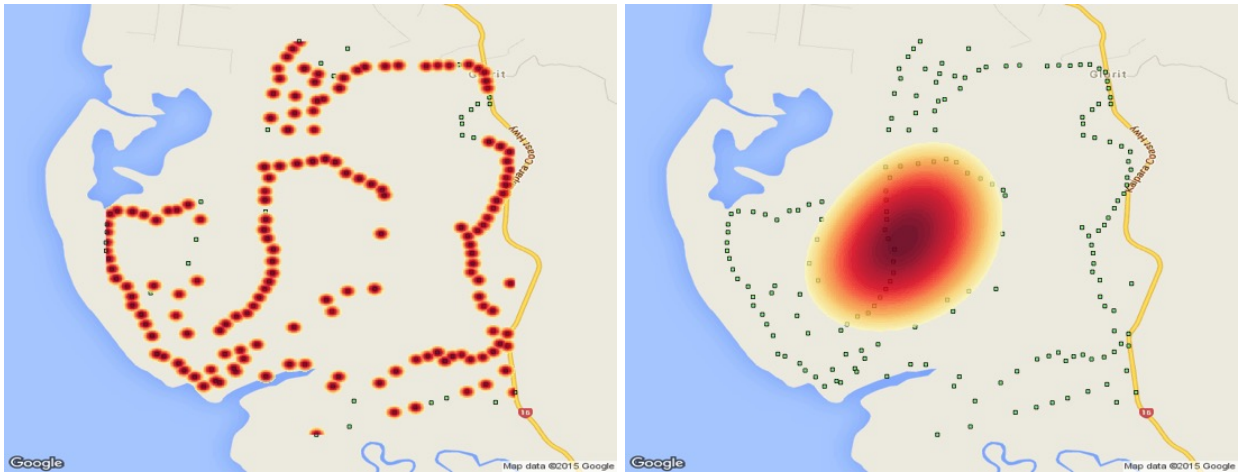
The initial approach for selecting the bandwidth for the heat map overlay was to use the R function `bw.diggle`, which uses cross-validation to select a smoothing bandwidth value (defined as σ), for the kernel estimation of point process intensity (Baddeley and Turner, 2005). This value for σ is chosen to minimise the mean-square error criterion defined by Diggle (1985). The heat map application then takes this value, and multiplies it by the currently selected value for the 'Focus' option (default = 1), and returns the heat map.

```
## Default bandwidth for the Kaipupu project
bw.diggle(data.ppp)
   sigma
7.335026
```

$$\text{Bandwidth} = \texttt{bw.diggle(data.ppp)} \times \text{Focus} \tag{1}$$

Since the catch maps already represent the number of catches made by each individual trap, the idea of the heat maps is to indicate areas of traps where high numbers of catches are made.

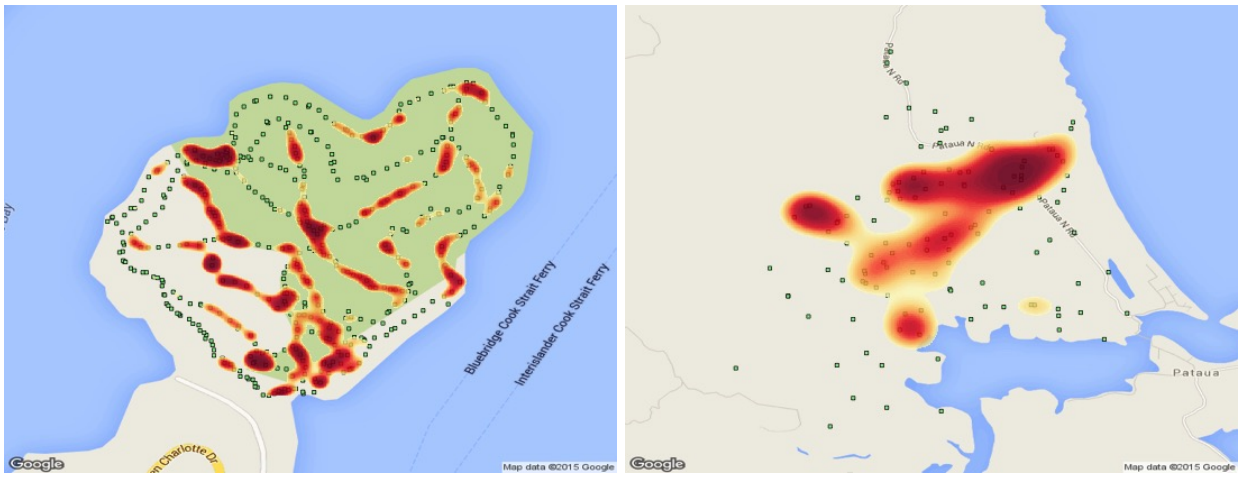
Not enough smoothing by the `bw.diggle` function will result in each trap being given an individual overlay, looking very similar to the catch map, while being less informative as there is no accompanying legend. However, too much smoothing will result in the overlay being too focused on a single area which is equally uninformative.



