

3301

John Duncan and Bryan Boruff

Table of contents

1 3301: Advanced GIS and Remote Sensing	5
1.1 Google Earth Engine Labs	5
1.1.1 Sign Up	6
1.2 Google Earth Engine	6
1.2.1 Applications	7
1.3 Useful Resources	7
2 JavaScript Introduction	8
2.1 Setup	8
2.1.1 Code Editor	8
2.1.2 Create a Repository	9
2.2 Programming	11
2.2.1 Data Types	11
2.2.2 Variables	15
2.2.3 Object Data Model	15
2.2.4 Functions	16
2.2.5 Syntax and Code Style	17
3 Introduction	19
3.0.1 Setup	19
3.1 Client and Server	19
3.2 The ee object	20
3.3 Spatial Data Models	20
3.3.1 Vector Data Model	21
3.3.2 Raster Data Model	21
3.4 Spatial Data Structures	22
3.4.1 Images	22
3.4.2 Geometry Objects	26
3.4.3 Features	29
3.4.4 Collections	30
4 Introduction	34
4.0.1 Setup	34
4.1 Colour Theory	34
4.1.1 Colour	35

4.1.2	Colour Models / Colour Spaces	35
4.1.3	Choosing Colours	38
4.2	Colour to Highlight Features	40
4.3	Colour to Represent Groups	41
4.4	Colour to Represent Data Values	43
4.5	Multiband Images	45
4.5.1	True Colour Composite Image	46
4.5.2	False Colour Composite Image	48
5	Introduction	50
5.0.1	Which Perth university has the greenest and coolest campus?	51
5.0.2	Setup	51
5.0.3	Data Import	52
5.1	Filter	53
5.2	Buffer	54
5.3	Zonal Statistics	57
5.4	Join	59
5.5	Descriptive Statistics	60
5.6	Visualisation	63
6	Introduction	67
6.0.1	Where has vegetation cover changed between 2000-2002 and 2017-2019?	68
6.0.2	Setup	68
6.1	Filter	69
6.1.1	Cloud Masks	71
6.2	Create new variables	79
6.2.1	Image Math and Local Map Algebra Operations	79
6.2.2	Spectral Indices	80
6.3	Join / Combine	83
6.4	Summarise	84
6.5	Change Detection	85
7	Introduction	87
7.0.1	Phenology	87
7.0.2	Remote Sensing of Phenology	87
7.1	Urban Park Phenology	89
7.2	Setup	89
7.3	Data Preprocessing	90
7.4	Noise Removal and Gap Filling	92
7.5	Smoothing	94
7.6	Non-vegetation Mask	99
7.7	Phenology Metrics	100
7.8	Phenology and Climate	102

8	Introduction	107
8.0.1	Supervised Classification	107
8.0.2	Unsupervised Classification	108
8.1	Task	109
8.2	Setup	109
8.2.1	Data Import	109
8.3	Supervised Classification	110
8.3.1	Training (Feature) Engineering	110
8.3.2	Train	112
8.3.3	Predict	113
8.3.4	Accuracy Assessment	115
8.4	Unsupervised Classification	119
9	Introduction	121
9.0.1	Predicting Land Surface Temperatures	121
9.0.2	Random Forests	121
9.1	Setup	126
9.2	Data Import	126
9.3	Training (Feature) Engineering	129
9.4	Model Training	131
9.5	Model Interpretation	132
9.6	Model Testing	135

1 3301: Advanced GIS and Remote Sensing

1.1 Google Earth Engine Labs

These labs introduce Google Earth Engine as a tool for geospatial data analysis. Geospatial data analysis can be broken down into a workflow comprising four stages (that are often iterative):

- Data (variables to represent spatial concepts, phenomena, or entities - data collection, data entry, data download, and data import)
- Transform (transforming data so it can be used for geospatial data analysis - summarising, combining, filtering)
- Visualisation (looking for patterns, features, and relationships in your data - maps, charts, tables)
- Model (use mathematical and statistical tools to answer questions using your data)

These stages are based on the data science workflow presented by Wickham and Grolemund (2017). As the figure below indicates, the stages of the data analysis workflow are wrapped in a *program* box. You will use a programming language to specify the tasks to be performed at each of the stage of the data analysis workflow. In these labs you will be introduced to JavaScript, a programming language for writing commands in Google Earth Engine programs. While learning to code is not the primary aim of these labs, an awareness of programming concepts is useful for geospatial data analysis and often makes your life easier through the ability to automate tasks.

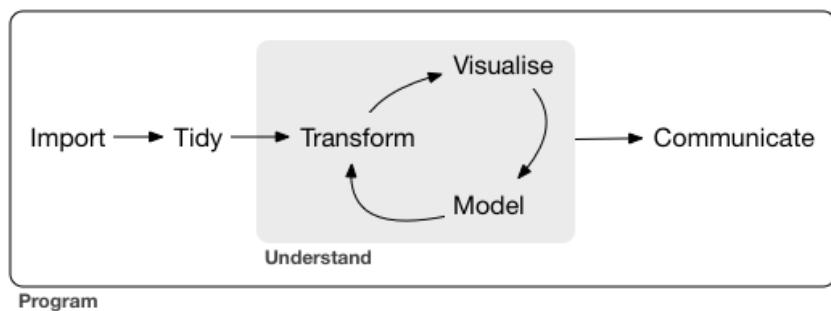


Figure 1.1: Data science workflow (source: [Wickham and Grolemund \(2017\)](#))

Each of the labs relate to a stage in the geospatial data analysis workflow outlined above. Thematically, the labs will focus on geospatial data analysis in urban areas (primarily using data from the Perth region).

- **Preliminary Resource:** Introduction to JavaScript and programming concepts
- **Lab 4a:** Data: Spatial data models and data structures
- **Lab 4b:** Visualisation: Spatial data visualisation
- **Lab 5:** Transform: Vector data operations
- **Lab 6:** Transform: Raster and remote sensing data processing
- **Lab 7:** Model: Land cover classification
- **Lab 8:** Model: Predicting urban land surface temperatures
- **Lab 9:** Model: Trends in urban vegetation and temperature

A series of questions are posed throughout the lab as demonstrated below; these are to help you engage with the content, have a go at answering them and click on the question to reveal the answer.

Example Question?

Answer.

1.1.1 Sign Up

Sign up for Google Earth Engine here. Identify that you are using Google Earth Engine for educational purposes as part of the Advanced GIS and Remote Sensing undergraduate course at the University of Western Australia.

1.2 Google Earth Engine

Google Earth Engine is a platform for geospatial data analysis. It combines databases of big geospatial data that are updated daily, a range of geospatial data analysis and processing functions, and access to cloud computing resources to apply these functions to geospatial datasets.

You can access Google Earth Engine through the Code Editor - a web-based interactive development environment (IDE) for creating Google Earth Engine programs for geospatial data analysis and visualising the results in web maps, interactive charts, or text summaries.

Gorelick et al. (2017) provide a detailed description of the Google Earth Engine platform.

1.2.1 Applications

Applications of Google Earth Engine span a variety of disciplines which utilise geospatial data:

- Monitoring global forest change
- Mapping habitat ranges
- Global water security
- Monitoring global croplands
- Mapping travel time to urban centres

More examples of how Google Earth Engine is used can be found on the Google Earth Engine blog.

1.3 Useful Resources

There are a range user resources for Google Earth Engine. You should use these resources to supplement the work done in the labs. Becoming familiar with these resources will help you troubleshoot problems. Using these resources will develop your independent problem solving skills when undertaking geospatial data analysis.

1. Google Earth Engine introduction - comprehensive overview of Google Earth Engine's capabilities.
2. Google Earth Engine tutorials - range of introductory and advanced tutorials on using Google Earth Engine for geospatial data analysis.
3. Google Earth Engine for education - range of training resources.
4. User Forum and help tab in the code editor (see below).

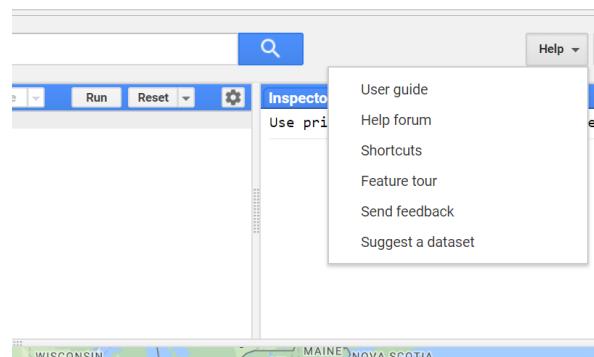


Figure 1.2: Google Earth Engine user forum and help tab

2 JavaScript Introduction

You will be using Google Earth Engine to perform geospatial data analysis. Google Earth Engine programs comprise a series of statements written in a programming language (JavaScript or Python) that outline steps taken to perform specific tasks using geospatial data.

This lab is an introduction to programming using JavaScript. It introduces key concepts that are important to understand when using Google Earth Engine. However, also view this section as a *reference resource* to refer back to as you work through the labs and become more proficient in using Google Earth Engine. A good exercise to consolidate your understanding of these concepts is to try and identify where, and explain how, the concepts that are introduced here are used to perform various geospatial data analysis tasks in later labs.

2.1 Setup

Load the Google Earth Engine code editor in your browser via the URL: <https://code.earthengine.google.com/>.

2.1.1 Code Editor

You will create Google Earth Engine programs using the code editor. The code editor is a web-based interactive development environment (IDE) which provides access to the Google Earth Engine JavaScript API. The Google Earth Engine Developers Guide provides an overview of the code editor tools.

The code editor provides a range of tools that make geospatial data analysis and visualisation easy. These tools will be introduced in the subsequent labs. Some key code editor features include:

- **code editor:** where you write JavaScript statements.
- **Scripts tab:** save the JavaScript code for your Google Earth Engine programs.
- **Map:** web map to visualise spatial data.
- **Docs:** JavaScript API reference - lists all the in-built functions and operations.
- **Console:** print results from analysis and metadata.
- **Inspector tab:** interactive query of spatial objects on the map.
- **Geometry tools:** digitise vector features.
- **Run:** Run your script.

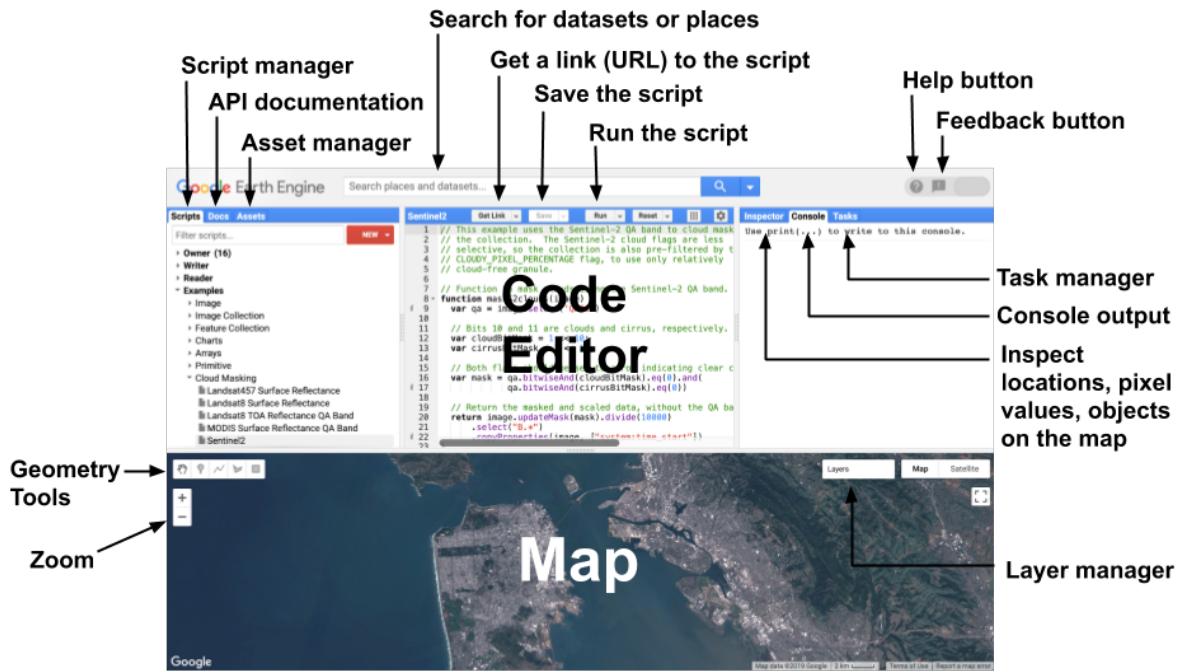


Figure 2.1: Google Earth Engine code editor (source: Google Earth Engine [Developers Guide](#)).

2.1.2 Create a Repository

Create a repository called *labs-gee* where you will store the scripts containing the code for programs you write in the labs. Go to the *Scripts* tab and click the **NEW** button to create a new *labs-gee* repository.

Enter the following code into the *Code Editor* and save the script to your *labs-gee* repository. Name the script *JS-intro*. This code is just some comments that define what the script does and who wrote it and when. Replace the author name and date as appropriate. Comments are not executed when your program runs. **Under path in the save widget make sure you select the correct repository (i.e. not default)**.

```

/*
JavvScript Introduction
Author: Test
Date: XX-XX-XXXX

*/
    
```

Create repository and save script.

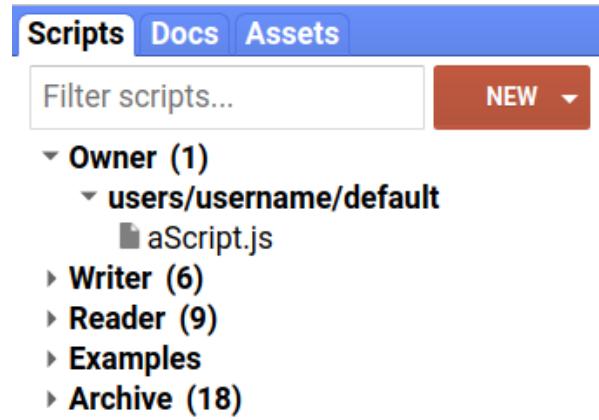


Figure 2.2: Scripts tab and button to create new repositories (source: Google Earth Engine Developers Guide).

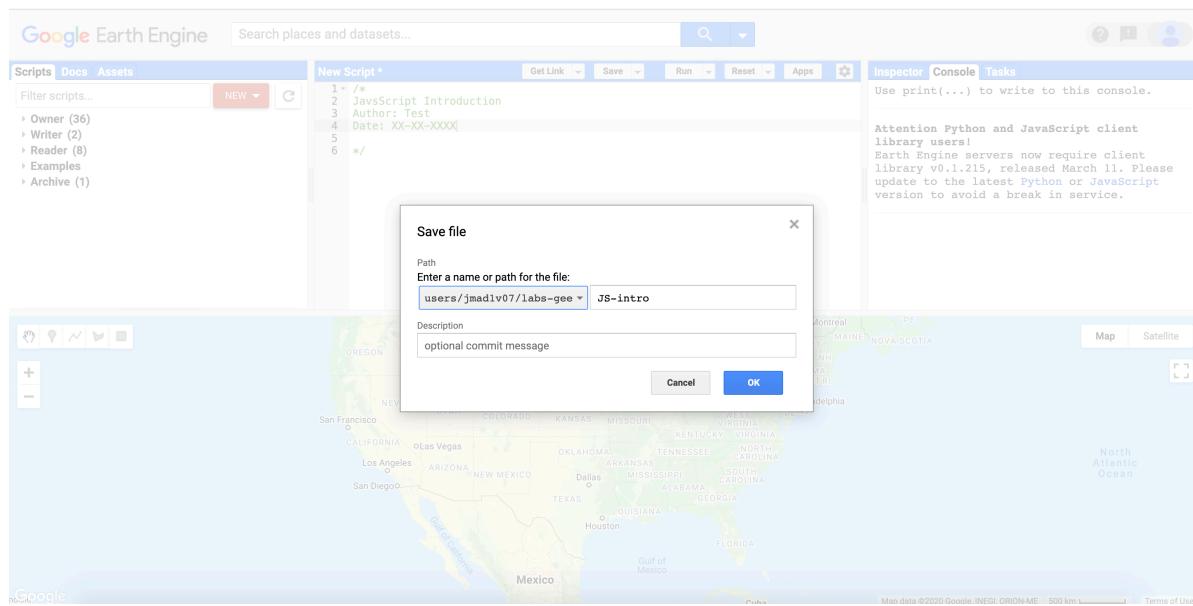


Figure 2.3: Save script to labs-gee repository.

2.2 Programming

Programming (coding) is the creation of source code for programs that run on computers. You will be writing programs using JavaScript.

2.2.1 Data Types

Programs need data to work with, perform operations on, and to return outputs from computation and analysis. Geographic and non-geographic phenomena and entities are represented in computer programs as data of specific types. In JavaScript there are seven primitive data types:

- undefined
- String
- Number
- Boolean
- BigInt
- Symbol
- null

undefined types are variables that have not been assigned a value. Variables of null data type intentionally have no value.

All other data types in JavaScript are of type object.

Strings

Variables of string data type contain characters and text which are surrounded by single ' or double " quotes. There are several cases where string variables are used when working with geospatial data; for example, in the metadata of satellite images the name of the sensor used to collect an image could be stored as a string.

What other geospatial data could be stored as a string data type?

Anything that needs to be represented as text data such as place names, road names, names of weather stations.

Enter the following command into the code editor to create a string variable.

```
var stringVar= 'Hello World!';
```

You have created a string variable called `stringVar` which contains the text information ‘Hello World!’. This is data that you can use in your program.

You can use the `print()` operation to print the data in `stringVar` onto the *Console* for inspection.

```
print(stringVar);
```

You should see ‘Hello World!’ displayed in the *Console*. You have just written a simple program that creates a string object storing text data in a variable named `stringVar` and prints this text data to a display.

In reality, programs that perform geospatial data analysis will be more complex, contain many variables of different data types, and perform more operations than printing values to a display (instead of printing results to the *Console* a GIS program might write a `.tif` file containing the raster output from some analysis).

Hello World!

Numbers

The number data type in JavaScript is in double precision 64 bit floating point format. Add the following code to your script to make two number variables.

```
var x = 1;
var y = 2;
print(x);
print(y);
```

Storing numbers in variables enables programs to perform mathematical and statistical operations and represent geographic phenomena and entities using quantitative values. For example, spectral reflectance values in remote sensing images are numeric which can be combined mathematically to compute vegetation indices (e.g. NDVI).

Execute the following code to perform some basic maths with the variables `x` and `y`.

```
var z = x + y;
```

What numeric value do you think variable `z` will contain? How could you check if the variable `z` contains the correct value?

```
3 print(z);
```

Boolean

The Boolean data type is used to store true or false values. This is useful for storing the results of comparison (equal to, greater than, less than) and logical (and, or, not) operations.

```
var demoBool = z == 4;
print(demoBool);

var bool1 = x == 1 && y == 2;
```

```
var bool2 = y < x;
```

You can read up on JavaScript logical and comparison operators here or look at the table below.

What do you think the value of `bool1` and `bool2` will be?

`bool1`: true `bool2`: false

Table 2.1: JavaScript comparison and logical operators

Operator	Description	Example
<code>==</code>	equal to	<code>x == 5</code>
<code>!=</code>	not equal	<code>x != 5</code>
<code>></code>	greater than	<code>x > 5</code>
<code><</code>	less than	<code>x < 5</code>
<code>>=</code>	greater than or equal to	<code>x >= 5</code>
<code><=</code>	less than or equal to	<code>x <= 5</code>
<code>&&</code>	and	<code>x == 5 && y == 4</code>
<code> </code>	or	<code>x == 5 y == 5</code>
<code>!</code>	not	<code>!(x <= 5)</code>

Objects

An object in JavaScript is a collection properties where each property is a name:value pair and the value can be any primitive data type (e.g. String, Number, Boolean, null) or a type of object. You can create custom data types using objects; for example, you could create an object to represent a point with two name:value pairs: `longitude: 25.55` and `latitude: 23.42` where the values are number type coordinates.

You can access properties of an object using the dot operator: `.` with the format `<object name>. <property name>`.

```
var lon = 25.55;
var lat = 23.42;

// create an object named point
var point = {
  longitude: lon,
  latitude: lat
};
print(point);
```

```
// access value in object  
print(point.longitude);
```

Arrays

Arrays are a special list-like object that store an ordered collection of elements. Arrays are declared by placing values in square brackets [1, 2, 3] and you access values inside an array using the value's array index. The first value in an array has an index of 0, the second value has an index of 1, and the final value has an index of $n - 1$ where n is the number of elements in the array. This is the distinction between arrays and objects where elements are represented by name:value pairs. The elements in arrays are ordered and accessed by their index position; the elements in objects are unordered and accessed by their property name.

Below is an example of how to create an array of numbers that represent years.

```
var years = [2000, 2001, 2002, 2003, 2004, 2005, 2006];
```

You can see the data inside arrays using the `print()` command or extract information from arrays using square brackets [] and the index of the element.

```
print(years);  
var year0 = years[0];  
print(year0);  
var year1 = years[1];  
print(year1);
```

You can also put strings inside arrays.

```
var stringList = ['I', 'am', 'in', 'a', 'list'];  
print(stringList);
```

Remember, each item in an array is separated by a comma. You can create n-Dimensional arrays.

```
var squareArray = [  
  [2, 3],  
  [3, 4]  
];  
print(squareArray);
```

What kind of geospatial data is well suited to being represented using arrays?

raster data (grids of pixels with each pixel assigned a value).

2.2.2 Variables

Variables are named containers that store data.

To create a variable you need to declare it using the `var` keyword. Once a variable is declared you can put data inside it and use that variable, and therefore the data inside it, in your program. You assign data to a variable using the assignment operator `=`.

The code block below declares a variable `temp` and then assigns the value 25 to this variable. As demonstrated by the variable `temp1` you can declare a variable and assign values to it in one statement.

```
var temp;  
temp = 25;  
  
var temp1 = 26;
```

Using variables makes code easier to organise and write. For example, if you want to perform multiple operations on temperature data you can refer to the data using the variable name as opposed to either writing out the temperature values or reading them from a file separately for each operation. You can use variables in operations and functions too:

What value do you think the variable `tempDiff` would store after executing this statement: `var tempDiff = temp1 - temp;`?

```
1 print(tempDiff);.
```

You only need to declare a variable once. `var` is a reserved keyword; this means a variable cannot be named `var`. Other reserved keywords in JavaScript include `class`, `function`, `let`, and `return` with a full list here.

2.2.3 Object Data Model

Except for the seven primitive data types, everything in JavaScript is an object. An object is a programming concept where each object can have properties which describe attributes of the object and methods which are operations or tasks that can be performed.

Real world phenomenon or entities can be represented as objects. For example, you can define an object called `field` to represent data about fields. The `field` object can have a numeric array property storing the vertices representing the field's location, a crop type string property stating what crops are grown in the field, and a numeric type property stating crop yield. The `field` object could have a `computeArea()` method which would calculate and return the area of the field. The `field` object is a spatial object so it could also have methods such as `intersects()` which would return spatial objects whose extent intersects with the field.

An object definition, which outlines the properties and methods associated with an object, is called a class; you can create multiple objects of the same class in your program.

2.2.4 Functions

There are many methods and operations already defined in JavaScript that you can use in your program. However, there will be cases where you need to create your own operation to perform a task as part of your program; user-defined functions fill this role. First, you declare or define your function which consists of:

- The `function` keyword.
- The name of the function.
- The list of parameters the function takes in separated by commas and enclosed in parentheses (e.g. `function subtraction(number1, number2)`).
- A list of statements that perform the function tasks enclosed in braces `{ }`.
- A `return` statement that specifies what data is returned by a call to the function.

```
// subtraction function

//function declaration
function subtraction(number1, number2) {
    var diff = number1 - number2;
    return diff;
}
```

Once a function has been declared you can call it from within your program. For example, you can call the function `subtraction` declared above and pass the two numeric variables `temp` and `temp1` into it as arguments. This will return the difference between the numeric values stored in `temp` and `temp1`.

```
// use subtraction function
var tempFuncDiff = subtraction(temp, temp1);
print(tempFuncDiff);
```

You should see the result `-1` printed in the *Console*.

This is a very simple example of how to declare and use a function. However, creating your own functions is one of the key advantages of programming. You can flexibly combine functions together to create complex workflows.

The following example declares and calls a function `convertTempToK` that takes in a temperature value in degrees centigrade as a parameter and returns the temperature in Kelvin.

```
// temperature conversion function
function convertTempToK(tempIn) {
    var tempK = tempIn - (-273.15);
    return tempK;
}
var tempInK = convertTempToK(temp);
print(tempInK);
```

2.2.5 Syntax and Code Style

There are various syntax rules that need to be followed when writing JavaScript statements. If these rules are not followed your code will not execute and you'll get a syntax error.

As you see and write JavaScript programs, syntax and style will become apparent. This is not something you need to get right first time but is part of the process of learning to write your own programs. Error messages when you run your program will alert you to where there are syntax errors so you can fix them.

Some important syntax rules:

- Strings are enclosed within " or ' quotes.
- Hyphen - cannot be used except as the subtraction operator (i.e. perth-airport is **not** valid).
- JavaScript identifiers are used to identify variables or functions (i.e. a variable of function name - x = 23 is identified by variable name x). Identifiers are case sensitive and can only start with a letter, underscore (_), or dollar sign (\$).
- Identifiers cannot start with a number.
- Variables need to be declared with the **var** keyword before they are used.
- Keywords (e.g. **var**) are reserved and cannot be used as variable or function names.

Code Style

Alongside syntax rules, there are stylistic recommendations for writing JavaScript. These are best adhered to as they'll make your code easier for you, future you, or somebody else to read. This is important if you require help debugging a script.

Some common style tips:

- Use camel case for variables - first word is lower case and all other words start with an upper case letter with no spaces between words (e.g. `camelCase`, `perthAirport`).
- Finish each statement with a semi-colon `var x = 23;`.
- At most, one statement per line (a statement can span multiple lines if required or improves readability).
- Consistency in code style throughout your script.

- Indent each block of code with two spaces.
- Sensible and logical variable names - variable names should be nouns and describe the variable.
- Sensible and logical function names - function names should be verbs that describe what the function does.
- Keep variable and function names short to avoid typos.
- One variable declaration per line.

The Google JavaScript style guide is a useful resource for writing clear JavaScript programs.

Comments

You can write text in your script that is not executed by the computer. These are comments and are useful to describe what parts of your script are doing. In general, you should aspire to write your code so that it is legible and easy to follow. However, comments augment good code, can help explain how a program works, and are useful to someone else using your script or to future you if you return to working on it.

Some useful things to comment:

- Start the script with brief description of what it does.
- Author and date of script.
- Outline any data or other programs the script depends on.
- Outline what data or results are returned by the script.
- Avoid commenting things which are outlined in documentation elsewhere (e.g. Google Earth Engine documentation).
- Outline what arguments (and type) a function takes and returns.

Lines of code can be commented using `//` or `/* ... */`.

```
/*
Script declares variables to store latitude and longitude values.
Author: XXXXX
Date: 01/02/0304
*/

// longitude
var lon = 25.55;

// latitude
var lat = 23.42;
```

3 Introduction

This lab introduces spatial data models for representing geographic entities and phenomena in Google Earth Engine.

3.0.1 Setup

Create a new script in your *labs-gee* repository called *lab4a*. Enter the following comment header to the script.

```
/*
Lab 5a
Author: Test
Date: XX-XX-XXXX

*/
```

3.1 Client and Server

In the preliminary lab you have been writing JavaScript programs that are executed in your browser and run on the hardware in your local machine (i.e. any data in variables you declare resides in your computer's memory and the functions you call run on your computer's CPU).

However, your machine has limited storage, memory, and processing power. Google Earth Engine allows you to access cloud servers comprising more powerful computers and access to larger datasets. You still write a Google Earth Engine program in JavaScript using the code editor in your browser; however, the servers storing and processing the geospatial data in your program are remotely located in the cloud.

The execution of a Google Earth Engine program is as follows:

1. You write a series of JavaScript statements that identify geospatial data, and operations to perform on this data, that run on Google servers and the results to be returned from this processing.
2. Your browser sends these statements to the Google servers.

3. The Google servers process your message, access data you requested, and perform the operations outlined in your script.
4. Results your program requests back from the Google servers are returned to your browser and displayed (e.g. a map is drawn in your browser display, results are printed to the *Console*, a file is made available to download).

3.2 The ee object

It is important to distinguish between variables that are stored, and operations that are run, locally on your machine and data and operations that run in the cloud. The `ee` prefix indicates that the data being referred to in your script is a server side object. For example, `var localString = 'on my computer'` is a string type variable stored locally on your machine where as `var cloudString = ee.String('in the cloud')` is a proxy object for a variable containing string data located on servers in the cloud.

In general, any variable that is declared as `ee.<Thing>()` is server side and any method or operation of the form `ee.<Thing>().method()` is a server side operation. One way of understanding `ee.<Thing>()` is as a container that you put instructions inside to send to the Google servers; for example, in `var cloudString = ee.String('in the cloud')` you are putting a client side string '`in the cloud`' in a container and that data is sent to servers in the cloud. Similarly, you could put the ID of geospatial data that is stored in cloud databases and assign it to server side variables that are used in your program; executing `var landsatImage = ee.Image('LANDSAT/LC8_L1T_TOA/LC81130822014033LGN00')` will assign the Landsat image with the specified ID to the variable `landsatImage` in your script.

If the geospatial data and operations used in your program are server side how do you access or visualise the results? There are a range of functions in Google Earth Engine that let you request data from the server to be displayed in your browser. For example, the `print()` function can request server side objects and print them to the *Console* and the `Map.addLayer()` function requests spatial data which is displayed in the map.

3.3 Spatial Data Models

A spatial data model refers to a conceptual model for describing geographic phenomena or entities. A spatial data model typically contains two pieces of information:

- Positional information describing location, shape, and extent (e.g. an `(x, y)` coordinate pair representing the location of a weather station).
- Attribute information describing characteristics of the phenomenon or entity (e.g. a name:value pair recording the name of the weather station `name:'Perth Airport'`).

A spatial data model is a *representation* of geographic phenomena or entities; therefore, some detail is abstracted away.

3.3.1 Vector Data Model

The vector data model represents geographic phenomena or entities as geometric features:

- points (i.e. a coordinate pair of values)
- lines (i.e. two or more points connected by a line)
- polygons (i.e. three or more points connected by a non-intersecting line which “closes” the polygon)

Along with coordinates that represent the position of the geometry, vector data also stores non-spatial attribute information which describe characteristics of the geographic phenomenon or entity represented by the geometry feature.

The figure below demonstrates how geographic entities in Perth can be represented using the vector data model. The blue line feature broadly captures the shape of the river; however, it is a simplification as it does not provide information about how the river’s width varies across space. The red point feature is used to represent the location of Perth; this might be an appropriate way to represent Perth’s location on a zoomed out map but it does not capture Perth’s actual extent.

What detail is abstracted away by representing Kings Park using the green polygon feature?

Shape of Kings Park is simplified using only 6 vertices.

Variation in land cover types and land uses within the park is not captured.

3.3.2 Raster Data Model

The raster data model represents geographic phenomena or entities as a grid of cells (pixels). Attribute information about geographic phenomena or entities is described by assigning a value to each pixel. The dimensions of a pixel relative to distance on the Earth’s land surface determines the complexity and detail of spatial features that can be resolved in raster data. A pixel that represents a 1 km x 1 km footprint on the Earth’s surface will not be able to represent an individual tree or a single building. Pixel values can be continuous (e.g. values represent precipitation) or categorical (e.g. values represent a land cover type).

The figure below shows the 2018 European Space Agency (ESA) Climate Change Initiative (CCI) land cover map for 2018. This is a raster data model representation of land cover; each pixel represents a 300 m x 300 m area on the Earth’s land surface and a pixel can only represent a single land cover type. If you look at the bottom two zoomed in maps you can see some limitations of modelling land cover using 300 m x 300 m spatial resolution raster data. The

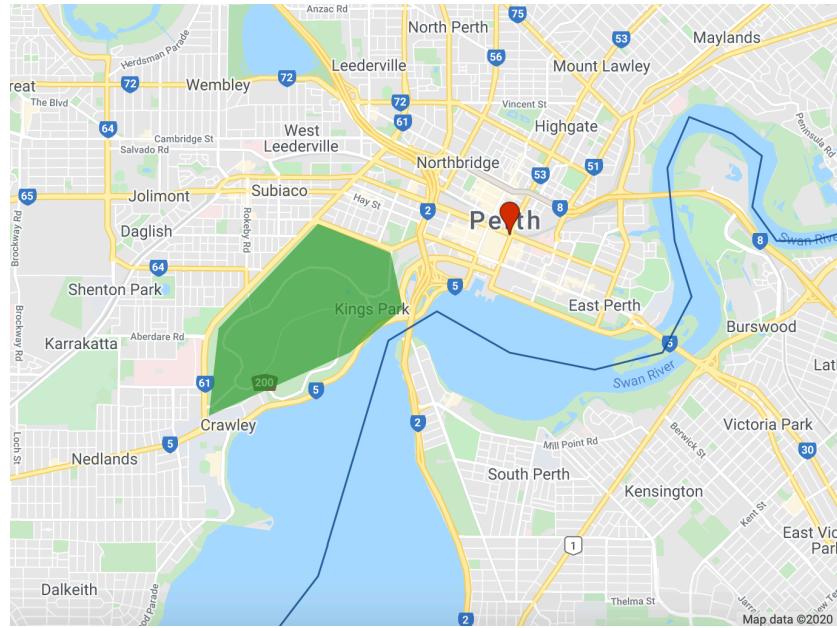


Figure 3.1: Representing geographic entities using the vector data model.

shape of land cover features are poorly represented by the “block-like” arrangement of pixels and there is variation in land cover within a single pixel (a mixed pixel problem).

