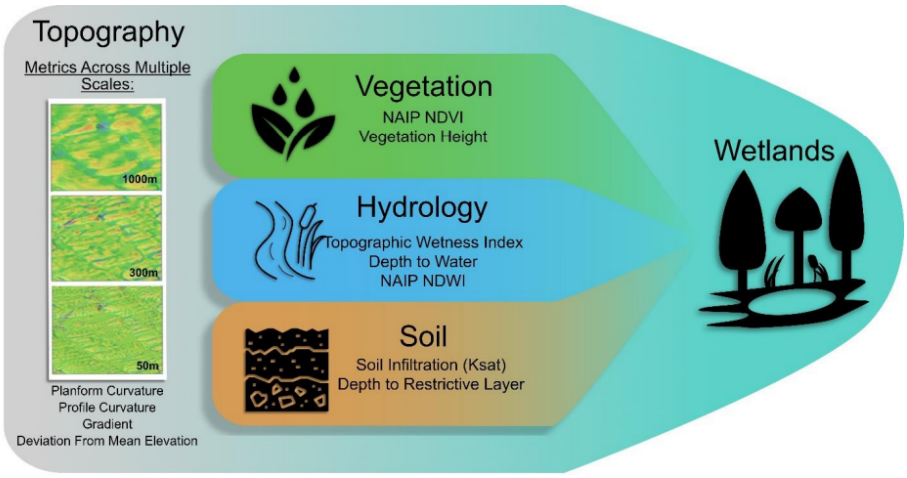
Guide to Running the Wetland Intrinsic Potential (WIP) Tool in R

Table of contents

Preface

Wetland inventories are essential for tracking loss of wetlands. The WIP tool was developed to identify wetlands that are missing from existing wetland inventories. Our wetland indicator framework, which includes spatial variables representing vegetation, hydrology, soils, and multi-scale topographic attributes can be used to quantify probability of wetland occurrence, as well as predict the type of wetland. Our wetland indicator framework provides a flexible approach that can be adapted to identify diverse wetland types across varied landscapes. The reasoning behind this framework is shown below



The following article provides a skeleton of how to run this tool in the R programming language. It must be noted that there are some things, including many of the wetland indicators, like vegetation and soils, that must be obtained externally for now. However, the development of topographical indices and everything to do with the building and running of the model, can be run in R.

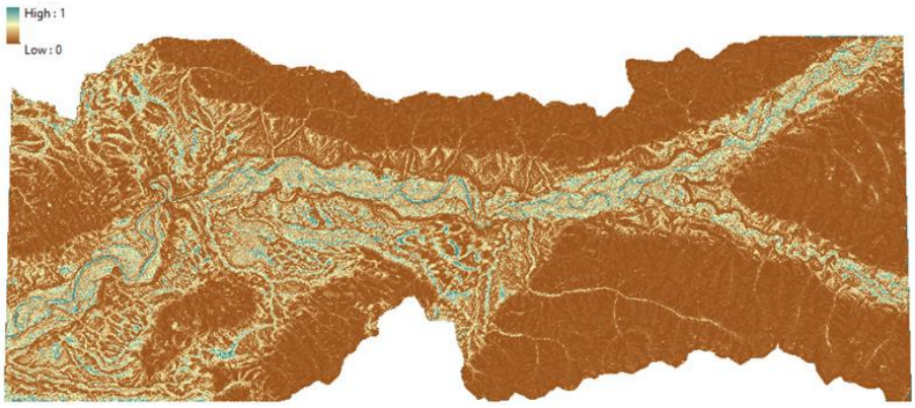


Figure 1: Example of WIP Tool in Hoh Rainforest

Setup

1. Install base R from CRAN
2. Install R Studio from Posit
3. Run RStudio (if needed, a quality introduction to R can be found here), then install packages that may or not be needed in the code chunk below. More information on each packages as follows:
    - The main package for spatial data and statistics: `terra`. The WIP tool will simply not run if `terra` is not installed
    - The `MultiscaleDTM` package is a necessity if one wants to calculate the surface metrics (such as gradient, curvature, etc.) within R
    - The packages `randomForest`, `caret`, and `nnet` may or may not need to be installed, depending on the type of model you want to run. I recommend installing all of them just in case

```
install.packages("terra")
install.packages("MultiscaleDTM")
install.packages("randomForest")
install.packages("caret")
install.packages("nnet")
```

4. Load in the functions. Since the WIP tool is not a package as of yet, that means to run the tool, you will have to load the functions by running them directly in R. The functions are all in the **Functions** section towards the end of this document
5. (Optional) Load in the data that will be used in the tool. This step is optional because all of the functions used in the WIP tool can receive file names as inputs. However, typing out file names over and over again can be a drag, so loading them in once can be the superior option.

```
dem <- terra::rast("Data/PF_DTM3.tif")
region <- terra::vect("Data/PF_studyarea.shp")
wetlands <- terra::vect("Data/PF_wetlands.shp")
```

The data that we loaded above to use for the book is from an area in Eatonville, WA (near Mount Rainier) called Pack Forest. It is a small land mass (making it ideal for an example), with four classified wetlands: Riverine, Freshwater Emergent Wetland, Freshwater Forested/Shrub Wetland, and Freshwater Pond. The three variables that were just loaded are the Digital Elevation Model (DEM), a polygon file describing a area, and a polygon file describing wetlands within that area, respectively.

Training Points

In order to be able to build a model that calculates wetland probabilities, we first need to have points to train the model (it has to gain knowledge from somewhere). That is what the `build_train_pts` function is here to accomplish. Using polygon data of both wetlands and the area of interest (these are best obtained from the National Wetlands Inventory, or NWI), the function will randomly sample both wetland (including an option doing so for multiple wetland types) and non-wetland points.

This step can be done either before or after calculating surface metrics, but neither can be done before building the model.

Parameters

Inputs	Description
`region_poly`	Polygon (shape file) input that contains the shape of the whole area of interest
`wet_poly`	Polygon input that contains areas where the (known) wetlands are located
`multi_class`	Binary variable indicating whether or not to return points from each of the wetland types or just have binary wetland/non-wetland points
`wet_types`	Vector input that lists all the types of wetlands we are considering draw sample points of that are listed in the `wet_poly` input
`wet_field`	String input indicating what the field name of the type of wetland in `wet_poly` is. The default is "WETLAND_TY", given that is the name NWI uses
`sample_points`	Vector indicating number of points to sample that are wet and non-wetland. First element in vector is the number of wetland points and second element is the number of non-wetland

Inputs	Description
`export`	Binary parameter that determines whether or not the function exports the output to a file

Running training points

```
simple_pts <- build_train_pts(region_poly = region,
                              wet_poly = wetlands,
                              wet_types = c("Riverine",
                                            "Freshwater Emergent Wetland",
                                            "Freshwater Forested/Shrub Wetland",
                                            "Freshwater Pond"),
                              multi_class = FALSE)
```

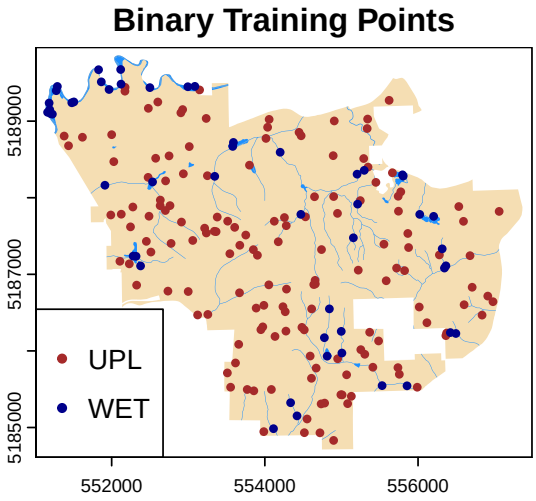
```
multi_pts <- build_train_pts(region_poly = region,
                              wet_poly = wetlands,
                              wet_types = c("Riverine",
                                            "Freshwater Emergent Wetland",
                                            "Freshwater Forested/Shrub Wetland",
                                            "Freshwater Pond"),
                              sample_points = c(20, 150),
                              multi_class = TRUE)
```

If you run this function, as is done above, it returns a Spatvector input of points, with some labeled as UPL (for upland) and others either labeled as WET or, if chosen `multi_class`, the particular type of wetland (such as Freshwater Pond), as shown below.

```
multi_pts
```

```
class       : SpatVector
geometry    : points
dimensions  : 230, 1  (geometries, attributes)
extent      : 551155.6, 557066, 5184777, 5189817  (xmin, xmax, ymin, ymax)
coord. ref. : NAD83 / UTM zone 10N (EPSG:26910)
names       :    class
type        :   <fact>
values      : Riverine
              Riverine
              Riverine
```

Here's a visual of what the training points look like



If you run the function without specifying the `wet_type`, it might still run (especially if the data was obtained from NWI), but it will also likely give warnings, for some wetland types that are not present in the file, as shown below.

```
build_train_pts(region, wetlands)
```

```
class       : SpatVector
geometry    : points
dimensions  : 200, 1  (geometries, attributes)
extent      : 551161.9, 556979, 5184869, 5189818  (xmin, xmax, ymin, ymax)
coord. ref. : NAD83 / UTM zone 10N (EPSG:26910)
names       :  class
type        : <fact>
values      :    WET
                 WET
                 WET
```

External Training Points

If you obtained a set of training points through some sort of external source, the following functions will still work, as long as there are points on the same area as what is set as the `ref_raster`, which will be discussed more later. In

other words, the training points can use a different projection system or spatial extent and still work

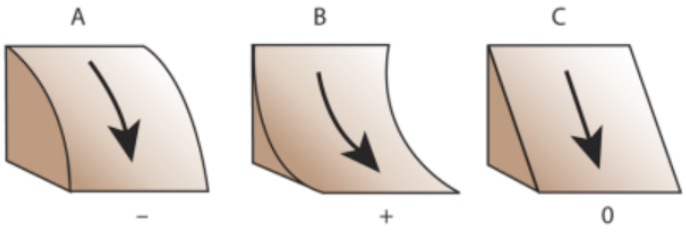
Surface Metrics

Also known as topographical indices, surface metrics are a key component of the model. They are able to capture the variability of topographic features conducive to wetland formation (think of how rivers and lakes make distinct divots into the ground around them), making this type of information is extremely useful. Starting with a DEM or a DTM (Digital Terrain Model), the functions derive the following types of surface metrics.