(a) A heat map too little enough smoothing *(b)* A heat map with too much smoothing

Figure 31. Extreme levels of smoothing

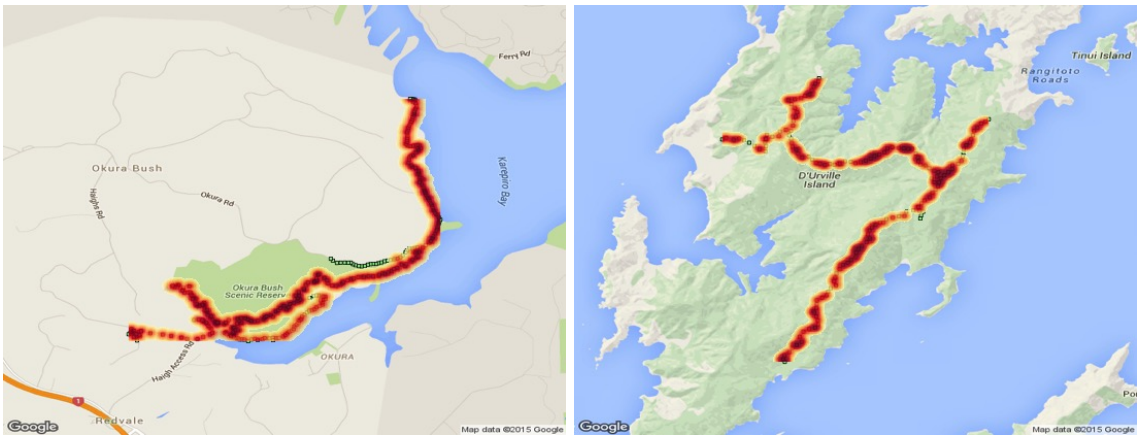
For projects with equally spaced traplines, the `bw.diggle` approach appears to apply an appropriate amount of smoothing and does a reasonably good job at producing a well-balanced overlay. The heat maps in *Figure 32* are both attractive and informative, clearly depicting areas and groups of traps that have made high numbers of catches.



(a) Heat map for Kaipupu *(b)* Heat map for Tahi

Figure 32. Default heat maps produced using the `bw.diggle` function

However, the `bw.diggle` approach does not do such a great job when it is applied to the projects with more linear traplines, such as Okura Bush and D'Urville Island. The default heat maps produced in *Figure 33* have not had enough smoothing, and suggest that almost every trap is a hotspot for catches, which is not the case.



(a) Heat map for Okura Bush *(b)* Heat map for D'Urville Island

Figure 33. Default heat maps produced using the `bw.diggle` function

Therefore, an adjustment to the `bw.diggle` approach must be made. The aim is that this adjustment should improve the default heat maps for projects with linear traplines, without affecting the projects with equally spaced traplines.

7.2 Bandwidth adjustment

For the eight largest projects affiliated with CatchIT, the default bandwidth values were calculated using `bw.diggle`. A rough approximation by eye was also made for each area as to how much this default bandwidth value needed to be multiplied by to give a more informative heat map.

	Area	Default Bandwidth	Approx Bandwidth Multiplier
1	Okura Bush	2.25	9.00
2	Kaipupu	7.34	1.60
3	Te Henga	23.78	0.75
4	Mataia	14.78	0.80
5	MEG Coro Kiwi Project	7.34	2.50
6	Durville	3.84	2.75
7	East Taranaki	14.14	2.00

8	Tahi	24.90	1.00

There appears to be no logical pattern in the default bandwidth values calculated by `bw.diggle`, however, this exercise does identify the projects Okura Bush and D'Urville Island, to be those most in need of adjustment. As noted earlier, both of these projects have relatively linear trapline layouts, giving us a starting point to work from.