How could you represent spatial variation in elevation using vector and raster data models?

Vector data model: contour lines.

Raster data model: digital elevation model (DEM) - each pixel value represents the elevation at that location.

3.4 Spatial Data Structures

3.4.1 Images

Raster data in GEE are represented as `Image` objects.

To create an `Image` object that stores raster data on the GEE server use the `ee.Image()` constructor. You pass arguments into the parentheses of the `ee.Image()` constructor to specify what raster data should be represented by the `Image` object. If you pass a number into `ee.Image()` you will get a constant image where each pixel value is the number passed in.

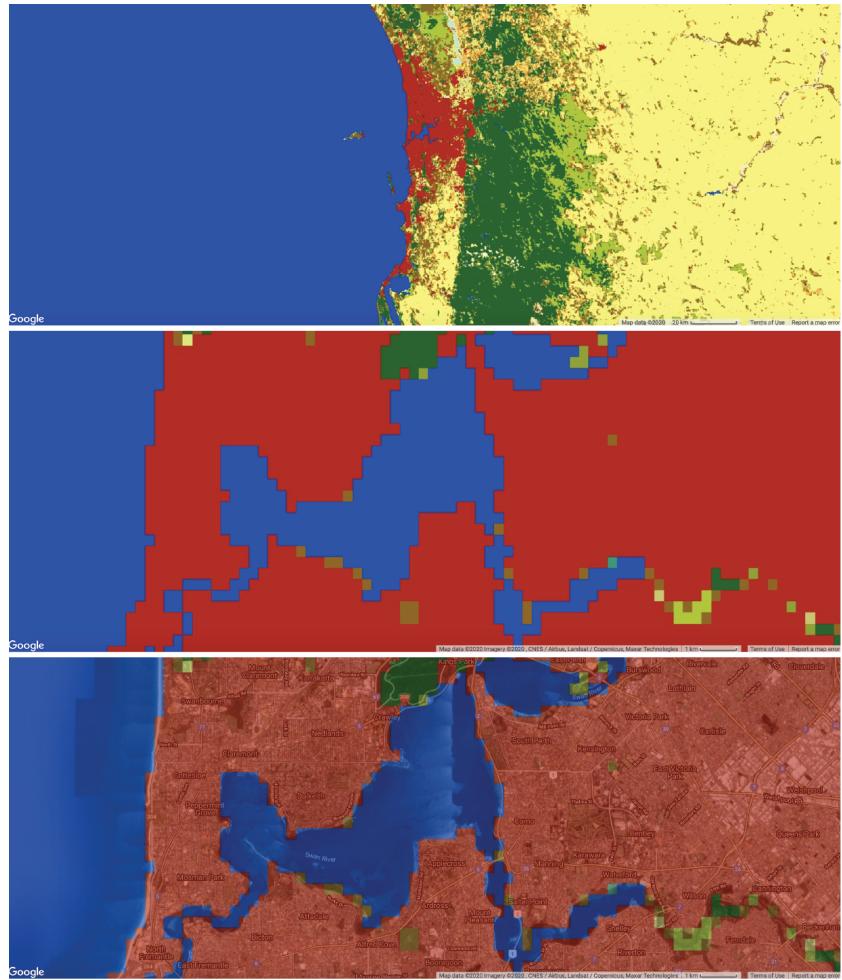


Figure 3.2: Representing land cover using the raster data model.

Add the following code to your GEE script. This will create an image object where each pixel has the value 5 which can be referred to using the variable `img5`. Click on the *Inspector* tab and then click at locations on the map. You should see the value 5 printed in the *Inspector*.

```
// Raster where pixel values equal 5
var img5 = ee.Image(5);
print(img5);
Map.addLayer(img5, {palette:['FF0000']}, 'Raster with pixel value = 5');
```

Alternatively, you can pass a string id into the `ee.Image()` constructor to specify a Google Earth Engine asset (e.g. a Landsat image). Google Earth Engine assets are geospatial data that are stored in cloud databases on Google servers, are available for use in your programs, and are frequently updated - see the available data at the Google Earth Engine data catalog.

The variable `img` in the code block below refers to an `Image` object on the Google servers storing Landsat 8 data. This variable can be used in your program to access, query, and analyse the Landsat data. Pass the variable `img` into the `print()` function to view the Landsat 8 `Image`'s metadata. The `Image` metadata should be printed in the *Console*. Exploring the `Image` metadata in the *Console* is demonstrated in the video below.

```
// Pass Landsat 8 image id into Image constructor*
var img = ee.Image('LANDSAT/LC8_L1T_TOA/LC81130822014033LGN00');
print(img);
```

An `Image` can have one or more bands, each band is a georeferenced raster which can have its own set of properties such as data type (e.g. Integer), scale (spatial resolution), band name, and projection. The `Image` object itself can contain metadata relevant to all bands inside a dictionary object.

Go to the *Console* and you should see the Landsat 8 `Image` has 12 bands. Click on a band and you should see some band specific properties such as its projection (`crs: EPSG:32650`). Click on the `Image properties` to explore metadata that applies to the `Image` such as cloud cover at the time of `Image` capture (`CLOUD_COVER: 11.039999961853027`) or the satellite carrying the sensor (`SPACECRAFT_ID: LANDSAT_8`).

You can visualise the Landsat 8 `Image` on the map display in your browser. To do this you use the `Map.addLayer()` function to request the `Image` stored in the variable `img` on the Google servers to be displayed in your browser. The following code block will visualise an RGB composite map of the Landsat 8 data stored in `img` in your browser's display.

```
/* Define the visualization parameters. The bands option allows us to specify which bands
var vizParams = {
  bands: ['B4', 'B3', 'B2'],
```

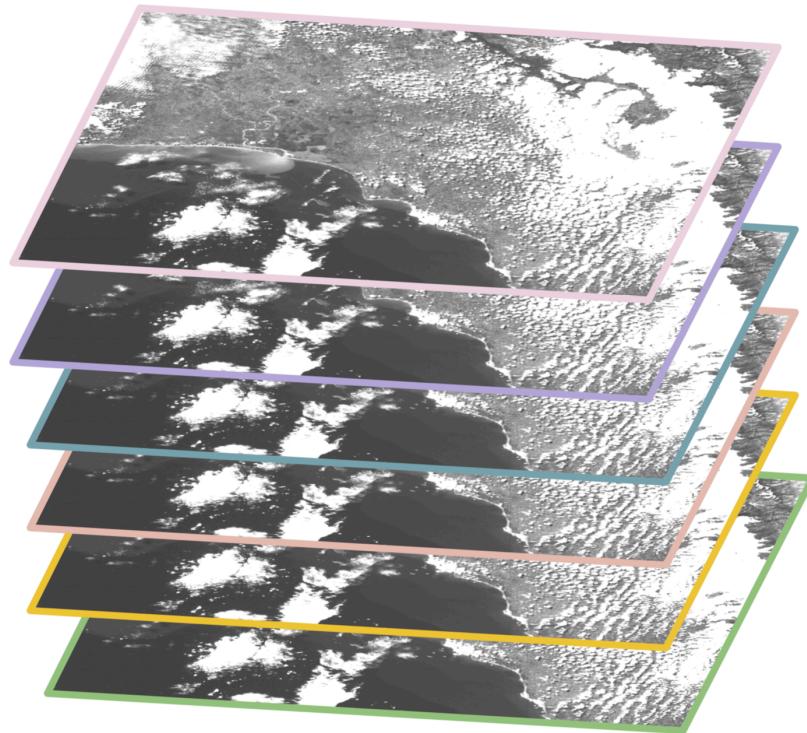


Figure 3.3: Schematic of an Image data structure in Google Earth Engine where an image can contain multiple georeferenced bands (source: [What is Google Earth Engine?](#)).

```

    min: 0,
    max: 0.5,
};

// Centre the display and then map the image
Map.centerObject(img, 10);
Map.addLayer(img, vizParams, 'RGB composite');

```

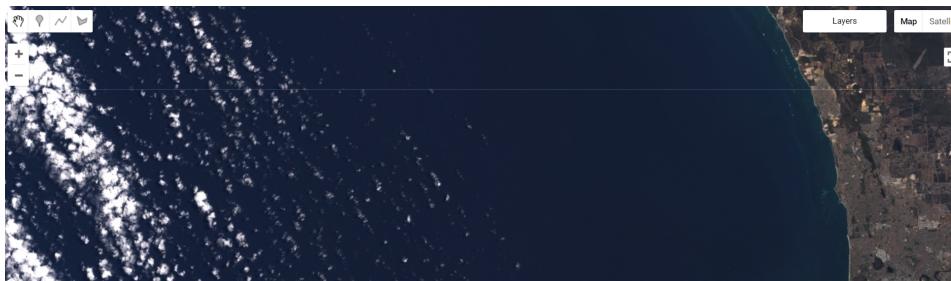


Figure 3.4: Visualising Landsat 8 data as a RGB composite image.

Images in Google Earth Engine

3.4.2 Geometry Objects

The spatial location or extent of vector data is stored as **Geometry** objects. Google Earth Engine implements the **Geometry** objects outlined in the GeoJSON spec:

- Point
- MultiPoint
- LineString
- MultiLineString
- Polygon
- MultiPolygon

To create a **Geometry** object programmatically use the `ee.Geometry.<geometry type>()` constructor (e.g. for a LineString object use `ee.Geometry.LineString()`) and pass the coordinates for the object as an argument to the constructor. Look at the code block below to observe that coordinates for a location in Kings Park are passed as arguments to the `ee.Geometry.Point()` constructor to create a point **Geometry** object (`locationKP`).

```

//location of Kings Park
var locationKP = ee.Geometry.Point(115.831751, -31.962064);
print(locationKP);

```

```
// Display the point on the map.
Map.centerObject(locationKP, 11); // 11 = zoom level
Map.addLayer(locationKP, {color: 'FF0000'}, 'Kings Park');
```

If you explore the metadata for `locationKP` in the *Console* you will see the object has a `type` field which indicates the object is of `Point` type and a `coordinates` field which contains the the coordinates for the point as an array object. The value of the `coordinates` field is an ordered x y pair.

You can create `LineString` objects in a similar way. Here, you can pass the coordinates as an array into the `ee.Geometry.LineString()` constructor. As noted in the GeoJSON spec, coordinates for `LineString` objects are an array of ordered x y pairs.

```
// May Drive as a LineString object
var mayDr = ee.Geometry.LineString(
    [[115.84063447625735, -31.959551722179764],
     [115.8375445714722, -31.957002964307144],
     [115.83303846032717, -31.956201911510334],
     [115.82994855554202, -31.957403488085628],
     [115.827244888855, -31.9606440253292],
     [115.82625783593753, -31.961445039381488],
     [115.82368291528323, -31.96217322791136],
     [115.82127965600588, -31.963811630990566],
     [115.82055009515383, -31.96563204456937],
     [115.82278169305422, -31.96690631259952],
     [115.82325376184085, -31.968471817682193],
     [115.82218087823489, -31.969818858827356],
     [115.82222379357913, -31.970401356984638]]);
print(mayDr);
Map.addLayer(mayDr, {color: '00FF00'}, 'May Drive');
```

`Geometry` objects in Google Earth Engine are by default geodesic (i.e. edges are the shortest path on spherical surface) as opposed to planar (edges follow the shortest path on a 2D surface). You can read more about the difference between geodesic and planar geometries [here](#).

You can also import `Geometry` objects into your scripts by manually drawing them on the map display using the *Geometry Tools*. The *Geometry Tools* are located in the upper left corner of the map display.

The following video illustrates how to use the *Geometry Tools* to create a `Polygon` object representing Kings Park and how to use variable storing the geometry object in your script.

Some things to note:

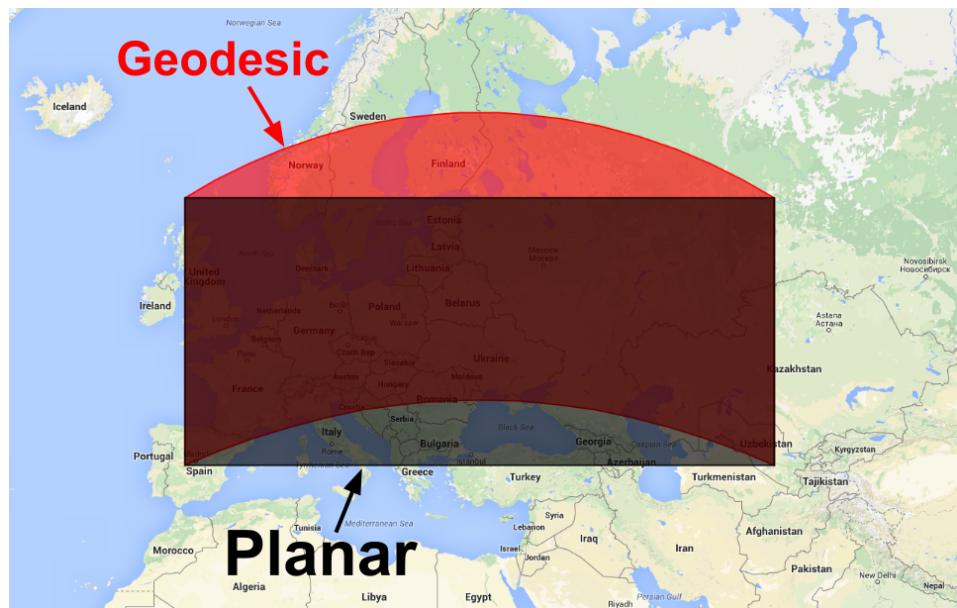


Figure 3.5: Illustration of difference between geodesic and planar geometries (source: [Google Earth Engine: Geodesic vs. Planar Geometries](#)).

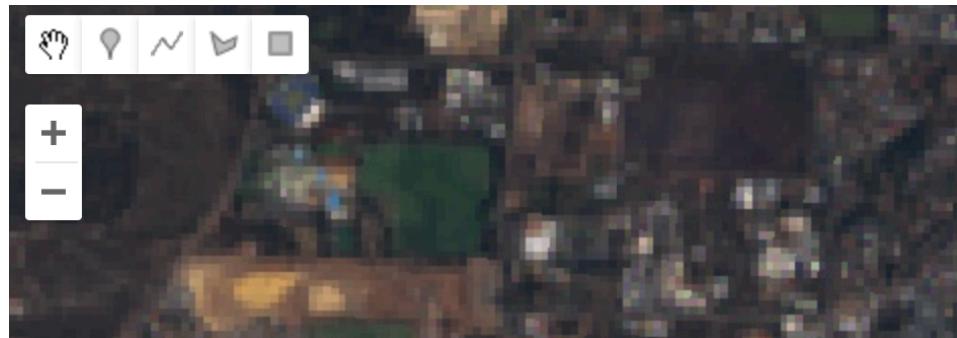


Figure 3.6: Geometry Tools.

- Use the placemark icon to create Point or MultiPoint objects.
- Use the line icon to create Line or MultiLine objects.
- Use the polygon icon to create Polygon or MultiPolygon objects.
- Use the spanner icon to configure how geometry objects that you create using *Geometry Tools* are imported into your script and styling options for display on the map.
- Use + new layer to create new **Geometry** objects. If you want to create separate **Geometry** objects for different geographic features remember to click this button before digitising a new feature.

Geometry Tools.

3.4.3 Features

Geometry objects describe the positional information of vector data; however, there is also a need to represent attribute information about geographic phenomena or entities represented by **Geometry** objects. Vector data in Google Earth Engine which contains geometry data (representing location and shape of geographic phenomenon or entities) and attribute data are **GeoJSON Feature** objects.

A **Feature** object is of type **Feature** with a **geometry** property which contains a **Geometry** object or **null** and a **properties** property which stores a dictionary object of name:value pairs of attribute information associated with the geographic feature represented by the **Geometry** object.

Execute the code block below to convert the **Geometry** object representing Kings Park to a **Feature** object with a **properties** property which with a name attribute. Inspect the **Feature** object in the *Console*.

```
// Create a Feature from the Geometry.
var kpFeature = ee.Feature(locationKP, {name: 'Kings Park'});
print(kpFeature);
```

<pre> ▼ Feature (Point, 1 property) type: Feature ▶ geometry: Point (115.83, -31.96) ▼ properties: Object (1 property) name: Kings Park </pre>	JSON
--	------

Figure 3.7: Kings Park Feature object.

*How would a **Feature** object differ if the Kings Park **geometry** property was of **Polygon** type rather than **point**? Can you convert **kpPoly** to a **Feature** object?*

The `geometry` property of the `Feature` object would contain an array object of coordinates for the outline of the Polygon.

```
// Create polygon Feature
var kpPolyFeature = ee.Feature(kpPoly, {name: 'Kings Park'});
print(kpPolyFeature);

▼ Feature (Polygon, 1 property)                                JSON
  type: Feature
  ▼ geometry: Polygon, 15 vertices
    type: Polygon
    ▼ coordinates: List (1 element)
      ▼ 0: List (15 elements)
        ▷ 0: [115.82986272485354, -31.953725885968524]
        ▷ 1: [115.82419789941409, -31.959041976263876]
        ▷ 2: [115.81810392053225, -31.964794658805754]
        ▷ 3: [115.81836141259768, -31.973240982570168]
        ▷ 4: [115.82291043908694, -31.972804122813177]
        ▷ 5: [115.82660115869143, -31.97091103984024]
        ▷ 6: [115.82831777246096, -31.97244007142741]
        ▷ 7: [115.83381093652346, -31.970328544917116]
        ▷ 8: [115.83775914819338, -31.968581037970846]
        ▷ 9: [115.83999074609378, -31.96494029166108]
        ▷ 10: [115.8428231588135, -31.962828592651096]
        ▷ 11: [115.84488309533694, -31.961663496546766]
        ▷ 12: [115.84333814294436, -31.95409001155769]
        ▷ 13: [115.83338178308108, -31.950885656835897]
        ▷ 14: [115.82986272485354, -31.953725885968524]

  ▼ properties: Object (1 property)
    name: Kings Park
```

Figure 3.8: Kings Park Polygon Feature object.

You can read more about `Feature` objects in Google Earth Engine here.

3.4.4 Collections

Collections in Google Earth Engine comprise groups of related objects. `ImageCollections` contain stacks of related `Image` objects and `FeatureCollections` contain sets of related `Feature` objects. Storing objects together in a collection means that operations can be easily applied to all the objects in the collection such as sorting, filtering, summarising, or other mathematical operations. For example, all Landsat 8 surface reflectance `Images` are stored in an `ImageCollection` with the ID '`LANDSAT/LC08/C01/T1_SR`'. You can pass this string ID into the `ee.ImageCollection()` constructor to import all Landsat 8 surface reflectance `Images` into your program.

If you were creating a program to monitor land surface changes over Kings Park in 2018, you might want to import an `ImageCollection` of all Landsat 8 `Images` into your program and then filter the `ImageCollection` for Landsat 8 scenes that intersect with the extent of Kings Park and were captured in 2018. The following code block demonstrates this. You can then apply subsequent analysis or summary operations to the `ImageCollection` stored in the variable `18ImCollKP`.

```
// Landsat 8 Image Collection
var 18ImColl = ee.ImageCollection("LANDSAT/LC08/C01/T1_SR");

// Filter Image Collection for 2018 and Images that intersect Kings Park
var 18ImCollKP = 18ImColl
  .filterBounds(kpPoly)
  .filterDate("2018-01-01", "2018-12-31");
print(18ImCollKP);
```

You should find 45 Landsat 8 surface reflectance `Images` that intersected with Kings Park in 2018. You can inspect all the `Images` in the `ImageCollection` `18ImCollKP` in the *Console*. The ability to store spatial data in collections makes creating programs that need to access and analyse big geospatial data easier.

You have already created your own `ImageCollection` that contains only the Landsat 8 `Images` for the spatial and temporal extent of interest to you (Kings Park in 2018). Now you can easily apply a range of functions and operations to all the `Images` in the `ImageCollection`. For example, you could apply a function that identifies maximum greenness observed at each pixel in 2018 to analyse spatial variability in vegetation cover. You will learn how to apply functions to `Images` in `ImageCollections` in subsequent labs.

You can find more information on `ImageCollections` here and `FeatureCollections` here.

How would you represent multiple weather stations and observations recorded at these stations as a `FeatureCollection`?

Each weather station would be a `Feature` object in the `FeatureCollection`. Each weather station `Feature` would have a `geometry` property containing a Point `Geometry` object representing the location of the station and a `properties` property containing objects of name:value pairs of weather observations for a given day.

```
// Example structure of weather stations Feature Collection
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
```

```

▼ ImageCollection LANDSAT/LC08/C01/T1_SR (45 elem... JSON
  type: ImageCollection
  id: LANDSAT/LC08/C01/T1_SR
  version: 1595665411245719
  bands: []
  ▼ features: List (45 elements)
    ▶ 0: Image LANDSAT/LC08/C01/T1_SR/LC08_112082_201...
      type: Image
      id: LANDSAT/LC08/C01/T1_SR/LC08_112082_20180105
      version: 1522740042342356
      ▼ bands: List (12 elements)
        ▶ 0: "B1", signed int16, EPSG:32650, 7721x778...
        ▶ 1: "B2", signed int16, EPSG:32650, 7721x778...
        ▶ 2: "B3", signed int16, EPSG:32650, 7721x778...
        ▶ 3: "B4", signed int16, EPSG:32650, 7721x778...
        ▶ 4: "B5", signed int16, EPSG:32650, 7721x778...
        ▶ 5: "B6", signed int16, EPSG:32650, 7721x778...
        ▶ 6: "B7", signed int16, EPSG:32650, 7721x778...
        ▶ 7: "B10", signed int16, EPSG:32650, 7721x77...
        ▶ 8: "B11", signed int16, EPSG:32650, 7721x77...
        ▶ 9: "sr_aerosol", unsigned int8, EPSG:32650, ...
        ▶ 10: "pixel_qa", unsigned int16, EPSG:32650, ...
        ▶ 11: "radsat_qa", unsigned int16, EPSG:32650...
      ▶ properties: Object (23 properties)
        ▶ 1: Image LANDSAT/LC08/C01/T1_SR/LC08_112082_201...
        ▶ 2: Image LANDSAT/LC08/C01/T1_SR/LC08_112082_201...
        ▶ 3: Image LANDSAT/LC08/C01/T1_SR/LC08_112082_201...
        ▶ 4: Image LANDSAT/LC08/C01/T1_SR/LC08_112082_201...
        ▶ 5: Image LANDSAT/LC08/C01/T1_SR/LC08_112082_201...
        ▶ 6: Image LANDSAT/LC08/C01/T1_SR/LC08_112082_201...
        ▶ 7: Image LANDSAT/LC08/C01/T1_SR/LC08_112082_201...
        ▶ 8: Image LANDSAT/LC08/C01/T1_SR/LC08_112082_201...

```

Figure 3.9: ImageCollection of all Landsat 8 scenes that intersect with Kings Park in 2018.

```

"properties": {
    "station-id": XXXX,
    "date": "01-01-2018",
    "temperature": 29
},
"geometry": {
    "type": "Point",
    "coordinates": [
        119.1796874999999,
        -26.74561038219901
    ]
}
},
{
    "type": "Feature",
    "properties": {
        "station-id": XXXX,
        "date": "02-01-2018",
        "temperature": 27
    },
    "geometry": {
        "type": "Point",
        "coordinates": [
            124.1015625,
            -29.535229562948455
        ]
    }
}
]
}

```

1. Can you use the **Geometry Tools** to create a **LineString Geometry** object representing a road? and 2. can you convert the **LineString Geometry** object to a **Feature** object by giving it a **road_name** property?

Create a **LineString Geometry** object to represent a road and create a **Feature** object with a **road_name** property.

Point , Line , and Polygon marker symbols obtained from Google Earth Engine Developers Guide

4 Introduction

This lab will introduce tools for spatial data visualisation in Google Earth Engine. Data visualisation is the activity of relating observations and variation in your data to visual objects and properties on a display. How you relate your data values to display objects will determine what insights you can derive from your data, what patterns and relationships it will reveal, and what messages you can convey from your data.

This lab will focus on the use of colour to represent features or patterns in geospatial data but you should also be aware of other visual properties of your map display that can be adjusted; for example, you don't want to use a line width so thick that it obscures variation in polygons represented by colour fill values.

4.0.1 Setup

Create a new script in your *labs-gee* repository called *lab4b*. Enter the following comment header to the script.

```
/*
Lab 5b
Author: Test
Date: XX-XX-XXXX

*/
```

4.1 Colour Theory

One of the key aspects of spatial data visualisation is using colour to represent variation in data values. This process involves mapping data values to colours and then assigning colours to objects (e.g. points, lines, polygons, or pixels) on your display. As stated by Wilke (2019) there are three main uses of colour in data visualisation:

1. distinguish groups in your data
2. represent data values
3. to highlight features in your data

4.1.1 Colour

Color is defined by the characteristics of a mix wavelengths of light in the visible spectrum Wickham (2020). A particular colour is defined by levels of intensity of light in different parts of the visible spectrum (e.g. yellow is a mixture of light in red and green wavelengths). The human eye can distinguish millions of colours CRCSI (2017); thus, colour is useful for representing variation, patterns, or interesting features in your data.

An individual colour can be described in terms of hue, value, or chroma (CRCSI, 2016):

- **Hue:** the attribute commonly associated with colour. Hues have an order which follows the colours of the spectrum and spectral hues can be created by mixing adjacent wavelengths. Purple, for example, is a non-spectral hue as it is a mixture of blue and red whose wavelengths are not adjacent.
- **Brightness (Value / Intensity):** is the perceived brightness of a colour and is related to the amount of energy in all wavelengths of light reflected by an object.
- **Chroma (Saturation):** is related to the purity of a colour. It can be thought of as the distribution of intensity in wavelengths around the wavelength of average (peak) intensity of light reflected by an object. Adding white, grey, or black to light reduces the chroma and produces pastel colours.

4.1.2 Colour Models / Colour Spaces

Additive Primaries (RGB Cube)

Colour is represented by combinations (addition) of red, green, and blue light. Red, green, and blue are primary colours and combine to form white. An absence of red, green, and blue is black. Secondary colours can be formed by the addition of primary colours of varying intensities (e.g. yellow is the addition of red and green, magenta is the addition of red and blue, and cyan is the addition of green and blue). A related colour model uses subtractive primary colours (yellow, magenta, or cyan) which are subtracted from a white background to produce different colours.

Colour can be represented by coordinates in 3D space using the RGB colour cube where each dimension is represented by a primary colour. The intensity of a colour is represented by its position along a dimension. Grey colours, equal intensities of each of the primary colours, is represented by the diagonal axis from black (absence of primary colours) to white (complete presence of the spectrum of colours).

Hue, Saturation, Intensity (HSI) Colour Space

Colour coordinates in the RGB cube colour model can be transformed to coordinates in Hue, Saturation, and Intensity (HSI) space. Hue represents saturated pure colours as angular values surrounding a central axis of achromatic colour with black at the bottom and white at the top (red hue = 0 or 360; green = 120; blue = 240). This central axis represents the lightness of the

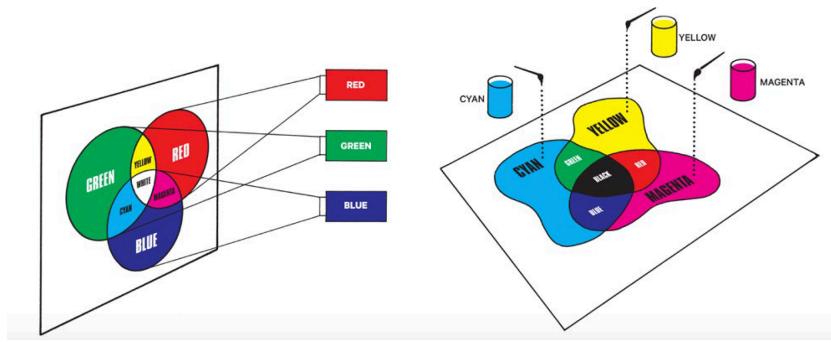


Figure 4.1: Additive and subtractive colour models (source: [CRCSI \(2017\)](#)).

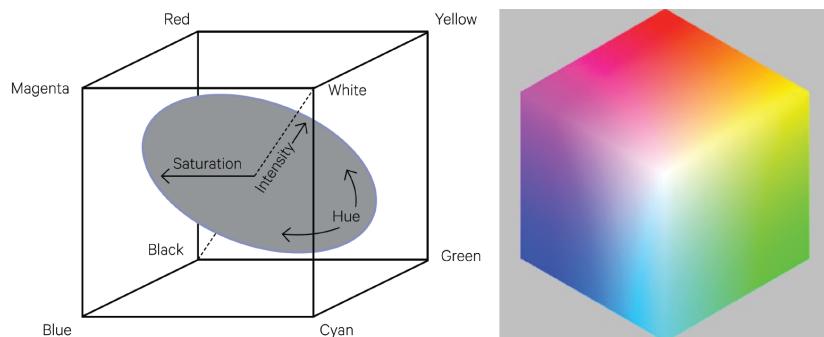


Figure 4.2: RGB Colour Cube (source: [CRCSI \(2017\)](#)).

colour (black = 0 % and white = 100 %). The saturation of a colour represents the amount of grey in the colour (grey = 0 % and pure colour = 100 %).

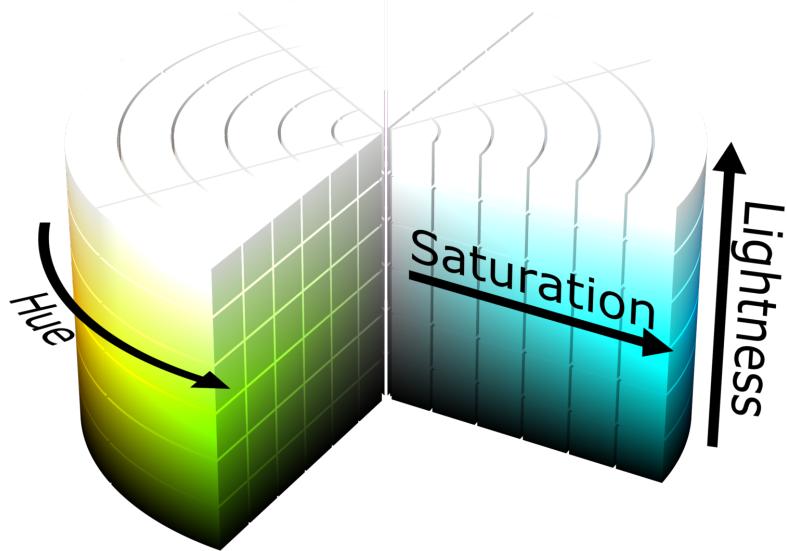


Figure 4.3: HSL colour model (source: [Wikimedia Commons](#)).

Colour on Computer Displays

Computer displays consist of red, green, and blue sub-pixels, which when activated with different intensities, are perceived as different colours. The range of colours that can be displayed on a computer display is called the *gamut*. Colour in computer programs is represented as a three byte hexadecimal number with byte 1 corresponding to red, byte 2 corresponding to green, and byte 3 corresponding to blue. Each byte can take the range of 00 to FF in the hexadecimal system (or 0 to 255 in decimal). 00 indicates the absence of that colour and FF indicates saturation of that colour. FF0000 is red, 00FF00 is green, and 0000FF is blue.

Use this RGB colour picker to see how changing red, green, and blue intensities creates hexadecimal number representations of the colour.

Similarly, you can use this HSL colour picker to see how changing hue, saturation, and lightness results in different hexadecimal number representations of colour.

How would you represent pure yellow as a 3 byte hexadecimal number?

FFFF00

You want to represent a flooded location as a blue polygon. You want this object to stand out and appear bright. What would be suitable hue, saturation, and lightness values for this object? What hexadecimal number would represent these HSL values?

Choose a colour with a distinctive blue hue (around 240), pure colour (saturation close to 1 or 100 %), and a lightness with minimal white or black tints and shades (around 0.5 or 50 %). Depending on the basemap and colour of other objects on the map you could adjust these values to maximise visual discrimination of the flooded object. #0000FF

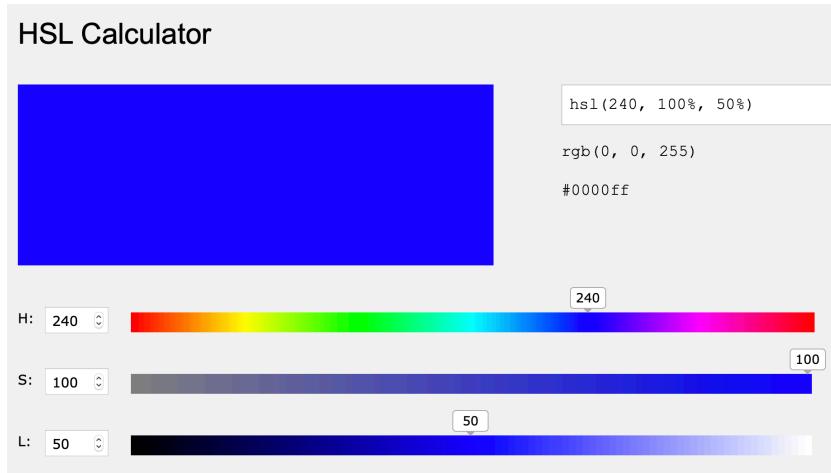


Figure 4.4: Distinctive blue for displaying a flood object (source: [W3Schools](#)).

4.1.3 Choosing Colours

Choose colours and colour palettes that account for colour blindness. There are online tools that you can use to simulate colour blindness (e.g. color oracle).

Choose colours / colour palettes that have a logical interpretation (e.g. greens for vegetation; blue for wetter areas; red for hot).

