# CAR Bushfire Smoke Variable Extraction - Handover

## Introduction

This should give an overview of the approaches taken to generate different types of variables for the CAR bushfire smoke project.

Basically there are five types of extraction that we need to accomplish in this project, which vary wildly in time taken to process, they are –

|  |  |  |
| --- | --- | --- |
| **Type of extraction** | **Examples** | **Performance across approaches** |
| Extraction from raster at point locations | * Elevation * Climate variables | Quite quick across all approaches, with R slightly slower |
| Extraction from raster at buffers around point locations | * Population density | Relatively slow in R, both for calculating the buffer and extraction from it. Quicker in Python and ArcGIS |
| Count of points within a buffer of point locations | * Fire density | ArcGIS has the advantage, but R is not far behind, provided the ‘st\_contains’ command is used as opposed to computing an intersection |
| Intersection of polygon and buffer around point locations | * Burnt area * Meshblock categorisation | Very slow in R and even in Python. Manageable if polygon is rasterized using ArcGIS and extractions taken |
| Density of line shapefile within a given buffer of point locations | * Road networks and density | I haven’t had to do this yet, I imagine the process would be quite similar for intersections of polygons and buffers |

I’ll go through each process in turn

## Some notes on software/packages

#### Basic formatting

So far it seems lighter (and easier) to format extracts by FID, instead of a latitude and longitude pair. Not only does this allow for better joining, if necessary, it reduces the size of the output dataset. I have kept an RDS file in CloudStor located at –

*/Bushfire\_Smoke\_for\_CAR\_Project/Predictors/data\_derived\_for\_predicting/AUS\_points\_5km.rds*

This has FID and the respective latitude and longitude (in GDA94/Australian Albers, EPSG 3577 projection). It’s lightweight and serves as an easy lookup table for all output, as well as input locations, with lat/lon easily converted between coordinate reference systems, such as the unprojected GDA94 (EPSG 4283) by:

grid <- sf::st\_as\_sf(grid, coords = c(‘X’, ‘Y’), crs = 3577)

grid <- sf::st\_transform(grid, crs = 4283)

#### R

The approaches I’ve used have been in line with my experience and knowledge. As such, I’ve mostly used the sf package in R since it maps best to PostGres commands, and the approaches can likely be implemented there with minimal re-writing. The exception is that I use the raster package to handle rasters, even though it works with the sp package. The stars package (which makes rasters sf compatible) is pretty rubbish, and doesn’t have the same functionality that raster does. The velox package speeds everything up and is cross-compatible across the raster, sp, and the sf packages. The issue with velox is that it’s no longer supported by CRAN, so needs to be installed from Github, which is a headache, but absolutely worth your while. There’s a guide [here](https://rdrr.io/github/hunzikp/velox/), but it was definitely not that straightforward when I first tackled it.

I also mainly use *dplyr*/*tidyverse* to work with data frames, and I use a lot of *purrr* functions, which work similarly to apply functions but have more versatility. Since recent updates, the performance gain using *data.table* over *tidyverse* is much smaller, with the two almost comparable now, and I find the code easier to read.

#### Python

I’m by no means a Python expert, but I’ve found an easy way to get things working is to install Anaconda, and run it through there. Maybe some Python expert would baulk at this, but hey, it works for me. I use the *geopandas* packages for shapefile handling and the *rasterio* package as both interface well with *numpy* and *pandas* which seem to be the standard approaches.

#### ArcGIS

This is the most unwieldly for me. While it’s inarguably quicker to do a lot of tasks, it’s also frustrating, with some functions not having what should be standard options, and outputs not saving anywhere near as easily as they should. Also, you need to work within ‘projects’ that accumulate a lot of temporary junk files over time, so it takes up a lot of space. Finally, some loops written in ArcGIS stop unexpectedly, with no warning, and you have to restart them. I can’t find any explanation for this but if you can solve it let me know. Basically I just monitor loops now, and when they inevitably stop, I just restart the loop at the point it got up to. Not the end of the world, just annoying.

I have found that loops tend to run better if you code entirely in arcpy and don’t use other packages (such as ‘glob’ for getting files in a directly). The ListFeatureClasses command does a similar task but seems to be more resistant to breaking. The issue there is you can only list feature classes within a working directory rather than, say, anywhere you want. So I’ve been working within a temporary working directory, and I output any final tables/files to an ‘out’ directory. Again, there’s probably a better way of doing this, but I find the programming side of ArcGIS quite opaque.