Therefore, the idea of trap density was introduced to help determine how much the default bandwidth value needs to be adjusted to produce an informative heat map. Drawing a convex hull around the perimeter of the traps for each project gives a rough idea of a project's trap density. As seen in *Figure 34*, the trap density for projects with equally spaced traplines is fairly consistent within the boundaries of the project.

(a) Convex hull for Kaipupu *(b)* Convex hull for Tahi

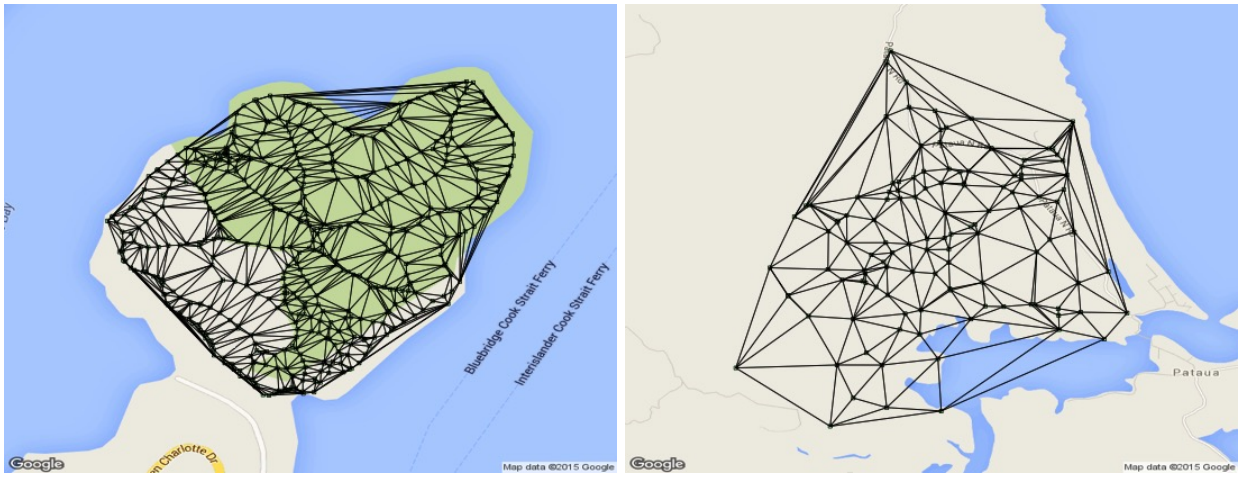
Figure 34. Convex hulls for projects with equally spaced traplines

However, for projects with linear traplines, the trap density is clearly very high in some locations of the project (specifically the edges), but very low in others, as seen in *Figure 35*.

(a) Convex hull for Okura Bush *(b)* Convex hull for D'Urville Island

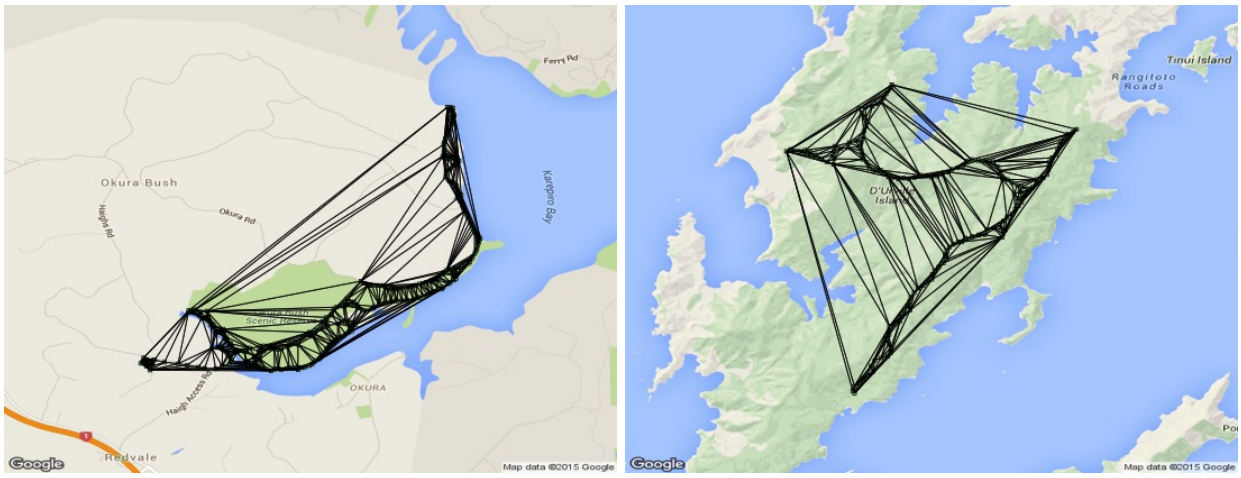
Figure 35. Convex hulls for projects with linear traplines