If your data values don't have a natural order (e.g. land cover data) don't use a colour scale that implies order (e.g. dark to light colours, low to high saturation, warm to cool hues). Section 4 in Wilke (2019) outlines key points to consider when choosing a colour scale to represent variation in your data and common pitfalls to avoid when using colour for data visualisations.

If there is order in your data, *sequential* colour palettes which indicate large and small values and distance between values should be used. Sequential colour palettes can be single-hue or multi-hue. The top group of colour palettes in the below figure depict sequential colour palettes from Color Brewer.

In some cases, your data might have a logical midpoint value (e.g. median) and you want your colour palette to represent variation away from this value. In this instance a *diverging* colour palette should be used (see the bottom group of colour palettes in the below figure).

Qualitative colour palettes (middle group in the figure below) assign colours to data values or categories where each colour appears equivalent and distinct. They should be used for categorical and unordered data.

The Color Brewer website is a good resource for generating colour palettes for spatial data which also account for colour blindness.

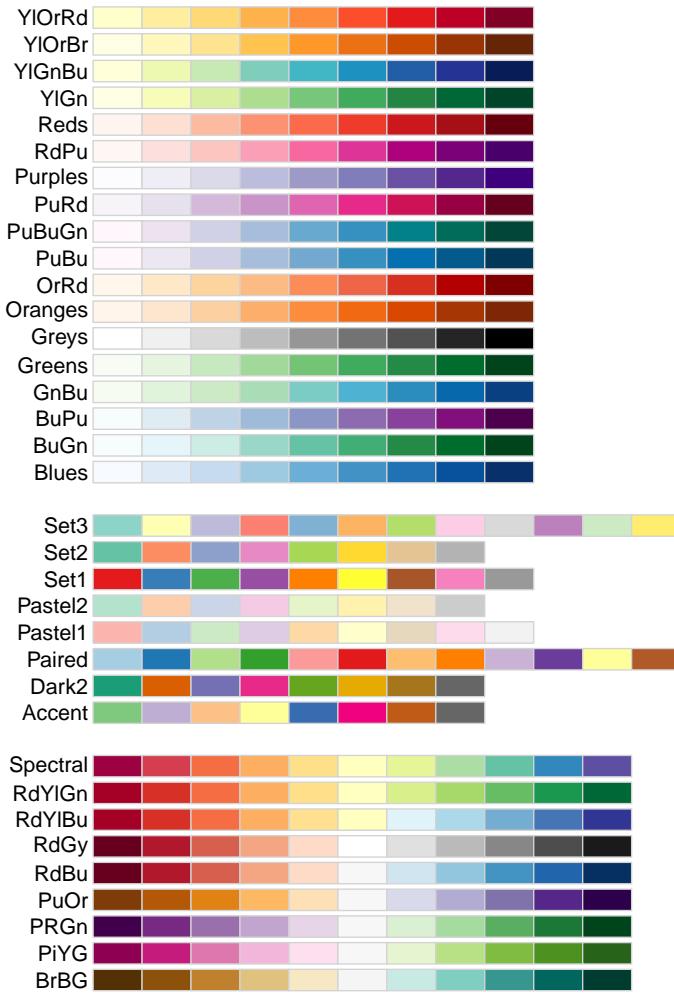


Figure 4.5: ColourBrewer scales (source [R 4 Data Science](#))

Be aware of context and how the relationship between objects on a map display can affect how they are perceived by human eyes. For example, thin geometric features are not easily detected when represented in blue hues, the perceived hue of an object changes with background colour (blue features are hard to detect on black backgrounds, yellow is hard to detect on white

backgrounds), and the perceived chroma of an object is related to feature size. Excursus 5.2 (CRCSEI, 2017; p. 75) highlights how color composition can obscure or misrepresent variation in spatial data.

Go to the Color Brewer website and choose a colour palette to represent spatial variation in precipitation? Justify why this is a suitable colour palette. Look out for how to copy the hexadecimal values representing the colours in your palette as a JavaScript array.

Diverging colour palette emphasises variation between wet and dry areas with red representing dry areas and blue wet areas. Pure red and blue will highlight extreme values. People who are colour blind will be able to distinguish variation in precipitation.

Selecting a colour palette to represent precipitation using Color Brewer.

4.2 Colour to Highlight Features

You can use colour to highlight certain features or patterns in your data, or, as is often the case in spatial data visualisation, make certain features stand out from a base map. Look at the colours used in Google Maps and also the typical styling for open street maps; they use subdued and relatively impure pastel colours. You can use strong and pure colours to visualise features that stand out against these backgrounds.

In the map display zoom and scroll to an area of parkland in Perth. Use the *Geometry Tools* to digitise a Polygon object to represent green space or a park. Change the name of the variable storing the Polygon **Geometry** object to **park**. Add the Polygon object in **park** to the map display with the green colour: #99e6b3. To do this you need to use the following functions:

- **Map.addLayer()** - adds a Google Earth Engine object (i.e. spatial data stored in a variable) to the map display. You pass the variable you wish to display (i.e. **park**), a dictionary of visualisation parameters that specify how the spatial data is visualised (i.e. `{color:"#99e6b3"}`), and a string name of the layer (i.e. "Park") into the **Map.addLayer()** function as arguments.
- **Map.centerObject()** - centers the map display on the location of an object passed as a variable to function (i.e. **park**) with the second argument being a number specifying the zoom level (i.e. 15).

```
// Add Polygon geometry object representing a park to the map
Map.centerObject(park, 15);
Map.addLayer(park, {color:"#99e6b3"}, "Park");
```

Use Geometry Tools to create a Polygon object representing a park and display in green.

Assess the use of the green colour: #99e6b3 to represent the park on Google Maps? Can you pick a different colour to visualise the park and justify your choice?

The green colour: #99e6b3 has a logical relationship with greenspace or parks - the real world geographic entity it is representing. However, the green colour could be edited to make it stand out from the basemap. For example, #99e6b3 has a hue of 140, a saturation of 60 %, and a lightness of 75 %. This indicates the hue of #99e6b3 is a spectral mix of blue and green, a saturation of 60 % indicates that the colour is not pure and includes some grey, and a lightness greater than 50 % indicates that white tints are introduced to the colour. You could reduce the white tints by reducing the lightness value to 50 %, increase the saturation, and change the hue value to be closer to primary green (hue = 120).

Use the HSL Calculator to adjust hue, saturation, and lightness values and find a suitable colour to highlight your park against the Google Maps basemap. Edit your script to visualise your park object in a more distinctive green colour. You should include a different hexadecimal number in the value referenced by the color key in the visualisation parameters. The code snippet below is an example using the hexadecimal number representation of primary green 00FF00.

```
// Use colour to highlight the park against the basemap
Map.addLayer(park, {color:"#00FF00"}, "Park - Primary Green");
```

4.3 Colour to Represent Groups

You can use colour to represent categorical groups in your data. You do not want to choose a colour palette that implies order in your data. Use a qualitative colour palette that ensures groups in your data can be distinguished from each other and the colours are perceived as equivalent (Wilke, 2019). Also, if it makes sense with your data choose colours that have a logical relationship with the group or category in your data (e.g. vegetated land cover classes as green in a land cover map).

Execute the below code. This loads the tree raster layer from the 2016 Urban Monitor data (Caccetta, 2012) and displays tree pixels in black. The Urban Monitor data is derived from 20 cm spatial resolution multispectral aerial images collected and processed by CSIRO (resampled to 40 cm spatial resolution here). Tree land cover is a categorical group in your data (the only group in this raster data).

Is black the most suitable colour to represent trees?

```
// UM Tree
var umTree = ee.Image("users/jmad1v07/gee-labs/um-lake-claremont-tree-2016");
Map.centerObject(umTree, 15);
Map.addLayer(umTree, {min: 0, max: 1, palette:["000000"]}, "UM Tree - black colour");
```

*Display the Urban Monitor tree **Image** on your map choosing a more suitable colour than black for display.*

```
// UM Tree - green  
Map.addLayer(umTree, {min: 0, max: 1, palette:[ "#009900" ]}, "UM Tree - better colour????");
```

Change the base map to satellite. You can do this by clicking the satellite button in the top right corner of the map display.

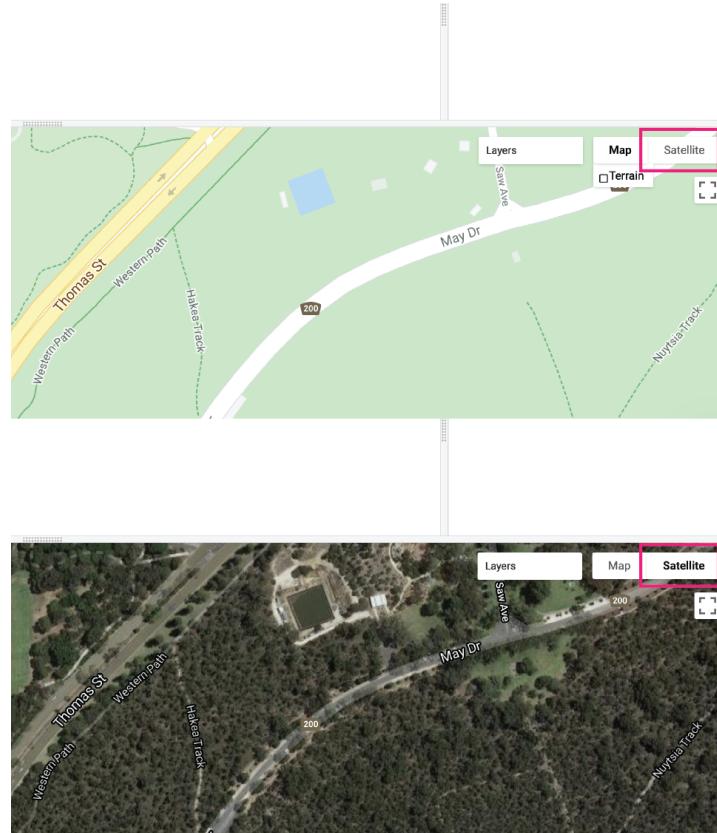


Figure 4.6: Change to satellite basemap.

Change the base map to satellite. Choose a suitable colour to represent trees so that they stand out against the satellite base map? Justify your choice of colour.

You could use Color Brewer to help pick out a colour that is distinct from the green and brown colours that dominate the base map but are safe for colour blind viewers. One option could be to use a blue colour; this will be distinct from green but the trade off is that blue is not logically associated with trees.

```
// UM Tree - blue
Map.addLayer(umTree, {min: 0, max: 1, palette:["#009E73"]}, "UM Tree - satellite basemap")
```

Tip: You can use the Layers widget in the top right corner of the map display to turn layers on and off, change a layer's opacity, and also control other visualisation and display parameters.

Change layers on map display.

You can also specify colours to visualise multiple categories in spatial data. If you execute the below code you will assign one of the colours specified in the array assigned to the `palette` key in the `igbpVis` object to one of land cover classes in the 500 m spatial resolution MODIS MCD12Q1 land cover data for 2019. Each pixel has a value from 1 to 17 which relates to the land cover classes (categories) of the Annual International Geosphere-Biosphere Programme (IGBP) classification.

```
// Visualise MODIS MCD12Q1 Land Cover data for 2019
var lcModis = ee.Image("users/jmad1v07/gee-labs/lc-mcd12q1-2019-lc-type-1");

// Define a palette for the distinct land cover classes.
var igbpVis = {
  min: 1.0,
  max: 17.0,
  palette: [
    '05450a', '086a10', '54a708', '78d203', '009900', 'c6b044', 'dcd159',
    'dade48', 'fbff13', 'b6ff05', '27ff87', 'c24f44', 'a5a5a5', 'ff6d4c',
    '69fff8', 'f9ffa4', '1c0dff'
  ],
};

Map.centerObject(lcModis, 9);
Map.addLayer(lcModis, igbpVis, "Land Cover MODIS MC12Q1 2019");
```

4.4 Colour to Represent Data Values

Up until now you have been associating discrete values or categories to colours that are rendered on your display. However, if your data is continuous you will need to relate your data values to a range of colours in a palette. How you relate data values to a range of colours will determine how variation in your data is visualised.

The 2016 Urban Monitor data introduced above also includes a vegetation height raster layer (Caccetta, 2012). Again, this data is resampled to a 40 cm spatial resolution and each pixel

value represents the height of vegetation in metres if a pixel was vegetated. Height is a continuous variable with a clear order from low to high.

Would a sequential or diverging colour palette be suited to visualising the height of vegetation?

Sequential. There is not an obvious mid-point from which height diverges. Height increases from 0 m.

Execute the below code to visualise the Urban Monitor vegetation height data with a colour palette to represent low vegetation height in black ("000000") through to red colours for high vegetation ("0000FF").

```
// UM Vegetation Height
var umVht = ee.Image("users/jmad1v07/gee-labs/um-lake-claremont-vht-2016");
Map.centerObject(umVht, 15);
Map.addLayer(umVht, {min: 0, max: 100, palette:["000000", "FFFF00", "FF0000"]}, "UM Vegeta
```

You should see something like the below image in your map display. It was just mentioned that the colour palette relates high vegetation to the colour red but tree canopy is appearing in black-yellow colours. Why is this?

If you look at the dictionary of visualisation parameters you passed as an argument to the `Map.addLayer()` function you will see `min: 0, max: 100`. This means you are relating a pixel value of 0 metres to black colours and a pixel value of 100 metres to pure red colour. Pixel values between 0 to 100 are related to the range of colours spanning the black-yellow-red colour palette that can be displayed by your screen. A tree height of 100 metres exceeds the expected tree height in the Lake Claremont area in Perth. This means that data values are assigned to display colours that are not in your dataset; in other words you are not utilising the range of display colours to highlight variation in your data values.

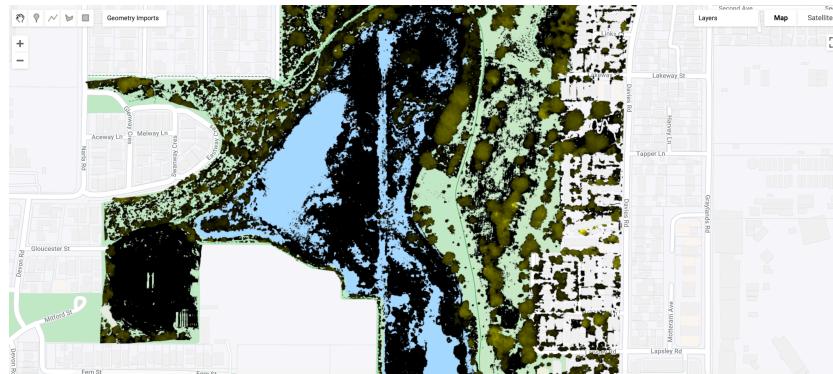


Figure 4.7: Visualising Urban Monitor vegetation height data with a black-yellow-red colour scale.

What would be appropriate min and max vegetation height values to assign to the minimum and maximum display range to highlight variation in vegetation height? Adjust the visualisation parameters to display the vegetation height data to emphasising variation in vegetation height.

```
// UM Vegetation Height - adjusted display range  
Map.addLayer(umVht, {min: 0, max: 30, palette:["000000", "FFFF00", "FF0000"]}, "UM Vegetat
```

How effective do you think visualising vegetation height with a black-yellow-red colour palette is? Do you usually associate red with high vegetation (i.e. tree canopy)? Choose a different colour palette to represent vegetation height and justify your choice.

Use a single hue green colour palette with darker green associated with higher vegetation. This colour palette was selected using Color Brewer. Think about the limits of this colour palette when using it with the Google base map.

```
// UM Vegetation Height - adjusted colour palette  
Map.addLayer(umVht, {min: 0, max: 30, palette:[ '#edf8e9', '#c7e9c0', '#a1d99b', '#74c476', '#4
```

The following code will load average land surface temperature (LST; Kelvin) across Perth for the summer months (December, January, and February) for the years 2014 to 2019. This LST data is derived from Landsat 8 observations and computed using the algorithm of Jiménez-Muñoz et al. (2014).

```
// Visualise Landsat 8 land surface temperature (K)  
var lstLandsat8 = ee.Image("users/jmad1v07/gee-labs/landsat8-lst");
```

Sensible minimum and maximum data values to assign to the limits of the display colour range are 295 to 315 K. Can you create a colour palette to visualise variation in land surface temperature and map the Landsat 8 data in the variable `lstLandsat8` using this colour palette?

```
// Example colour palette to visualise land surface temperature data  
Map.centerObject(lstLandsat8, 12);  
var lstVisParam = {min:295, max:315, palette:["000066", "00ffff", "ffff00", "ff0000"]};  
Map.addLayer(lstLandsat8, lstVisParam, "Surface Temperature (K)");
```

4.5 Multiband Images

Images in Google Earth Engine can have multiple bands where each band comprises georeferenced raster data. As discussed above, computer displays represent colour through varying

the intensity of sub-pixel displays of red, green, and blue light. Variability in data values in multiband `Images` can be visualised by relating data values in one band of the `Image` to the intensity of one the primary colours on the computer display. Visualising a multiband `Image` in this way creates an additive RGB or colour composite image - it is called a composite image because each pixel is a composite of red, green, and blue light Excursus 5.2 (CRC, 2017).

4.5.1 True Colour Composite Image

Multiband `Images` are common in remote sensing where each band contains measures of spectral reflectance in different wavelengths. When a sensor records spectral reflectance in the visible blue, green, and red wavelengths, variation in these bands can be related to intensities of blue, green, and red on the computer display. This should display features on your map in colours similar to how you would see these spatial features if you were looking down towards the Earth's land surface.

The Urban Monitor data contains a 4-band multispectral `Image` corresponding to spectral reflectance measures in the blue, green, red, and near infrared (NIR) wavelengths. You can use the blue, green, and red bands in the Urban Monitor data to display this data as a true colour composite RGB image on your map display.

The following code will visualise a multiband Urban Monitor `Image` as a colour composite on your display. As you have done previously, you pass a dictionary object of visualisation parameters (stored in the variable `visTrueColourParams` in the below snippet) to the `Map.addLayer()` function. The key:value pairs inside this dictionary object determine how the bands in the multiband `Image` are rendered as a colour composite.

You have an array `["b1", "b2", "b3"]` assigned to the `bands` key. This array specifies which bands should be assigned to red, green, and blue intensities on the display. The Urban Montior multispectral data has band 1 (`"b1"`) storing red spectral reflectance, band 2 (`"b2"`) storing green spectral reflectance measures, and band 3 (`"b3"`) storing blue spectral reflectance measures. NIR is stored in band 4 (`"b4"`) in the Urban Monitor product. This band ordering does not correspond to the order of wavelengths so be careful to assign the band storing red reflectance values to red on the display.

You have assigned a value of zero spectral reflectance to the minimum of the display range and a spectral reflectance value of 0.3 to the maximum of the display range. Most of the features in this Urban Monitor scene have spectral reflectance values in this range even though the maximum possible spectral reflectance is 1. Assigning these values to the limits of the display range ensures you maximise the use of display colours to discriminate features in your image.

```
// UM multispectral
var umDom = ee.Image("users/jmad1v07/gee-labs/um-lake-claremont-dom-2016");
```

```

// UM True Colour Composite
// Define the visualization parameters.
var visTrueColourParams = {
  bands: ["b1", "b2", "b3"],
  min: 0,
  max: 0.3
};
Map.centerObject(umDom, 17);
Map.addLayer(umDom, visTrueColourParams, "UM True Colour Composite");

```

Your visualisation of the Urban Monitor Image data as a true colour composite should look like the figure below. The colours on the display clearly correspond to how we would perceive this scene with our eyes if we were looking down on it.



Figure 4.8: Urban Monitor - true colour composite.

4.5.2 False Colour Composite Image

You can associate other **Image** bands to intensities of red, green, and blue light on your display even if these bands do not actually measure spectral reflectance in the red, green, and blue wavelengths. This is a false colour composite image. Some features of Earth's land surface have distinct reflectance characteristics in portions of the electromagnetic spectrum outside the visible wavelengths. For example, vegetation has high reflectance in the NIR wavelengths.

A common false colour composite image associates NIR reflectance with red intensities on your display, red spectral reflectance with green on your display, and green spectral reflectance with blue on your display Excursus 5.2 (CRCSE, 2017). This false colour composite will visualise vegetation in red shades (due to high reflectance in the NIR wavelengths), red soils as green (due to soils having high reflectance in the red wavelengths, and water as blue (due to water having relatively higher reflectance in the green wavelengths)).

Bare soil has high spectral reflectance in the blue, green, red, and NIR wavelengths. What colour will bare soil be visualised in on your map display? (Hint: use this RGB colour picker to create a colour that is composed of high intensities in red, green, and blue).

White and light shades. White light is a combination of reflectance across all wavelengths.

In a true colour composite image cloudy areas will be displayed with high intensities in the red, green, and blue display values. Why is this?

Clouds are typically white which means they have high spectral reflectance in all visible wavelengths. Therefore, high values of blue, green, and red spectral reflectance measures will be related to high intensities of blue, green, and red on the display.

```
// UM False Colour Composite
var visFalseColourParams = {
  bands: ['b4', 'b1', 'b2'],
  min: 0,
  max: 0.3
};
Map.addLayer(umDom, visFalseColourParams, "UM False Colour Composite");
```

Your display of the Urban Monitor **Image** as a false colour composite should look like the display below. You can clearly see vegetation in shades of red due the red colour on the display representing NIR spectral reflectance.

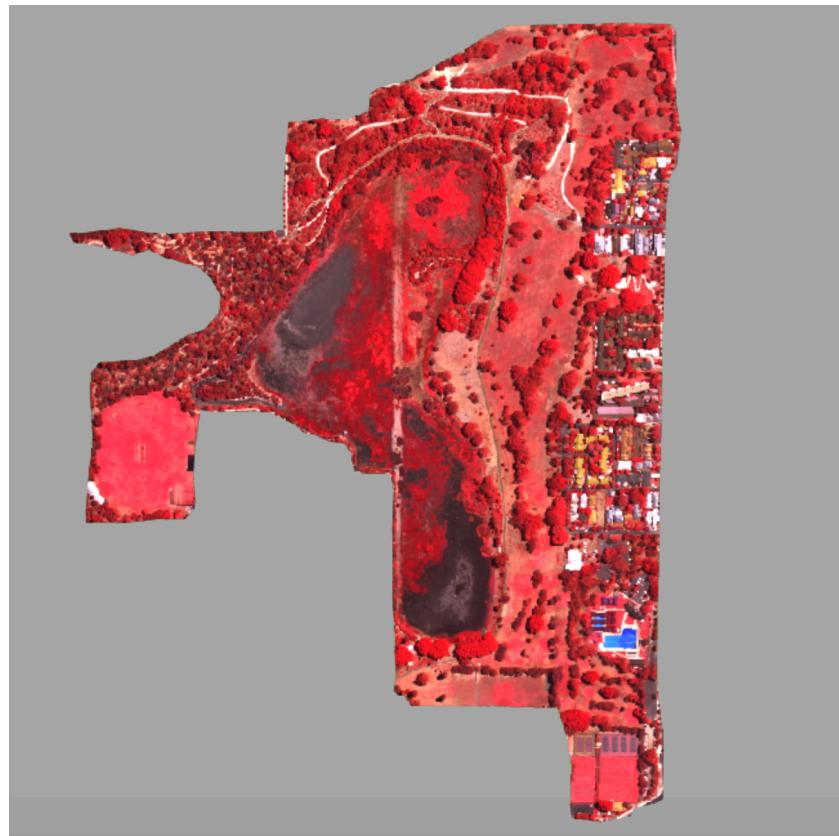


Figure 4.9: Urban Monitor - false colour composite.

5 Introduction

This lab will introduce data transformation and manipulation operations for vector data.

Data transformation operations transform data into a format ready for subsequent analysis. For non-spatial data, common data transformation operations include filtering and subsetting observations, creating new variables through applying functions to existing variables, joining and combining data, and summarising or aggregating data. For example, consider the following data transformation operations that could be applied to climate data:

1. *Filter and subset*: filter the data for all observations that occurred within a time-period (e.g. since the year 2010) and select a variable of interest (e.g. temperature in Fahrenheit).
2. *Create a new variable*: create a new variable (temperature in Celsius) by converting the observations of temperature in Fahrenheit using the equation: $T_{(C^\circ)} = (T_{(F^\circ)} - 32) * 5/9$.
3. *Join and combine*: join your temperature data to precipitation data (e.g. joining on weather station ID and date and time of weather observation).
4. *Summarise*: summarise your data by computing the average daily temperature and precipitation for each weather station.

Data transformation operations can be applied to the non-spatial attribute information associated with spatial **Feature** objects. However, most data transformation operations have a spatial equivalent. For example:

1. *Filter and subset*: filter a **FeatureCollection** based on geographic location and topological relations (e.g. selecting only weather stations that intersect with the Western Australia extent or only weather stations contained within a 10,000 km radius from Perth).
2. *Create a new variable*: the **Geometry** objects can be converted into spatial objects with new shapes or extents (e.g. through applying a 1 km buffer operation).
3. *Join and combine*: spatial data can be combined through spatial join operations (e.g. join weather station data with Statistical Area Level 1 (SA1) polygon geometries based on the intersection between a weather station's location and the extent of the SA1)
4. *Join and combine AND create a new variable*: vector data can be combined with raster data via zonal statistics (e.g. computing the area of each land cover class within a polygon **Geometry**).
5. *Summarise*: you could perform aggregations or summary operations for observations within a spatial extent (e.g. compute the average temperature of all weather stations within each SA1 extent).

This lab will demonstrate several spatial and non-spatial data transformation operations. These operations will form part of a workflow to address the question: *Which Perth university has the greenest and coolest campus?*

5.0.1 Which Perth university has the greenest and coolest campus?

You will start with a `FeatureCollection` containing `Feature` objects of a building (a polygon `Geometry` object) and attribute information indicating if the building is part of a university (the `building` property), the name of the university (the `uni_name` property), and a building ID (the `osm_id` property). The building footprint data are from Open Street Map and include buildings from the University of Western Australia (UWA) Crawley Campus, Curtin University Bentley Campus, Murdoch University Perth (Murdoch) Campus, Edith Cowan University (ECU) Mount Lawley Campus, and some non-university buildings near to each campus.

You will use the area of tree canopy cover within a certain distance of a building as an indicator of greenness; the tree canopy data is in raster format and derived from the Urban Monitor data (Caccetta, 2012).

The temperature data is also in raster format and is a measure of average summer (December, January, and February) land surface temperature (LST) derived from Landsat 8 (Jiménez-Muñoz et al. 2014).

You will need to produce summary statistics that describe the greenness and temperature of each university campus.

Your analysis will comprise the following steps:

1. *Filter and subset*: filter the building footprint `FeatureCollection` to include only university buildings.
2. *Create a new variable*: perform a buffer operation on each university building footprint `Geometry` object.
3. *Create a new variable*: compute the area of tree canopy cover within the buffer of each building footprint.
4. *Create a new variable*: compute the average LST for each building's buffer.
5. *Join and combine*: join the area of tree canopy cover and average LST within each building's buffer to a `FeatureCollection` storing the building footprint `Geometry`.
6. *Summarise*: compute the average tree canopy cover and LST for buildings on each campus.

5.0.2 Setup

Create a new script in your `labs-gee` repository called `lab6`. Enter the following comment header to the script.

```

/*
Lab 6
Author: Test
Date: XX-XX-XXXX

*/

```

5.0.3 Data Import

Execute the following code to import the data. The OSM buildings near Perth university campuses are a `FeatureCollection` of building `Feature` objects. The Urban Monitor tree cover data are clips of the area surrounding each university campus from a larger raster layer. Import these four clipped rasters as `Image` objects and `mosaic()` them into one `Image`. Each 40 cm pixel in the `uniTree` `Image` has a value 1 if it covers a tree canopy and a masked no data value if not. The `multiply(ee.Image.pixelArea())` operation is converting the pixel value of one into the area of the pixel in square metres.

```

// Data import

// Perth OSM university buildings and buildings near universities
var perthBuildingOSM = ee.FeatureCollection('users/jmad1v07/gee-labs/perth-uni-osm');
print(perthBuildingOSM);
Map.centerObject(perthBuildingOSM, 13);
Map.addLayer(perthBuildingOSM, {color: 'FF0000'}, 'OSM buildings near Perth university campuses');

// import urban monitor tree cover data
var curtinTree = ee.Image('users/jmad1v07/gee-labs/curtin-tree-2016');
var ecuTree = ee.Image('users/jmad1v07/gee-labs/ecu-tree-2016');
var murdochTree = ee.Image('users/jmad1v07/gee-labs/murdoch-tree-2016');
var uwaTree = ee.Image('users/jmad1v07/gee-labs/uwa-tree-2016');

// mosaic urban monitor tree cover data covering Perth universities
var uniTree = ee.ImageCollection([curtinTree, ecuTree, murdochTree, uwaTree]).mosaic();
print(uniTree);

// convert each pixel value to represent area of tree cover (SqM)
var uniTreePixelArea = uniTree.multiply(ee.Image.pixelArea());
print(uniTreePixelArea);

// import Landsat 8 summer land surface temperature

```

```
var landsatLST = ee.Image('users/jmad1v07/gee-labs/landsat8-lst');
```

Explore the `perthBuildingOSM` data in the map display and in the *console*.

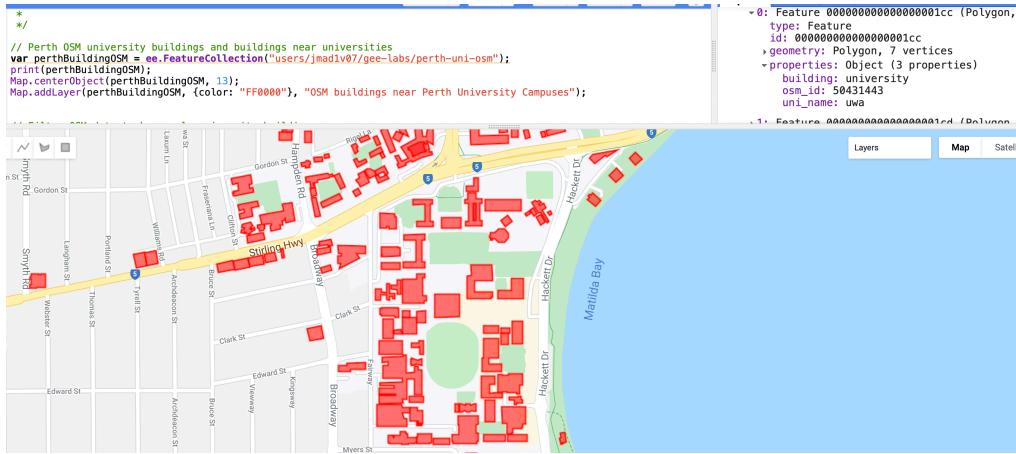


Figure 5.1: OSM building footprints and properties near UWA.

Can you visualise the `uniTree` Image on the map display? Look back to lab 5 for examples of how to visualise the Urban Monitor data and for appropriate colour schemes.

::: {.cell}

```
// UM uni tree
Map.addLayer(uniTree, {min: 0, max: 1, palette:[ '#009900' ]}, 'UM Tree');
```

:::

5.1 Filter

Filtering subsets observations from your data based on their values (Wickham and Grolemund, 2017). In Google Earth Engine, comparison operators (e.g equals to `eq`, not equals to `neq`, less than `lt`, greater than `gt`) are used to filter observations based on attribute values.

Google Earth Engine allows you to specify custom `filter()` functions. The following code snippet demonstrates how to create your own filter to subset `Features` in the `perthBuildingOSM FeatureCollection` whose `building` property value is 'university'. If you execute the following code snippet and inspect the filtered `FeatureCollection` in the *console* and the cyan building footprints on the map display you should see that `perthUniBuildingOSM` contains fewer `Features` than `perthBuildingOSM`.

```
// Filter OSM data to keep only university buildings
var perthUniBuildingOSM = perthBuildingOSM.filter(ee.Filter.eq('building', 'university'));
print('Uni Buildings:', perthUniBuildingOSM);
Map.addLayer(perthUniBuildingOSM, {color: '000000'}, 'OSM university buildings');
```

Let's quickly unpack the `filter()` function. The `filter()` function takes an `ee.Filter.eq(name, value)` object as an argument. The `ee.Filter.eq()` object is constructed by specifying a name and value which correspond to the name of the property and a value that property should take for a filter's comparison operation to evaluate to true.

*Look at the filter documentation on the Google Earth Engine documentation website. Which filter would you use to return non-university building Features from perthBuildingOSM
`ee.Filter.neq()` ee.Filter.neq("building", "university").*

5.2 Buffer

To compute the area of tree cover or average LST near each university building you need to define the area which corresponds to a building's surrounding neighbourhood. You can compute this area by applying a geometric `buffer()` operation to each building `Geometry` object.

The `buffer` operation is a unary geometric operation as it is applied to just one geometric object. Along with the `buffer` operation, examples of unary operations include computing the centroid of a polygon object, simplifying geometries, or shifting or rescaling a geometry (Lovelace et al. 2020). In contrast, binary geometry operations modify a geometry based upon another; for example, clipping one geometry using the extent using the extent of another.

Applying a buffer to a geometry returns a polygon encompassing the area within a specified distance of the input geometry; for example, applying a 1 km buffer to a point object would return a circular polygon with a 1 km radius surrounding the point. In Google Earth Engine the `buffer()` operation can be applied to `Geometry` objects and returns a buffer polygon `Geometry` object. The `buffer()` function in Google Earth Engine has a distance parameter which is a number specifying the size of the buffer to compute (in metres unless otherwise specified).