I also use R to compile outputs from ArcGIS, by converting any ArcGIS output to a dbf file, and then reading into R using the *foreign* package.

## Extraction Approaches

### Extraction from raster at point locations

Since this performs pretty comparable across all software, and I’m most comfortable in *R*, I’ve just been doing it using *R*. I’ve done side by side comparisons with *R*, *Python* and *ArcGIS* and the speed gain in *Python* and *ArcGIS* isn’t worth the additional learning. This is especially quick if you parallelise the process in R, which the Lismore PC is perfect for.

An example of this is getting wind speed at (10/100) at grid point locations, located below –

/Bushfire\_Smoke\_for\_CAR\_Project/Predictors/data\_derived\_for\_predicting/Spatiotemporal/Wind/Wind Extract.R

An alternative would be to write a bunch of scripts and submit to Artemis (see process below – Count of points) since this effectively parallelises the process.

#### Extraction from raster at buffers around point locations

Python or ArcGIS can accomplish this relatively quickly, you just need to loop it. Python performs about as quickly as ArcGIS, so that’s what I went with. Example code is located at –

/Bushfire\_Smoke\_for\_CAR\_Project/Predictors/data\_derived\_for\_predicting/Spatiotemporal/Population

I’ve annotated the process in the Python script. The basic process involves defining a function for extraction, converting the output to a data frame, and then labelling it accordingly.

Note this could also be done with an ArcGIS loop, if you wanted to have a look at that too. The code is much simpler, but requires a bit more processing afterwards to get the density.

#### Count of points within a buffer of point locations

I’ve done this using *R* because it’s actually quite quick. This is because we can make use of the ‘st\_contains’ command, that doesn’t actually compute an intersection polygon, just counts the number of points inside the buffer. For big jobs I would use Artemis, because you can submit the jobs in parallel, and it’ll just run through them in order.

I generate individual scripts using a loop, as in the ‘Generate Codes for Artemis.R’ script saved in –

/Bushfire\_Smoke\_for\_CAR\_Project/Predictors/data\_derived\_for\_predicting/Spatiotemporal/ActiveFire/

This outputs both an R script and a pbs file which adds the script to the Artemis queue. Read up on Artemis scheduling in the ‘Using Artemis guide’ I wrote up and stored in ‘/PRJ-env\_health/’ folder on the RDS network. You can submit pbs files in a loop in Artemis, by running something like –

For i in \*.pbs; do qsub $i; done

In the folder where you keep the pbs files.

Alternately you can just run this on a loop on your computer with a similar approach. See the scripts generated in the above folder (under ‘Codes’) for an example.

#### Intersection of polygon and buffer around point locations

This is the most time consuming of all approaches and this is where ArcGIS is unquestionably better than the other approaches. Both R and Python are agonisingly slow at computing intersections between polygons (i.e. a polygon and a buffer), and ArcGIS is similarly slow *unless* you convert the polygon to a raster first. The process is as follows –

1. First process the shapefile. Split the shapefile up into individual temporal units (or whatever unit you’re iterating over) using the arcpy command.
2. Output to a temporary folder.
3. Read each split shapefile and add a field that is the value of the area of each grid point, which is calculated in the next step. For example, at 500m resolution the area would be 250m², or 0.25km². This step isn’t strictly necessary, as you can just count the number of points and then multiply that by the area each point represents, but this makes things slightly easier.
4. Convert this polygon to raster, at the highest resolution necessary, using the area field calculated above as the value. I found 500m metres to be fine for the burnt fires polygon. I did a test conversion and compared the two outputs and they look basically identical with some very slight differences that didn’t affect any estimations. Other polygons might need higher resolutions, or you may be able to get away with lower ones.
5. Compute focal statistics around each point in that raster at the required buffer. Note that this calculates a buffer at *every* point, i.e. at a 500m resolution, irrespective of what your output is.
6. Extract that value of these focal statistics at the points required using zonal statistics.
7. Output this into a table

Despite the excess computation in step 5, this is *still* much faster than any other approach. If you can work out how to get it to extract focal statistics at only the desired output resolution, then that will speed things up even more. Computing buffers around the incoming points in advance is not the solution as ArcGIS has to rasterise *every* buffer polygon around each point in the grid, which takes ages.