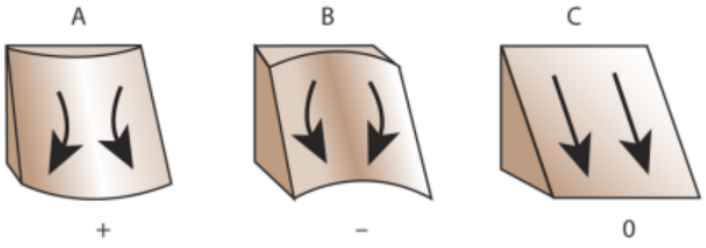
Types of Metrics

There are five types of surface metrics that we are currently able to calculate here: gradient, planar curvature, profile curvature, local relief, and topographical wetness index (TWI).

- **Gradient**: essentially just the rate of change in elevation
- **Profile curvature**: the curvature in the direction of the maximum slope



- **Planar curvature**: the curvature that is perpendicular to the direction of the maximum slope (sideways curvature)



- **Local relief (DEV)**: is the representation of where a spot is in elevation in comparison to its surroundings
- **TWI**: essentially a model of how water will flow beneath the surface, calculated from topography attributes

To calculate the above metrics, we have two different choices of function. One calculates the metrics in R, while the other function connects to a set of external files, called *Executable Files*.

Using Executable files

The first option we have is to calculate them using Executable files. These Executable files are files of FORTRAN code that are executed quickly. These files, which can be accessed here, produce the output(s) on an exported file and not in R itself, so you would have to load in the data once again.

Running the `surface_met1` function sets up the bridge between R and the Executable files. There are a few differences between running this function versus the `surface_met` function. Since the files are run externally, the user needs to type in the file directories rather than the variable in R. Another thing to note is that currently there is not a setup to run TWI with these files, so we are limited to just the other four types of metrics

Parameters

Inputs	Description
`len`	The length that the metrics are calculated at
`metrics`	Vector indicating which metrics to calculate. Options include "grad", "prof", "plan", and "dev"
`dem_dir`	String (text) input that indicates the file directory of the file that contains the DEM
`exec_dir`	String input that indicates the file directory
`out_dir`	String input of the directory where the output is
`re_sample`	Number indicating the re-sampling rate for the DEV/local relief

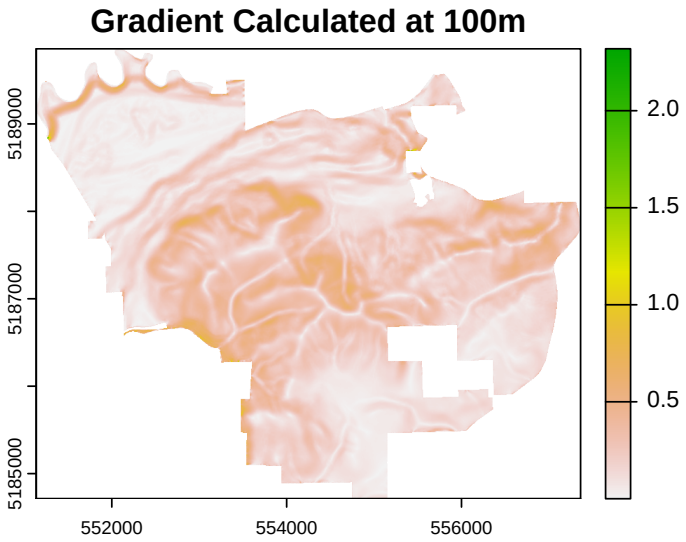
Using function example

```
surface_met1(len = 100, dem_dir = "Data/PF_DTM3.tif",
             metrics = c("grad", "prof", "plan"),
             exec_dir = "../ExecutableFiles")
```

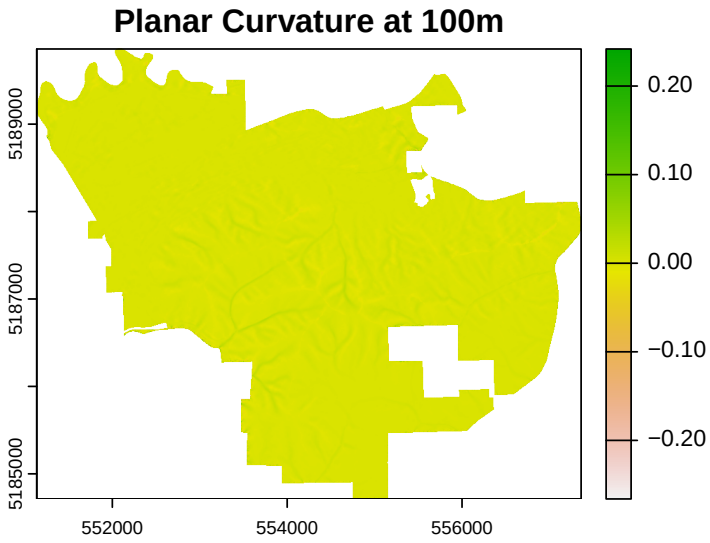
The functions creates a `.txt` file named `input_makeGrids` (or `input_localRelief` for DEV) which, as the name suggests, is an input file for the Executable Files. It lists all the important file directories (including the outputs), as well as the length scale that the metrics will be calculated at.

Visuals of example

Here is what a map of gradient looks like at 100 meter length scale, calculated above.



As for the curvatures, the minute differences might be hard to see on a visualization, but rest assured, the assumed minute differences are still important in the models that are calculated later on.


Planar Curvature at 100m
5189000
5187000
5185000
552000
554000
556000
0.20
0.10
0.00
−0.10
−0.20

Completely in R

Now, we will go into the function that calculates the metrics internally, `surface_met`. This option is less complicated to get to work, but also comes with drawbacks. It will not only take longer to run, but also has slightly less accurate results.

This function can calculate any and all five of the types of metrics described above. The default is to calculate all five, but that can be changed with a different input of `elev_dev` .

Parameters

Inputs	Description
`DEM`	As the name suggests, this input is the DEM/DTM
`len`	Number input represents the length at which the metric is calculated
`elev_dev`	Vector input of selection of metrics. The choices are "grad", "prof", "plan", "dev", "twi"
`export`	This is a binary (true/false) parameter that determines whether or not the function exports the output(s) to a file

Using the function

```
elev1 <- surface_met(dem, len = 20, elev_dev = c("grad", "prof", "plan"))
```

```
|---------|---------|---------|---------|
=========================================
```

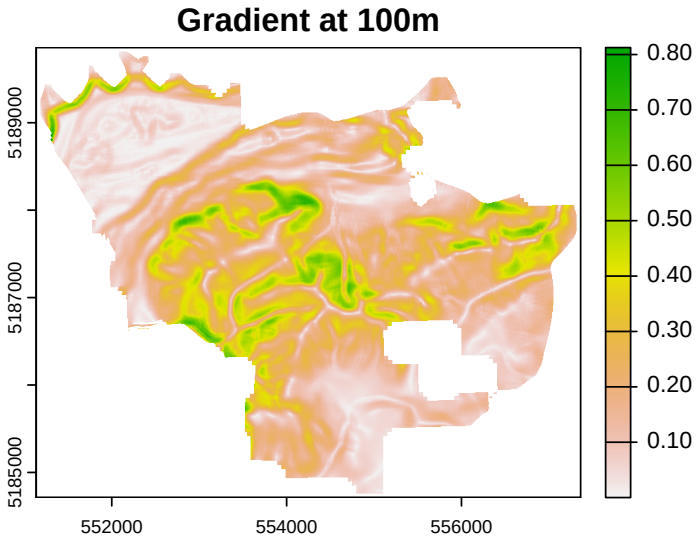
```
elev2 <- surface_met(dem, len = 100, elev_dev = c("grad", "prof", "plan"))
```

```
|---------|---------|---------|---------|
==========================================
```

```
elev3 <- surface_met(dem, len = 10, elev_dev = c("twi"))
```

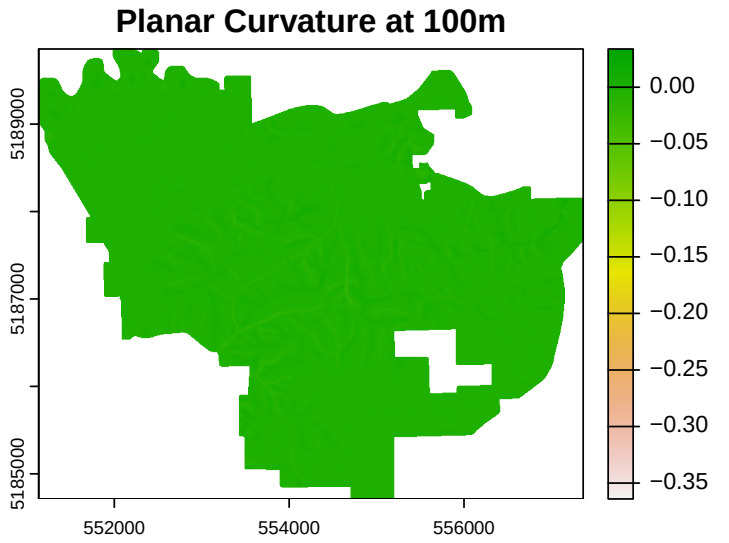
A progress bar is printed below a few of the functions, in case you were wondering what those odd lines were. This is added so that the user can have some semblance of how much is left to run. Most of the functions within the entirety of the tool do something similar, as they will often take some time to run.