Here, you will compute each building's surrounding neighbourhood using a 50 m buffer. The following code snippet creates a function that computes a 50 m buffer for a `Geometry` object. Let's quickly recap how user-defined functions are created in Google Earth Engine.

1. *function name*: first, you have given the function an informative name that describes what it does; `bufferFunc` clearly indicates this function will compute a buffer.

2. *parameters*: the function parameters are enclosed within parentheses (`feature`) following the function declaration. This function takes in a single parameter `feature` which is passed onto the operations enclosed in {}.
3. *function operations*: the operation enclosed within this function computes the 50 m buffer for the `feature` passed into the function.
4. *return*: this function `returns` a `Feature` object containing the 50 m buffered polygon surrounding the `Feature` passed into the function.

```
// This function computes a 50 m buffer around each university building footprint
var bufferFunc = function(feature) {
  return feature.buffer(50);
};
```

You have created a function that will compute the 50 m buffer. Next, you need to apply this function to each university building. You do this by `mapping` the function over each `Feature` in the `perthUniBuildingOSM FeatureCollection`. You can think of this as a “for each” operation; for each `Feature` in the `FeatureCollection` compute this function and return the result.

The concept of mapping a function over elements in a collection can be represented graphically:

Each of the orange boxes is an element in a collection and `f` is a function that is applied to each element. Here, `f` is `bufferFunc()`. Mapping the function `f` over each element in the collection returns a collection.

`buildingFootprint → bufferFunc(50) → bufferedBuildingFootprint ↓ buildingFootprint
→ bufferFunc(50) → bufferedBuildingFootprint ↓ buildingFootprint → bufferFunc(50)
→ bufferedBuildingFootprint ↓ buildingFootprint → bufferFunc(50) → bufferedBuildingFootprint`

To avoid confusion, map here refers to the mathematical meaning of an “an operation that associates each element of a given set with one or more elements of a second set” and NOT representing objects in space (Wickham, 2020).

If you execute the following code snippet you will map the buffer function `bufferFunc` over each `Feature` representing a university building in the `FeatureCollection perthUniBuildingOSM`. This returns a `FeatureCollection` stored in the variable `perthUniBuildingOSMBuffer`. Each `Feature` in `perthUniBuildingOSMBuffer` should contain a `Geometry` object representing a 50 m buffer around a building.

```
// map buffer function over university buildings feature collection
var perthUniBuildingOSMBuffer = perthUniBuildingOSM.map(bufferFunc);
Map.addLayer(perthUniBuildingOSMBuffer, {color: '33FF00'}, 'Uni building 50 m buffer');
```

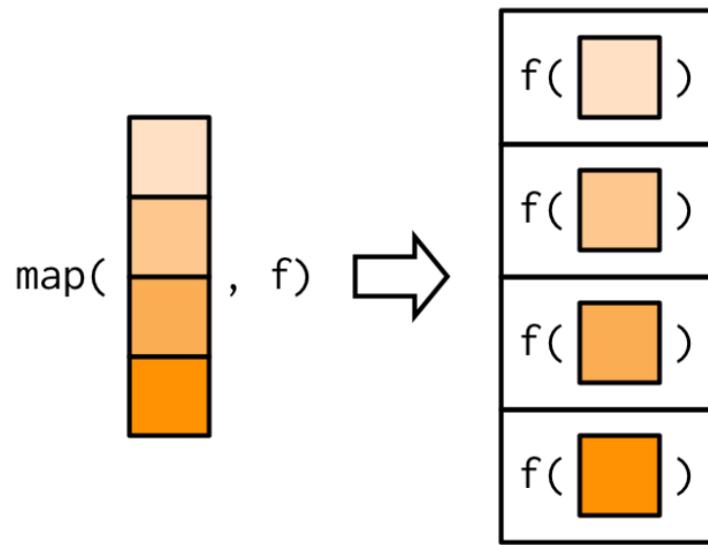


Figure 5.2: Graphical representation of mapping a function over a collection and returning a collection as an output (source: [Wickham \(2020\)](#)).



Figure 5.3: 50 m buffer (green) computed for buildings at Curtin University.

5.3 Zonal Statistics

You need to compute the area of tree cover and average LST surrounding each university building. You can use your buffered polygon `Geometry` objects to represent the area surrounding a building. Tree cover and LST data are in raster format. Zonal operations can be used to summarise the raster tree cover or LST values that intersect with a building's buffer.

Data aggregation and summaries are computed in Google Earth Engine using reducer objects of the `ee.Reducer` class. You can find an overview of reducer functions in Google Earth Engine here. Reducers aggregate data over space, time, or another dimension in attribute data using an aggregation or summary function (e.g. mean, max, min, sum, standard deviation).

You pass values for the pixels that intersect with a building's buffer into a reducer function. There are `reduceRegion()` and `reduceRegions()` functions that can be used to summarise raster values that intersect with a specified region (i.e. a building's buffer); these functions return one summary value per region.

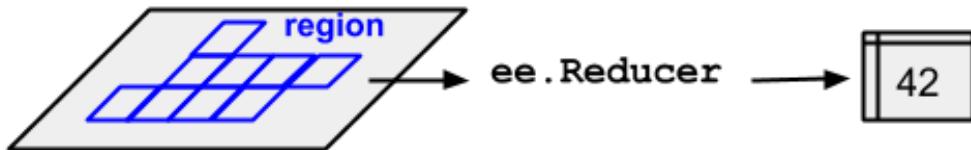


Figure 5.4: Reduce region (source: [Google Earth Engine developers guide](#)).

The following code snippet applies the `reduceRegions()` function to the `uniTreePixelArea` `Image` where each pixel value is the area of tree cover in square metres. If you look at the arguments to `reduceRegions()` you will see that the regions over which raster values are summarised are taken from the `perthUniBuildingOSMBuffer` `FeatureCollection`, a sum reducer function was used to summarise the raster values, and the summary operation was performed on raster data with a spatial resolution of 0.4 metres.

The result of the `reduceRegions()` function is a `FeatureCollection` with the same number of `Features` as the input `FeatureCollection` but with a name:value pair in the `properties` object which contains the result of the summary of raster values within that region.

```
// Zonal stats: reduceRegions to sum tree cover within a building's buffer
var perthUniBuildingTree = uniTreePixelArea.reduceRegions({
  collection: perthUniBuildingOSMBuffer,
  reducer: ee.Reducer.sum(),
  scale: 0.4,
});

// helper function to give result of reduceRegions an informative name
```

```

perthUniBuildingTree = perthUniBuildingTree.map(function(feature){
  return ee.Feature(feature.geometry(), {
    building: feature.get('building'),
    osm_id: feature.get('osm_id'),
    uni_name: feature.get('uni_name'),
    treeAreaSqM: feature.get('sum')
  });
});

print('zonal stats - tree area:', perthUniBuildingTree);

```

You can perform a similar `reduceRegions()` operation to compute average LST surrounding each university building. Inspect the results of the `reduceRegions()` in the *console*.

```

// Zonal stats: reduceRegions to average LST within a building's buffer
var perthUniBuildingLST = landsatLST.reduceRegions({
  collection: perthUniBuildingOSMBuffer,
  reducer: ee.Reducer.mean(),
  scale: 0.4,
});

// helper function to give result of reduceRegions an informative name
perthUniBuildingLST = perthUniBuildingLST.map(function(feature){
  return ee.Feature(feature.geometry(), {
    building: feature.get('building'),
    osm_id: feature.get('osm_id'),
    uni_name: feature.get('uni_name'),
    lstK: feature.get('mean')
  });
});

print('zonal stats - ave. LST:', perthUniBuildingLST);

```

What is different about the reducer used to compute average LST for a building's buffer?

Instead of using a sum reducer which sums all the raster values that intersect with the region a mean reducer was used which computes the average of all raster values that intersect with a region - `ee.Reducer.mean()`.

5.4 Join

You now have four `FeatureCollections` that contain information about university buildings:

- `perthUniBuildingOSM`: the `Geometry` objects for university building footprints.
- `perthUniBuildingOSMBuffer`: the `Geometry` objects for each university building's polygon buffer.
- `perthUniBuildingTree`: `Geometry` objects for each university building's polygon buffer and a `properties` dictionary with the area of tree cover within each buffer.
- `perthUniBuildingLST`: `Geometry` objects for each university building's polygon buffer and a `properties` dictionary with the average LST within each buffer.

You need to combine these `FeatureCollections` into one data set without duplicating `Features`.

You can use join operations to combine elements in a `FeatureCollection` through matching observations based on a common variable in both data sets (if you are familiar with relational database management systems this common variable(s) is often called a key - in Google Earth Engine these variables are called `leftField` and `rightField`).

In Google Earth Engine what constitutes a match, between observations in two data sets, is determined by an `ee.Filter()` object; an `ee.Filter.eq()` object would join the attributes for two `Features` if their values for the specified `leftField` and `rightField` are equivalent. The graphic below illustrates the concept of joining two data sets based upon matching values in a common variable.

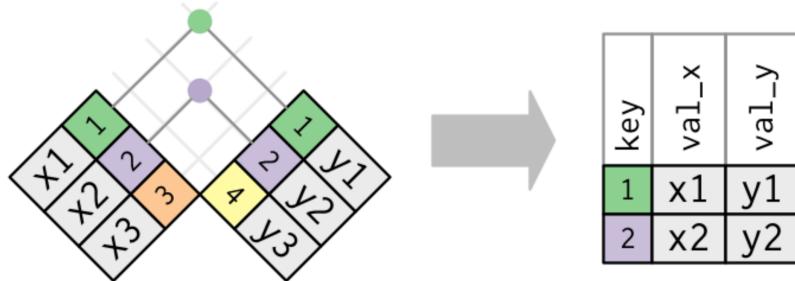


Figure 5.5: Illustration of an inner join between two data sets based upon matching values in a common field (source: [Wickham and Grolemund \(2017\)](#)).

The first step is to specify an `ee.Filter.equals()` object that will match values in a common field between two `Feature` objects. You can use the `osm_id` property which uniquely identifies a building object to match common buildings across `FeatureCollections`.

```
// Use an equals filter to specify how the collections match.
var osmFilter = ee.Filter.equals({
  leftField: 'osm_id',
  rightField: 'osm_id'
});
```

Next, you need to specify the type of join to apply. Here, you will use an inner join which keeps all attributes from both `FeatureCollections` being joined where there are matching observations for the joining field `osm_id`.

```
// Define the join.
var innerJoin = ee.Join.inner('primary', 'secondary');

// Apply the join.
var lstTreeJoin = innerJoin.apply(perthUniBuildingTree, perthUniBuildingLST, osmFilter);
print('joined:', lstTreeJoin);
```

The matching `Features` from `perthUniBuildingTree` and `perthUniBuildingLST` are stored in a `primary` and `secondary` dictionary object of the output from the join. Execute the following helper function to add the name:value pairs in `secondary` to the `properties` in the `primary` dictionary. This puts all your name:value pairs in one `properties` dictionary and makes it easy for you to query, summarise, and visualise this data. You can inspect the tidied `FeatureCollection` in the *Console* to see the output from this function.

Not necessary, but it might be a good activity to consolidate understanding: work through the function in the below code snippet and describe what each line is doing.

```
// tidy up properties of output from join
lstTreeJoin = lstTreeJoin.map(function(feature) {
  var f1 = ee.Feature(feature.get('primary'));
  var f2 = ee.Feature(feature.get('secondary'));
  return f1.set(f2.toDictionary());
});

print('joined and tidied:', lstTreeJoin);
```

5.5 Descriptive Statistics

You have now transformed your raw data (open street map buildings near Perth university campuses, a raster layer of tree cover, and a raster layer of LST) into a format where you can

answer the question at the beginning of the lab: *Which Perth university has the greenest and coolest campus?*

Your FeatureCollection, `lstTreeJoin`, should contain 215 Features with each Feature comprising a `Geometry` object and a dictionary of properties: `building`, `osm_id`, `uni_name`, `lstK`, and `treeAreaSqM`. One approach to addressing the question is to perform a *group by* and *summarise* operation. Group your data by the `uni_name` property and compute summary statistics for all observations within each group. Comparing the summary statistics between groups would indicate which university campus has buildings that are surrounded by more trees and cooler temperatures.

You have already used reducers in Google Earth Engine to aggregate values across space. There are other useful reducer functions: `reduceColumns()` aggregates values in `FeatureCollection` `properties` and a `reducer.group()` applies summary operations to groups of observations.

The following code snippet demonstrates how to apply `reduceColumns()` to the FeatureCollection `lstTreeJoin`. Let's go through this snippet line by line:

The `reduceColumns()` function has a:

- `selectors` parameter which is a list of `properties` that the reducer will group by and summarise values for.
- a `reducer` parameter which specifies the type of reducer function that will be applied to the `properties` specified in the `selectors` argument.
- pass a mean reducer `ee.Reducer.mean()` as the reducer argument into `reduceColumns()` indicating you want to aggregate values using the mean function.
- specify `repeat(2)` to apply this reducer twice (one reducer for '`treeAreaSqM`' and one reducer for '`lstK`').
- use `.group({.....})` to define how to group `Features` in your `FeatureCollection` before reducing their values. `groupField` specifies the grouping property in `selectors` (index location 2 corresponds to the third element in the list - `uni_name`). `groupName` is the name of the property for the grouping variable in the output.

```
// group by and summarise tree area and LST within each university campus
var campusSummaryStats = lstTreeJoin.reduceColumns({
  selectors: ['treeAreaSqM', 'lstK', 'uni_name'],
  reducer: ee.Reducer.mean().repeat(2).group({
    groupField: 2,
    groupName: 'uni_name'
  })
};

print(campusSummaryStats);
```

If you inspect the `print()` of `campusSummaryStats` in the *Console* you will see that it returned a dictionary object which contains a list of dictionary objects. This is an unfriendly data structure for storing and querying the data it contains.

```
▼ Object (1 property)
  ▼ groups: List (4 elements)
    ▼ 0: Object (2 properties)
      ▼ mean: [3116.349772479891, 307.4715005672809]
        0: 3116.349772479891
        1: 307.4715005672809
      uni_name: curtin
    ▼ 1: Object (2 properties)
      ▼ mean: [1865.2467230333511, 307.33004852440274]
        0: 1865.2467230333511
        1: 307.33004852440274
      uni_name: ecu
    ▶ 2: Object (2 properties)
    ▶ 3: Object (2 properties)
```

Figure 5.6: Structure of data returned by grouped `reduceColumns()`.

The following code snippet tidys up this data returning a `FeatureCollection` where each `Feature` has a null `geometry` property and a dictionary of properties: `uni_name`, `lstK`, and `treeAreaSqM`.

Again, it is not necessary to understand what is going on here but working through it line by line would be a good extra exercise to consolidate understanding of programmatically transforming data into more friendly formats.

```
// tidy up campus summary stats
var campusSummaryStats = ee.Dictionary(campusSummaryStats).values();
var campusSummaryStatsFlat = ee.List(campusSummaryStats).flatten();

var tidySummaryStats = function(listElement) {
  var groups = ee.Dictionary();
  var stats = ee.Dictionary(listElement).get('mean');
  var treeArea = ee.List(stats).get(0);
  var temp = ee.List(stats).get(1);
  var uni = ee.Dictionary(listElement).get('uni_name');
  groups = groups.set('uni_name', uni)
    .set('treeAreaSqM', treeArea)
    .set('lstK', temp);
  var groupsFeat = ee.Feature(null, groups);
```

```

    return groupsFeat;

};

var tidyCampusStats = campusSummaryStatsFlat.map(tidySummaryStats);
tidyCampusStats = ee.FeatureCollection(tidyCampusStats);
print('tidy campus stats:', tidyCampusStats);

tidy campus stats:
  ▾ FeatureCollection (4 elements, 4 columns)
    ▷ type: FeatureCollection
    ▷ columns: Object (4 properties)
    ▷ features: List (4 elements)
      ▷ 0: Feature 0
        type: Feature
        id: 0
        geometry: null
        ▷ properties: Object (3 properties)
          lstK: 307.4715005672809
          treeAreaSqM: 3116.349772479891
          uni_name: curtin
      ▷ 1: Feature 1
      ▷ 2: Feature 2
      ▷ 3: Feature 3

```

Figure 5.7: Tidier data structure for storing the results of grouped reduceColumns().

Let's look at the results. You should have `print()`d `tidyCampusStats` onto the *Console*. The `properties` object for each `Feature` stores the average area of tree canopy cover and LST within a 50 m buffer of buildings on each university campus. The figure above shows that, on average, buildings on Curtin University's Bentley Campus have an LST of 307.47 K. Look at the values reported for the other university campuses.

5.6 Visualisation

To make comparisons between campuses in terms of their greenness and coolness, you can look up the values in the *Console* for the `properties` object storing the results of the group by and summarise operations. However, this is not a visually friendly way to inspect your data, identify patterns, or detect relationships between variables. Google Earth Engine provides a range of tools to generate interactive charts from spatial data.

Chart objects can be rendered in the *Console* to visualise your data. The `ui.Chart.feature.byFeature()` function creates a chart from a set of `Features` in a `FeatureCollection` plotting each `Feature` on the X-axis and the value for a `Feature`'s property on the Y-axis.

The first argument to the `ui.Chart.feature.byFeature()` function is the `FeatureCollection` - `tidyCampusStats`. The second argument to `ui.Chart.feature.byFeature()` is the `label` property for `Features` plotted on the X-axis - `'uni_name'`. The final argument is a list object of properties whose values are plotted on the Y-axis - `['treeAreaSqM']`.

Use the `.setChartType()` method to specify the type of chart to create. View possible charts in this gallery. A dictionary of name:value pairs is passed into the `setOptions()` method to control various style elements of the chart (e.g. chart title, axis title).

To render your chart in the *Console* use the `print()` function.

```
// Make a chart by feature
var treeColumnChart =
  ui.Chart.feature.byFeature(tidyCampusStats, 'uni_name', ['treeAreaSqM'])
    .setChartType('ColumnChart')
    .setSeriesNames([''])
    .setOptions({
      title: 'Average tree cover near university buildings (SqM)',
      hAxis: {title: 'Uni. Campus'},
      vAxis: {title: 'Tree Cover (SqM)'}
    });
print(treeColumnChart);

// Make a chart by feature.
var lstColumnChart =
  ui.Chart.feature.byFeature(tidyCampusStats, 'uni_name', ['lstK'])
    .setChartType('ColumnChart')
    .setSeriesNames([''])
    .setOptions({
      title: 'Average LST near university buildings (K)',
      hAxis: {title: 'Uni. Campus'},
      vAxis: {title: 'LST (K)'}
    });
print(lstColumnChart);
```

The `ui.Chart.feature.groups()` function creates a chart from a set of `Features` in a `FeatureCollection` plotting values for `Feature` properties on the X-axis and Y-axis. This chart can be used to visualise the relationships between variables stored in `FeatureCollection` data.

The first argument to the `ui.Chart.feature.groups()` function is the `FeatureCollection`

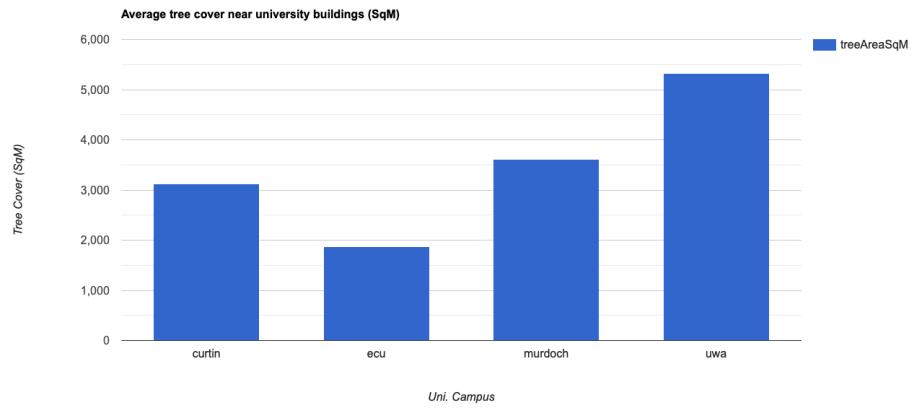


Figure 5.8: Average area of tree cover within a 50 m buffer of buildings on university campuses.

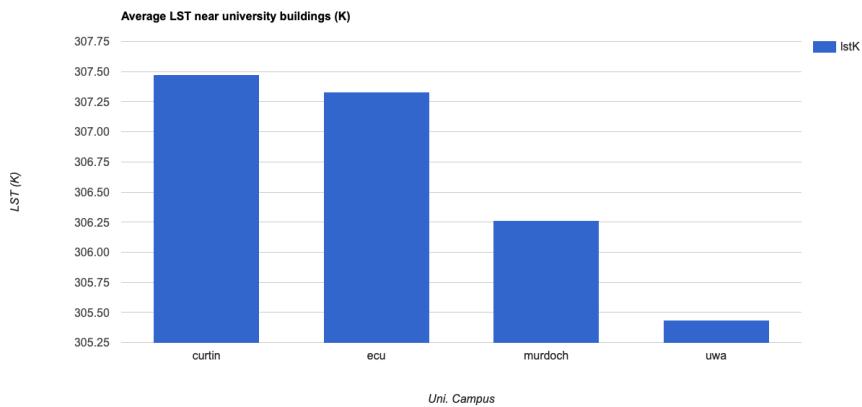


Figure 5.9: Average LST (K) within a 50 m buffer of buildings on university campuses.

- `lstTreeJoin` here as we want to visualise data for individual university buildings. The second argument to `ui.Chart.feature.groups()` is the property to be plotted on the X-axis
- `'treeAreaSqM'`. The third argument to `ui.Chart.feature.groups()` is the property to be plotted on the Y-axis - `'lstK'`. The final argument is the series property used to determine groups within the data - `'uni_name'` here (setting this argument will mean each University's data points will be rendered in different colours).

```
// Make a scatter chart
var tempVsTree =
  ui.Chart.feature.groups(lstTreeJoin, 'treeAreaSqM', 'lstK', 'uni_name')
    .setChartType('ScatterChart')
    .setOptions({
      title: '',
      hAxis: {title: 'Building neighbourhood tree cover (SqM)'},
      vAxis: {title: 'Temperature (K)'}
    });
  print(tempVsTree);
```

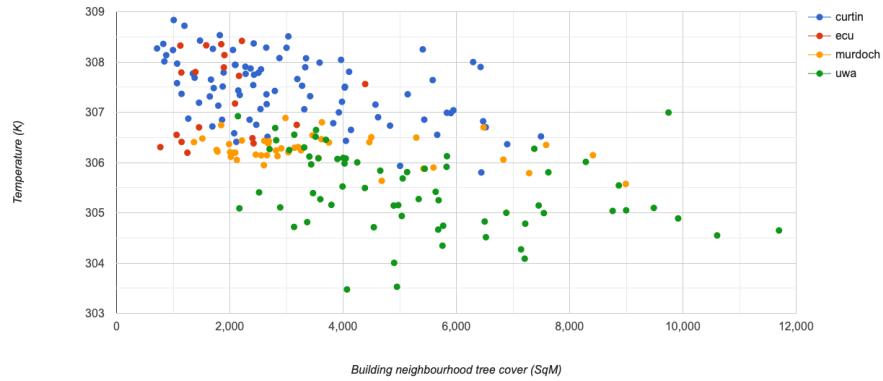


Figure 5.10: Scatter chart showing the relationship between average LST (K) and tree cover (SqM) within a 50 m buffer of buildings on university campuses.

6 Introduction

This lab will introduce data transformation and manipulation operations for raster data.

Quick recap: In Google Earth Engine, raster data is represented using `Image` objects which include one or more bands and each band is a georeferenced raster. Each band can have its own set of `properties` such as data type (e.g. integer), scale (spatial resolution), band name, and projection. The `Image` object itself can contain metadata relevant to all bands inside a `properties` dictionary object (e.g. date of `Image` capture). A collection of `Images` are stored in an `ImageCollection`. For example, all Landsat 8 `Images` are stored in an `ImageCollection` object.

[Lab 6](#) introduced data transformation operations that can be applied to vector data. These operations can be categorised into:

1. filtering or subsetting data
2. creating new variables from existing / raw data
3. joining or combining data sets
4. summarising or aggregating data

These categories of data transformation operations also apply to raster data. Vector data transformation operations can be spatial or non-spatial. Similarly, raster data transformation operations can be spatial (e.g. changing the spatial resolution of raster pixels, clipping a raster extent), non-spatial and applied to metadata (e.g. filtering raster `Images` by date), or applied to values stored in raster pixels (e.g. applying a function to convert raster `Image` pixel values that represent temperature in Fahrenheit to Celsius).

Data transformation operations applied to raster data can also be categorised as:

1. local operations: a pixel value is computed using per-pixel arithmetic, comparison, or logical operations.
2. focal operations: a pixel value is computed using aggregation or summary operations applied to neighbouring pixel values defined by a moving window or kernel.
3. zonal operations: a pixel value is computed using aggregation or summary operations applied to pixels within an irregular region.
4. global operations: an output is the result of aggregation or summary operations applied to all pixels in a raster data set.

6.0.1 Where has vegetation cover changed between 2000-2002 and 2017-2019?

This lab demonstrates a range of raster data transformations as part of a workflow to address the question: *Where has vegetation cover changed between 2000-2002 and 2017-2019?* The study area is the urban Perth and Peel region.

To address this question you will be using surface reflectance data from the Landsat 7 and Landsat 8 satellites.

The Landsat 7 surface reflectance data is derived from measurements recorded by the Enhanced Thematic Mapper (ETM+) sensor which has a revisit period of 16 days and a 30 m spatial resolution. Data from Landsat 7 is available from 1999 in Google Earth Engine.

The Landsat 8 surface reflectance data is derived from measurements recorded by the Operational Land Imager (OLI) sensor. Again, Landsat 8 data has a revisit period of 16 days and a spatial resolution of 30 m. Landsat 8 observations are available from 2013.

The following are good resources for information about Landsat data:

- A survival guide to Landsat preprocessing
- Landsat-8: Science and product vision for terrestrial global change research
- Current status of Landsat program, science, and applications

This lab will also introduce you to Google Earth Engine's capacity to process big geospatial data sets. Here, you will use all Landsat 7 and 8 scenes that intersect with the Perth and Peel region between 2000 and 2002 and 2017 and 2019.

Your workflow will encompass the following steps:

1. Subset all Landsat 7 and 8 `Image`s that intersect with the study region.
2. Mask cloudy pixels out of all Landsat 7 and 8 `Image`s returned from step 1.
3. Merge Landsat 7 and Landsat 8 `Image`s into one `ImageCollection`.
4. Compute the NDVI for all cloud free Landsat pixels between 2000-2002 and 2017-2019.
5. Compute a median NDVI composite `Image` for 2000-2002 and 2017-2019.
6. Compute a difference `Image` showing the change in NDVI between 2000-2002 and 2017-2019.

6.0.2 Setup

Create a new script in your `labs-gee` repository called `lab6`. Enter the following comment header to the script.

```
/*
Lab 7
Author: Test
```

```

Date: XX-XX-XXXX

*/
// import data

// import Landsat 7 and 8 ImageCollections
var l7 = ee.ImageCollection('LANDSAT/LE07/C01/T1_SR');
var l8 = ee.ImageCollection('LANDSAT/LC08/C01/T1_SR');

var bBox = ee.Geometry.Polygon(
  [[[115.73339493500093, -32.00885297734888],
    [115.73339493500093, -32.17638807304199],
    [115.97097428070406, -32.17638807304199],
    [115.97097428070406, -32.00885297734888]]], null, false);

```

6.1 Filter

`ImageCollections` can be filtered using spatial and non-spatial filters. The `filterBounds()` function filters `Images` in an `ImageCollection` that intersect with the extent of a `Geometry` object (passed as an argument to the `filterBounds()` function). This is a spatial filtering operation.

`ImageCollections` can also be filtered using non-spatial operations using the values of properties in an `Images` metadata. Each Landsat `Image` has a `properties` dictionary object of metadata attributes including the date and time of image capture (the `system:time_start` property). This property can be used with the `filterDate()` function to subset all Landsat `Images` captured within a specified date range. String data for the start and end date are passed as arguments into the `filterDate()` function and an `ImageCollection` containing only `Images` captured during that period is returned.

Executing the code below will filter the Landsat 7 and Landsat 8 `ImageCollections` using the `Geometry` object in `bBox` and subset `Images` captured between the dates specified.

```

/* filter Landsat 7 and 8 ImageCollections to return only Images
that intersect the bBox and were captured within specified date range */
var l72000 = l7
  .filterBounds(bBox)
  .filterDate('2000-01-01', '2002-12-31');

var l72017 = l7

```

```

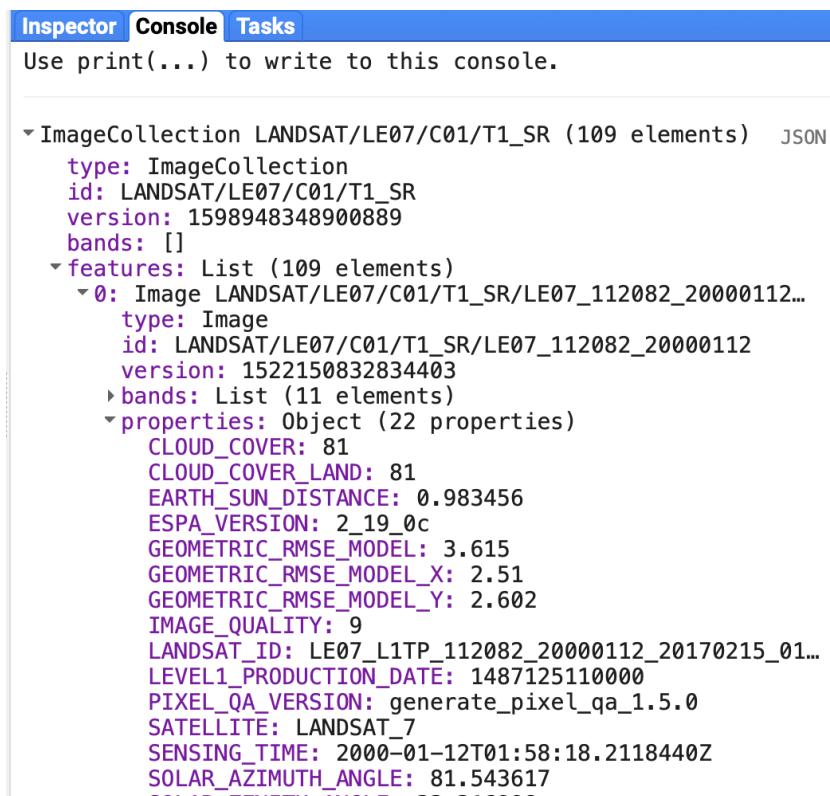
.filterBounds(bBox)
.filterDate('2017-01-01', '2019-12-31');

var 182017 = 18
.filterBounds(bBox)
.filterDate('2017-01-01', '2019-12-31');

print('Landsat 7 Im Coll 2000-2002:', 172000);

```

Print the Landsat 7 `ImageCollection` that is returned from filtering the `ImageCollection` storing the Landsat 7 archive for the `Images` that intersect with the `Geometry` object `bBox` and were captured between 1 January 2000 and 31 December 2002. The filtered `ImageCollection` is stored in the variable `172000` and should contain 109 Landsat 7 `Images`. If you inspect the `SENSING_TIME:` property for each of the `Images` in the `172000` you should see dates between 1 January 2000 and 31 December 2002.



```

Inspector Console Tasks
Use print(...) to write to this console.

▼ ImageCollection LANDSAT/LE07/C01/T1_SR (109 elements) JSON
  type: ImageCollection
  id: LANDSAT/LE07/C01/T1_SR
  version: 1598948348900889
  bands: []
  ▼ features: List (109 elements)
    ▶ 0: Image LANDSAT/LE07/C01/T1_SR/LE07_112082_20000112...
      type: Image
      id: LANDSAT/LE07/C01/T1_SR/LE07_112082_20000112
      version: 1522150832834403
      ▶ bands: List (11 elements)
      ▶ properties: Object (22 properties)
        CLOUD_COVER: 81
        CLOUD_COVER_LAND: 81
        EARTH_SUN_DISTANCE: 0.983456
        ESPA_VERSION: 2_19_0c
        GEOMETRIC_RMSE_MODEL: 3.615
        GEOMETRIC_RMSE_MODEL_X: 2.51
        GEOMETRIC_RMSE_MODEL_Y: 2.602
        IMAGE_QUALITY: 9
        LANDSAT_ID: LE07_L1TP_112082_20000112_20170215_01...
        LEVEL1_PRODUCTION_DATE: 1487125110000
        PIXEL_QA_VERSION: generate_pixel_qa_1.5.0
        SATELLITE: LANDSAT_7
        SENSING_TIME: 2000-01-12T01:58:18.2118440Z
        SOLAR_AZIMUTH_ANGLE: 81.543617
        ...

```

Figure 6.1: `Print()` of filtered Landsat 7 `ImageCollection`.

What would happen if you passed the dates '2010-01-01', '2010-12-31' into the `filterBounds()` for a Landsat 8 `ImageCollection`?

Error message - there should be no Landsat 8 Images captured in 2010. Landsat 8 started collecting data in 2013.

6.1.1 Cloud Masks

You have just applied spatial and non-spatial filtering operations to an `ImageCollection`. You can also apply filtering operations to an individual `Image`. A common application of spatial filtering operations applied to an `Image` is masking out a pixel value based upon the pixel value in the corresponding location in another raster.

Remote sensing data products often have a pixel quality band which indicates if the observation for a pixel is high quality or not. Cloud cover and atmospheric contamination are common sources of low quality observations.