In order to quantify the differences in trap densities, a Delaunay triangulation was applied to each project, producing a tessellation of triangles between traps. We expected projects with fairly equally spaced traplines, to have triangles that are mostly of similar size. In contrast, we expected projects with linear traplines, to have triangles that have greater variation in size. This appears to be the case as shown in *Figures 36* and *37*.



(a) Delaunay triangulation for Kaipupu *(b)* Delaunay triangulation for Tahi

Figure 36. Delaunay triangulations for projects with equally spaced traplines



(a) Delaunay triangulation for Okura Bush *(b)* Delaunay triangulation for D'Urville Island

Figure 37. Delaunay triangulations for projects with linear traplines

Therefore, a relationship appears to exist between the shape of a project's layout, and the variation in size of the triangles produced by the Delaunay triangulation. To further investigate this relationship, the plotting ratio and coefficient of variation for the Delaunay triangulation was calculated for each project.

The plotting ratio was defined as the ratio between the actual plotting area of the Google map, and the plotting area of the heat map overlay, examples of which can be seen below in *Figure 38*.

(a) Plotting ratio for Kaipupu = 0.44 *(b)* Plotting ratio for Okura Bush = 0.31

Figure 38. Plotting ratios

The coefficient of variation for the Delaunay triangulation was calculated as follows:

$$\mu = \text{mean area of triangles}$$
$$\sigma^2 = \text{variance of triangle edge lengths}$$

$$\text{Coefficient of variation} = \frac{\sqrt{\sigma^2}}{\mu} \tag{2}$$

The coefficient of variation was expected to be larger for projects that have linear traplines, while the plotting ratio informs us how much of the screen the overlay will cover.

	Area	CoV	Plotting Ratio	CoV/PR	Bandwidth Multiplier
1	Okura Bush	2.06	0.31	6.65	9.00
2	Kaipupu	0.87	0.44	1.98	1.60
3	Te Henga	1.04	0.63	1.65	0.75
4	Mataia	0.90	0.56	1.61	0.80
5	MEG Coro Kiwi Project	0.90	0.23	3.91	2.50
6	Durville	1.45	0.31	4.68	2.75
7	East Taranaki	1.31	0.61	2.15	2.00
8	Tahi	0.82	0.51	1.61	1.00

These results appear to suggest that areas with large values for $\frac{\text{coefficient of variation}}{\text{plotting ratio}}$ will likely have a larger value for the desired bandwidth multiplier. This relationship is plotted below in Figure 39.