Visualizing output



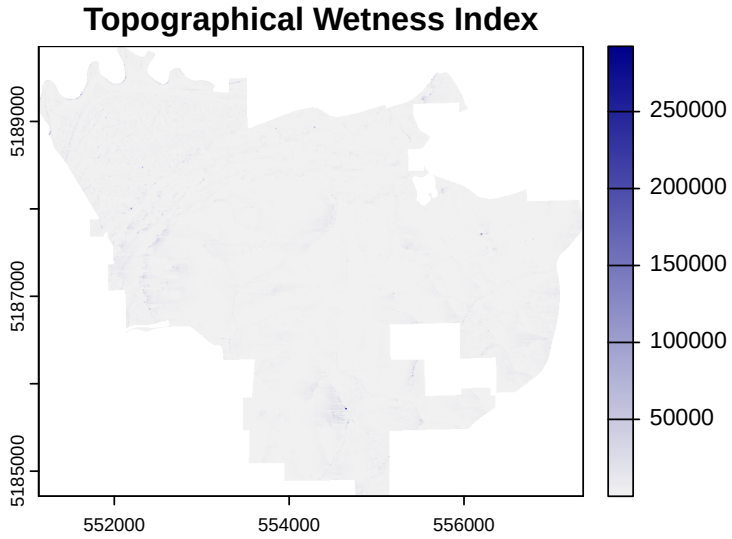
As we can see, the shape is similar to the one calculated by the Executable Files, though the scale is a little different

For the types of curvature, the edges are puffed up, as shown below.



This is done to ensure that data is not eliminated, since the other option would make it so that plenty of area around the edges would be eliminated, exponentially so with larger length scales. These will get smoothed out later on in the WIP process, so the final output will not look like this.

One last visualization that you might find interesting is how the TWI looks, so this is shown below.



Build Model

Now that we have both the surface metrics and training points, now it is finally time to start calculating wetland probabilities. The first step is building, or training, the model. This allows the model to figure out how to calculate probabilities and make predictions.

See `build_model` section for the function

Model options

The model options are decision tree, random forest, logistic regression, and k-nearest neighbors. More on these types of models in the coming pages The default setting is to run random forest with `n=200` trees. To change this, the user would just need to type `model_type =` and choose their selection of `c("forest", "tree", "glm", "knn")`.

Also, depending on the model choice, you might also have to add an extra input, `model_params`. If the choice is `knn`, then you would have to add in the number of neighbors. For example, if you want to use ten neighbors, just type in `model_params = list(k = 10)`. If you want to use a different number of tress in the random forest than `n=200`, just type in `model_params = list(ntree = )` , with the selected choice of trees after the equal sign.

Parameters

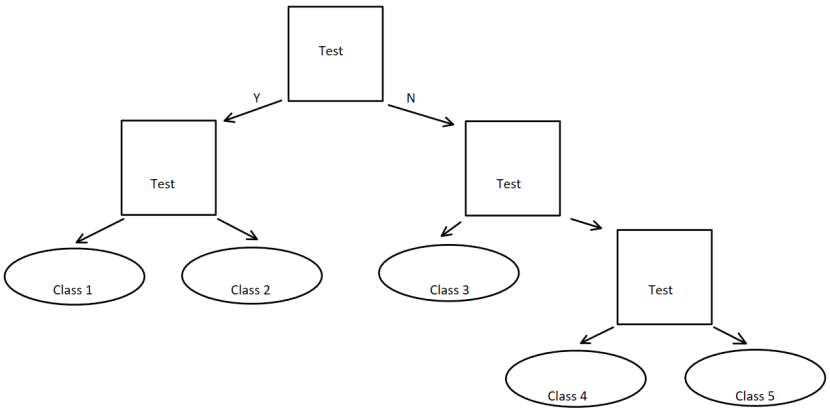
Input	Description
`in_rasts`	List input that contains all the rasters that are chosen to be included in the model
`poly_inputs`	List input that contains the polygons that are chosen to be included in the model, if any
`train`	Spatial vector input of the training points
`ref_raster`	Spatial raster input of the reference raster, which is used to align all of the other inputs. The DEM is usually a good choice here
`model_type`	String input indicating type of machine learning model. Options include “forest”, “tree”, “glm”, and “knn”
`model_params`	List input representing necessary parameters to the model. Depending on the model, it may require an extra input, such as the number of trees or number of neighbors.

Input	Description
`class_field_name`	String input indicating the field where the wetland classification is. If calculated training points using the `build_train_pts` function above, this input may be skipped. However, very important if training points came externally

Decision Tree

Description

A decision tree is a method of machine learning that works a lot like a flow chart. Using the features, or input data, the tree then conducts tests on the data. An example of one of these tests, would be if the gradient at 100 meters is greater than 0.5, or whether the planar curvature is greater than 0.05, or if TWI is higher than zero. After a test, then depending on its result, the tree either assigns a class or checks another test. The tree keeps conducting tests until it assigns a class to the observation. A diagram of how this whole system works can be found below



Example of use

```
multi_mod <- build_model(in_rasts = c(elev1, elev2, elev3),
                         train = multi_pts, ref_raster = dem,
                         class_field_name = "class",
                         model_type = "tree")
```

```
[1] "Formatting inputs"
[1] "Setting up training data"
[1] "Building model"
[1] "Done!"
```

```
simple_mod <- build_model(in_rasts = c(elev1, elev2, elev3),
                         train = simple_pts, ref_raster = dem,
                         model_type = "tree")
```

```
[1] "Formatting inputs"
[1] "Setting up training data"
[1] "Building model"
[1] "Done!"
```

The returned object is a `randomForest` object (more on why on the next page), which contains a large amount of information. If run, as is done below, it returns the estimated error as well as a confusion matrix on the training data

```
multi_mod
```

```
Call:
 randomForest(formula = class ~ ., data = df_train, ntree = 1)
               Type of random forest: classification
                     Number of trees: 1
No. of variables tried at each split: 2

        OOB estimate of  error rate: 44.09%
Confusion matrix:
                                  Freshwater Emergent Wetland
Freshwater Emergent Wetland                                 1
Freshwater Forested/Shrub Wetland                           0
Freshwater Pond                                             1
Riverine                                                    0
UPL                                                         4
```

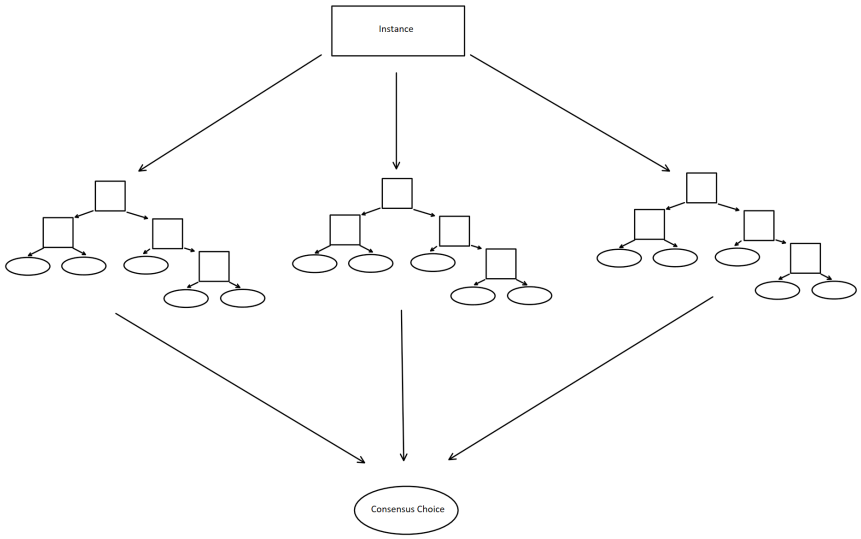
```
                                  Freshwater Forested/Shrub Wetland
Freshwater Emergent Wetland                                       0
Freshwater Forested/Shrub Wetland                                 0
Freshwater Pond                                                   0
Riverine                                                          1
UPL                                                               5
                                  Freshwater Pond Riverine UPL class.error
Freshwater Emergent Wetland                    10        0   1   0.9166667
Freshwater Forested/Shrub Wetland               3        0   3   1.0000000
Freshwater Pond                                 2        0   0   0.3333333
Riverine                                        3        2   3   0.7777778
UPL                                             6        1  47   0.2539683
```

Random Forest

Our next model is a random forest, which is set as the default model for the `build_model` function.

Description

A random forest is a model that generates a large number of decision trees (hence the name). To calculate probabilities, the model checks the decisions and then calculates the proportion of the trees that chose each class. Those proportions are the estimated probabilities. To make an overall prediction, the model just takes the class with the highest amount trees that decided in its favor.