The following code block displays an `Image` from the filtered Landsat 8 `ImageCollection` 182017 as an RGB composite. Cloud cover contamination is clearly visible.

```
// cloud mask function

// visualise a cloudy Landsat Image
var cloudyL8 = ee.Image('LANDSAT/LC08/C01/T1_SR/LC08_112082_20170510');

/* Define the visualization parameters. The bands option allows us to specify which bands
Here, we choose B4 (Red), B3 (Green), B2 (Blue) to make a RGB composite image.*/
var vizParams = {
  bands: ['B4', 'B3', 'B2'],
  min: 0,
  max: 3500,
};
Map.centerObject(cloudyL8, 8);
Map.addLayer(cloudyL8, vizParams, 'Cloudy L8 Image');
```

Landsat surface reflectance pixel quality attributes are stored as a bitmask within the `pixel_qa` band generated by the CFMASK algorithm. Other remote sensing products (e.g. MODIS and Planet) also provide pixel quality information as a bitmask; therefore, bitmasks are an important concept to understand when working with remote sensing data.

Each pixel in a quality band stores a number which can be represented as a decimal number (e.g. 32) or as a binary number (e.g. 00100000 - this is an 8 bit integer or one byte).

Binary numbers

- each bit in the binary number can take a value of 0 or 1

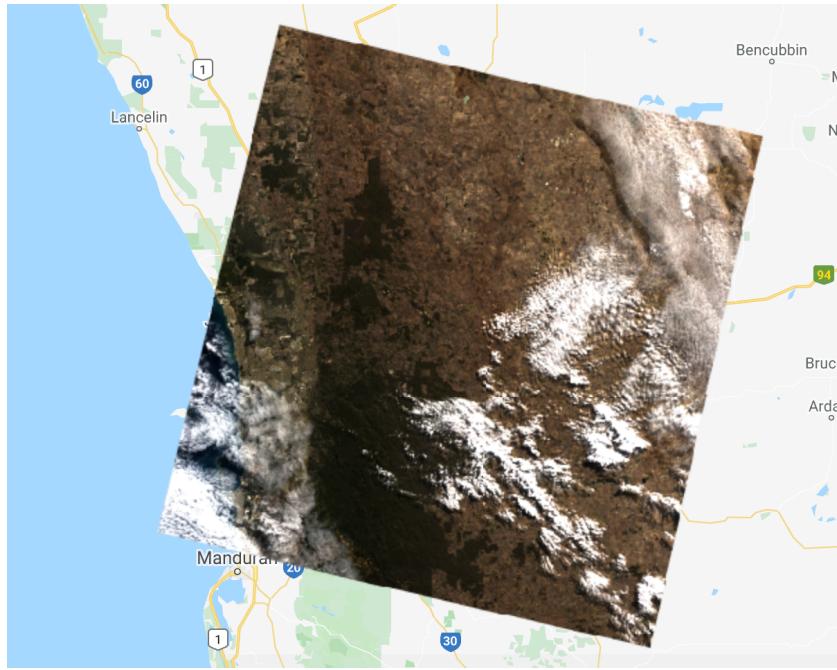


Figure 6.2: Cloud contamination of Landsat 8 Image.

- bits are ordered from right to left (i.e. 00001001 - bit 0 and bit 3 are represented by the binary digit 1)
- bit order starts at 0 (i.e. bit 5 is set to 1 here - 00100000)
- binary numbers have base 2
- convert binary 00001001 to decimal $\rightarrow 00001001 = (0 * 2^7) + (0 * 2^6) + (0 * 2^5) + (0 * 2^4) + (1 * 2^3) + (0 * 2^2) + (0 * 2^1) + (1 * 2^0) = 9$

In a bitmask `Image`, each bit corresponds to an indicator of quality information for that pixel. Keeping with the binary number 00100000, the bitmask represented by bit 5 evaluates to true and all other bitmasks are false. The bitmasks represented by values in the `pixel_qa` band of Landsat `Images` are shown below. The `pixel_qa` band stores pixel values as unsigned 16 bit integers with bits 0 to 10 representing different bitmasks representing different aspects of pixel quality.

A high quality pixel observation has the digit 1 in bit 1 (i.e. 0000000000000010). A cloudy pixel would have the digit 1 in bits 3 and 5 (i.e. 0000000000101000). While using bitmasks to store pixel quality information is more complicated than using separate binary `Image` bands for each indicator of pixel quality, it is a more efficient way of storing and transporting this information.

If a pixel was clear and snow what would its `pixel_qa` value be in binary?

00000000000010010 (bit 1 for clear and bit 4 for snow)

- | | |
|-----------------------|---|
| <code>pixel_qa</code> | <ul style="list-style-type: none">• Bit 0: Fill• Bit 1: Clear• Bit 2: Water• Bit 3: Cloud Shadow• Bit 4: Snow• Bit 5: Cloud• Bits 6-7: Cloud Confidence<ul style="list-style-type: none">◦ 0: None◦ 1: Low◦ 2: Medium◦ 3: High• Bits 8-9: Cirrus Confidence<ul style="list-style-type: none">◦ 0: None◦ 1: Low◦ 2: Medium◦ 3: High• Bit 10: Terrain Occlusion |
|-----------------------|---|

Figure 6.3: `pixel_qa` bitmask derived from CFMASK for Landsat surface reflectance.

You can use the bitmask contained in the `qa_band` to mask out cloudy pixels. To do this you need identify which pixels have the digit 1 in bit 3 or bit 5 (i.e. they are cloudy) and then mask those pixels in the Landsat 8 `Image`. The following steps demonstrate how to do this:

1. Create a binary number representing cloud shadow (digit 1 in bit 3): `var cloudShadowBitMask = (1 << 3);.` The `<<` shifts bits left by filling in zeros from the right. Here, 1 is shifted three bits left and the resulting value is stored in `cloudShadowBitMask`. `cloudShadowBitMask` now represents the value of the bitmask when cloud shadow is present.
2. Create a binary number representing clouds (digit 1 in bit 5): `var cloudsBitMask = (1 << 5);.` The `<<` shifts bits left by filling in zeros from the right. Here, 1 is shifted five bits left and the resulting value is stored in `cloudsBitMask`. `cloudsBitMask` now represents the value of the bitmask when cloud is present.

Left shift operator to create bitmask values for cloud and cloud shadow.

Variable

Operation

Operand

Output

`cloudShadowBitMask`

`(1 << 3)`

`00000000000000001`

`00000000000000001000`

`cloudsBitMask`

`(1 << 5)`

`00000000000000001`

`000000000000100000`

3. Extract the `pixel_qa` band into its own `Image` object `pixelQA`: `var pixelQA = cloudyL8.select('pixel_qa');`

4. Use `bitwiseAnd` and comparison operators, that evaluate to boolean true or false values, to identify if pixel values in `pixelQA` have the digit 1 in bit 3 or bit 5 when the pixel value is represented as a binary number. The `bitwiseAnd` operator compares two binary numbers and returns a number with same number of bits and the digit 1 in bit locations where both input binary numbers have the digit 1.

`bitwiseAnd` operation.

Situation

Operation

Operand

Output

cloud shadow present in `pixelQA`

```
pixelQA.bitwiseAnd(cloudShadowBitMask)
```

0000000001001000 & 0000000000001000

00000000000000001000

cloud shadow NOT present in `pixelQA`

```
pixelQA.bitwiseAnd(cloudShadowBitMask)
```

0000000001000000 & 0000000000001000

00000000000000000000

cloud present in `pixelQA`

```
pixelQA.bitwiseAnd(cloudsBitMask)
```

0000000001111000 & 0000000000100000

000000000000100000

cloud shadow NOT present in `pixelQA`

```
pixelQA.bitwiseAnd(cloudsBitMask)
```

0000000001011000 & 0000000000100000

00000000000000000000

5. If the result of applying a `bitwiseAnd` operation to a pixel value in `pixelQA` and either `cloudShadowBitMask` or `cloudsBitMask` is not equal to zero then the bitmask indicates either cloud or cloud shadow is present at that pixel location and it should be masked from subsequent processing. The `.eq(0)` comparison can be used to test if the result of a `bitwiseAnd()` operation is equal to zero and the `.and()` logical operator can be used to test if the `pixelQA` pixel value has the digit 0 for the bitmask for cloud and cloud shadow. If this logical operator evaluates to true then it indicates the pixel is cloud free.

The following code puts all these commands together. `cloudMask` stores a raster `Image` with a pixel value of 1 indicating no cloud and a pixel value 0 indicating cloud. When visualised on the map, cloud pixels should render in black.

```
// make a cloud mask
var cloudShadowBitMask = (1 << 3);
var cloudsBitMask = (1 << 5);

var pixelQA = cloudyL8.select('pixel_qa');
var cloudMask = pixelQA.bitwiseAnd(cloudShadowBitMask).eq(0)
    .and(pixelQA.bitwiseAnd(cloudsBitMask).eq(0));
Map.addLayer(cloudMask, {}, 'cloud mask');
```

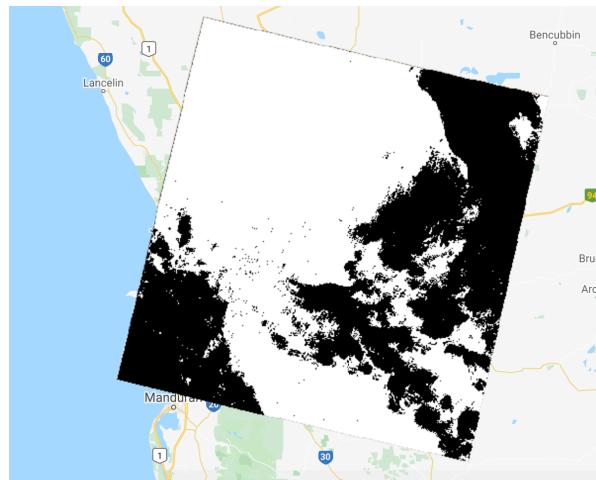


Figure 6.4: `cloudMask` Image derived from the `pixelQA` bitmask.

You can use the `Image` stored in the variable `cloudMask` to mask pixel values in the Landsat 8 `Image` `cloudyL8` where a pixel is cloudy. Masking pixels in Google Earth Engine makes pixels transparent and removes them from subsequent processing, analysis, or visualisation. To mask `Image` pixel values in Google Earth Engine pass a raster `Image`, where pixel values of zero indicate locations to mask, into the `updateMask()` function.

```
// mask out clouds in the cloudyL8 image
var cloudyL8Mask = cloudyL8.updateMask(cloudMask);
Map.addLayer(cloudyL8Mask, vizParams, 'Cloud Masked L8 Image');
```

Use the Layers widget to toggle the unmasked and masked Landsat 8 `Image` on and off to see the effect of cloud masking using the bitmask in the `pixel_qa` band.

You have gone through the process of masking out cloudy pixels from a single Landsat 8 `Image`.

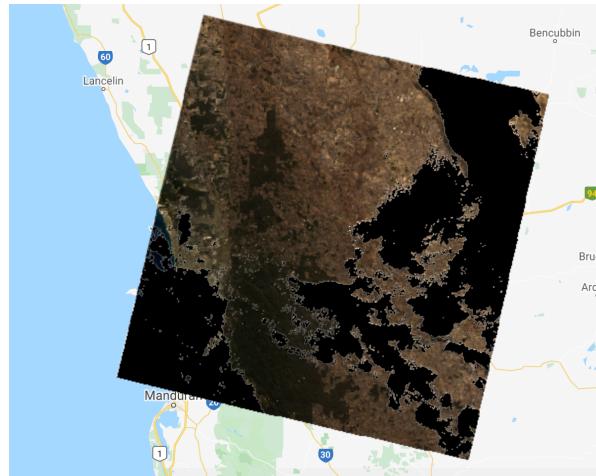


Figure 6.5: cloudMask applied to Landsat 8 Image.

However, repeating this process manually for all Landsat 8 Images would be time consuming.

This is where you can take advantage of a programmatic approach to GIS. You can wrap up the steps to create a cloud mask from an Images pixel_qa band and use that cloud mask to mask an Images spectral reflectance values in a function. You can then map that function over an ImageCollection of Landsat Images masking out cloudy pixels in each Image.

```
// Function to mask clouds based on the pixel_qa band of Landsat data.
function cloudMaskFunc(image) {
  // Bits 3 and 5 are cloud shadow and cloud, respectively.
  var cloudShadowBitMask = (1 << 3);
  var cloudsBitMask = (1 << 5);
  // Get the pixel QA band.
  var qa = image.select('pixel_qa');
  // Both flags should be set to zero, indicating clear conditions.
  var mask = qa.bitwiseAnd(cloudShadowBitMask).eq(0)
    .and(qa.bitwiseAnd(cloudsBitMask).eq(0));
  return image.updateMask(mask);
}
```

Next, you need to map this function over each of your filtered ImageCollections of Landsat 7 and 8 data.

```
// map cloudMask function over Landsat ImageCollections.
172000 = 172000.map(cloudMaskFunc);
```

```
172017 = 172017.map(cloudMaskFunc);
```

```
182017 = 182017.map(cloudMaskFunc);
```

In Lab 6 the concept of mapping a function over a `FeatureCollection`, applying the function to each `Feature` in the `FeatureCollection`, and returning a `FeatureCollection` of `Features` storing the results returned from the function was introduced. The same concept of mapping a function over an `ImageCollection` applies here. The function `cloudMaskFunc` is mapped over the `ImageCollections` 172000, 172017, and 182017, each `Image` in these `ImageCollections` is cloud masked, and an `ImageCollection` of cloud masked `Images` is returned.

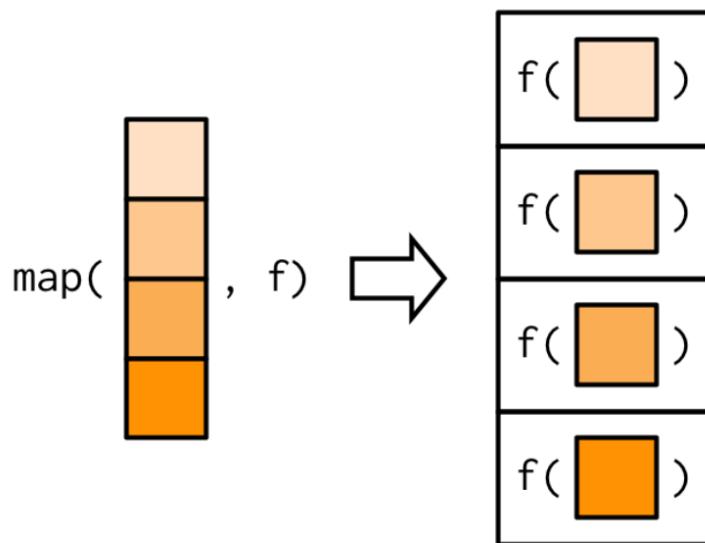


Figure 6.6: Graphical representation of mapping a function over a collection and returning a collection as an output (source: [Wickham \(2020\)](#)).

The concept of masking pixel values based on a cloud mask can be considered spatial subsetting; you are subsetting pixels for subsequent analysis based on the pixel values and their location in another raster `Image` (Lovelace et al. 2020). This is also an example of a local raster operation where operations are applied on a per-pixel basis.

6.2 Create new variables

6.2.1 Image Math and Local Map Algebra Operations

Image math operations are applied to Images on a per-pixel basis. The output from an Image math operation is a raster where each pixel's value is the result of the Image math operation. The input to the Image math operation is either:

1. two or more Images where the operation is applied on a per-pixel basis.
2. one or more Images and a constant number where the constant number is combined with each pixel value using the specified math operator.

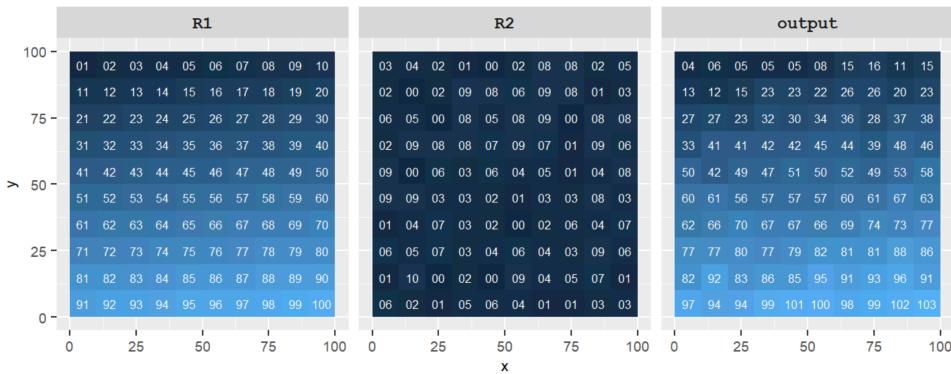


Figure 6.7: Illustration of Image math with two or more Images where the operation is applied on a per-pixel basis - output = R1 + R2 (source: [Gimond \(2019\)](#)).

Pixel values across Images or pixel values and constant numbers can be combined using math operators in Google Earth Engine:

- `add()`
- `subtract()`
- `multiply()`
- `divide()`

Image math operations can also be combined with per-pixel comparison and logical operators that evaluate to true or false.

- `lt()` - less than
- `gt()` - greater than
- `lte()` - less than or equal to
- `gte()` - greater than or equal to
- `eq()` - equal to
- `neq()` - not equal to

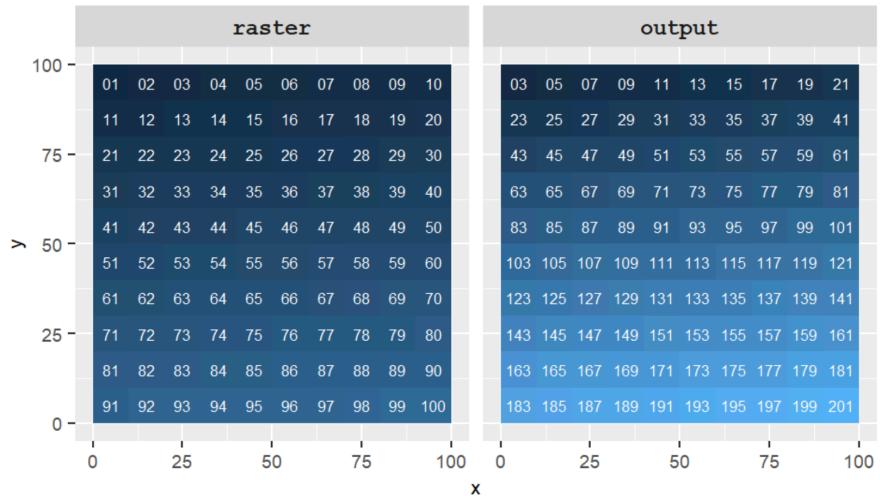


Figure 6.8: Illustration of Image math with two or more Images and a constant number where the constant number is combined with each pixel value using the specified math operator - output = 2 * + raster + 1 (source: [Gimond \(2019\)](#)).

- `and()` - AND

In Google Earth Engine only the intersection of unmasked pixels between input `Images` are returned from `Image` math operations.

6.2.2 Spectral Indices

`Image` math is used to combine bands in remote sensing `Images` that correspond to measures of spectral reflectance (ie. reflectance in different wavelengths). Mathematical per-pixel combinations of spectral reflectance measures are called spectral indices. Different land surface features have different spectral signatures (they reflect differently across wavelengths of the electromagnetic spectrum). Spectral indices use `Image` math to combine information about levels of reflectance in different wavelengths into a single value. Thus, computing spectral indices can provide more information about the condition or characteristics of a pixel than could be obtained from spectral reflectance measures in a single band.

Spectral indices are commonly used to monitor vegetation (vegetation indices). The normalised difference vegetation index (NDVI) is computed using spectral reflectance in red and near infrared wavelengths.

The NDVI equation is:

$$NDVI = \frac{NIR - red}{NIR + red}$$

NDVI values have a range of -1 to 1; a higher NDVI value indicates greater vegetation cover, greenness, or biomass within a pixel.

The NDVI is based on the reflectance characteristics of green vegetation which absorbs red light and reflects near infrared electromagnetic radiation. Red light is absorbed by chlorophyll in leaves (which also explains why we see vegetation in green). Near infrared electromagnetic radiation is reflected by green vegetation as it passes through the upper layers of leaves and is scattered by mesophyll tissue and openings between cells. Some of this scattered near infrared radiation is reflected upwards and detected by remote sensors. For vegetated land covers there will be a larger difference between red and near infrared reflectance than is observed for other non-vegetated land covers.

The following two functions compute the NDVI for Landsat 7 and Landsat 8. They return an `Image` with a band named '`nd`' as set by the `rename()` function and a `system:time_start` property which is set in the returned `Images` metadata. This is important as it records what date and time the NDVI data corresponds to.

Note the different band designations for Landsat 7 and Landsat 8. The near infrared band in Landsat 7 is band 4 ('`B4`') and the red band is band 3 ('`B3`'). For Landsat 8 the near infrared band is band 5 ('`B5`') and the red band in band 4 ('`B4`').

```
// Image math - NDVI
var ndviL7 = function(image) {

  var ndvi = image.select('B4').subtract(image.select('B3'))
    .divide(image.select('B4').add(image.select('B3')));
  ndvi = ndvi.rename('nd');
  var startDate = image.get('system:time_start');
  return ndvi.set({'system:time_start': startDate});
};

var ndviL8 = function(image) {

  var ndvi = image.select('B5').subtract(image.select('B4'))
    .divide(image.select('B5').add(image.select('B4')));
  ndvi = ndvi.rename('nd');
  var startDate = image.get('system:time_start');
  return ndvi.set({'system:time_start': startDate});
};
```

Finally, map these functions over the Landsat 7 and Landsat 8 `ImageCollections` to return an `ImageCollection` of NDVI `Images`. Display the first NDVI `Image` returned in the `182017NDVI` on the map to visualise the output of computing the NDVI using Landsat data.

```

//Map NDVI functions of Landsat ImageCollections
var l2000 = 172000.map(ndviL7);

var l72017NDVI = 172017.map(ndviL7);

var l82017NDVI = 182017.map(ndviL8);
print(l82017NDVI);

// display first Landsat 8 NDVI Image on the map
Map.addLayer(l82017NDVI.first(), {min: 0.2, max: 0.8, palette: ['#f7fcfd', '#e5f5f9', '#ccece6']});

```

Image math or local map algebra operations are raster data transformation operations where you apply a function to input data to compute a derived variable for subsequent analysis. If your study required monitoring vegetation condition, you can apply a function to spectral reflectance data to compute per-pixel vegetation index values.

There are many other spectral and vegetation indices that are used for different land surface monitoring. For example, the soil adjusted vegetation index (SAVI) is used to account for soil brightness effects when vegetation cover is low. The SAVI equation is:

$$SAVI = \left(\frac{NIR - red}{NIR + red + L} \right) (1 + L)$$

L is 0 for high vegetation cover and 1 for low vegetation covers. The following table shows the band designations for Landsat data.

TABLE 1. Summary of band designations and pixel size (m) for all Landsat satellites (LS) and sensors.

Landsat sensor	LS 1–5 MSS	LS 4–5 TM	LS 7 ETM+	LS 8 OLI/TIRS	Pixel size (m)
Coastal aerosol				B1 (0.43–0.45)	30
Blue		B1 (0.45–0.52)	B1 (0.45–0.52)	B2 (0.45–0.51)	30
Green	B1 (0.5–0.6)	B2 (0.52–0.60)	B2 (0.52–0.60)	B3 (0.53–0.59)	30 (60† for MSS)
Red	B2 (0.6–0.7)	B3 (0.63–0.69)	B3 (0.63–0.69)	B4 (0.64–0.67)	30 (60† for MSS)
NIR 1	B3 (0.7–0.8)				60
NIR	B4 (0.8–1.1)	B4 (0.76–0.90)	B4 (0.77–0.90)	B5 (0.85–0.88)	30 (60† for MSS)
SWIR 1		B5 (1.55–1.75)	B5 (1.55–1.75)	B6 (1.57–1.65)	30
SWIR 2		B7 (2.08–2.35)	B7 (2.09–2.35)	B7 (2.11–2.29)	30
Thermal		B6 (10.40–12.50)	B6‡ (10.40–12.50)	B10 (10.60–11.19) B11 (11.50–12.51)	30†
Pan-Chromatic			B8 (0.52–0.90)	B8 (0.50–0.68)	15
Cirrus				B9 (1.36–1.38)	30

Figure 6.9: Landsat bands and wavelength coverage (source: [Young et al \(2017\)](#)).

Can you create functions that compute the SAVI from Landsat 7 and Landsat 8 data? Remember that band designations differ between Landsat 7 and 8.

```

// Image math - SAVI
var saviL7 = function(image) {

  var savi = ((image.select('B4').subtract(image.select('B3'))))
    .divide(image.select('B4').add(image.select('B3')).add(0.5)))
    .multiply(1.5);
  savi = savi.rename('nd');
  var startDate = image.get('system:time_start');
  return savi.set({'system:time_start': startDate});
};

var saviL8 = function(image) {

  var savi = ((image.select('B5').subtract(image.select('B4'))))
    .divide(image.select('B5').add(image.select('B4')).add(0.5)))
    .multiply(1.5);
  savi = savi.rename('nd');
  var startDate = image.get('system:time_start');
  return savi.set({'system:time_start': startDate});
};

//Map SAVI functions of Landsat ImageCollections
var saviL72000 = 172000.map(saviL7);

var saviL82017 = 182017.map(saviL8);
print(saviL82017);

// display first Landsat 8 SAVI Image on the map
Map.addLayer(saviL82017.first(), {min: 0.2, max: 0.8, palette:['#a6611a', '#dfc27d', '#f5f5f5']}

```

6.3 Join / Combine

The time period for Landsat 7 observations spans 1999 to 2020. Therefore, you have Landsat 7 observations for the period 2000-2002 and 2017-2019. Landsat 8 observations start from 2013. You need to combine the Landsat 7 and Landsat 8 observations for period 2017 to 2019. To do this use the `merge()` function which merges two `ImageCollections` into one. You can then sort by the date of `Image` capture so the `Images` in the returned collection are in a temporal order.

```
// merge Landsat 7 NDVI and Landsat 8 NDVI ImageCollections and sort by time
var 12017 = ee.ImageCollection(182017NDVI.merge(172017NDVI)).sort('system:time_start');
print(12017);
```

Inspect the `ImageCollection` `12017` in the *Console* to see that it includes Landsat 8 and Landsat 7 `Images`.

6.4 Summarise

You now have two `ImageCollections` `12000` and `12017` that contain NDVI `Images` from 2000 to 2002 and 2017 to 2019, respectively. You need to summarise the NDVI data in these two `ImageCollections` to create a per-pixel measure of vegetation condition in each time-period.

The process of combining multiple, spatially overlapping pixel measures is called compositing. Composite vegetation index `Images` are often computed because spectral reflectance values for a signal time point are often noisy (e.g. due to cloud cover or atmospheric contamination). However, the summary of multiple measures at the same location is likely to reduce noise and provide a more accurate indication of the characteristics of that pixel.

Creating composite `Images` from `ImageCollections` in Google Earth Engine is straightforward. You can apply the `median()`, `mean()`, `max()` functions to an `ImageCollection` to compute the per-pixel median, mean, or max value for all `Images` in the collection. Here, use the `median()` function to create a median NDVI composite for the period 2000 to 2002 and 2017 to 2019. The median operation throws away dark pixels (cloud shadows) and bright pixels (clouds) that were not removed by the cloud masking operation.

```
// 3 year median NDVI composite
var 12000Composite = 12000
  .median()
  .clip(bBox);

// mask water
12000Composite = 12000Composite.updateMask(12000Composite.gt(0));
```

You will spot that you also `clipped` your median NDVI composite `Image` using the extent of the `Geometry` object `bBox`. You also masked out any pixels in your median NDVI composite with NDVI values less than or equal to zero. This is a quick way to remove water from your `Image` as water's NDVI values are typically below 0. Both of these steps are to enhance the visualisation of your NDVI composite `Images` on the map.

The `clip()` operation is another example of spatial subsetting.

Creating the mask of pixel locations with a median NDVI composite value greater than 0 `12000Composite.gt(0)` is an example of a local raster operation using a comparison operator as opposed to an arithmetic operator (each pixel value will evaluate to true or false).

Turn off the other layers on your map display using the *Layers* menu. Visualise your median NDVI composite `Image` on the map. Use the inspector to query NDVI values at different locations.

```
// 3 year median NDVI composite - 2000 - 2002
print(12000Composite);
Map.centerObject(12000Composite, 12);
Map.addLayer(12000Composite, {min: 0.1, max: 0.8, palette: ['#ffffcc', '#d9f0a3', '#add8e6', '#9acd32']});
```

Repeat the steps for the time period 2017 to 2019.

```
// 3 year median NDVI composite - 2017 - 2019
var 12017Composite = 12017
  .median()
  .clip(bBox);

// mask water
12017Composite = 12017Composite.updateMask(12017Composite.gt(0));

print(12017Composite);
Map.addLayer(12017Composite, {min: 0.1, max: 0.8, palette: ['#ffffcc', '#d9f0a3', '#add8e6', '#9acd32']});
```

Let the median NDVI composites for the period 2000-2002 and 2017-2019 (`12000Composite` and `12017Composite`) load on your map display. Toggle them on and off using the *Layers* menu to visualise change in vegetation over that time period. You should see something similar to the video below. Where you see change in NDVI can you explain why? Look at the satellite base map in areas where you note change in NDVI to see if that can provide any clues as to what caused the change in NDVI.

Change in NDVI between 2000-2002 and 2017-2019

6.5 Change Detection

You can use the median NDVI composite `Images` for the two time periods to answer the question at the beginning of the lab: *Where has vegetation cover changed between 2000-2002 and 2017-2019?*.

Using the `Image` math operations introduced earlier, you can compute a change detection `Image` which shows the location, direction (positive NDVI change, negative NDVI change, no change),

and magnitude of NDVI change between these two time periods. Temporal change detection is the operation of detecting change between two Images captured on different dates.

You can perform an Image differencing operation to compute a change Image.

```
// change detection
var ndviChange = 12017Composite.subtract(12000Composite);
print(ndviChange);
Map.addLayer(ndviChange, {min: -0.2, max: 0.2, palette:['#d73027','#f46d43','#fdade61','#fecc5c']});
```

You should see an Image similar to the figure below on your map visualising change in NDVI between 2000-2002 and 2017-2019. The areas in red indicate a decrease in NDVI, yellow little / no change in NDVI, and blue indicates an increase in NDVI.

Change the min and max values in the visualisation parameters to highlight different features of vegetation change (i.e. just areas of vegetation loss).

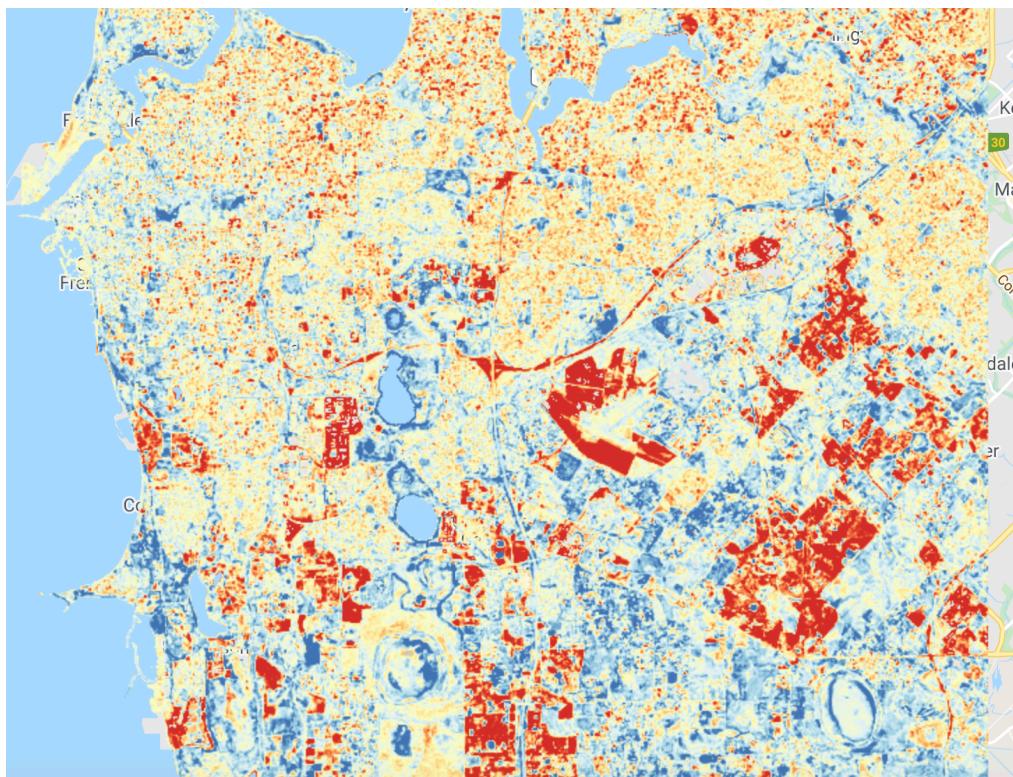


Figure 6.10: Image difference between NDVI in 2000-2002 and 2017-2019. Red indicates decrease in NDVI and blue indicates increase in NDVI.

7 Introduction

7.0.1 Phenology

Phenology refers to timing of biological events, phenomena, or life cycle phases, which are often driven by climatic and environmental factors. For example, budbreak or the onset and duration of flowering of plants are phenological events.

As remote sensors capture repeat observations of the same location over time, it is possible to track the dynamics of vegetation growth and the occurrence of phenological events.

The ability to monitor phenology can help us understand: how year-to-year variation in weather (e.g. extreme heat events or drought) affects vegetation growth (e.g. crop yields); how ecosystems are responding to changing climates (e.g. changing timing and duration of growing seasons); and how human modification of landscapes affects vegetation dynamics (e.g. differences in observed phenology in urban areas versus proximal natural or rural areas).