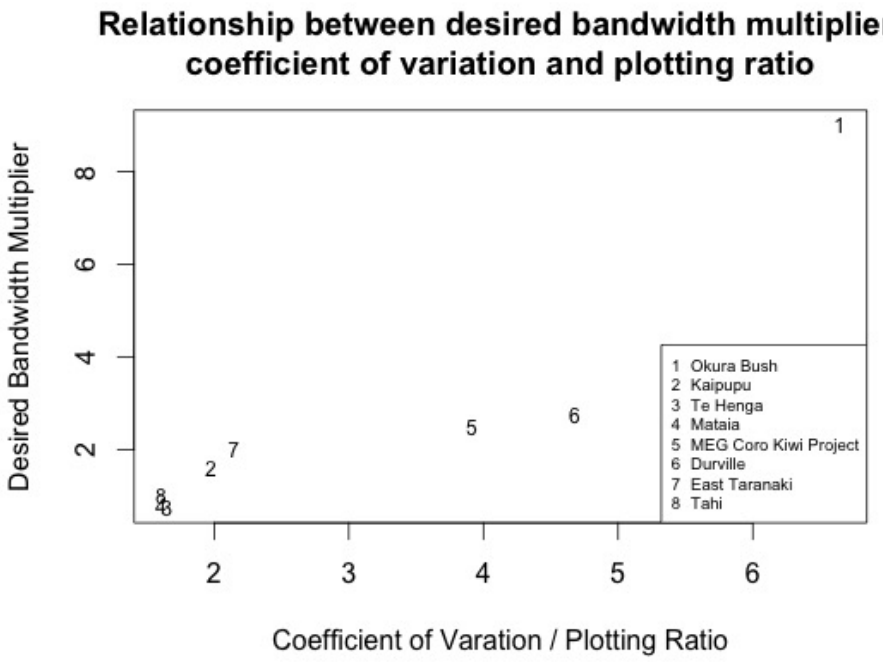


Figure 39. Plot of desired bandwidth versus $\frac{\text{coefficient of variation}}{\text{plotting ratio}}$

Taking the log of the y-axis variable, desired bandwidth multiplier; the relationship becomes somewhat linear, allowing us to perform a linear regression.

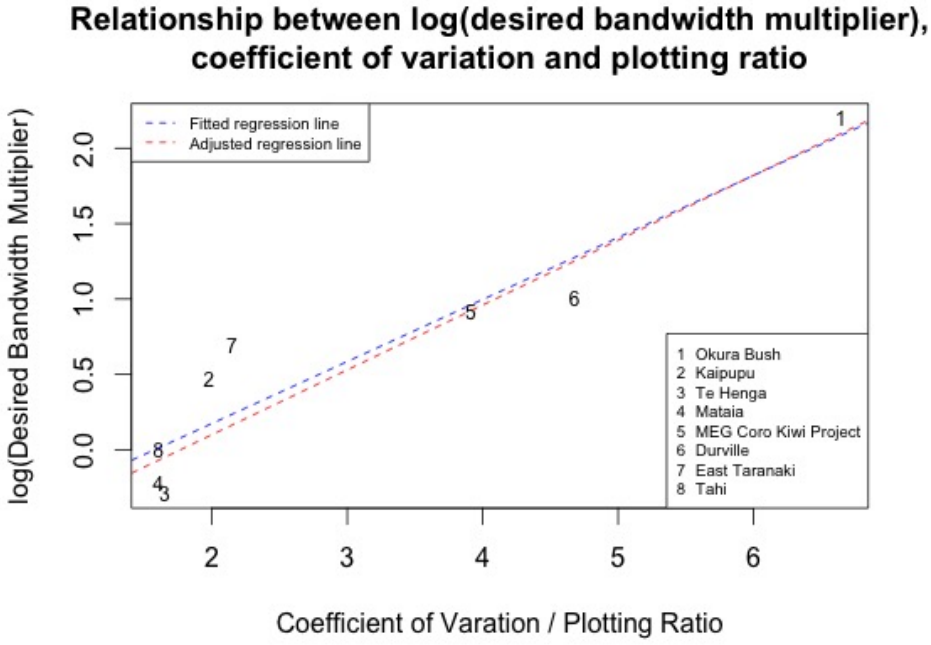


Figure 40. Plot of log(desired bandwidth) versus $\frac{\text{coefficient of variation}}{\text{plotting ratio}}$

```
lm(log(desired.bandwidth) ~ I(CoV/Plot.ratio))$coef

      (Intercept)

           -0.648

I(CoV/Plot.ratio)

            0.411
```

The fitted line for this regression is plotted in Figure 40 in blue. A marginal adjustment (plotted in red) has been made to this regression, as some projects were more sensitive to the adjustment than others (specifically the MEG Coro Kiwi Project). This gives a final adjustment of:

$$\text{Bandwidth adjustment} = \exp\left(0.76 + 0.43 \times \frac{\text{coefficient of variation}}{\text{plotting ratio}}\right) \tag{3}$$