Examples of Use

```
multi_mod <- build_model(in_rasts = c(elev1, elev2, elev3),
                         train = multi_pts, ref_raster = dem)
```

```
[1] "Formatting inputs"
[1] "Setting up training data"
[1] "Building model"
[1] "Done!"
```

```
simple_mod <- build_model(in_rasts = c(elev1, elev2, elev3),
                          train = simple_pts, ref_raster = dem)
```

```
[1] "Formatting inputs"
[1] "Setting up training data"
[1] "Building model"
[1] "Done!"
```

Since they are very similar algorithms, the returned output for random forest models is the same as with trees, as we can see from the printed output below:

```
multi_mod
```

```
Call:
 randomForest(formula = class ~ ., data = df_train, ntree = model_params$ntree)
               Type of random forest: classification
                     Number of trees: 200
No. of variables tried at each split: 2

        OOB estimate of  error rate: 30.41%
Confusion matrix:
                                  Freshwater Emergent Wetland
Freshwater Emergent Wetland                                 7
Freshwater Forested/Shrub Wetland                           1
Freshwater Pond                                             1
Riverine                                                    2
UPL                                                         4
                                  Freshwater Forested/Shrub Wetland
Freshwater Emergent Wetland                                       0
Freshwater Forested/Shrub Wetland                                 1
Freshwater Pond                                                   3
```

```
Riverine                                                1
UPL                                                     4
                                  Freshwater Pond Riverine UPL class.error
Freshwater Emergent Wetland                   2        0  11  0.65000000
Freshwater Forested/Shrub Wetland             2        1  12  0.94117647
Freshwater Pond                               8        0   7  0.57894737
Riverine                                      0        4  12  0.78947368
UPL                                           0        3 131  0.07746479
```

Logistic Regression

Description

Logistic regression is another type of machine learning model. Different from the first two models, logistic regression is much more mathematically involved. In short, it tries to find relationships between each field of input data and the resulting class (i.e. finding connections between gradient and wetland type). Using these relationships, it then creates a regression formula (similar to the slope equation $y = mx + b$ in Algebra) that is then used to probabilities. Then, whichever class is calculated to have highest probability is what is the predicted outcome.

One caveat of logistic regression is that it assumes that each of the input data is independent of each other. However, this is often the case, especially using spatial data. As a result, the predictions/probabilities could end up looking a little unsual, if not careful.

Examples of Use

```
multi_mod <- build_model(in_rasts = c(elev1, elev2, elev3),
                         train = multi_pts, ref_raster = dem,
                         model_type = "glm")
```

```
[1] "Formatting inputs"
[1] "Setting up training data"
[1] "Building model"
# weights:  45 (32 variable)
initial  value 349.248027
iter  10 value 223.826226
iter  20 value 195.529526
iter  30 value 188.498465
```

```
iter  40 value 184.766843
iter  50 value 183.599588
iter  60 value 181.373417
iter  70 value 179.208432
iter  80 value 178.122418
iter  90 value 176.997883
iter 100 value 176.512240
final  value 176.512240
stopped after 100 iterations
[1] "Done!"
```

```
simple_mod <- build_model(in_rasts = c(elev1, elev2, elev3),
                          train = simple_pts, ref_raster = dem,
                          model_type = "glm")
```

```
[1] "Formatting inputs"
[1] "Setting up training data"
[1] "Building model"
[1] "Done!"
```

Returned is a list object, which contains lots and lots of information, including the predicted values of the training data, different levels of accuracy assessments, and the coefficients. These coefficients are a measure of the relationship between that data field and the class. Farther the coefficient is from zero, the larger the amount of change is supposed to be when picking between classes.

Running the object in console, like done below, shows us these coefficients

```
simple_mod
```

```
Call:  glm(formula = class ~ ., family = "binomial", data = df_train)

Coefficients:
(Intercept)       grad20       prof20       plan20      grad100      prof100
 -6.695e-01   -7.598e+00   -1.306e+01   -1.149e+02   -1.749e+00   -2.549e+02
    plan100        twi10
 -1.263e+02    3.905e-05

Degrees of Freedom: 178 Total (i.e. Null);  171 Residual
Null Deviance:      185.1
Residual Deviance: 133.5    AIC: 149.5
```

K-Nearest Neighbors

Description

Last, but not least, we have reached the k-Nearest Neighbors machine learning algorithm. This one (may) be the easiest to comprehend. How it works is that it finds a certain number of observations, k to be exact, that have features (the input data) that most closely match the observation we are trying to predict. We will call these close observations *neighbors*. Using those neighbors, the model predicts the class by picking the class that has the highest number of neighbors. For probabilities, it uses the proportion of neighbors of that class out of the total. Here is an example below.

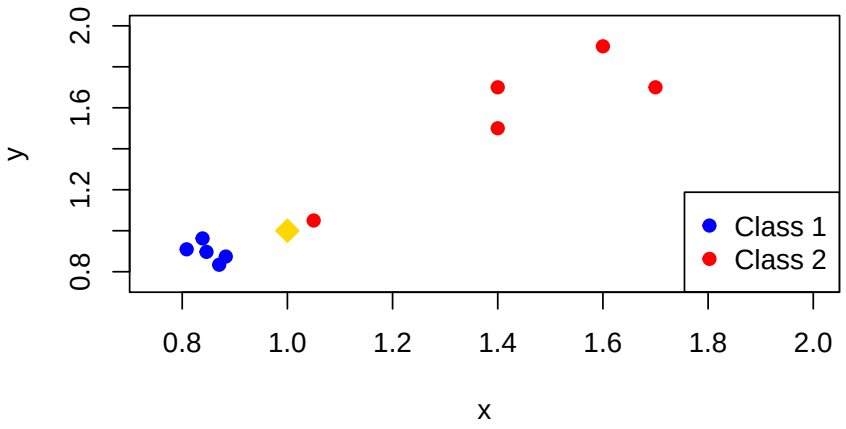


Figure 2: Two classes of train data; gold diamond is new observation

From the example above, we see how the decision can change with different

levels of k. If $k = 1$, then the decision would be `Class 2`, since the closest point to the diamond point is red. However, with any number of neighbors higher than 2, then decision would be `Class 2`.

One piece of advice is to normally choose either an odd number or a large number for k. This significantly lessens the chance that there is a tie when the model makes a decision. As a result, it might also be a good idea to run it with a larger amount of training points.

Examples of Use

```
multi_mod <- build_model(in_rasts = c(elev1, elev2, elev3),
                         train = multi_pts, ref_raster = dem,
                         model_type = "knn",
                         model_params = list(k = 15))
```

```
[1] "Formatting inputs"
[1] "Setting up training data"
[1] "Building model"
[1] "Done!"
```

```
simple_mod <- build_model(in_rasts = c(elev1, elev2, elev3),
                         train = simple_pts, ref_raster = dem,
                         model_type = "knn",
                         model_params = list(k = 5))
```

```
[1] "Formatting inputs"
[1] "Setting up training data"
[1] "Building model"
[1] "Done!"
```

The returned output is a list that contains the input data and the predicts. As we run the object below, we can see the number the model predicted for each class

```
multi_mod
```

```
15-nearest neighbor model
Training set outcome distribution:
```

```
Freshwater Emergent Wetland Freshwater Forested/Shrub Wetland
                         20                                 17
            Freshwater Pond                          Riverine
                         19                                 19
                        UPL
                        142
```

Running Model

We have surface metrics and have built a model. Now it is time to run the model, which means it is finally time to make predictions and/or obtain probabilities. To accomplish this, we use the `run_model` function. The function either returns a classification raster or a set of probability rasters, depending on your choice.

Parameters

Inputs	Description
`in_rasts`	List input that contains all the rasters that are chosen to be included in the model
`poly_inputs`	List input that contains the polygons that are chosen to be included in the model, if any
`ref_raster`	Spatial raster input of the reference raster, which is used to align all of the other inputs. The DEM is usually a good choice here
`model_type`	String input indicating type of machine learning model. Options include "forest", "tree", "glm", and "knn"
`class_rast`	Binary parameter that determines whether the output will be probability rasters or a classification raster
`export`	Binary parameter that determines whether or not the function exports the output to a file

Executing the function

```
prob_multi <- run_model(multi_mod,
                        in_rasts = c(elev1, elev2, elev3),
                        ref_raster = dem)
```

```
[1] "Formatting inputs"
[1] "Stacking rasters"
[1] "Running model"

|---------|---------|---------|---------|
=========================================

[1] "Done!"
```

```
prob_simple <- run_model(simple_mod,
                         in_rasts = c(elev1, elev2, elev3),
                         ref_raster = dem)
```

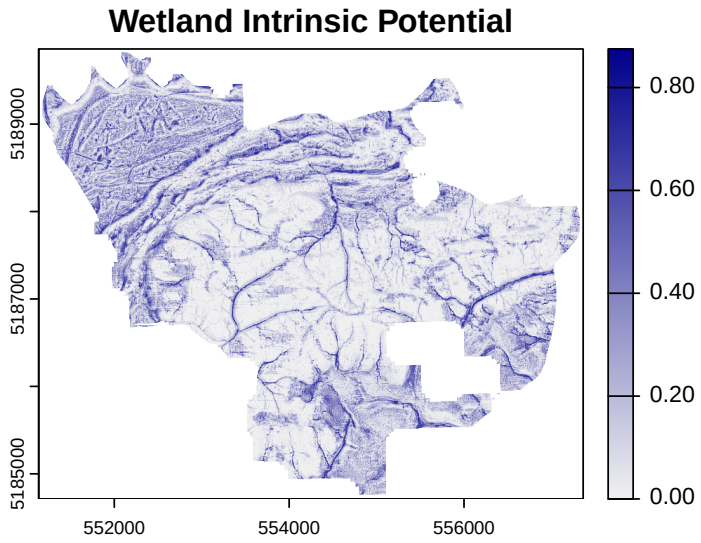
```
[1] "Formatting inputs"
[1] "Stacking rasters"
[1] "Running model"
[1] "Done!"
```

The function outputs a SpatRaster object has probabilisitic (or categorical) values and has the same extent (meaning) as the reference raster, as seen below.

```
prob_simple
```

```
class       : SpatRaster
dimensions  : 1287, 1557, 2  (nrow, ncol, nlyr)
resolution  : 3.999648, 3.998959  (x, y)
extent      : 551135.5, 557362.9, 5184713, 5189859  (xmin, xmax, ymin, ymax)
coord. ref. : NAD83 / UTM zone 10N (EPSG:26910)
source(s)   : memory
names       :  UPL,  WET
min values  : 0.12, 0.00
max values  : 1.00, 0.88
```