7.0.2 Remote Sensing of Phenology

Spectral vegetation indices, mathematical combinations of surface reflectance in visible and near infrared wavelengths, capture various aspects of vegetation condition at the time of image capture. These vegetation indices (VI) are used to monitor vegetation dynamics, and, thus, phenology.

Would you expect the phenology profiles to differ for evergreen and deciduous forests?

Yes, a deciduous forest's phenology profile would have a more exaggerated hump-shape with a distinct start-of-growing season, greenup to peak greenness, and then a senescence to leaf off. An evergreen forest would have a less exaggerated greenup and senescence and would maintain higher VI values in winter.

Due to the coarse spatial resolution of remote sensing data, relative to plants, it is challenging to directly observe specific phenology events such as flowering or leaf out. Often, remotely sensed data is used to indirectly estimate the timing of phenology events or to compute metrics that characterise the stages and nature of vegetation growth. These metrics are referred to as *land surface phenology* metrics ([Zeng et al., 2020](#)) or seasonal parameters ([Jönsson and Eklundh, 2004](#)).

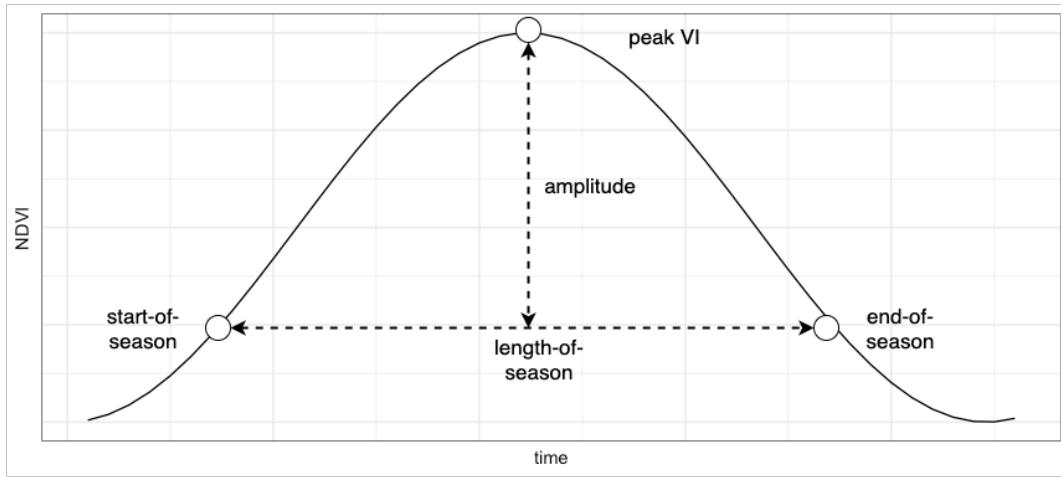


Figure 7.1: Phenology as represented by a temporal profile of vegetation index values.

Phenology metrics are computed from time-series of vegetation index values and are used to characterise the phenology at a location and for a given year. Common phenology metrics include start-of-season, end-of-season, length-of-season, time-of-peak-VI, peak VI, amplitude, area under the curve, and rates of greenup and senescence. Peak VI and area under the curve are often used as indicators of biomass.

Time-series of vegetation index values are used as inputs to functions which contain rules for computing phenology metrics. For example, peak VI can be estimated by applying the `max()` function to a subset of vegetation index values between start-of and end-of-season. Start-of-season can be estimated by a search of vegetation index values for exceedance of a threshold value or the detection of a valley point in vegetation index time-series. Rate-of-greenup can be estimated by the slope between start-of-season and peak VI.

What are advantages of using satellite remote sensing over field-based monitoring of phenology?

Spatial coverage - it is easier to monitor vegetation and ecosystem dynamics at spatial extents not possible to cover by field-based observations.

Temporal coverage - it is costly to make repeat observations over time using field-based or in-situ methods. Satellite remote sensing provides an affordable way to monitor the same location within-seasons and over many years.

What are advantages of using in-situ or field-based methods to monitor phenology?

Monitor the phenology of specific species or plants.

Monitor specific phenological events which are hard to detect with satellites at coarse and moderate resolutions such as flowering or budding.

7.1 Urban Park Phenology

In this lab you will monitor the phenology over an urban park in Perth and you will be working with Sentinel-2 satellite images. Sentinel-2 data is a European Space Agency product with a 5-day temporal resolution and a 10 m spatial resolution. You will:

- Subset Sentinel-2 images for 2020.
- Mask out cloudy pixels.
- Transform the Sentinel-2 multispectral images into NDVI images.
- Fill gaps caused by cloudy pixels in NDVI time-series.
- Use a smoothing algorithm to remove remnant noise in NDVI time-series.
- Estimate start-of-season and length-of-season for park vegetation.
- Explore relationships between phenology metrics and climate variables.

To complete these tasks you will use various data transformation and visualisation functions that have been introduced in previous labs.

7.2 Setup

Add a header to the start of your script.

```
/*
Lab 8
Author: Test
Date: XX-XX-XXXX

*/
```

Next we need to import a module that provides functions for time-series and phenology analysis.

```
// module with functions for phenology analysis
var phenology = require('users/jmad1v07/Phenology:phenology.js');
```

We also need to specify our study area. Here, we'll be working in a section of Kings Park.

```
var geometry = ee.Geometry.Polygon(
  [[[115.83102144426752, -31.95262028223891],
    [115.83102144426752, -31.956297928621538],
    [115.83679355806761, -31.956297928621538],
    [115.83679355806761, -31.95262028223891]]]);
```

```

// centre the map on the study area
Map.centerObject(geometry, 16);

// create an empty image into which to paint the features, cast to byte.
var empty = ee.Image().byte();

// study area outline
var outline = empty.paint({
  featureCollection: geometry,
  color: 1,
  width: 3
});
Map.addLayer(outline, {palette: '000000'}, 'study area');

```

7.3 Data Preprocessing

Before we can use Sentinel-2 surface reflectance data for phenology analysis, it needs to go through several data transformation and preprocessing steps. These include:

- **filter** operations to subset the Sentinel-2 `ImageCollection` to the time-period and study area of interest. This requires implementing a non-spatial filter using `Image` metadata (date of `Image` capture) and spatial filtering based on the `Image` footprint intersecting the study area.
- **creating a new variable** using *local operations* that are applied at the pixel level to combine bands representing red and near infrared reflectance into a normalised difference vegetation index (NDVI) band.
- **filter** and *local operations* to compare `Image` bands to a cloud mask and mask out pixels where cloud was detected.

To implement data preprocessing, we will need two functions. One function to generate per-`Image` cloud masks and another function to compute the NDVI.

```

// function to mask clouds in Sentinel 2 images
function maskS2Clouds(image) {
  var qa = image.select('QA60');
  var cloudBitMask = 1 << 10;
  var cirrusBitMask = 1 << 11;
  var mask = qa.bitwiseAnd(cloudBitMask).eq(0).and(
    qa.bitwiseAnd(cirrusBitMask).eq(0));
  return image.updateMask(mask).divide(10000)
}

```

```

    .select("B.*")
    .copyProperties(image, ["system:time_start"]);
}

// function that adds an NDVI band per-image
function addNDVIAndClip(image) {
  var ndvi = image.normalizedDifference(['B8', 'B4']).rename('ndvi');
  image = image.addBands(ndvi);
  return image.clip(geometry);
}

```

Finally, import the Sentinel-2 surface reflectance `ImageCollection` and chain together the various data preprocessing operations.

```

// import Sentinel 2 ImageCollection, cloud mask, and add NDVI band
var s2 = ee.ImageCollection("COPERNICUS/S2_SR");

var startDate = ee.Date('2020-01-01');
var endDate = ee.Date('2021-03-01');

var s2Ndvi = s2
  .filter(ee.Filter.date(startDate, endDate))
  .filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 30))
  .filter(ee.Filter.bounds(geometry))
  .map(maskS2Clouds)
  .map(addNDVIAndClip)
  .select('ndvi');

```

Print the `s2Ndvi` `ImageCollection` to the *Console* to check that it stores Sentinel-2 Images with an `ndvi` band.

```
print('Check the 2020 S2 NDVI collection:', s2Ndvi);
```

Why is it important to remove clouds from remote sensing images before computing phenology metrics?

If cloudy observations remain in VI time-series it will create short-run variation in greenness which could be mistaken for phenological events or bias estimates of phenological events.

7.4 Noise Removal and Gap Filling

Let's explore the NDVI time-series in the study area. Run the following code to load an interactive chart generator; click on a location within the study area on the map and an NDVI time-series chart will be rendered in the *Console*.

```
// draw NDVI time series chart at clicked points
var drawChart = function(point) {
    var clickedPoint = ee.Geometry.Point(point.lon, point.lat);

    var tsChart = phenology.timeSeriesChart(clickedPoint, s2Ndvi.select(['ndvi']), 'NDVI',
        print(tsChart);

};

Map.add(ui.Label('Click anywhere within study area - NDVI time-series will appear in Console');
Map.onClick(drawChart);
```

As you click and render time-series charts of vegetation growth there are a couple of things to look out for. First, the seasonal profile of vegetation growth with greenup beginning in late austral summer and peak greenness in austral winter should be apparent. Second, it is likely that you will see short-run variation in NDVI values that are superimposed on the dominant seasonal profile. There could also be some gaps in the time-series which correspond to cases where the cloud mask was applied. This short-run variation in NDVI values could be noise and is not caused by actual changes in vegetation condition. The source of this noise could be clouds missed by the cloud mask, other sources of atmospheric contamination, or changes in illumination. Vegetation growth is typically characterised by smooth, continuous, and sustained increases and then decreases in greenness. Functions to detect phenology metrics look for transitions from increasing to decreasing vegetation growth, or vice versa, or rates of increase or decrease in vegetation growth over time. This noise in NDVI time-series could confound detection of phenology parameters; thus, NDVI time-series are filtered to remove anomalous values.

Let's apply a function to our NDVI time-series to remove any anomalously high or low NDVI values. The approach we'll use here is based on [Kong et al. \(2021\)](#) and defines a noisy spike or low NDVI value as cases where the difference between a data point and the NDVI values of its temporal neighbours is greater than one and a half times the standard deviation of the entire NDVI time-series.

We can use the `removeSpikes()` function from the `phenology` module to filter out anomalous NDVI values. The `removeSpikes()` function expects to receive as arguments:

- An `ImageCollection` of VI values.
- A number that defines the window size of temporal neighbours for each data point.
- The band name.

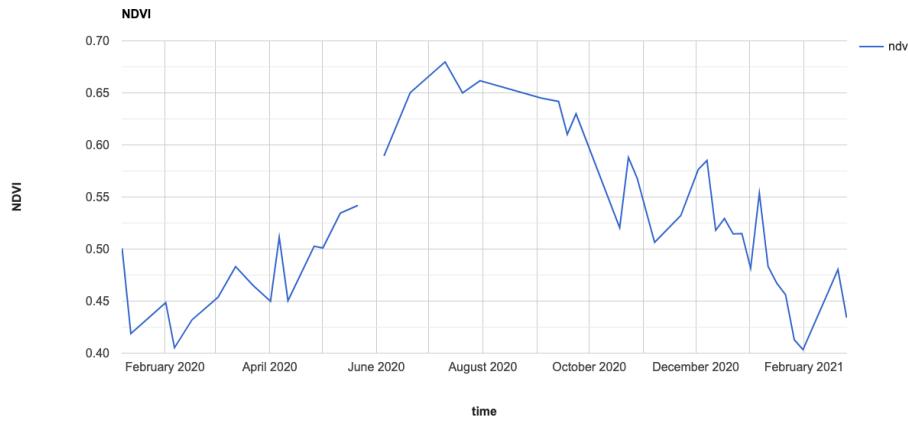


Figure 7.2: NDVI time-series showing dominant seasonal profile of vegetation growth, short-run noise in NDVI time-series, and gaps due to cloud masking.

- The number of standard deviations that defines a noisy value.
- The time property for each `Image` in the collection.

```
// add a timestamp band for each image
var s2NdviTs = s2Ndvi.map(phenology.addTimeBand('system:time_start', 'timestamp'));

// remove spikes and drop outs
// there can be noisy pixel values that are missed by cloud masks
// these noisy pixel values can affect representations of vegetation growth profiles and p
// remove noisy pixel values - anomalous values
var s2RemovedSpikes = phenology.removeSpikes(s2NdviTs, 15, 'ndvi', 1.5, 'system:time_start')
```

We can explore the result of removing anomalous NDVI values from our time-series. Run the following code snippet and click on the map to generate charts of NDVI time-series with spikes removed.

```
// Removed spikes NDVI chart
var drawChart = function(point) {
    var clickedPoint = ee.Geometry.Point(point.lon, point.lat);

    var tsChart = phenology.timeSeriesChart(clickedPoint, s2RemovedSpikes.select(['ndvi'])
    print(tsChart);

};
```

```
Map.onClick(drawChart);
```

Through cloud masking and filtering anomalous NDVI values we have removed some of the noise from our time-series. However, now there are gaps in the time-series which is problematic when using these time-series to represent profiles of vegetation growth. To fill in the gaps in the NDVI time-series, we can use the `gapFill()` function from the `phenology` module. This function takes in as arguments:

- An `ImageCollection` where masked pixels in its constituent images will be filled with sensible VI values.
- A number specifying a date range to search for neighbouring images for each data point in the time-series. The valid VI values of these neighbouring images will be used to interpolate and fill the missing VI values.
- The time property for each `Image` in the collection.
- The name of the timestamp band in each `Image`.

Gap fill the NDVI `ImageCollection` and explore the results of gap filling by rendering charts in the *Console*.

```
// gap fill cloudy pixels using average of nearest cloud free before and after images
var s2GapFilled = phenology.gapFill(s2RemovedSpikes, 30, 'system:time_start', 'timestamp')

// gap filled NDVI chart
var drawChart = function(point) {
    var clickedPoint = ee.Geometry.Point(point.lon, point.lat);

    var tsChart = phenology.timeSeriesChart(clickedPoint, s2GapFilled.select(['ndvi']), 'timestamp');
    print(tsChart);

};

Map.onClick(drawChart);
```

You should see cases where spikes in NDVI values have been caught, removed, and gap filled.

7.5 Smoothing

Even after removing anomalous NDVI values and then gap filling the time-series, the NDVI profiles still do not completely represent the smooth continuous greenup and senescence that is characteristic of vegetation growth. Therefore, many phenology workflows use smoothing models to remove the short-run fluctuations in VI values and retain the dominant profile of

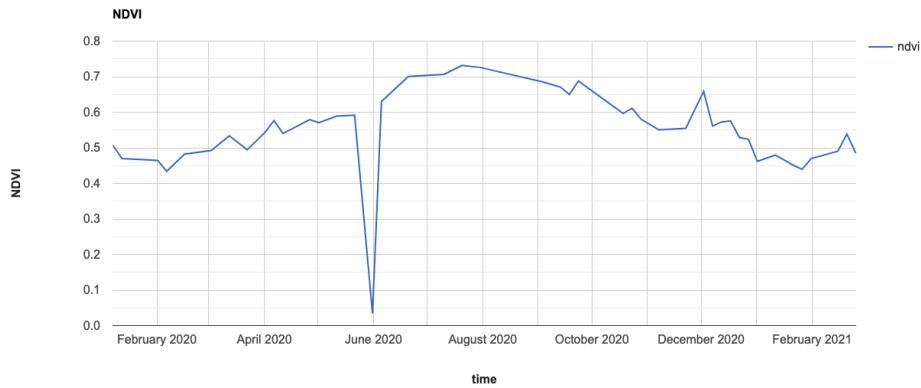


Figure 7.3: NDVI time-series with noisy values remaining after cloud masking.

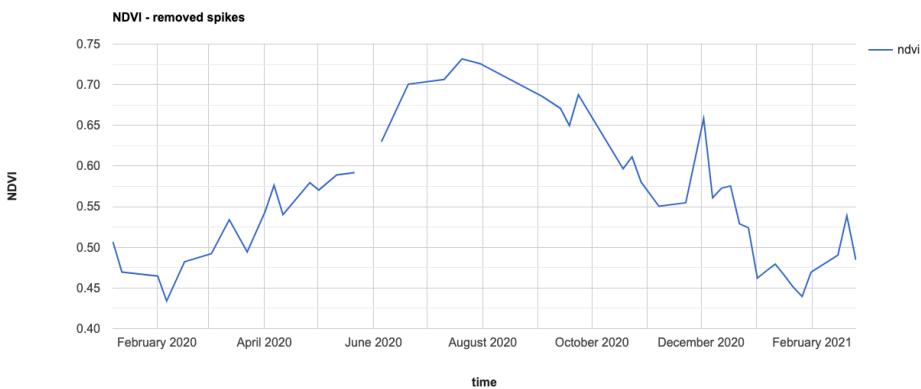


Figure 7.4: NDVI time-series with noisy values removed.

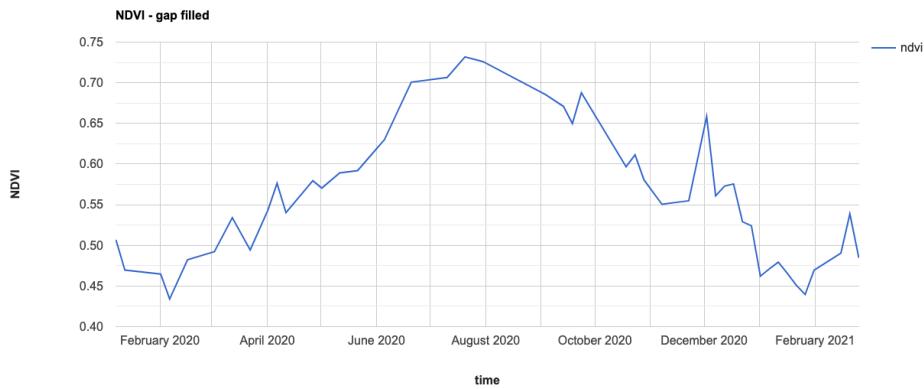


Figure 7.5: NDVI time-series with noisy values removed and gap filled.

seasonal vegetation growth. Generally, it is easier to accurately detect key phenological events such as start-of-season and length-of-season from smoothed vegetation index time-series.

A range of different smoothing models exist and have been successfully applied to phenology applications. You can read [Atkinson et al's. \(2012\)](#) intercomparison of different smoothing approaches. Generally, smoothing is a form of time-series analysis where a model is specified to represent characteristics of the time-series and then fitted using observed time-series values. A key feature of smoothing is the selection of model parameters that determine the degree to which a time-series is smoothed.

There is a trade-off between models that are more or less sensitive to short-run variation in VI time-series. Models that are more sensitive to short-run VI variation might preserve the signal of short-lived phenological events such as flowering, but at the cost of retaining more noise in the time-series.

A simple model of a time-series is representing it as a linear model of time:

$$y_t = \beta_0 + \beta_1 t + \epsilon_t$$

The goal of model fitting is to find estimates of β_0 and β_1 , $\hat{\beta}_0$ and $\hat{\beta}_1$. You can think of this as fitting a linear regression model to generate an estimate, \hat{y}_t , at time t . Here, you would be representing your time-series with a model of a linear trend in NDVI values through time. The blue line in the figure below is a linear model of vegetation growth fitted to the black points which are a time-series of NDVI values.

Is a linear model of time (i.e. linear trend model) a good way to represent the profile of vegetation growth in NDVI time-series?

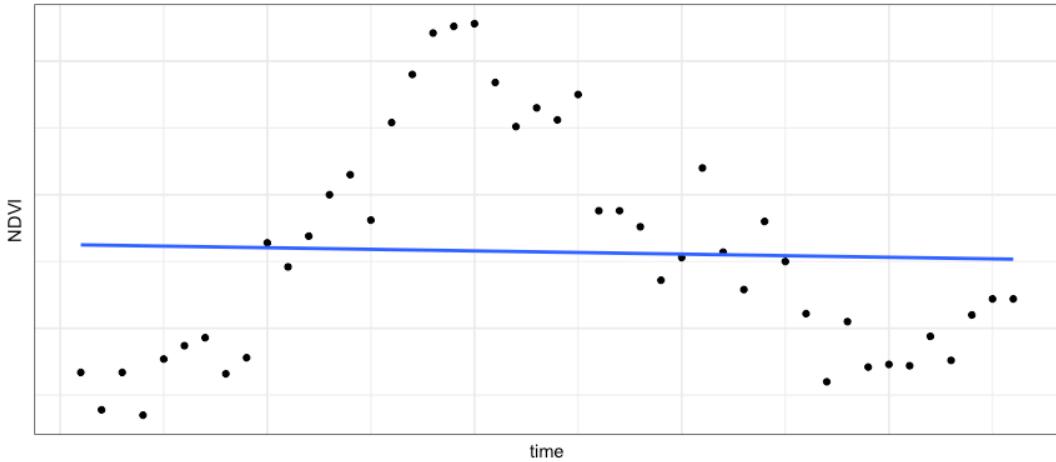


Figure 7.6: Fitting a linear trend model to represent the profile of vegetation growth.

No, vegetation growth has a clear cyclical cycle of greenup, peak greenness, followed by senescence. A linear trend model can only represent vegetation growth as a linear pattern of increasing or decreasing greenness.

Think about specifying a model of a NDVI time-series that accurately reflects vegetation growth patterns. What features of the time-series should the model be able to represent?

Non-linear increases and decreases in NDVI values.

The cyclical nature of vegetation growth dynamics.

If you are familiar with sinusoid waves, you might notice similarities between the seasonal profiles of vegetation growth and sine or cosine waves.

Sinusoidal waves are described by the amplitude, frequency, and phase of the curve. Without needing to go into too much detail, which can be found [here](#) and [here](#), sinusoidal waves can be expressed mathematically as a function of their amplitude A , frequency f , and phase ϕ :

$$y_t = A \cos(2\pi w t + \phi)$$

Where w is $1/f$, which is the number of complete cycles of a sinusoidal wave. This model can be rewritten as:

$$y_t = \beta_0 + \beta_1 t + \beta_2 \cos(2\pi w t) + \beta_3 \sin(2\pi w t) + \epsilon_t$$

This is just a case of estimating another linear regression model that generates predictions, \hat{y}_t , that represent a sinusoidal wave with a frequency, f , that best fits the NDVI values in our

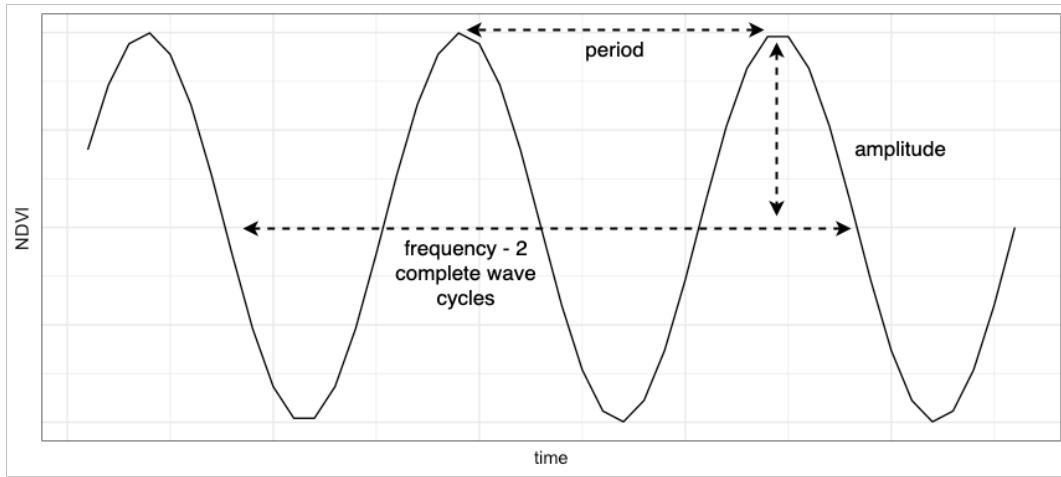


Figure 7.7: Representing NDVI time-series as a sinusoidal wave.

time-series. The assumption here is that the best fitting sinusoidal wave accurately captures the signal of smooth continuous vegetation growth and does not capture short-run noise. This is often called a harmonic model of time.

In the Kings Park study area, there is generally one growing season per year. So, we'll estimate this model with w set as 1 where a single sinusoidal wave will represent the growing season.

We can use the `hants()` function in the phenology module to fit this model to NDVI time-series in each pixel and return an `ImageCollection` with smoothed NDVI values.

Run the following code and compare the `fitted` and input `ndvi` values on charts by clicking in the study area. The `fitted` time-series should be a smooth curve the fits input NDVI values.

```
// even after gap filling there is noise in the NDVI profiles
// the NDVI profiles do not reflect the smooth and continuous green up and senescence of vegetation
// smooth the gap filled time series with a harmonic model of time
var s2Hants = phenology.hants(s2GapFilled.select(['ndvi']), 1, 'ndvi');

// draw NDVI time series chart at clicked points
var drawChart = function(point) {
    var clickedPoint = ee.Geometry.Point(point.lon, point.lat);

    var tsChart = phenology.timeSeriesChart(clickedPoint, s2Hants.select(['fitted', 'ndvi']));
    print(tsChart);

};
```

```
Map.onClick(drawChart);
```

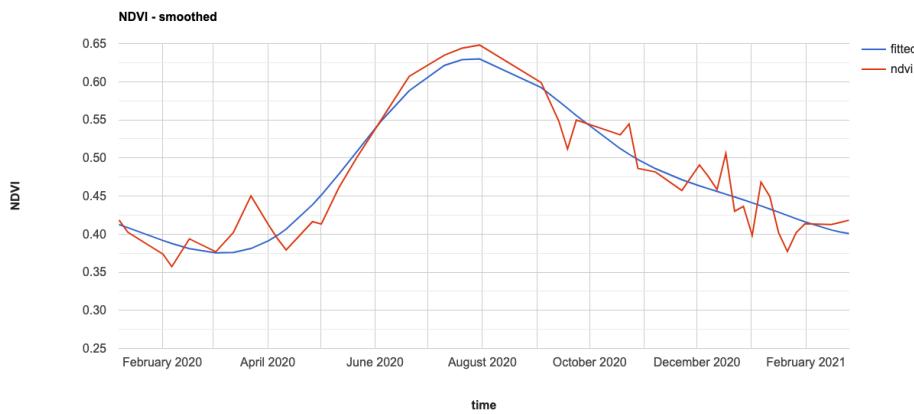


Figure 7.8: Time-series of NDVI values smoothed using a harmonic model of time.

7.6 Non-vegetation Mask

To ensure that we are only monitoring the phenology of vegetated land cover, we will mask out non-vegetated areas by flagging pixels with NDVI time-series that do not display clear patterns of vegetation growth.

The approach used here is based rules used in NASA's Vegetation Index and Phenology (VIP) product to mask out non-vegetated pixels. For a pixel to be classed as having a valid growing season, the:

- maximum growing season NDVI should be greater than a threshold value (NDVI > 0.12 is the NASA VIP default).
- a minimum delta NDVI (maximum NDVI - minimum NDVI) greater than a threshold ($\Delta\text{NDVI} > 0.05$ is the NASA VIP default).
- there should be a sustained increase in NDVI values for n days (here we will require there to be at least 40 days of sustained vegetation growth).

Can you explain why these rules are sensible for masking out locations where there is no vegetation growth?

A maximum growing season NDVI threshold ensures that there is some green vegetation present in a pixel.

A Δ NDVI greater than a threshold ensures the difference between peak greenness and low greenness is larger than small noisy fluctuations in spectral reflectance.

A sustained increase in NDVI values is a characteristic of vegetation growth dynamics and will distinguish actual vegetation growth for small fluctuations in spectral reflectance that could still occur over non-vegetated pixels.

The `checkSeasonsVip` function requires four arguments:

- An `ImageCollection` with `Images` to check for valid pixels.
- The maximum growing season NDVI threshold (here 0.12).
- The Δ NDVI threshold (here 0.1).
- The time step, in days, between `Images` in the `ImageCollection` (here 5).
- The required number of days of a sustained increase in NDVI for a growing season to be valid (here 40).

```
// check for valid seasons using the NASA VIP rules
// max season NDVI > a threshold (default 0.12)
// minimum season delta NDVI > a threshold (default 0.05)
// sustained increase in VI values for n days
var seasonMask = phenology.checkSeasonsVip(s2Hants.select('fitted'), 0.12, 0.1, 5, 40);
```

We will keep our valid season mask in the variable `seasonMask` and use it at the end of our workflow to mask maps of phenology metrics such as start-of-season and length-of-season.

7.7 Phenology Metrics

From smoothed time-series of NDVI values, we will estimate the following phenology metrics:

- Start-of-season.
- End-of-season.
- Length-of-season.
- Peak greenness.

One approach to estimating start-of-season and end-of-season is using a fixed NDVI threshold. When a sequence of increasing NDVI values exceed the fixed threshold, start-of-season is detected. End-of-season is defined in a similar way, when a sequence of decreasing NDVI values pass the fixed threshold, end-of-season is detected.

What is a potential limitation of using fixed NDVI thresholds for detecting start-of-season and end-of-season?

Different plant species, mixes of plant species in a pixel, or canopy coverage within a pixel will result in different NDVI time-series. Thus, a fixed NDVI threshold that works for one context is unlikely to transfer to different locations.

Similarly, defining thresholds in one growing season might not be suitable for future or historical seasons due to inter-annual climate variation.

To overcome the limitations of using fixed NDVI thresholds to detect phenology metrics, a range of different approaches have been proposed and used. One approach that advances upon fixed NDVI thresholds is to use local (pixel-specific) thresholds applied to a time-series of NDVI values that have been transformed into NDVI ratio values. The NDVI ratio is computed as:

$$NDVI_{ratio} = \frac{NDVI - NDVI_{min}}{NDVI_{max} - NDVI_{min}}$$

NDVI ratio values fall between 0 and 1. A threshold value such as 0.5 ([While et al., 2009](#)) or 0.3 ([Didan et al., 2018](#)) have been used. When increasing NDVI ratio values exceed this threshold, start-of-season is detected. As this threshold is applied to NDVI values transformed into the NDVI ratio, the absolute values of the threshold will vary from pixel to pixel depending on the amplitude of vegetation growth in each pixel's time-series.

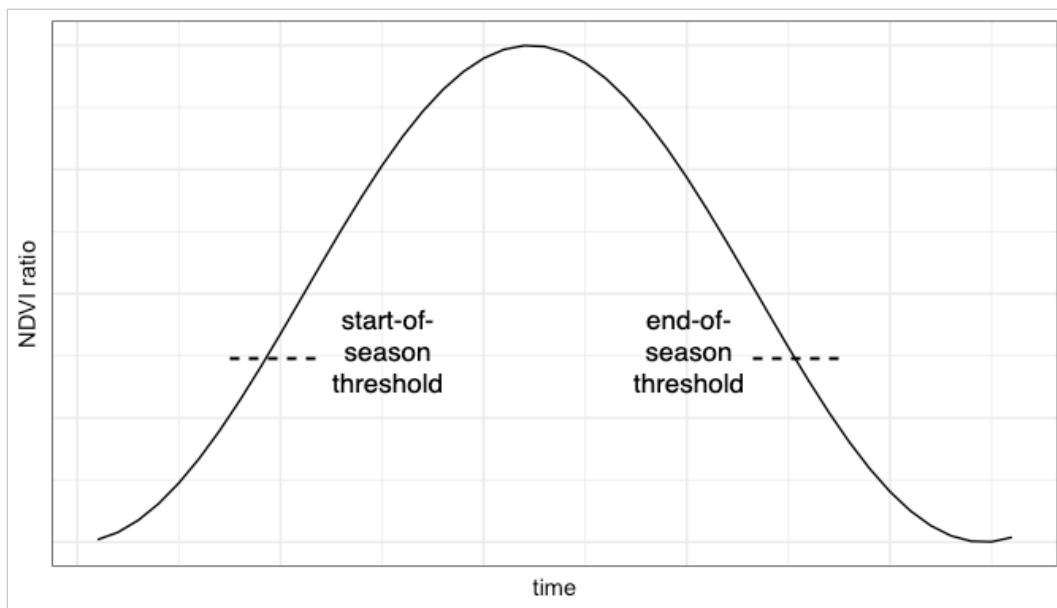


Figure 7.9: Detecting start-of-season and end-of-season using thresholds applied to NDVI ratio time-series.

We can use the `sosRatioThreshold()` function to generate a start-of-season date `Image` from an `ImageCollection` of smoothed NDVI values. The `eosRatioThreshold()` function can

be used to generate an end-of-season date Image and the `peakVi()` function can be used to generate an Image with bands corresponding to peak NDVI and date of peak NDVI.

Run the following code to generate start-of-season, end-of-season, and peak NDVI Images and use the `seasonMask` to mask pixels where there was no vegetation growing.

```
// phenology parameters
// start-of-season
var sos = phenology.sosRatioThreshold(s2Hants, ['fitted'], 10, 0.3)
  .updateMask(seasonMask);
Map.addLayer(sos.select('day'), {min: 0, max: 150, palette:['440154', '433982', '30678D',
  // end-of-season
  var eos = phenology.eosRatioThreshold(s2Hants, ['fitted'], 10, 0.3)
    .updateMask(seasonMask);
  Map.addLayer(eos.select('day'), {min: 200, max: 350, palette:['440154', '433982', '30678D',
  // peak NDVI
  var peakVi = phenology.peakVi(s2Hants, ['ndvi'])
    .updateMask(seasonMask);
  Map.addLayer(peakVi.select('fitted'), {min: 0.6, max: 0.85, palette:['ffffcc', 'd9f0a3', 'ffcc99']});
```

Length-of-season is the difference between end-of-season and start-of-season? You have used Image math operations in Lab 7 where you subtracted one Image from another. Can you use the sos and eos Images to generate a length-of-season Image?

```
var los = eos.subtract(sos)
  .updateMask(seasonMask);
Map.addLayer(los.select('day_from_start'), {min: 200, max: 350, palette:['440154', '433982']});
```

If you head to the *Inspector* tab, you can click on the map display and explore Image values at clicked locations. You can toggle the `sos`, `eos`, and `peakVi` layers on and off to see maps of the phenology metrics. You can also compare start-of-season and end-of-season at clicked locations with their NDVI time-series charts which will be generated in the *Console*.