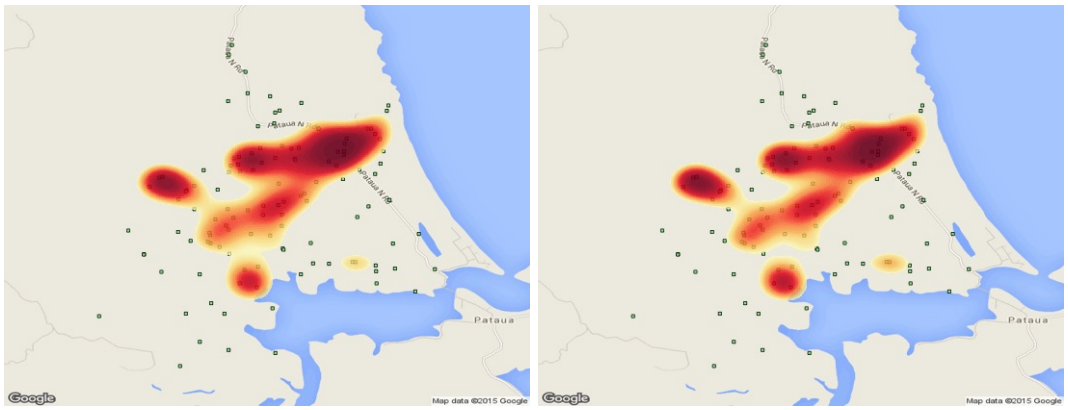
Therefore, the final bandwidth used by the heat map application when calculating the overlays is:

$$\text{Final bandwidth} = \text{Bandwidth adjustment} \times \texttt{bw.diggle(data.ppp)} \times \text{Focus} \tag{4}$$

This chapter again goes to show that producing informative, attractive graphics is one hurdle to overcome, but the real challenge lies in automating the production. It cannot be denied that the bandwidth adjustment detailed in this chapter is heuristic in nature, however its effectiveness speaks for itself. Projects that previously produced rather poor heat maps when using the default bandwidth calculated by `bw.diggle`, now produce much more informative graphics. Additionally, the projects for which the default bandwidth value worked fine, are barely changed. The heat maps of each project before and after the bandwidth adjustment are shown in Figures 41 - 48 below.

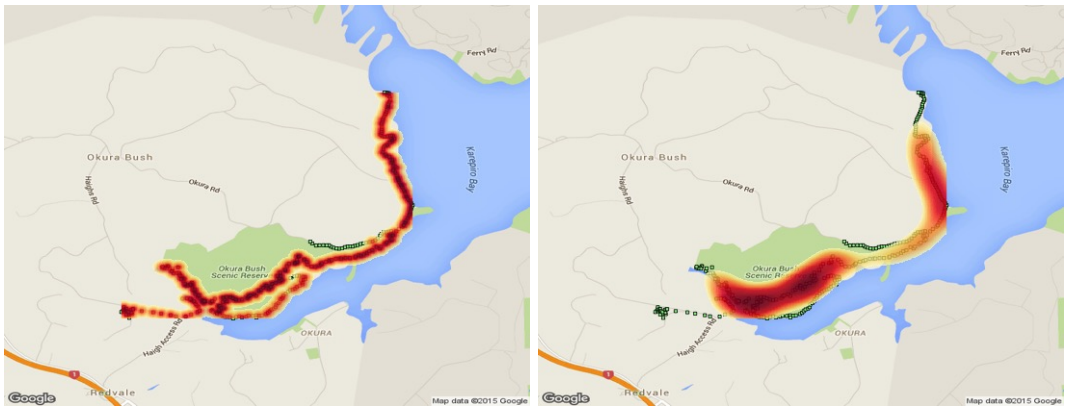
(a) Heat map using default bandwidth *(b)* Heat map using adjusted bandwidth

Figure 41. Kaipupu heat maps



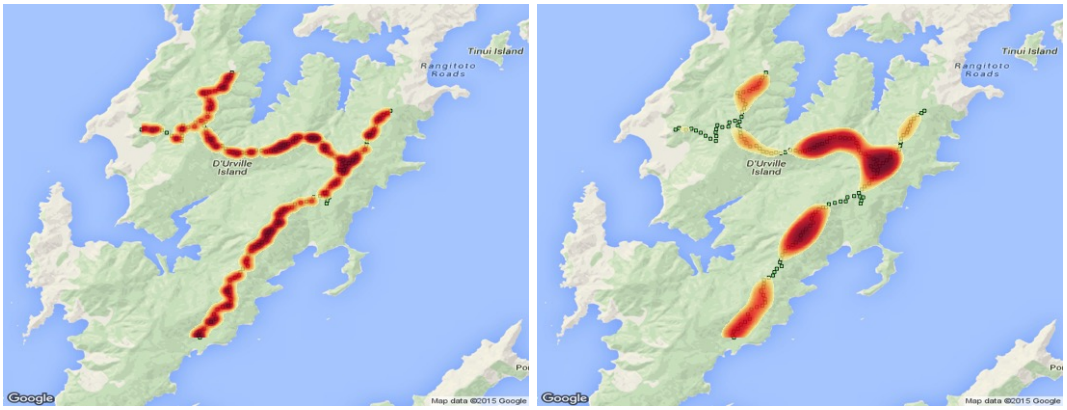
(a) Heat map using default bandwidth *(b)* Heat map using adjusted bandwidth