Below is how the output looks like on a map



Classification Option

Instead of returning probabilities, the `run_model` function also has an option to return a classification raster. This is often useful for when we testing the model to predict multiple types of wetlands (multi-class classification), since it is easier to comprehend one prediction rather than lots of probabilities.

```
class_multi <- run_model(multi_mod, class_rast = TRUE,
                         in_rasts = c(elev1, elev2, elev3),
                         ref_raster = dem)
```

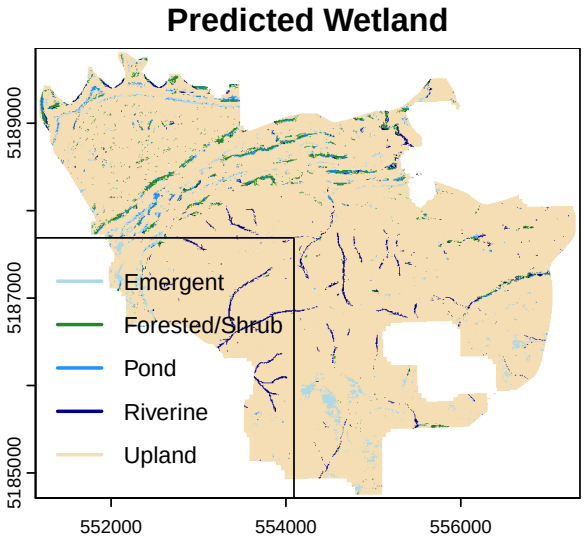
```
[1] "Formatting inputs"
[1] "Stacking rasters"
[1] "Running model"
[1] "Done!"
```

```
class_multi
```

```
class       : SpatRaster
dimensions  : 1287, 1557, 1  (nrow, ncol, nlyr)
resolution  : 3.999648, 3.998959  (x, y)
extent      : 551135.5, 557362.9, 5184713, 5189859  (xmin, xmax, ymin, ymax)
```

```
coord. ref. : NAD83 / UTM zone 10N (EPSG:26910)
source(s)   : memory
categories  : class
name        :                          class
min value   : Freshwater Emergent Wetland
max value   :                            UPL
```

Here is what the output looks like on a map:



Testing model

So we have model predictions on the whole area. But just how accurate are these predictions? That is what the `CV_err` function is here to find out. The function uses cross validation, which is done by splitting the data into a certain number of chunks, called folds (which is represented by the `kfold` parameter), sets one aside to be the test data, trains the model on the rest of the data, and then cross-references the model's predictions against the test data. More explanation on how it works can be found here. One thing to note is these estimates might be slight overestimates (since the models here are not built using all of the data).

The estimates will be produced as a printed output, meaning it does not need to be saved to a variable.

Parameters

The inputs for this function are almost the exactly the same as the `build_model` function, with one addition, `kfold`, the number of folds used in cross-validation.

Inputs	Descriptions
`in_rasts`	List input that contains all the rasters that are chosen to be included in the model
`poly_inputs`	List input that contains the polygons that are chosen to be included in the model, if any
`train`	Spatial vector input of the training points
`ref_raster`	Spatial raster input of the reference raster, which is used to align all of the other inputs. The DEM is usually a good choice here
`model_type`	String input indicating type of machine learning model. Options include "forest", "tree", "glm", and "knn"
`model_params`	List input representing necessary parameters to the model. Depending on the model, it may require an extra input, such as the number of trees or number of neighbors.

Inputs	Descriptions
`class_field_name`	String input indicating the field where the wetland classification is. If calculated training points using the `build_train_pts` function above, this input may be skipped
`kfold`	Numeric input indicating number of folds data is split into. Represents number of times we test the model

Examples of use

The function has the same inputs as the `build_model` functions. However, there is one more feature to customize, which is `k_folds`, or the number of times the cross validation is run. The default for the function is `kfolds = 5`

```
  CV_err(in_rasts = c(elev1, elev2, elev3), train = multi_pts,
         ref_raster = dem)
```

```
[1] "Test Error Estimate: 33.3%"
[1] "95% Confidence Interval: [32.3, 34.3]"
```

```
  CV_err(in_rasts = c(elev1, elev2, elev3), train = simple_pts,
         ref_raster = dem)
```

```
[1] "Test Error Estimate: 22.2%"
[1] "95% Confidence Interval: [19.1, 25.4]"
```

As we can tell above, the binary model is doing a much better job of prediction than the multi-class model

Extraneous Tips

Visualizing model

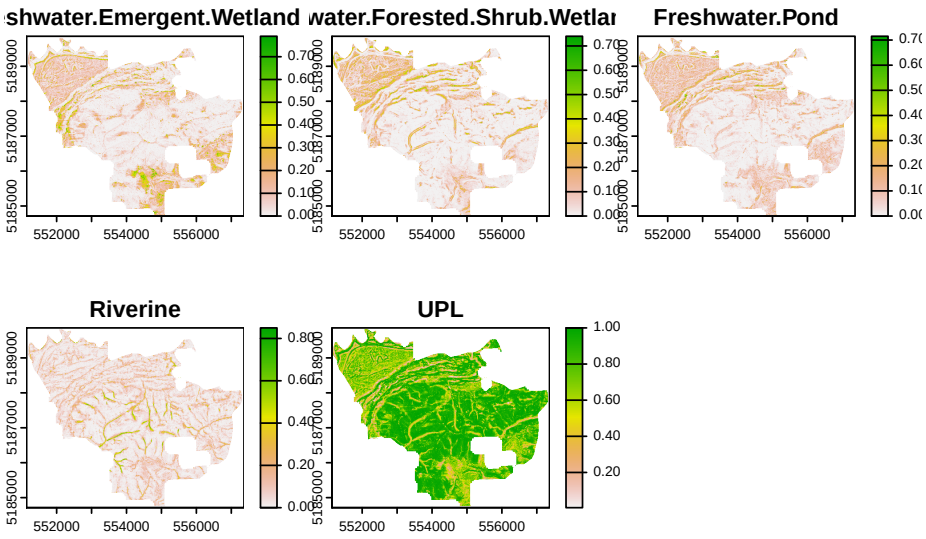
To conduct basic plotting, it is easiest to just load in the `terra` package, as done below.

```
library(terra)
```

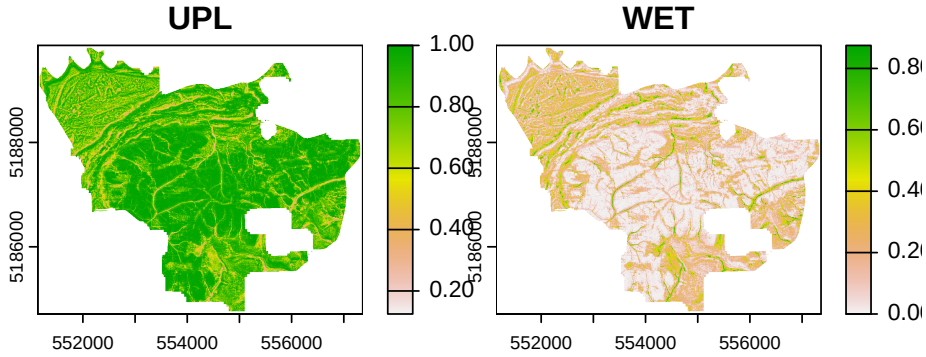
```
terra 1.7.55
```

Now, you can just type in the `plot` function and add the raster/vector object inside

```
plot(prob_multi)
```

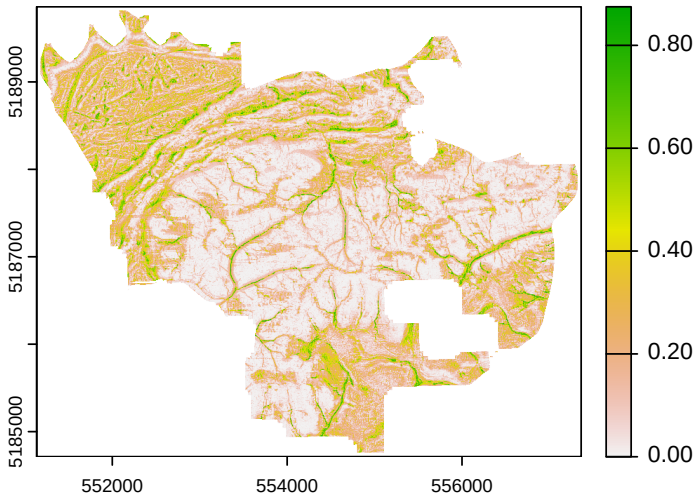



```
plot(prob_simple)
```



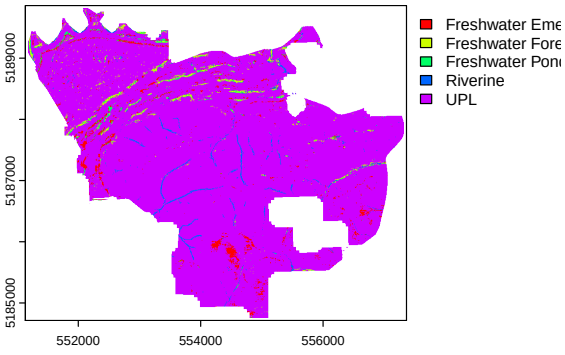
Since having both of these plots above is redundant, we can also choose to plot just one

```
plot(prob_simple["WET"])
```



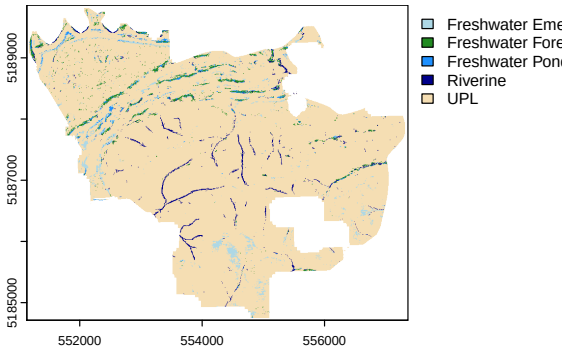
If we want to be fun and add rainbow colors, type in `rainbow(n)`, with `n` being the number of meadows. Since there are 5 different classes in `class_multi`, we will type in `rainbow(5)`

```
plot(class_multi, col = rainbow(5))
```