7.8 Phenology and Climate

Now we have extracted phenology metrics from our NDVI time-series, we can use them in analysis and modelling tasks. Common analyses using phenology metrics derived from remote sensing images include assessing drivers of phenology (e.g. how does changes in climate or weather affect the timing of phenology events) or undertaking spatio-temporal trend analysis to characterise how ecosystems are changing over time.

Here, we will undertake a simple analysis to explore the relationship between summer land surface temperatures and start-of-season dates in Kings Park. Several observational studies have detected urban climates affecting the timing of phenological events.

Let's create an `Image` of average land surface temperatures from October 2019 to the end of February 2020. Broadly, this period corresponds to just before the start-of-season. We will use the land surface temperature band included in the Landsat 8 surface reflectance product and convert it from Kelvin to Degrees Celsius.

```
// Climate and phenology relationships

// apply scaling factors
function maskL8Sr(image) {
    // Bit 0 - Fill
    // Bit 1 - Dilated Cloud
    // Bit 2 - Cirrus
    // Bit 3 - Cloud
    // Bit 4 - Cloud Shadow
    var qaMask = image.select('QA_PIXEL').bitwiseAnd(parseInt('11111', 2)).eq(0);
    var saturationMask = image.select('QA_RADSAT').eq(0);

    // Apply the scaling factors to the appropriate bands.
    var opticalBands = image.select('SR_B_').multiply(0.0000275).add(-0.2);
    var thermalBands = image.select('ST_B.*').multiply(0.00341802).add(149.0);

    // Replace the original bands with the scaled ones and apply the masks.
    return image.addBands(opticalBands, null, true)
        .addBands(thermalBands, null, true)
        .updateMask(qaMask)
        .updateMask(saturationMask);
}

var lst = ee.ImageCollection("LANDSAT/LC08/C02/T1_L2")
    .filterDate('2019-10-01', '2020-03-01')
    .filterBounds(geometry)
    .map(maskL8Sr)
    .mean()
    .select('ST_B10')
    .subtract(273.15)
    .rename('surface_temp_C')
    .clip(geometry);
```

```
Map.addLayer(lst, {min: 30, max: 45, palette:['00FFFF', 'FFFF00', 'FF0000']}, 'LST (C)');
```

You should see a map of land surface temperature for the study area.

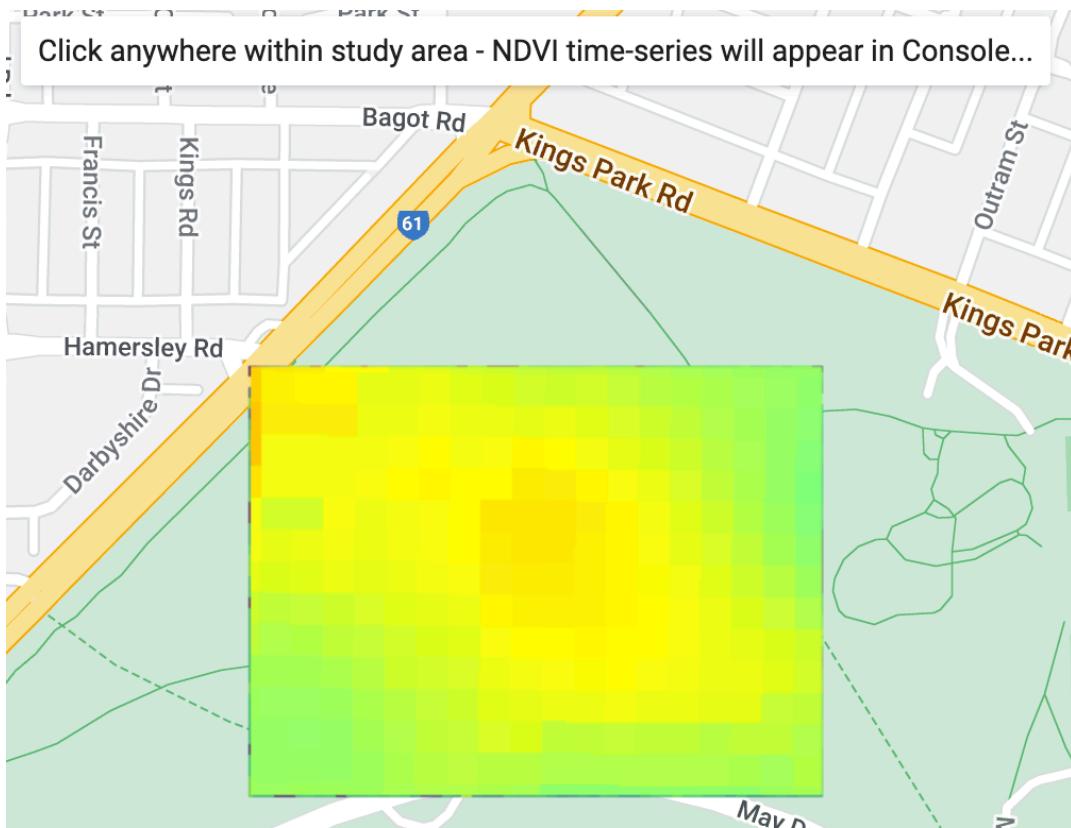


Figure 7.10: Landsat 8 land surface temperature over Kings Park.

Scatter plots can visualise bivariate relationships between two variables. We will use a scatter plot to explore if there is a relationship between land surface temperature start-of-season in our study area.

First, we need to combine the start-of-season `Image` with our land surface temperature `Image`. Let's add `lst` as a band to the `sos Image`. Then, we will sample 300 locations in the study area extracting the land surface temperature and start-of-season values at each location.

```
var sosLst = sos.addBands(lst);

var sosLstPoints = sosLst.sample({
  region: geometry,
  scale: 10,
```

```

    numPixels: 300,
    seed: 123,
    geometries: true
});

Map.addLayer(sosLstPoints, {}, 'sample points');

```

Finally, let's generate a scatter plot to see if there is a relationship between land surface temperature and start-of-season.

```

// print lst v start-of-season chart on the console
// add a linear trendline
var lstSosChart =
    ui.Chart.feature
        .byFeature({
            features: sosLstPoints,
            xProperty: 'surface_temp_C',
            yProperties: ['day']
        })
        .setSeriesNames([''])
        .setChartType('ScatterChart')
        .setOptions({
            title: 'LST and start-of-season relationship',
            hAxis:
                {title: 'Temperature (°C)', titleTextStyle: {italic: false, bold: true}},
            vAxis: {
                title: 'Start of season',
                titleTextStyle: {italic: false, bold: true},
                viewWindow: {min: 50, max: 150}
            },
            pointSize: 10,
            trendlines: { 0: {} }
        });
print(lstSosChart);

```

This has just been a quick demo of how to explore relationships between climatic drivers and phenology. If you were going to expand this analysis to complete a more comprehensive analysis of climate-phenology relationships, what would you need to consider?

We should consider the vegetation species present in the case study area, and, if, temperature before the growing season is an important driver of their phenology. Other climatic drivers (e.g. precipitation and moisture availability or insolation) could affect phenology. The analysis could be expanded to incorporate other drivers of phenology to guard against omitted variable

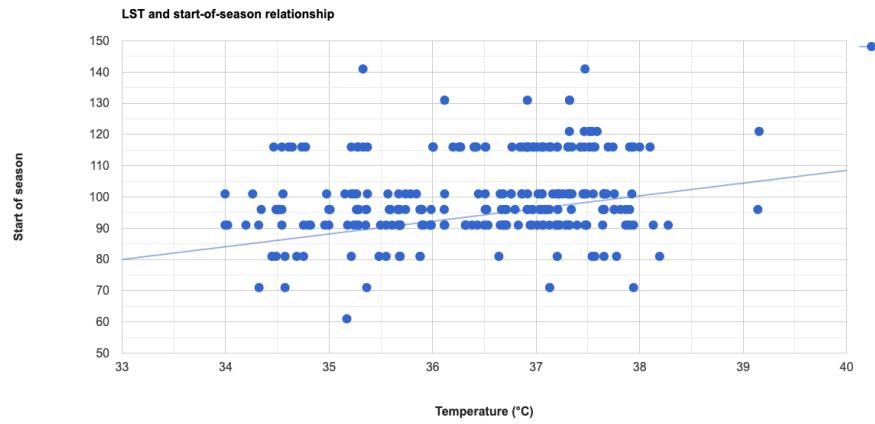


Figure 7.11: Relationship between land surface temperature and start-of-season date (day of year).

bias (e.g. (mis)attributing the effect of precipitation on phenology to surface temperatures); this could be done using multiple regression analysis. There are a few other things to explore such as the mix of vegetation species within a 10 m Sentinel-2 pixel footprint, and, if they're likely to have similar phenology profiles.

8 Introduction

This lab will introduce a workflow to classify **Images** in Google Earth Engine using supervised classification algorithms.

Image classification is the process of classifying pixels in an **Image** into similar classes (groups or clusters of similar pixels) or categories (a label that relates to physical or land surface characteristics).

You will use a supervised classification algorithm to assign each pixel a label based upon spectral reflectance values. Land cover mapping is a common image classification task. Different land cover types have different spectral signatures (i.e. they have different levels of reflectance in different portions of the electromagnetic spectrum); an algorithm can be trained to predict each pixel's land cover type by learning rules that relate spectral reflectance properties to land cover types.

You have a predictor dataset X which contains p predictor variables X_1, X_2, \dots, X_p ; in the case of land cover classification using remotely sensed data p often corresponds to wavelengths of the electromagnetic spectrum. Your goal is to train a classifier (algorithm) f which relates predictor variables X to an outcome Y .

$$Y = f(X)$$

The video below presents a good overview of tools for **Image** classification in Google Earth Engine. It is quite long, so best watched after the lab to consolidate what you have learnt.

An overview of **Image** classification in Google Earth Engine.

8.0.1 Supervised Classification

In supervised classification, to train a classifier the target values of Y need to be known. The goal is to train a classifier, \hat{f} , that predicts target labels when it is given a set of input predictor variables, X_1, X_2, \dots, X_p . Training a classifier involves learning rules that relate values of predictor variables, X_1, X_2, \dots, X_p , to the known target labels of Y . For each training observation x_i there is a known y_i . You train a classifier by finding rules that minimise a metric of classification error when predicting the known y_i .

A classification tree is one of the simplest supervised classifiers to see inside and to visualise the algorithm's rules for making predictions. A classification tree consists of a series of yes / no splits for values of X_p , the predictor variables (e.g. spectral reflectance measured in p wavelengths). The leaves of a classification tree correspond to the target label (e.g. a land cover type). Training a decision tree classifier involves finding splits in the values of X_p (i.e. learning rules) that minimise the classification error that occurs when a data point with a known target label is assigned to the wrong leaf node. When using a decision tree to predict an unknown target label, you move down the tree following the yes / no splits to arrive at a leaf node. The leaf node you end up in is the predicted target label.

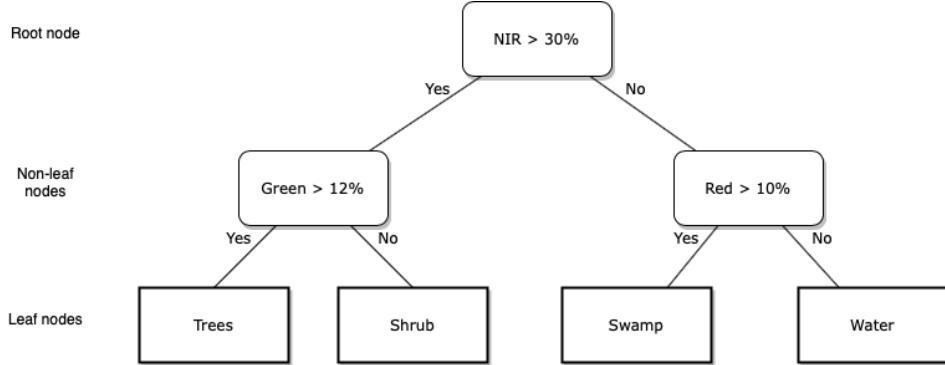


Figure 8.1: Example classification tree showing how land cover can be predicted from spectral reflectance values (adapted from Tso and Mather (2009)).

The final step of a supervised classification workflow is to assess the accuracy of your model. You see how well your model performs in predicting Y for data that was not used in training the model (test data). This gives you an indication of your model's performance and how well it will generalise to cases where Y is not known.

8.0.2 Unsupervised Classification

In unsupervised classification, for each observation x_i in the training data there is no observed outcome y_i . An unsupervised classification algorithm f assigns observations to clusters based upon them having similar values in the predictor variables X . In the context of remote sensing image classification, an unsupervised classification algorithm would assign a pixel i to a class j based upon its x_i spectral reflectance values. Your goal, given some training data, is to train an unsupervised classification algorithm \hat{f} that assigns pixels to classes so that the within-class (within-cluster) variability in pixel values is minimised. Unsupervised classifiers in Google Earth Engine are called **clusterers**; this name refers to goal of unsupervised classifiers to create clusters of similar observations. Classes (clusters) do not directly relate to ground objects, real world phenomenon or entities, or land cover types; instead, they define clusters of similar pixels in terms of their spectral reflectance values. Often, image classes

relate to interesting land surface characteristics but it is a post-processing step to formalise these relationships and assign meaningful labels to image classes.

8.1 Task

You will perform a supervised classification of a Landsat 8 Image to predict a pixel's land cover type.

Supervised classification

1. Prepare labelled training data comprising observations of Landsat 8 spectral reflectance values and a land cover type property.
2. Train a supervised classifier to predict a pixel's land cover type (water, built up, or vegetation).
3. Predict each pixel's land cover type in the Landsat 8 Image.
4. Assess the accuracy of your land cover prediction using independent test data.

8.2 Setup

```
/*
Lab 9
Author: Test
Date: XX-XX-XXXX

*/
```

8.2.1 Data Import

```
// Data import

// study area
var bBox =
    /* color: #d63000 */
    /* displayProperties: [
        {
            "type": "rectangle"
        }
    ] */
```

```

ee.Geometry.Polygon(
  [[[115.23492202085255, -31.663337604046916],
    [115.23492202085255, -32.220373273114554],
    [116.13442763608693, -32.220373273114554],
    [116.13442763608693, -31.663337604046916]]], null, false);

//Get Landsat 8 SR Image
var l8SR = ee.Image('LANDSAT/LC08/C01/T1_SR/LC08_113082_20160802').clip(bBox);
print(l8SR);

// Use these Landsat 8 bands and rescale spectral reflectance values
var bands = ['B2', 'B3', 'B4', 'B5', 'B6', 'B7'];
l8SR = l8SR.select(bands).divide(10000);

/* Define the visualization parameters. The bands option allows us to specify which bands
var vizParams = {
  bands: ['B4', 'B3', 'B2'],
  min: 0,
  max: 0.4,
};

// Centre the display and then map the Landsat 8 image
Map.centerObject(l8SR, 10);
Map.addLayer(l8SR, vizParams, 'RGB composite');
print(l8SR);

```

Use the *Inspector* tab to visualise the difference in Landsat 8 spectral reflectance across different wavelengths for different land cover types.

Visualise spectral reflectance using the *Inspector* tab

8.3 Supervised Classification

8.3.1 Training (Feature) Engineering

The first task in a supervised classification workflow is to prepare the training data. This is called training or feature engineering (in machine learning predictor variables are often called features).

Classifiers in Google Earth Engine require a `FeatureCollection` for training where one property in the `properties` object is the outcome variable and the other properties are the predictors.

The `FeatureCollection` `trainingPoints` contains three `Feature` objects. Each `Feature` object has a multipoint `Geometry` object and a `properties` object containing the property `lc`. `lc` refers to the land cover type at each point's location.

An `lc` value of 0 corresponds to vegetated land cover, an `lc` value of 1 corresponds to built up land cover, and an `lc` value of 2 corresponds to water.

These labelled training points were created through manual digitisation guided by visual inspection of the Google satellite basemap.

```
// Training engineering
// Import points labelled with land cover type property
var trainingPoints = ee.FeatureCollection("users/jmad1v07/gee-labs/training_points");
print(trainingPoints);
Map.addLayer(trainingPoints, {color:'red'}, 'training data');
```

The date of the Google satellite basemap used to guide the creation of the training data was not known. What problem could this introduce when using this training data to train a classifier to predict the land cover type of Landsat 8 pixels?

Land cover might have changed between the date of capture of the Google satellite basemap and the date of the Landsat 8 `Image` you wish to classify. This means that the labelled training data may not accurately reflect the land cover type at the time the Landsat 8 `Image` was captured.

Explore the red points on the map that represent the location of the labelled training points. You should see clusters of points in Perth CBD and Kings Park. What problem could this clustering of training data present?

Training data observations which are close to each other in space might have spatial correlation among their data values. This spatial correlation reduces the effective number of independent training data observations. This can be problematic as non-parametric classifiers, such as CART, require large numbers of training data observations to characterise the relationships between predictor variables and outcome categories.

You now need to find the Landsat 8 spectral reflectance values that intersect with each of the point locations.

There is a convenient `sampleRegions()` function in Google Earth Engine that helps with feature engineering. The `sampleRegions()` function samples the pixels of an input `Image` (`18SR` here) using the `Geometry` objects specified as the `collection` argument (`trainingPoints` here). This function extracts the values in each band of `18SR` that intersect with each point's location.

```
// Sample spectral reflectance at training points
var trainingData = 18SR.sampleRegions({
  collection: trainingPoints,
  properties: ['lc'],
  scale: 30
});
print(trainingData);
```

Inspect the `FeatureCollection` `trainingData` in the *Console*. You should see it contains 129 `Feature` objects and each `Feature` has a `properties` object containing an `lc` property which corresponds to a land cover type and properties B2 through to B7 which correspond to spectral reflectance measurements.

```
FeatureCollection (129 elements, 0 columns)
  type: FeatureCollection
  columns: Object (0 properties)
  ▼ features: List (129 elements)
    ▼ 0: Feature 00000000000000000000_0
      type: Feature
      id: 00000000000000000000_0
      geometry: null
      ▼ properties: Object (7 properties)
        B2: 0.026000000536441803
        B3: 0.03929999843239784
        B4: 0.045899998396635056
        B5: 0.21130000054836273
        B6: 0.11919999867677689
        B7: 0.0674000009894371
        lc: 0
```

Figure 8.2: FeatureCollection storing training data to train a supervised classifier to predict land cover type.

You can now use `trainingData` to train a classifier to predict land cover type based on Landsat 8 spectral reflectance data values.

8.3.2 Train

You are going to use `trainingData` to train a classification and regression tree (CART) model to predict a pixel's land cover type based on its spectral reflectance values.

The first step is to create a CART object using the `ee.Classifier.smileCart()` constructor function. You then use the `train()` function to train the CART classifier. You pass your labelled training data `trainingData` into the `train()` function and it will learn the rules that best relate spectral reflectance values in your training data to the observed land cover type specified by the `lc` property.

```
// Train a CART classifier with default parameters.
var trainedCART = ee.Classifier.smileCart()
  .train(trainingData, 'lc', bands);
print(trainedCART);
```

The video below presents a short demonstration of how CART classifiers work. You can also read Chapter 8 of [James et al., \(2013\)](#) to understand the process of how CART classifier rules are trained via minimising an error function.

Overview of CART classifiers

8.3.3 Predict

Now you can use your trained CART classifier object to predict each Landsat 8 pixel's land cover. To classify an `Image` in Google Earth Engine using a trained classifier use the `classify()` function. You pass your trained classifier into the `classify()` function as an argument; this function will apply the algorithm defined in your trained classifier (`trainedCART` here) to each pixel's spectral reflectance values to predict land cover type.

Display your predicted land cover `Image` on the map. Visually inspect your map of predicted land cover. How good a job has your trained classifier `trainedCART` done at predicting the land cover type of the Landsat 8 pixels?

```
// Classify (predict land cover type) 18SR using trained CART classifier
var lcImage = 18SR
  .select(bands)
  .classify(trainedCART);
print(lcImage);

//Make a palette for mapping
var lulcColor = [
  '00FF00', // vegetation
  '000000', // built up
  '0000FF', // water
];
Map.addLayer(lcImage, {palette: lulcColor, min: 0, max: 2}, 'supervised classification');
```

Use the *Layers* tab to toggle on and off the Landsat 8 RGB composite, the predicted land cover map, and the Google satellite basemaps. Zoom in on a few locations and explore the difference between the land covers captured by the Google satellite basemaps and the Landsat 8 data.

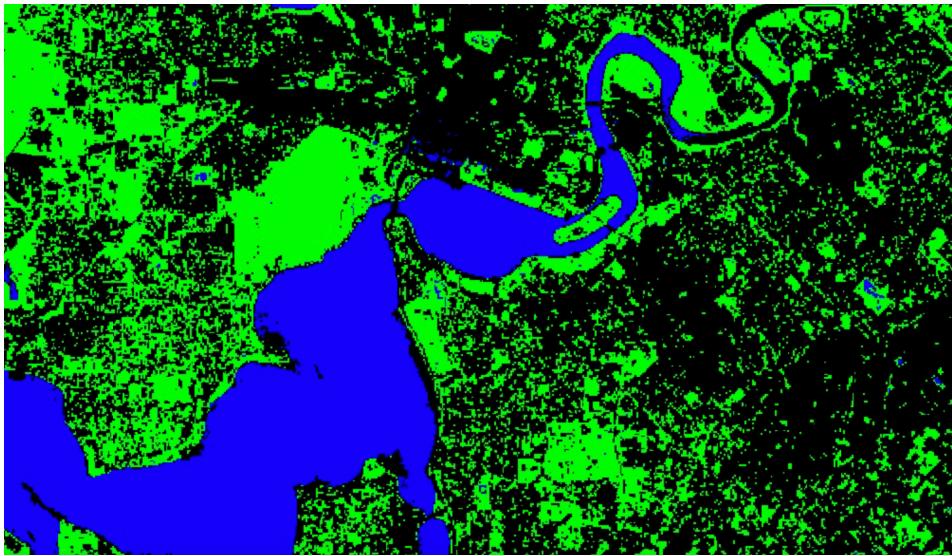


Figure 8.3: Predicted land cover type using the trained CART classifier

You should see that in many cases a single Landsat 8 pixel is comprised of multiple land cover types (e.g. a pixel could include trees, a grass lawn, and roofs). This is a mixed pixel problem; it is one example of how spatial data often simplifies reality. Here, a mix of land cover types with different spectral signatures will be represented by a single spectral reflectance value per-band and per-pixel or by a single land cover type label.

Visualise mixed pixels

How could you reduce the mixed pixel problem that is visible when predicting land cover over urban Perth using Landsat 8 data?

You could use finer spatial resolution data that would be able to resolve the different land cover objects present within a single Landsat pixel.

You could use a fuzzy classification system where a pixel can be a member of more than one land cover type.

The classifier you have just trained predicts whether a pixel is one of three land cover types (vegetation, built up, or water). In reality, there are far more than three land cover types present within the area covered by the Landsat 8 Image. An example of this is shown in the below figure where beach between City Beach and Scarborough in Perth is classified as built up land cover. Therefore, through limiting the possible land cover types we are introducing error into predicted land cover maps (i.e. there is a discrepancy between predicted land cover and reality).

How could you train a classifier that predicts whether a pixel belongs to more land cover types than vegetation, built up, or water?



Figure 8.4: Mixed pixels - a pixel is assigned a single land cover type category yet it contains a mix of land cover types

Ensure that your labelled training data covers the range of land cover types within the area of interest and that you have a sufficient number of training data observations per-land cover type in order to characterise each land cover type accurately.

8.3.4 Accuracy Assessment

Before using your trained classifier to predict the land cover type of new pixels (i.e. pixels that you don't already know the land cover type) you need to know how accurate the classifier's predictions are. You do this by assessing the classifier's accuracy; typically this is done by comparing predictions to observations on unseen data - data that has not been used to train the model. This will give you an indication of how well your classifier will generalise and work on new data.

First, you need some test data that contains labels of observed land cover types. Then you can use your trained classifier to predict the land cover type at each of these locations. Finally, you compare the predicted versus the observed land cover types to assess the accuracy of your trained classifier.

Import the the `FeatureCollection test_points` into the variable `testPoints`. This contains labelled points with an observed land cover type property. The process of predicting the land cover type at each of these test points is similar to the feature engineering process described above. Use the `sampleRegions()` function to extract the spectral reflectance measures for

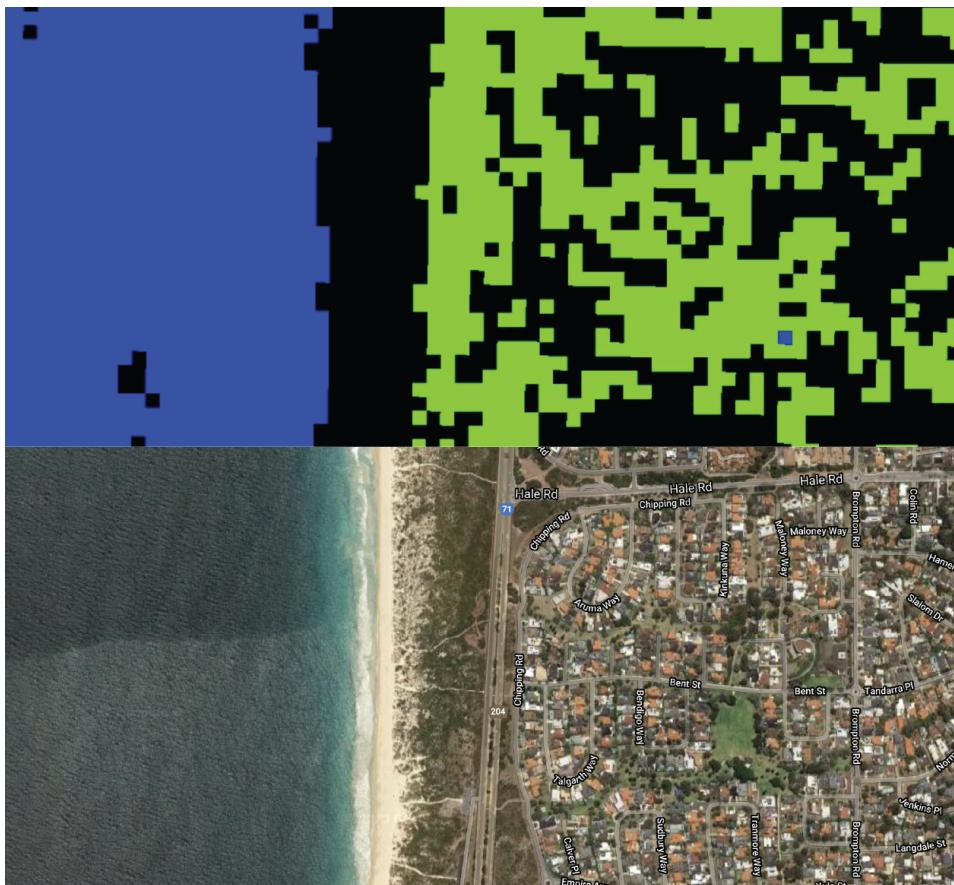


Figure 8.5: Beach (sandy land cover) classified as built up.

each of these points. Then use the trained classifier `trainedCART` to predict the land cover type at each test point based on its spectral reflectance values.

```
// Accuracy Assessment
var testPoints = ee.FeatureCollection('users/jmadi07/gee-labs/test_points');
Map.addLayer(testPoints, {color:'yellow'}, 'test points');

// Sample spectral reflectance at test points
var testData = 18SR.sampleRegions({
  collection: testPoints,
  properties: ['lc'],
  scale: 30
});
print(testData);

// Classify the test data.
var testClassified = testData.classify(trainedCART);
print(testClassified);
```

Using the predicted versus observed land cover types for the test data points you can generate an error matrix. This is an $n \times n$ matrix where n is the number of land cover types. In Google Earth Engine error matrices, the columns (1-axis) represent the predicted data and the rows (0-axis) represent the observed test data. The diagonal of the matrix represents pixels that were correctly classified (i.e. the classifier correctly predicted the observed land cover type in the test data). This code flips (transposes) the matrix for display purposes here so observed data are in the columns and predicted data are in the rows.

You can use the `errorMatrix()` function in Google Earth Engine to generate an error matrix from a `FeatureCollection`. The first argument to the `errorMatrix()` function is the observed land cover type (`lc` here) and the second argument is the predicted land cover type (`classification` here).

```
// Get an error matrix
var errorMat = testClassified.errorMatrix('lc', 'classification');
print('error matrix: ', errorMat.array().matrixTranspose(0, 1));
```

The sum of the diagonal elements is the number of correctly classified pixels. The overall accuracy of the classifier can be computed as:

$$accuracy = \frac{CorrectlyClassifiedPoints}{TotalNumberofPoints}$$

This indicator of classification accuracy can be interpreted as $x\%$ of an image is correctly

```
error matrix:  
▼ [[26,8,0],[0,29,12],[0,0,39]]  
  ▷ 0: [26,8,0]  
  ▷ 1: [0,29,12]  
  ▷ 2: [0,0,39]
```

```
overall accuracy:  
0.8245614035087719
```

```
kappa:  
0.7350685568208227
```

```
producers accuracy:  
▼ [[1],[0.7837837837837838],[0.76470588...]  
  ▷ 0: [1]  
  ▷ 1: [0.7837837837837838]  
  ▷ 2: [0.7647058823529411]
```

```
users accuracy:  
▼ [[0.7647058823529411,0.70731707317073...]  
  ▷ 0: [0.7647058823529411,0.70731707317073...]  
    0: 0.7647058823529411  
    1: 0.7073170731707317  
    2: 1
```

Figure 8.6: Error matrix and accuracy assessment indicators.

classified. It does not give any indication of the spatial variability in classification accuracy or if certain land cover types are classified more accurately than others.

There are helpful functions in Google Earth Engine that you can apply to error matrices to compute accuracy assessment statistics. You can apply the `accuracy()` function to your error matrix to compute the overall accuracy and you can apply the `kappa()` function to compute the kappa statistic.

```
print('overall accuracy: ', errorMat.accuracy());
print('kappa: ', errorMat.kappa());
print('producers accuracy: ', errorMat.producersAccuracy());
print('users accuracy:', errorMat.consumersAccuracy());
```

Using the error matrix you can compute producer's and user's accuracy that give an indication of the accuracy of classification for individual land cover types.

The producer's accuracy is computed by dividing the entry in cell (i, i) of the error matrix by the sum of column i . The producer's accuracy is a measure of omission error and informs on the number of test pixels in a given land cover type that were correctly classified. Here, all of the vegetation (class 0) pixels in the test dataset were correctly classified as vegetation by `trainedCART`. Eight of the test data points labelled as built up were classified as vegetation; therefore, built up land cover has a producer's accuracy of 78%.

The user's accuracy is computed by dividing the entry in cell (i, i) of the error matrix by the sum of row i . The user's accuracy is a measure of commission error (i.e. erroneously labelling a pixel a given land cover type). Of the 34 test data points that were predicted as being of vegetation land cover type, eight were observed as being built up. Thus, vegetation land cover has a user's accuracy of 76%.

8.4 Unsupervised Classification

Training unsupervised classification algorithms involves learning rules to classify a dataset into groups (classes) without requiring a label to be known or assigned to each group. The goal when training an unsupervised classification algorithm is to find splits in the data that ensure observations within each group are similar.

A common unsupervised classification algorithm is K -means clustering and the only input you need to provide to the classifier is the number of classes that you wish to cluster your dataset into.

We will demonstrate how to use unsupervised classification to cluster the Landsat 8 Image assigning all pixels to groups based on similar spectral reflectance values.

To train the the K -means clustering algorithm, we will generate a sample of pixel spectral reflectance values from the Landsat 8 Image. Here, we will generate a sample of 1000 points within the Landsat 8 scene.

```
// Unsupervised classification

// Make the training dataset.
var trainingKMeans = 18SR.sample({
  region: bBox,
  scale: 30,
  numPixels: 1000,
  seed: 123
});
```

Next, we will train a K -means clusterer with five classes and then use trained algorithm to classify the Landsat 8 Image pixels into one of the five classes.

```
// Instantiate the clusterer and train it.
var clusterer = ee.Clusterer.wekaKMeans(5).train(trainingKMeans);

// Cluster the input using the trained clusterer.
var clusteredL8SR = 18SR.cluster(clusterer);
```

Finally, we can display the result on the map by assigning a colour to each class.

```
// Display the clusters with random colors.
Map.addLayer(clusteredL8SR.randomVisualizer(), {}, 'clusters');
```

Compare the classified maps generated using supervised and unsupervised classifiers. Has the unsupervised classifier split pixels into classes that correspond to different land cover types?

What is an advantage offered by unsupervised classification algorithms?

They are a tool that we can use to extract and visualise patterns within large and complex datasets. They can also be used for data reduction exercises to reduce the size of the dataset.

What are some limits of using unsupervised classification algorithms for land cover mapping?

The clusters learnt by the algorithm might not relate to actual land cover classes. In some cases it can be unclear how many clusters to split the dataset up into.