Figure 42. Tahi heat maps



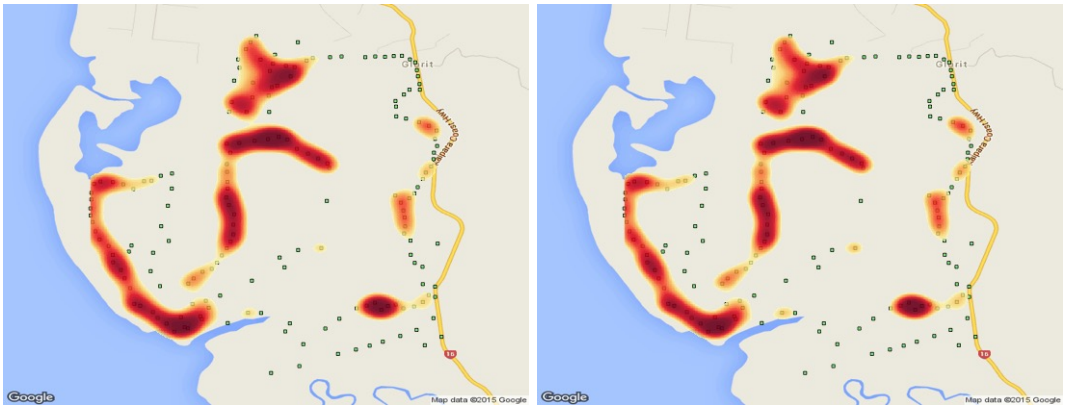
(a) Heat map using default bandwidth *(b)* Heat map using adjusted bandwidth

Figure 43. Okura Bush heat maps



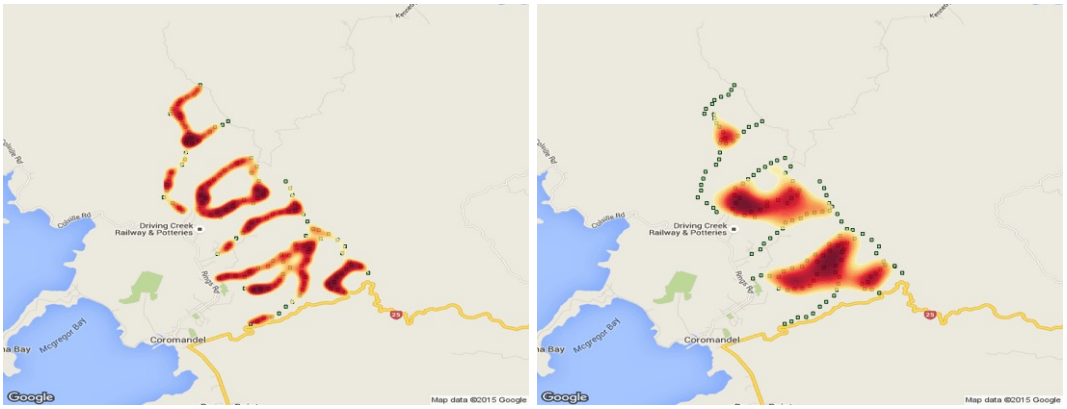
(a) Heat map using default bandwidth *(b)* Heat map using adjusted bandwidth