Also can choose the color for each class. Here is a description of all the colors that can be used

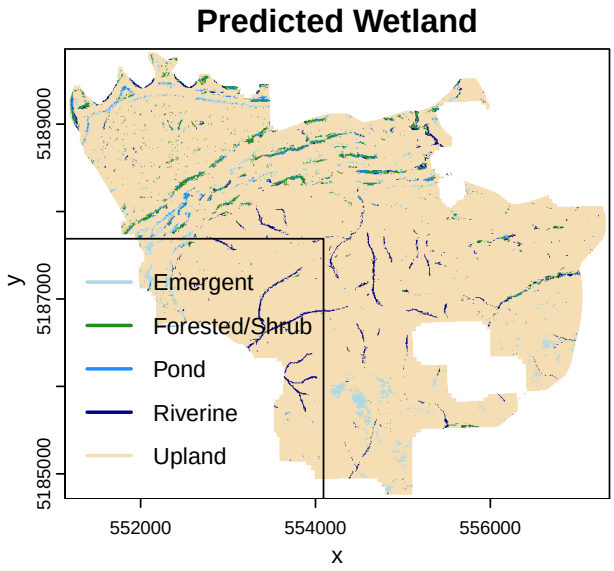
```
plot(class_multi, col = c("lightblue", "forestgreen", "dodgerblue", "darkblue", "whe
```



You can also add labels to the plots as well. Adding a title just requires an input of `main` and adding x and y labels just require inputs for `xlab` and `ylab`, respectively (though x and y labels are less important in spatial plotting).

Another thing that you can try is changing the legend. We can see from the above plots that the names for the values are simply too long for the legend. To improve on this, we need to add the `add_legend()` function in `terra` after `plot()`. The inputs of this are the position of the legend (should be typed in first), the labels for the legend, and the colors that are depicted (can just copy/paste from `plot`). A few other inputs are `cex` (legend size) and `lwd` (line width), both of which can be chosen by trial and error. Also, in order to prevent having multiple legends, type `legend = FALSE` into `plot()`.

```
plot(class_multi, main = "Predicted Wetland", xlab = "x",
     ylab = "y", col = c("lightblue", "forestgreen", "dodgerblue", "darkblue", "whea
add_legend("bottomleft",
           legend=c("Emergent", "Forested/Shrub", "Pond",
                    "Riverine", "Upland"),
           col= c("lightblue", "forestgreen", "dodgerblue", "darkblue", "wheat"), ce
```



More advanced plotting

If you desire more advanced and more customizable plots, then that is what the `tidyterra` package is for. More information on this package, including how it is used, can be found here. The package includes data manipulation features, which you might find interesting, but is not within the scope of the WIP tool.

Export

Say you to take the outputs, whether they are the probability rasters or the gradient, and be able to use or see them outside of R. This could be so you can take a look at them using a GIS software or so you can share your results. Whatever the reason, there are a number of ways to do this.

As mentioned above, many of the functions, such as `surface_met` and `run_model`, have export options (using the executable files exports them automatically). As such, the only action needed to be done in order to have an exported file would be to type in `export = T`.

However if that did not happen, then no worries, here is a guide to exporting on your own. For this, we will use the `terra::writeRaster()` function, which requires two inputs: the R input we want to export and the name of the exported file. Below we will export the `class_multi` raster into a file named PF_class.tif

```
terra::writeRaster(class_multi, filename = "PF_class.tif")
```

This file ends up in the current working directory. To find what directory you are in, run the function `getwd()`. To change the working directory, at the top of the screen, click on Session -> Set Working Directory -> Choose Directory.

Another way to choose where the exported file ends up is to type in the full directory into the `filename` input. This is shown below:

```
terra::writeRaster(class_multi,
                   filename = "C:Projects/WIP/PF_class.tif")
```

One last thing to note is for how to export when the object contains more than one raster inside. This happens above with objects like `prob_multi`, `elev1`, and `elev2` (among others). To export these correctly, you will need to export each raster one at a time. This is done by adding a `$` after the variable name, and then typing the name of the specific raster after, as shown below:

```
terra::writeRaster(prob_multi$Pond, filename = "Pond.tif")
terra::writeRaster(prob_multi$Riverbed, filename = "River.tif")
terra::writeRaster(prob_multi$Water, filename = "Water.tif")
```

Errors

As always in dealing with any sort of computer processing, it is just a matter of time until an error message pops up. This section covers some of the more common errors that may show up when running the WIP tool that are also not easily explained by the error message.

- If you are typing in a file directory and use "\" (backslash) once when acknowledging the break between files, R will throw an error such as `'\P' is an unrecognized escape in character string` (the P is a placeholder for whatever letter is after the backslash)
- `std::bad_aloc` is a memory error, meaning whatever R is trying to do is taking up too much memory for the computer to handle. Apart from just using a computer with more memory, the best way to solve this is to split up the area you are running into multiple parts, run each of the parts one at a time, and then mosaic them together afterwards.
- `No wetlands to sample` is an error from the `build_train_pts` function. Aside from the wetlands file actually being empty, there are two possible issues:

- Incorrect input in `wet_types`, meaning that the function is searching for wetlands that are actually not there
- Incorrect input in `wet_field`, which means that the function can't find where the types of wetlands are listed. This will happen when the wetlands are from a different database than NWI. To figure out what the field name is, try using the `names()` function and then see which field is most likely to contain the wetland types, then type that into `wet_field`

However, if a different error shows up, keep in mind that usually the best way to solve an error is to copy the error message R puts up, and then paste it into a search engine. It might seem like a cheat and a bit lazy, but it is a very successful debugging tactic.

Functions

This section contains the code for all of the functions mentioned throughout this whole book. As a review, a list of all of these functions is below:

- build_train_pts
- surface_met1
- surface_met2
- build_model
- run_model
- CV_err

To use these functions are your device, either click the *Functions* link under the **Downloads** tab in the right corner, which downloads an `.RData` file containing them and can be loaded using the `load` function, or copy/paste each of these sectiona into your own R session

build_train_pts

```
build_train_pts <- function(region_poly, wet_poly, multi_class = FALSE,
                            wet_types = c("Freshwater Forested/Shrub Wetland",
                                          "Freshwater Emergent Wetland",
                                          "Freshwater Pond",
                                          "Estuarine and Marine Wetland",
                                          "Riverine", "Lake",
                                          "Estuarine and Marine Deepwater",
                                          "Other"),
                            wet_field = "WETLAND_TY",
                            sample_points = c(50, 150),
                            export = FALSE) {

  # Loads in polygons if input is a file name
  if(is.character(wet_poly[[1]])) {
    temp_poly <- list()
    for(i in 1:length(wet_poly)) {
      temp_poly[[i]] <- terra::vect(wet_poly)
    }
    wet_poly <- temp_poly
  }

  if(is.character(region_poly)) {
    region_poly <- terra::vect(region_poly)
  }

  # Filters the wetland polygons to only include wanted types
  wet_poly <- wet_poly[unlist(wet_poly[[wet_field]]) %in% wet_types]
  if(length(wet_poly) == 0) {
    stop("No wetlands to sample!")
  }

  # Cropping the wetland polygon(s) to the overall region
```

```
wet_poly <- terra::project(wet_poly, region_poly)
wet_poly <- terra::crop(wet_poly, region_poly)

# Checks if output is supposed to be more than two classes before proceeding
if(multi_class) {
  # Initialize parameters
  train_crds <- NULL
  train_atts <- c()
  wet_samp <- sample_points[1]
  up_samp <- sample_points[2]

  # Sample points for each wetland class
  for(i in 1:length(wet_types)) {
    temp_poly <- wet_poly[unlist(wet_poly[[wet_field]]) == wet_types[i]]

    # Checking if polygons of that type of wetland exist
    if(length(temp_poly) == 0) {
      warning(paste0(wet_types[i], " not found!"))
    }
    else {
      wet_crds <- NULL
      samp_wet_pts <- terra::spatSample(temp_poly, wet_samp)
      coords <- terra::crds(samp_wet_pts)
      wet_crds <- rbind(wet_crds, coords)

      num_coords <- nrow(coords)
      while(num_coords < wet_samp) {
        new_points <- terra::spatSample(temp_poly,
                                        wet_samp-(num_coords))
        new_crds <- terra::crds(new_points)
        wet_crds <- rbind(wet_crds, new_crds)
        num_coords <- num_coords + nrow(new_crds)
      }

      train_crds <- rbind(train_crds, wet_crds)
      train_atts <- c(train_atts, rep(wet_types[i], wet_samp))
    }
  }

  # Sample points from non-wetland areas
  up_poly <- terra::erase(region_poly, wet_poly)
  samp_up_pts <- terra::spatSample(up_poly, up_samp)
  up_crds <- terra::crds(samp_up_pts)
```

```
    # Create the points
    train_crds <- rbind(train_crds, up_crds)
    train_atts <- c(train_atts, rep("UPL", up_samp))
    train_atts <- data.frame(class = factor(train_atts))
    pts <- terra::vect(train_crds, atts = train_atts,
                       crs = terra::crs(region_poly))
  } else {
    # Sample the wetland points
    wet_samp <- sample_points[1]
    up_samp <- sample_points[2]
    wet_crds <- NULL

    num_points <- c()
    total_area <- sum(terra::expanse(wet_poly))
    for(i in 1:length(wet_types)) {
      temp_poly <- wet_poly[unlist(wet_poly[[wet_field]]) == wet_types[i]]

      # Checking if polygons of that type of wetland exist
      if(length(temp_poly) == 0) {
        warning(paste0(wet_types[i], " not found!"))
      }
      else {
        prop_area <- sum(terra::expanse(temp_poly)) / total_area
        num_points[i] <- round(prop_area * wet_samp)
        if(num_points[i] != 0) {
          samp_wet_pts <- terra::spatSample(temp_poly,
                                            num_points[i])
          coords <- terra::crds(samp_wet_pts)
          wet_crds <- rbind(wet_crds, coords)
          num_coords <- nrow(coords)
          while(num_coords < num_points[i]) {
            new_points <- terra::spatSample(temp_poly,
                                            num_points[i]-(num_coords))
            new_crds <- terra::crds(new_points)
            wet_crds <- rbind(wet_crds, new_crds)
            num_coords <- num_coords + nrow(new_crds)
          }
        }
      }
    }
    if(sum(num_points, na.rm = T) != wet_samp) {
      stop("Please try another sample size")
    }
```

```
    # Sample points from non-wetland areas
    up_poly <- terra::erase(region_poly, wet_poly)
    samp_up_pts <- terra::spatSample(up_poly, up_samp)
    up_crds <- terra::crds(samp_up_pts)

    # Create the points
    train_crds <- rbind(wet_crds, up_crds)
    train_atts <- data.frame(class = factor(c(rep("WET", wet_samp),
                                              rep("UPL", up_samp))))
    pts <- terra::vect(train_crds, atts = train_atts,
                       crs = terra::crs(region_poly))
  }

  # Return the points and exports them, if desired
  if(export) {
    terra:writeVector(pts, filename = "trainingdata.shp")
  }

  return(pts)
}
```