9 Introduction

In Lab 9 you implemented a workflow to train and test a machine learning model to perform a classification task. A classification problem is when the outcome variable is qualitative (i.e. it can take on a value of one of K classes or categories). If the outcome variable is numerical and continuous this is termed a regression problem. This lab will introduce a workflow to train, test, and interpret a machine learning model for regression tasks.

You will use a range of predictor variables to train a model that predicts summer land surface temperature (LST) across a study area in Perth.

9.0.1 Predicting Land Surface Temperatures

Exposure to warmer temperatures can have adverse impacts on various indicators of well-being (see below figure from Carleton and Hsiang (2016)).

Populations in urban areas are exposed to warmer temperatures compared to populations; the urban heat island (UHI) effect. Land cover affects the energy balance, and, thus, temperature in urban areas; for example, trees can provide a cooling affect via evapotranspiration or shade. The ability to predict how different configurations of urban land cover affects temperature is important for planners seeking to develop urban areas in ways that minimise heat exposure. The Low Carbon Living CRC has produced a useful Guide to Urban Cooling Strategies demonstrating how different urban land cover types affect temperature.

In this lab you will use a range of spatial layers that represent the built environment to train and test a machine learning model that predicts urban LST. This model could be used for scenario analyses where the effect of different urban developments, which result in different land covers, on LST can be assessed.

9.0.2 Random Forests

As in lab 9, your goal will be to train a model f which relates predictor variables X to an outcome Y .

$$Y = f(X)$$

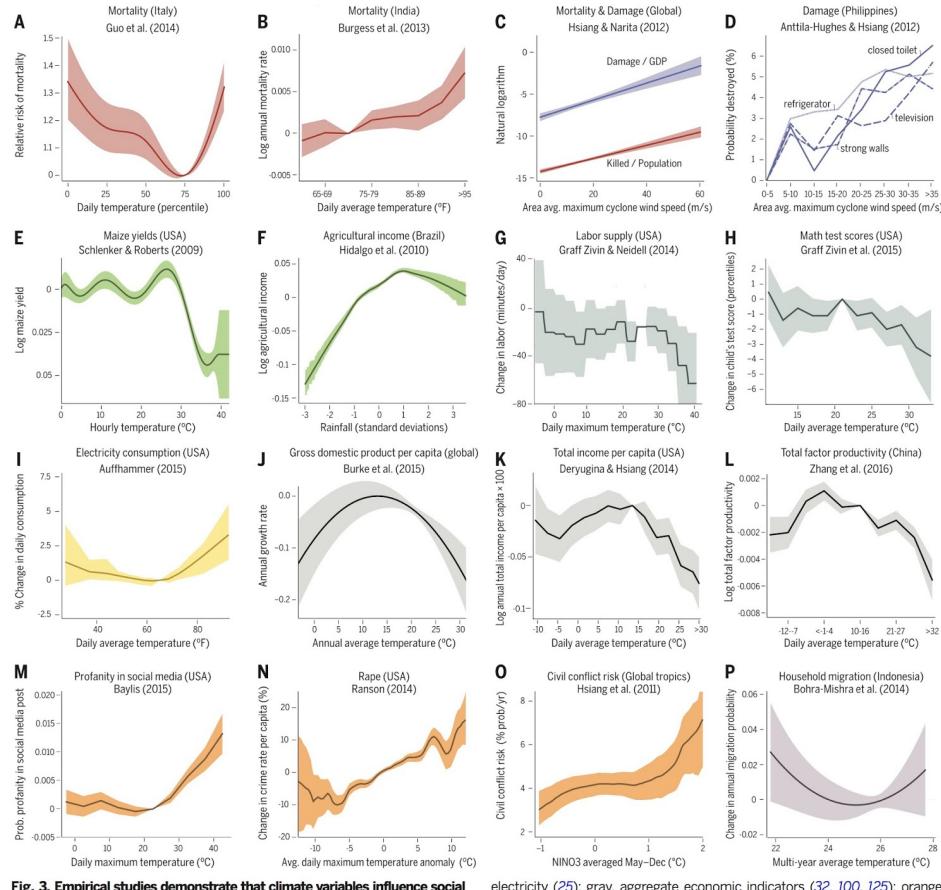


Fig. 3. Empirical studies demonstrate that climate variables influence social and economic outcomes in many sectors and contexts. (A to P) Examples of dose-response functions estimating the causal effect of climatological events on various social outcomes. Reproduced from authors' original estimation; titles list the outcome variable and location studied. Colors indicate categories of outcome variables: red, mortality (44, 46); blue, cyclone damage to assets (48, 116); green, agriculture (21, 153); teal, labor productivity (96, 97); yellow, electricity (25); gray, aggregate economic indicators (32, 100, 125); orange, aggression, violence, and conflict (27, 130, 134, 136); purple, migration (171). Climate variables differ by study but include temperature, cyclone wind speed, rainfall anomalies, and ENSO measures. Response functions only identify relative changes and are either normalized to "zero effect" at a designated climatic event, such as a minimum valued outcome, or the sample mean of an outcome. Shaded areas are confidence intervals, as computed by original authors.

electricity (25); gray, aggregate economic indicators (32, 100, 125); orange, aggression, violence, and conflict (27, 130, 134, 136); purple, migration (171). Climate variables differ by study but include temperature, cyclone wind speed, rainfall anomalies, and ENSO measures. Response functions only identify relative changes and are either normalized to "zero effect" at a designated climatic event, such as a minimum valued outcome, or the sample mean of an outcome. Shaded areas are confidence intervals, as computed by original authors.

Figure 9.1: Climate impacts on indicators of well-being (source: Carleton and Hsiang (2016)).

However, here Y is LST which is a continuous variable. Therefore, the model \hat{f} that you train will be in regression mode. This will still be a supervised learning task where you have observations of LST, Y , and predictor variables, X , which you use to train a model \hat{f} that learns rules that relate X to predicted LST values, \hat{Y} .

The model you will be using is a Random Forests model. A Random Forests model comprises an ensemble of decision (regression) tree models. In regression mode, the final prediction is the average of the predictions from all the regression trees in the ensemble.

9.0.2.1 Regression Trees

First, let's go through the process of training a single regression tree. A single regression tree divides the predictor space - the range of values the p predictor variables can take - into J distinct and non-overlapping regions R_1, R_2, \dots, R_J . If an observation ends up in region R_j it takes on the response value assigned to that region. This is the mean of all response values for that region in the training data. When training a regression tree, your task is to determine the rules that best divide the predictor space into J regions by minimising an error or loss function.

First, you find the predictor variable X_j and cutpoint value s that splits the training data such that the residual sum of squares (RSS) of response variable Y in the two regions generated by the split is minimised. For any j and s in your training data you can define two regions:

$$R_1(j, s) = \{X | X_j < s\} \text{ and } R_2(j, s) = \{X | X_j \geq s\}$$

and you seek X_j and s that minimises the sum of the RSS for the two regions R_1 and R_2 :

$$\sum_{i:x_i \in R_1(j, s)} (y_i - \hat{y}_{R1})^2 + \sum_{i:x_i \in R_2(j, s)} (y_i - \hat{y}_{R2})^2$$

\hat{y}_{R1} is the mean of the response variable Y in the region R_1 and \hat{y}_{R2} is the mean of the response variable Y in the region R_2 .

You recursively repeat this process for the data in each of the subsequent splits until a stopping criterion is reached (e.g. minimum number of training observations in a final leaf node or the maximum number of leaf nodes reached). The set of splits along predictor variables X and cutpoints s are the learned decision rules that determine how your model predicts \hat{y}_i for a set of known predictor variables x_i .

The figure below illustrates the process of creating a regression tree by recursively splitting the training data into regions that minimise the RSS between the observed response value and average of the response values in a given region $R_{(j,s)}$.

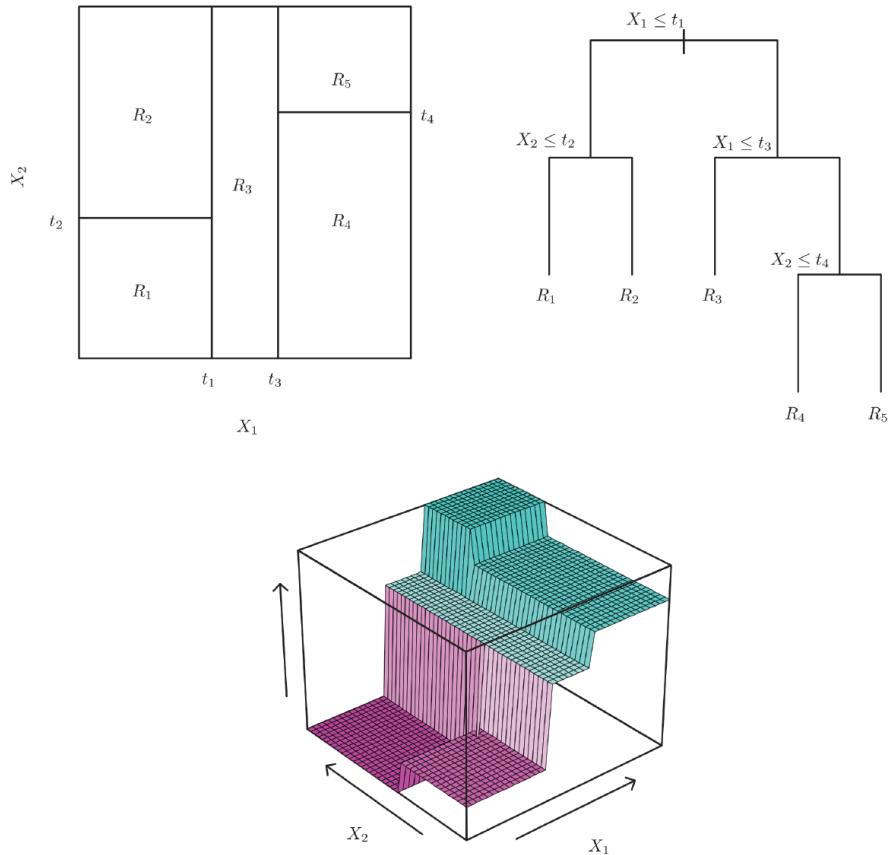


Figure 9.2: Top left is the outcome of recursively splitting training data that consists of two predictors variables X_1 and X_2 , top right is the same splits in the data represented as a tree, and the bottom panel shows a prediction surface with predicted values on the Y-axis (source: James et al. (2013)).

9.0.2.2 Random Forests

A Random Forests model is an ensemble of regression trees generated by taking **bootstrap samples** (sampling with replacement) from the training data and fitting a regression tree to each sample. One issue with a single regression tree is that it has high variance; if you train a model \hat{f} using different training datasets and you get different \hat{f} 's then that method has high variance. If you train a regression tree with sample 1, X_1 , and then train a regression tree with sample 2, X_2 , where both samples are drawn from the same population, small differences in the samples could result in different rules that characterise your trained regression trees \hat{f}_1 and \hat{f}_2 . This will result in different predictions for the same input data.

Bootstrap aggregation (bagging) is a method to reduce the variance in predictions. You draw B training samples from your training data and train a regression tree \hat{f}_b for each $1, 2, \dots, B$ training samples. For each x_i your prediction is the average prediction returned from all $\hat{f}_b(x)$.

$$\hat{y}_{bag}(x) = \frac{1}{B} \sum_{b=1}^B \hat{f}_b(x)$$

This video provides an overview of bootstrap aggregation (bagging).

When you generate bagged training samples some of your initial training dataset are not used to train a given regression tree \hat{f}_b . You can use these held-out data to compute the prediction error for each regression tree by comparing predictions \hat{y}_i for these data points to observed values y_i . This is termed the **out-of-bag** error. You can combine the out-of-bag error computed for each i observation in your training data to compute an overall out-of-bag mean square error (MSE) estimate for your bagged model.

Random Forests build on bagged tree models by decorrelating the decision trees within the ensemble. If there was one strong predictor variable in your training dataset, each time you train a regression tree in using a bagged training sample you would expect this variable to be dominant in determining the first split in your tree. This would result in your regression trees having a similar structure and possibly missing relationships between other predictor variables and the response. Therefore, when training a Random Forests model at each split only m out of p predictor variables are considered. This decorrelates the regression trees in the Random Forests ensemble reducing the variance in predictions and capturing more information about the relationships between the predictors and the response variable.

Chapter 8 and Chapter 2 (section 2.2.2) of James et al. (2013) provide a clear and detailed explanation of Random Forests models.

This video also provides an overview of the process of training a Random Forests model.

Random Forests models are used for a variety of prediction tasks involving spatial data. Some of the advantages of random forests models are:

- Tree based models can easily handle qualitative and quantitative predictor variables.
- Random Forests models are relatively robust to being trained with large numbers of predictor variables
- Random Forests models handle (multiple) interactions between predictor variables (when the relationship between predictors and response is complex)
- Random Forests models can handle non-linear relationships between predictor variables and response variables
- Random Forests models generally have good predictive performance
- Random Forests models are relatively quick to train

Disadvantages of Random Forests models include:

- Black box - it is difficult to see inside the model and visualise the relationships between predictors and response (how do you visualise > 100 regression trees?)
- Random Forests are not suited to inference problems - how does Y change as a function of X . The model f cannot be a black box in inference tasks as you are interested in f 's form

This free and open book (Interpetable Machine Learning) includes some accessible discussion of limits to non-parametric machine learning models, such as Random Forests, and approaches to interpret them.

9.1 Setup

```
/*
Lab 10
Author: Test
Date: XX-XX-XXXX

*/
```

9.2 Data Import

Import an **Image** `lab8Data` which contains a band `1st`; this is the response or outcome variable. `1st` is the average Landsat 8 LST for Perth's summer months (December, January, and February) for the years 2014 to 2019 and was computed using the method in Jiménez-Muñoz et al. (2014).

This **Image** also contains a range of land cover related predictor variables as bands; you will train a model to predict LST using these bands. These predictor variables include:

mb_cat16 - the [ABS Mesh Block](#) land use category for the Mesh Block with the largest intersection with a Landsat 8 pixel. The values correspond to land use categories:

1. Parkland
2. Transport
3. Residential
4. Water
5. Education
6. Commercial
7. Hospital / Medical
8. Industrial
9. Other

mb_road_length - the length in metres of all OpenStreetMap roads that are returned from the intersection of Open Street Map roads and the Mesh Block.

building_count - count of buildings whose footprint intersects with a Landsat 8 pixel footprint. Building data is derived from the [Geoscape Buildings](#) product.

building_area - sum of building footprint area (in square metres) whose footprint intersects with a Landsat 8 pixel footprint. Building data is derived from the [Geoscape Buildings](#) product.

median_eave_height - for all buildings that intersect with a Landsat 8 pixel, the mean of the relative height of the eave elevation to the ground elevation and is calculated by taking the median eave elevation minus the minimum ground elevation. Building data is derived from the [Geoscape Buildings](#) product.

max_building_height - for all buildings that intersect with a Landsat 8 pixel, the mean relative height of the roof elevation to the ground elevation and is calculated by taking the maximum roof elevation minus the minimum ground elevation. Building data is derived from the [Geoscape Buildings](#) product.

swim_pool_count - count of swimming pools at buildings that intersect with a Landsat 8 pixel footprint. Swimming pool data is derived from the [Geoscape Buildings](#) product.

solar_panel_count - count of buildings that intersect with a Landsat 8 pixel footprint and have solar panels. Solar panel data is derived from the [Geoscape Buildings](#) product.

tree_pct - percentage of a Landsat 8 pixel covered by tree canopy. Defined as vegetation greater than 3 m in height and derived from the Urban Monitor product ([Caccetta, 2012](#)).

shrub_pct - percentage of a Landsat 8 pixel covered by shrub. Defined as vegetation greater than 0.5 m and less than 3 m in height and derived from the Urban Monitor product ([Caccetta, 2012](#)).

grass_pct - percentage of a Landsat 8 pixel covered by grass. Defined as vegetation less than 0.5 m in height and derived from the Urban Monitor product ([Caccetta, 2012](#)).

Import `lab8Data` and visualise its `Image` bands on the map display. Visually inspect the relationship between different land cover types and LST. Do they conform to your expectations? For example, use the **Layers** to turn off all layers except LST and `building_area`; can you see the relationship between large buildings and surface temperature?

Visualisation of warmer LST over Sir Charles Gairdner Hospital.

```
// Data import

// mb_cat16 - ABS Mesh Block land use
// mb_road_length - length of OSM roads (m) in an ABS Mesh Block
// lst - Landsat 8 LST (K) Dec-Feb average for years 2014 - 2019
// building_count - count of building footprints intersecting Landsat 8 pixel
// building_area - area of building footprints intersecting Landsat 8 pixel
// median_eave_height - median eave height of buildings intersecting Landsat 8 pixel
// max_building_height - max building height of buildings intersecting Landsat 8 pixel
// swim_pool_count - count of swimming pools at properties intersecting Landsat 8 pixel
// solar_panel_count - count of solar panels at properties intersecting Landsat 8 pixel
// tree_pct - % tree cover of Landsat 8 pixel
// shrub_pct - % shrub cover of Landsat 8 pixel
// grass_pct - % grass cover of Landsat 8 pixel
var lab8Data = ee.Image('users/jmadiv07/gee-labs/lab-8-data');
lab8Data = lab8Data.rename(['mb_cat16', 'mb_road_length', 'lst', 'building_count', 'building_area']);
print(lab8Data);

// create study area from lab8Data
var bBox = lab8Data.geometry();
Map.centerObject(bBox, 13);
Map.addLayer(bBox);

// Visualise LST
Map.addLayer(lab8Data.select('lst'), {min:295, max:315, palette:['0000FF', '00FFFF', 'FFFFFF']});

// Visualise Building Count
Map.addLayer(lab8Data.select('building_count'), {min:0, max:8, palette:[ '#ffffd4', '#fee391']});

// Visualise Building Area
Map.addLayer(lab8Data.select('building_area'), {min:0, max:10000, palette:[ '#ffffd4', '#fee391']});

// Visualise Tree Cover
Map.addLayer(lab8Data.select('tree_pct'), {min:0, max:100, palette:[ '#edf8e9', '#c7e9c0', '#a1d99b']});
```

9.3 Training (Feature) Engineering

The first task in a supervised classification workflow is to prepare the training data. You have already imported and visualised the outcome variable (1st) and a range of predictor variables.

There are some additional variables in Google Earth Engine that you can add to your training data. There is often a relationship between elevation or distance from water bodies and temperature (e.g. cooling winds coming off the ocean).

There are a range of elevation datasets pre-loaded into Google Earth Engine; here, you will import the [DEM-H: Australian SRTM Hydrologically Enforced Digital Elevation Model](#) where each pixel value represents elevation in metres.

```
// Training engineering

// import elevation data
var dem = ee.Image('AU/GA/DEM_1SEC/v10/DEM-H').clip(bBox);
Map.addLayer(dem, {min: 0, max: 50, palette:[ '#8c510a', '#d8b365', '#f6e8c3', '#c7eae5', '#5ab2a9' ]});
```

To compute the distance to water you will use the [JRC Global Surface Water Mapping Layers v1.2](#) data. The `seasonality` band pixel values represent the number of months in a year a location was covered by surface water; using the relational operator `eq()` identify all pixels that are permanently water (i.e. `seasonality.eq(12)` where 12 represents the number months in a year that a pixel is water).

To compute the distance from each location in the study area `bBox` to a water body use the `fastDistanceTransform()` function. The `fastDistanceTransform()` function returns the squared number of pixels between a location and the nearest non-zero pixel in the input `Image` (`jrcWater` here where pixel values of one represent permanent water). To convert the return value from `fastDistanceTransform()` to metres use the square root function `sqrt()` and the multiply the pixel number by the pixel dimension `multiply(ee.Image.pixelArea().sqrt())`.

```
// water distance
var jrcWater = ee.Image('JRC/GSW1_2/GlobalSurfaceWater')
  .select('seasonality')
  .eq(12)
  .clip(bBox);

var waterDist = jrcWater.fastDistanceTransform(1000)
  .sqrt()
  .clip(bBox);
```

```

.multiply(ee.Image.pixelArea().sqrt())
.rename(['water_distance']);
Map.addLayer(waterDist, {min: 0, max: 3000, palette:[ '#253494', '#2c7fb8', '#41b6c4', '#7f

```

You need to add `dem` and `waterDist` as bands to the training data. You can use the `addBands()` function for this.

```

// add water distance and dem to training data
lab8Data = lab8Data.addBands(dem).addBands(waterDist);

```

Having prepared and inspected the outcome and predictor data you need to create samples to train and test the model. First, mask out water bodies from the `dem` and `waterDist` bands.

You will generate test and training samples by performing a stratified random sample where each of the Mesh Block land use categories are strata. You can use the `stratifiedSample()` function in Google Earth Engine to randomly sample a pre-specified number of points from strata (groups) in an `Image`. You need to convert the band `mb_cat16` to integer data type prior to using it as a strata argument in the `stratifiedSample()` function. The `seed` argument to the `stratifiedSample()` function ensures that you draw the same random sample each time you execute your code; this is important to ensure your results are reproducible.

Why did you take a stratified random sample as opposed to a random sample?

By sampling from each of the Mesh Block land use categories you ensure that your training data is drawn from the range of land cover types / contexts present in the study area. This ensures that land cover types that might cover a small area are included in model training; these locations might be missed in a normal random sampling procedure.

```

//mask water from elevation and water distance layers
var mask = lab8Data.select('lst').gt(0);
lab8Data = lab8Data.updateMask(mask);

var mb_cat16_int = lab8Data.select('mb_cat16').toInt().rename('mb_cat16_int');
lab8Data = lab8Data.addBands(mb_cat16_int);

var training = lab8Data.stratifiedSample({
  numPoints: 300,
  classBand: 'mb_cat16_int',
  region: bBox,
  seed: 0,
  scale: 30

```

```

    });

print(training);

var test = lab8Data.stratifiedSample({
  numPoints: 100,
  classBand: 'mb_cat16_int',
  region: bBox,
  seed: 1,
  scale: 30

});
print(test);

```

What could be one problem with how the `training` and `test` samples were generated?

A separate stratified random sample was generated for the test data. There is a chance that locations included in the training sample could also be in the test sample. Also, by stratifying the sample using Mesh Block land use categories it is likely that test and training samples are drawn from locations close to each other. This might result in training and test sample data being spatially correlated and not fully independent observations. There is an example here of how you can use a spatial filter to remove test data points that are geographically close to training data.

9.4 Model Training

Using the training data that you have just prepared, train a Random Forests model that will predict LST. First, create a list of bands naming the outcome variable (`lst`) and the predictor variables in the training data `training`.

```

// Model Training

// bands to use in training the model
var bands = ['mb_cat16', 'mb_road_length', 'building_count', 'building_area', 'median_eave

```

You need to create a Random Forests model object of the `ee.Classifier.smileRandomForest` class and set the `numberOfTrees` and `seed` arguments. The `numberOfTrees` argument specifies the number of regression trees to include in ensemble. The `seed` is used to initialise a pseudo-random number generator; setting the seed ensures that the same bootstrap samples are drawn within the Random Forests algorithm each time you execute your code. This ensures your results are reproducible.

Google Earth Engine implements the [smile Random Forests](#) model.

You need to set the output mode of the Random Forests model as REGRESSION. LST is a continuous numeric variable so you are using the Random Forests for a regression task here.

Finally, once you have set up the parameters for your Random Forests model (i.e. `numberOfTrees`) and set it to regression mode you are ready to train the model using your training data.

```
// Train a random forests classifier
var randomForest = ee.Classifier.smileRandomForest({
  numberOfTrees: 100,
  seed: 0})
.setOutputMode('REGRESSION')
.train({
  features: training,
  classProperty: 'lst',
  inputProperties: bands
});
```

9.5 Model Interpretation

Compared to a single classification or regression tree or the results from estimating a linear regression model interpreting Random Forests models is more challenging. As the Random Forests model is an ensemble of many regression trees it is difficult to visualise the relationships between predictors and response (think about the difference between looking at a single regression tree versus 100 regression trees). There is often a trade-off between the prediction accuracy provided by a Random Forests model and ease of interpreting the model.

You can call the `explain()` function on your trained Random Forests model `randomForest` to obtain some model diagnostics. The dictionary object returned by `explain()` includes an `outOfBagErrorEstimate` property. This reports on the root mean square error (RMSE) between observed and predicted LST for training data that was not included in a given regression tree due to the bootstrap resampling process (remember from the introduction that each regression tree in a Random Forests model is trained using a different sample of the training data - for each tree, some training data observations were not used in the training process).

Print the dictionary returned by `explain()` to the *Console*.

```
// Model interpretation
var explainRF = randomForest.explain();
print('Explain:', explainRF);
```

Your `explainRF` dictionary object should be similar to the below figure. This Random Forests model has an out-of-bag error estimate of 1.09 Kelvin (RMSE between predicted and observed LST for all data points not included in a given regression tree within the Random Forests ensemble).

```
Explain:  
  ▾ Object (3 properties)  
    ▶ importance: Object (13 properties)  
      numberOfWorkers: 100  
      outOfBagErrorEstimate: 1.0926147146967327
```

Figure 9.3: Printing the dictionary returned by `explain()` to the console.

You will also see a property in `explainRF` called `importance`. This is a measure of variable importance in terms of each variable's contribution to prediction accuracy. The variable importance measure represents the amount the error function (RSS) was reduced due to splits in a regression tree over a predictor variable and summed over all B trees in the ensemble. A larger variable importance measure indicates that a predictor variable is more important for prediction accuracy.

The [smile Random Forests model](#) documentation defines the variable importance measure as “*Every time a split of a node is made on variable the impurity criterion for the two descendant nodes is less than the parent node. Adding up the decreases for each individual variable over all trees in the forest gives a fast measure of variable importance that is often very consistent with the permutation importance measure.*”

You can extract the `importance` property from `explainRF` to a `Feature` object and then plot each variable's importance score as a column chart. Inspect your chart in the *Console*.

```
var variableImportance = ee.Feature(null, ee.Dictionary(explainRF).get('importance'));  
  
var chartVarImp =  
ui.Chart.feature.setProperty(variableImportance)  
  .setChartType('ColumnChart')  
  .setOptions({  
    title: 'Random Forest Variable Importance',  
    legend: {position: 'none'},  
    hAxis: {title: 'Bands'},  
    vAxis: {title: 'Importance'}  
});  
  
print(chartVarImp);
```

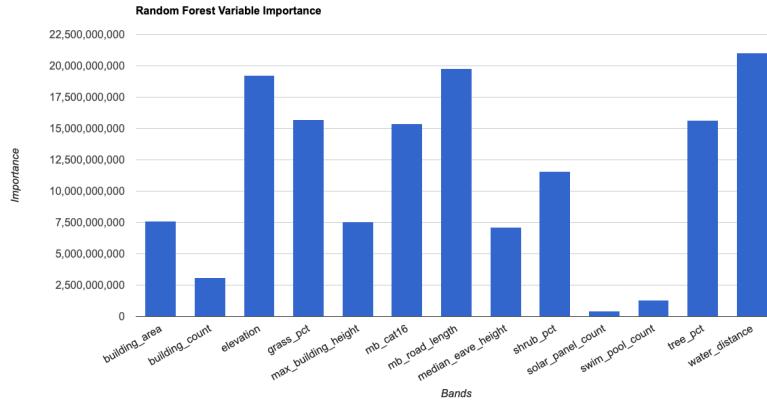


Figure 9.4: Variable importance chart for predictors of LST.

Can you explain why some predictors have higher variable importance scores?

Water distance, tree cover, road length, and elevation had high variable importance scores. Locations close to water (coastal or riverside locations) might be cooled by winds blowing off water bodies. Tree cover can cool via evapotranspiration or shade. Locations with higher altitude are cooler; however, the variation in elevation in this study area might not explain much variation in temperature. It is likely that the elevation variable is picking up that Kings Park is higher, cooler, and more vegetated. Roads are open surfaces (i.e. exposed to incoming solar radiation) that can absorb heat. It is plausible that locations with greater road coverage are warmer - can you check your display of `1st` on the map to see if roads are warmer?

Can you identify a limitation to interpreting models using variable importance plots?

Variable importance plots are useful for informing on variables that are important for prediction accuracy. However, they do not provide any information on the relationship between predictors and outcome variables. Models such as linear regression models are better suited to inference problems such as these. However, there are tools you can use to explore relationships between predictors and outcome variables in Random Forests models such as partial dependence plots.

A final model interpretation activity is predicting LST for all locations in the `lab8Data` Image, the data set you used to generate your `training` sample. You can visualise the prediction of LST on the map and compare it to observed LST.

```
// Predict LST
var predLST = lab8Data.classify(randomForest);
Map.addLayer(predLST, {min: 295, max: 310, palette:['0000FF', '00FFFF', 'FFFF00', 'FF0000']}
```

Your predicted LST should look similar to the figure below.

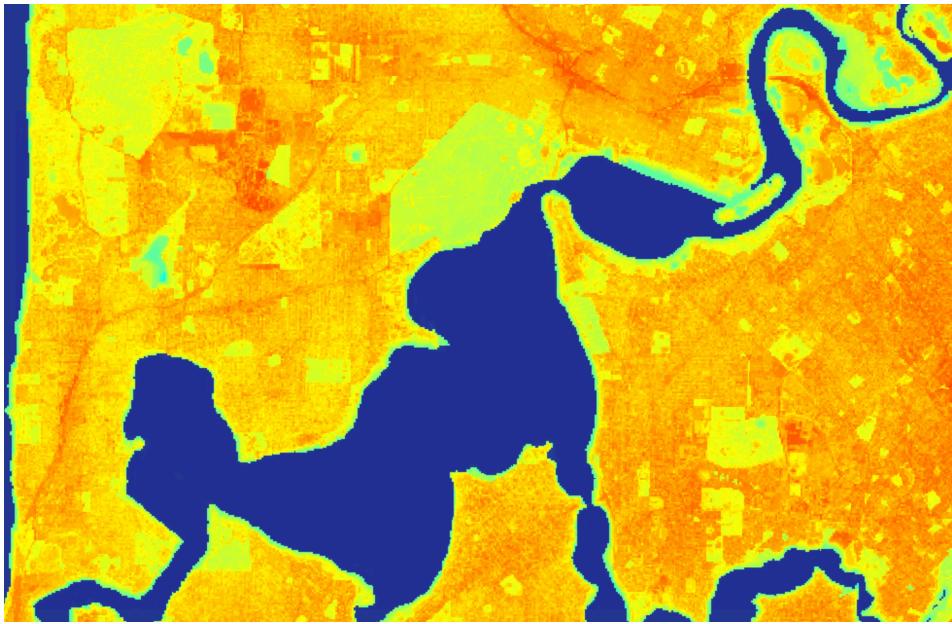


Figure 9.5: Predicted LST.

9.6 Model Testing

Your goal in this lab was to train a Random Forests model that could predict LST based on a suite of land cover indicators. As was stated at the beginning of the lab, such a model could be useful for urban planning providing a tool to predict LST under different land cover / development scenarios and identify where and how temperature exposure could be reduced. However, before a model can be used for such purposes it needs to be tested to assess how well it generalises to data that was not used in model training.

There are a range of [error statistics](#) (e.g. RMSE, mean square error (MSE)) you can compute to assess how well your model generalises to unseen data; these statistics measure the discrepancy between an observed and predicted outcome for data points the model has not seen before. The following code demonstrates how to compute the RMSE for your predictions of LST for the `test` dataset.

First, you need to use your trained model `randomForest` to predict LST values for your `test` data points. Inspect the data in the *Console* to confirm that each `Feature` in `test` has a predicted LST value stored in the new `classification` property. How well does the predicted LST match the observed LST?

```
// Model Validation
var predTestLST = test.classify(randomForest);
```

```
print(predTestLST);
```

Next, you can use your predicted and observed LST values to compute the RMSE for your model's prediction of LST on unseen test data. The RMSE is computed as:

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

For each data point i in your `test` dataset, you compute the squared difference between observed LST y_i and predicted LST \hat{y}_i . Then you compute the mean of the squared differences between y_i and \hat{y}_i in your `test` data. Finally, you take the square root of the mean of the squared differences so you can interpret your error metric in units of the response variable (K or Kelvin here).

The following code block will compute the the RMSE using you `test` data. First, you declare a function that computes the squared difference between observed and predicted LST for each `test Feature`. Then you call the `reduceColumns()` function which reduces (aggregates) all the values in a specified property of `Features` in a `FeatureCollection`; to compute RMSE use the mean function applied to the column `diff_squared`. Finally, apply the square root function `sqrt()` to the output of the call to `reduceColumns()`.

You should see your RMSE value printed to the *Console*.

```
var squaredDiff = function(feature) {
  var lstObs = ee.Number(feature.get('lst'));
  var lstPred = ee.Number(feature.get('classification'));
  var diff = lstObs.subtract(lstPred);
  return feature.set('diff_squared', diff.pow(2));
};

// Calculate RMSE
var rmse = ee.Number(predTestLST.map(squaredDiff)
  .reduceColumns(ee.Reducer.mean(), ['diff_squared'])
  .get('mean'))
  .sqrt();
print('RMSE: ', rmse);
```

What was the RMSE for Random Forests model when evaluated against `test` data? How can you interpret this error statistic? Is your model accurate enough, when applied to unseen data, to be useful for urban development and land use planning scenario analysis?

Things to consider are: how large is your model's prediction error relative to variation in LST you could expect to observe between different land cover types? How does your model

compare to other similar non-parametric machine learning models in the literature (and are these models evaluated using independent test data)?, and how does your model compare to predictions of temperature that could be derived from other methods (e.g. energy balance models)?

Hu et al. (2020) present a tree based model that predicts LST based on a range of land cover related datasets. How did they validate their models performance?

See Thom et al. (2016) for an example of a study validating the SOLWEIG energy balance model for urban temperature predictions in Adelaide.