Figure 44. D'Urville Island heat maps



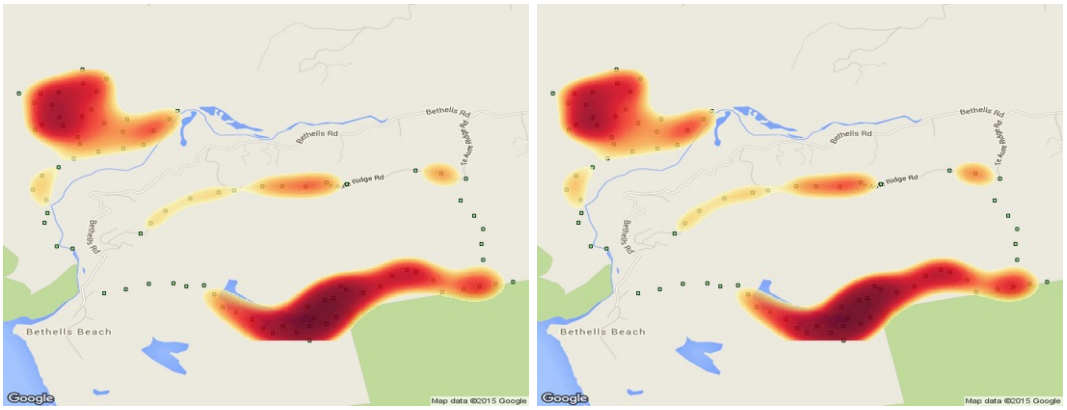
(a) Heat map using default bandwidth *(b)* Heat map using adjusted bandwidth

Figure 45. Mataia heat maps



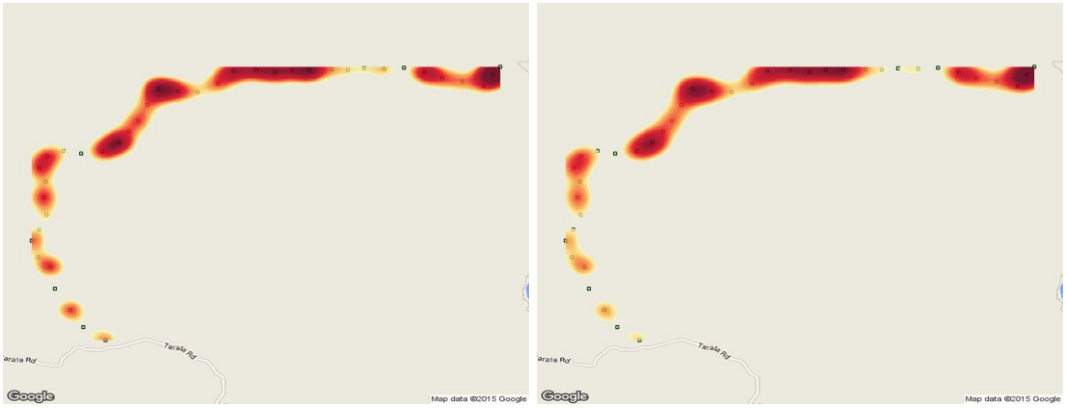
(a) Heat map using default bandwidth *(b)* Heat map using adjusted bandwidth

Figure 46. MEG Coro Kiwi Project heat maps



(a) Heat map using default bandwidth *(b)* Heat map using adjusted bandwidth

Figure 47. Te Henga heat maps



(a) Heat map using default bandwidth *(b)* Heat map using adjusted bandwidth

Figure 48. East Taranaki heat maps

8 Conclusions and further work

At the time of printing, the two applications are being distributed to various conservation projects in New Zealand communities and schools. In the near future, they will be utilised by hundreds of people, ranging from school children to retired volunteers. Initial feedback from volunteers indicates that the maps and graphics are both informative and easy-to-use; the applications are serving their intended purpose of presenting statistical data in a creative, attention-grabbing manner.

The flexibility of these applications allows new projects to be readily incorporated. The only requirement is to convert the data into the prescribed format, which is simply achieved by contacting the CatchIT developers. New users of the applications require no knowledge of R or statistics to get their data up and running.

The success of these applications for conservation volunteers in New Zealand highlights the scope for bridging the gap between lay-people and their understanding of data, in other disciplines.

References

Baddeley, A. and Turner, R. (2005). spatstat: Spatial point pattern analysis, model-fitting, simulation, tests. *Journal of Statistical Software*, 12(6):1–42.

Brewer, C. A. (1999). Color use guidelines for data representation. In *Proceedings of the Section on Statistical Graphics*, pages 55–60. American Statistical Association.

CatchIT (2011). About CatchIT. Retrieved from `https://www.stat.auckland.ac.nz/~fewster/CatchIT/`.

Diggle, P. (1985). A kernel method for smoothing point process data. *Applied Statistics*, 34:138–147.

Ihaka, R. (2003). Colour for presentation graphics. In *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*.

RStudio, Inc (2013). *Easy web applications in R.* Retrieved from `http://www.rstudio.com/shiny/`.

Wild, C. and Seber, G. (2000). *Chance encounters: A first course in data analysis and inference*. John Wiley & Sons Inc., New York, NY.