surface_met1

```
surface_met1 <- function(len, metrics = c("grad", "plan", "prof", "dev"),
                         dem_dir, exec_dir, out_dir=getwd(), re_sample = NA) {

  # Checking to see if directories exist
  if(!file.exists(dem_dir)) {
    stop("DEM directory does not exist!")
  }

  if(!dir.exists(exec_dir)) {
    stop("Executable Files directory does not exist!")
  }

  # Prepare inputs
  dem_dir <- normalizePath(dem_dir)
  out_dir <- normalizePath(out_dir)
  if(!endsWith(out_dir, "\\")) {
    out_dir <- paste0(out_dir, "\\")
  }
  exec_dir <- normalizePath(exec_dir)

  # Write input file
  file_name <- paste0(out_dir, "input_makeGrids.txt")
  file.create(file_name)

  writeLines(c("# Input file for makeGrids",
               "",
               paste0("DEM: ", dem_dir),
               paste0("SCRATCH DIRECTORY: ", out_dir),
               paste0("LENGTH SCALE: ", len)), con = file_name)

  if("grad" %in% metrics) {
    write(paste0("GRID: GRADIENT, OUTPUT FILE = ", out_dir, "grad", len, ".flt"),
```

```
          file = file_name, append = T)
  }

  if("plan" %in% metrics) {
    write(paste0("GRID: PLAN CURVATURE, OUTPUT FILE = ", out_dir,
                 "plan", len), file = file_name, append = T)
  }

  if("prof" %in% metrics) {
    write(paste0("GRID: PROFILE CURVATURE, OUTPUT FILE = ", out_dir,
                 "prof", len), file = file_name, append = T)
  }

  # Run surface metrics sans DEV
  system(paste0(exec_dir, "\\makeGrids"), input = file_name)

  # Writing input file for DEV
  if ("dev" %in% metrics) {
    if(is.na(re_sample)) {
      stop("Set re_sample level")
    }

    # Prepare inputs
    file_name <- paste0(out_dir, "input_localRelief.txt")
    rad <- len / 2

    # Create and write input file
    file.create(file_name)
    writeLines(c("# Input file for LocalRelief",
                 "# Creating by surfaceMetrics.R",
                 paste0("# On ", Sys.time()),
                 paste0("DEM: ", dem_dir),
                 paste0("SCRATCH DIRECTORY: ", out_dir),
                 paste0("RADIUS: ", rad),
                 paste0("DOWN SAMPLE: ", re_sample),
                 paste0("SAMPLE INTERVAL: ", re_sample),
                 paste0("OUTPUT LOCAL RASTER: ", out_dir, "local", len)),
               con = file_name)

    # Run DEV in console
    system(paste0(exec_dir, "\\localRelief"), input = file_name)
  }
}
```

surface_met

```
surface_met <- function(DEM, len, export = FALSE,
                        elev_dev = c("grad", "plan", "prof", "dev", "twi")) {
  # Checks if inputs are file names and loads them in
  if(is.character(DEM)) {
    if(!file.exists(DEM)) {
      stop("Cannot find DEM file")
    }
    DEM <- terra::rast(DEM)
  }
  # Sets up the resolution
  k <- round(len/terra::res(DEM)[1])
  if (k %% 2 == 0) {
    k <- k + 1
  }

  # Initialize the inputs for the model
  in_rast <- list()

  if("grad" %in% elev_dev) {
    j <- k/2 - 0.5

    xl.end <- matrix(c(1, rep(NA_real_, times=k-1)), ncol=k, nrow=1)
    xr.end <- matrix(c(rep(NA_real_, times=k-1), 1), ncol=k, nrow=1)

    x.mids <- matrix(NA_real_, ncol=k, nrow=j-1)

    xl.mid <- matrix(c(2, rep(NA_real_, times=k-1)), ncol=k, nrow=1)
    xr.mid <- matrix(c(rep(NA_real_, times=k-1), 2), ncol=k, nrow=1)

    xl.mat <- rbind(xl.end, x.mids, xl.mid, x.mids, xl.end)
    xr.mat <- rbind(xr.end, x.mids, xr.mid, x.mids, xr.end)
```

```
  yt.end <- matrix(c(1, rep(NA_real_, times=k-1)), ncol=1, nrow=k)
  yb.end <- matrix(c(rep(NA_real_, times=k-1), 1), ncol=1, nrow=k)

  y.mids <- matrix(NA_real_, ncol=j-1, nrow=k)

  yt.mid <- matrix(c(2, rep(NA_real_, times=k-1)), ncol=1, nrow=k)
  yb.mid <- matrix(c(rep(NA_real_, times=k-1), 2), ncol=1, nrow=k)

  yt.mat <- cbind(yt.end, y.mids, yt.mid, y.mids, yt.end)
  yb.mat <- cbind(yb.end, y.mids, yb.mid, y.mids, yb.end)

  dz.dx.l <- terra::focal(DEM, xl.mat, fun=sum, na.rm=T, na.policy = "omit")
  dz.dx.r <- terra::focal(DEM, xr.mat, fun=sum, na.rm=T, na.policy = "omit")
  dz.dy.t <- terra::focal(DEM, yt.mat, fun=sum, na.rm=T, na.policy = "omit")
  dz.dy.b <- terra::focal(DEM, yb.mat, fun=sum, na.rm=T, na.policy = "omit")

  wts.l <- terra::focal(!is.na(DEM), w=xl.mat, fun=sum, na.rm=TRUE,
                        na.policy = "omit")
  wts.r <- terra::focal(!is.na(DEM), w=xr.mat, fun=sum, na.rm=TRUE,
                        na.policy = "omit")
  wts.t <- terra::focal(!is.na(DEM), w=yt.mat, fun=sum, na.rm=TRUE,
                        na.policy = "omit")
  wts.b <- terra::focal(!is.na(DEM), w=yb.mat, fun=sum, na.rm=TRUE,
                        na.policy = "omit")
  dz.dx <- ((dz.dx.r/wts.r) - (dz.dx.l/wts.l))/(2*j*terra::xres(DEM))
  dz.dy <- ((dz.dy.t/wts.t) - (dz.dy.b/wts.b))/(2*j*terra::yres(DEM))

  grad <- sqrt(dz.dx^2 + dz.dy^2)
  in_rast <- c(in_rast, grad)

  names(in_rast)[length(in_rast)] <- paste0("grad", len)
}

if("plan" %in% elev_dev) {
  if ("prof" %in% elev_dev) {
    both <- MultiscaleDTM::Qfit(DEM, metrics = c("planc", "profc"),
                                w = k, na.rm = T)
    in_rast <- c(in_rast, both[[1]], both[[2]])

    names(in_rast)[length(in_rast)-1] <- paste0("plan", len)
    names(in_rast)[length(in_rast)] <- paste0("prof", len)
  } else {
    plan <- MultiscaleDTM::Qfit(DEM, metrics = "planc", w = k, na.rm = T)
    in_rast <- c(in_rast, plan)
```

```
      names(in_rast)[length(in_rast)] <- paste0("plan", len)
    }
  } else if("prof" %in% elev_dev) {
    prof <- MultiscaleDTM::Qfit(DEM, metrics = "profc", w = k, na.rm = T)
    in_rast <- c(in_rast, prof)

    names(in_rast)[length(in_rast)] <- paste0("prof", len)
  }

  if("dev" %in% elev_dev) {
    dev <- (DEM - focal(DEM, w = k, fun = "mean", na.rm = T, na.policy = "omit")) / focal(DEM,
    in_rast <- c(in_rast, rast_dev)

    names(in_rast)[length(in_rast)] <- paste0("dev", len)
  }

  if("twi" %in% elev_dev) {
    topidx <- topmodel::topidx(terra::as.matrix(DEM), res = terra::res(DEM)[1])
    a <- terra::setValues(DEM, topidx$area)
    twi <- a / tan(terra::terrain(DEM, unit = "radians"))
    terra::values(twi) <- ifelse(terra::values(twi) < 0, 0, terra::values(twi))
    twi <- terra::focal(twi, w = k, mean, na.rm = T, na.policy = "omit")

    in_rast <- c(in_rast, twi)

    names(in_rast)[length(in_rast)] <- paste0("twi", len)
  }

  # Exports the surface metrics
  if(export) {
    for(i in 1:length(in_rast)) {
      writeRaster(in_rast[[i]],
                  filename = paste0(names(in_rast[i]), len, ".tif"))
    }
  }
  return(in_rast)
}
```

build_model

```
build_model <- function(in_rasts, poly_inputs = list(), train, ref_raster,
                        model_type = "forest", model_params = list(ntree = 200),
                        class_field_name = "class") {

  # Checking if input rasters are file names, then load them in
  if(is.character(in_rasts[1])) {
    temp_rast <- rep(list(), length(in_rasts))
    for(i in 1:length(in_rasts)) {
      temp_rast[[i]] <- terra::rast(in_rasts[i])
    }
    names(temp_rast) <- in_rasts
    in_rasts <- temp_rast
  }

  # Checks if there are any polygon inputs
  if(length(poly_inputs) > 0) {

    # Checking to see the polygon inputs are filenames
    if(is.character(poly_inputs[[1]])) {
      temp_poly <- rep(list(), length(poly_inputs))
      for(i in 1:length(poly_inputs)) {
        if(!file.exists(poly_inputs[[i]])) {
          stop(paste0("Cannot find poly input file:", poly_inputs[i]))
        }
        temp_poly[[i]] <- terra::vect(poly_inputs[i])
      }
      names(temp_poly) <- poly_inputs
      poly_inputs <- temp_poly
    }

    # Rasterize polygon inputs
    for(i in 1:length(poly_inputs)) {
```

```
    vr_name <- names(poly_inputs)[i]
    temp_rast <- terra::rasterize(poly_inputs[i], ref_raster, field = vr_name)
    in_rasts <- c(in_rasts, temp_rast)
  }
}

# Ensure that all inputs are covering the same area
print("Formatting inputs")
for(i in 1:length(in_rasts)) {
  in_rasts[[i]] <- terra::project(in_rasts[[i]], ref_raster)
  in_rasts[[i]] <- terra::crop(in_rasts[[i]], ref_raster)
}


# Set up training data
print("Setting up training data")
train <- terra::project(train, ref_raster)
df_train <- data.frame(class = factor(as.vector(unlist(train[[class_field_name]]))))
for(i in 1:length(in_rasts)) {
  vals <- terra::extract(in_rasts[[i]], train, ID = F)
  df_train <- cbind(df_train, vals)
}
df_train <- na.omit(df_train)
colnames(df_train) <- c("class", names(in_rasts))

# Build the model
print("Building model")
if(model_type == "forest"){
  mod <- randomForest::randomForest(class ~ ., data = df_train,
                                    ntree = model_params$ntree)
} else if (model_type == "tree") {
  mod <- randomForest::randomForest(class ~ ., data = df_train, ntree = 1)
} else if(model_type == "glm") {
  if(length(levels(df_train$class)) > 2) {
    mod <- nnet::multinom(class ~ ., data = df_train)
  } else {
    mod <- glm(class ~ ., data = df_train, family = "binomial")
  }
} else if(model_type == "knn") {
  mod <- caret::knn3(formula = class ~ ., data = df_train, k = model_params$k)
} else {
  stop("Incorrect model type")
}
```

```
  print("Done!")
  return(mod)
}
```

run_model

```
run_model <- function(mod, in_rasts = list(), poly_inputs = list(), ref_raster,
                      model_type = "forest", class_rast = FALSE,
                      export = FALSE) {

  # Checking if inputs are file names, then load them in
  if(is.character(in_rasts[1])) {
    temp_rast <- rep(list(), length(in_rasts))
    for(i in 1:length(in_rasts)) {
      temp_rast[[i]] <- terra::rast(in_rasts[i])
    }
    names(temp_rast) <- in_rasts
    in_rasts <- temp_rast
  }

  if(length(poly_inputs) > 0) {
    if(is.character(poly_inputs[[1]])) {
      temp_poly <- rep(list(), length(poly_inputs))
      for(i in 1:length(poly_inputs)) {
        if(!file.exists(poly_inputs[[i]])) {
          stop(paste0("Cannot find poly input file:", poly_inputs[i]))
        }
        temp_poly[[i]] <- terra::vect(poly_inputs[i])
      }
      names(temp_poly) <- poly_inputs
      poly_inputs <- temp_poly
    }
    for(i in 1:length(poly_inputs)) {
      vr_name <- names(poly_inputs)[i]
      temp_rast <- terra::rasterize(poly_inputs[i], ref_raster, field = vr_name)
      in_rasts <- c(in_rasts, temp_rast)
    }
  }
```

```
# Ensure that all inputs are covering the same area
print("Formatting inputs")
for(i in 1:length(in_rasts)) {
  in_rasts[[i]] <- terra::project(in_rasts[[i]], ref_raster)
  in_rasts[[i]] <- terra::crop(in_rasts[[i]], ref_raster)
}

# Stacks the rasters on top of each other to create one raster
print("Stacking rasters")
input_raster <- in_rasts[[1]]
if(length(in_rasts) > 1) {
  for(i in 2:length(in_rasts)) {
    input_raster <- c(input_raster, in_rasts[[i]])
  }
}
names(input_raster) <- names(in_rasts)

# Run the model
print("Running model")

if(class_rast) {
  if(isTRUE(mod$call[[1]] == "glm")) {
    output <- terra::predict(input_raster, mod, na.rm = T,
                             type = "response")
    vals <- terra::values(output)
    vals <- ifelse(vals > 0.5, "WET", "UPL")
    terra::values(output) <- vals

  } else {
    output <- terra::predict(input_raster, mod, na.rm = T)
  }

} else {
  if(isTRUE(mod$call[[1]] == "glm")) {
    output <- terra::predict(input_raster, mod, na.rm = T,
                             type = "response")
  } else {
    output <- terra::predict(input_raster, mod, na.rm = T, type = "prob")
  }
}

if(export) {
  for(i in 1:length(output)) {
    file_name <- paste0(names(input_raster)[i], "prob.tif")
```

```
      terra::writeRaster(output[[i]], filename = file_name)
    }
  }

  print("Done!")
  return(output)
}
```

CV_err

```
CV_err <- function(in_rasts, poly_inputs = list(), ref_raster,
                   model_type = "forest", model_params = list(ntree = 200),
                   train, kfold= 5, class_field_name = "class") {

  # Checking if inputs are file names, then load them in
  if(is.character(in_rasts[1])) {
    temp_rast <- rep(list(), length(in_rasts))
    for(i in 1:length(in_rasts)) {
      temp_rast[[i]] <- terra::rast(in_rasts[i])
    }
    names(temp_rast) <- in_rasts
    in_rasts <- temp_rast
  }

  if(length(poly_inputs) > 0) {
    if(is.character(poly_inputs[[1]])) {
      temp_poly <- rep(list(), length(poly_inputs))
      for(i in 1:length(poly_inputs)) {
        if(!file.exists(poly_inputs[[i]])) {
          stop(paste0("Cannot find poly input file:", poly_inputs[i]))
        }
        temp_poly[[i]] <- terra::vect(poly_inputs[i])
      }
      names(temp_poly) <- poly_inputs
      poly_inputs <- temp_poly
    }
  }

  # Convert the polygons into rasters
  if(length(poly_inputs) > 0) {
    for(i in 1:length(poly_inputs)) {
      vr_name <- names(poly_inputs)[i]
```

```
    temp_rast <- terra::rasterize(poly_inputs[i], ref_raster, field = vr_name)
    in_rasts <- c(in_rasts, temp_rast)
  }
}

# Ensure that all inputs are covering the same area
for(i in 1:length(in_rasts)) {
  in_rasts[[i]] <- terra::project(in_rasts[[i]], ref_raster)
  in_rasts[[i]] <- terra::crop(in_rasts[[i]], ref_raster)
}

# Set up training data
train <- terra::project(train, ref_raster)
df_train <- data.frame(class = factor(as.vector(unlist(train[[class_field_name]]))))
for(i in 1:length(in_rasts)) {
  vals <- terra::extract(in_rasts[[i]], train, ID = F)
  df_train <- cbind(df_train, vals)
}
df_train <- na.omit(df_train)
colnames(df_train) <- c("class", names(in_rasts))

k <- kfold
test_err <- c()
index <- sample(k, nrow(df_train), replace = T)

for(i in 1:k) {
  train_df <- df_train[index != i,]
  test_df <- df_train[index == i,]
  y_test <- test_df$class

  if(model_type == "forest"){
    mod <- randomForest::randomForest(class ~ ., data = train_df,
                                      ntree = model_params$ntree)
  } else if (model_type == "tree") {
    mod <- randomForest::randomForest(class ~ ., data = train_df, ntree = 1)
  } else if(model_type == "glm") {
    if(length(levels(df_train$class)) > 2) {
      mod <- nnet::multinom(class ~ ., data = train_df)
    } else {
      mod <- glm(class ~ ., data = train_df, family = "binomial")
    }
  } else if(model_type == "knn") {
    mod <- caret::knn3(formula = class ~ ., data = train_df,
                       k = model_params$k)
```

```
    } else {
      stop("Incorrect model type")
    }

    if(model_type == "glm") {
      pred <- predict(mod, newdata = test_df, type = "response")
    } else {
      pred <- predict(mod, newdata = test_df)
    }
    test_err[i] <- mean(pred != y_test)
  }
  mean_err <- mean(test_err)
  ci_err <- round(100 * (mean_err + c(-1, 1)*qnorm(0.975)*sd(test_err)/k), 1)
  print(paste0("Test Error Estimate: ", round(mean_err * 100, 1), "%"))
  print(paste0("95% Confidence Interval: [", ci_err[1], ", ", ci_err[2], "]"))
}
```