

Continuous soil attribute modeling and mapping

Soil Security Laboratory

2017

The implementation of some of the most commonly used model functions used for digital soil mapping will be covered in this chapter. Before this is done however, some general concepts of model validation are covered.

1 Model validation

Essentially, whenever we train or calibrate a model, we can then generate some predictions. The question one needs to ask is how good are those predictions? Generally, we confront this question by comparing observed values with their corresponding predictions. Some of the more common “quality” measures are the root mean square error (RMSE), bias, coefficient of determination or commonly the R^2 value, and concordance. You will also find in the digital soil mapping and general statistical literature various other model assessment statistics. The RMSE is defined as:

$$RMSE = \sqrt{\left(\frac{\sum_{i=1}^n (obs_i - pred_i)^2}{n} \right)} \quad (1)$$

where obs is the observed soil property, $pred$ is the predicted soil property from a given model, and n is the number of observations i . Bias, also called the mean error of prediction and is defined as:

$$bias = \frac{\sum_{i=1}^n pred_i - obs_i}{n} \quad (2)$$

The R^2 is evaluated as the square of the sample correlation coefficient (Pearson’s) between the observations and their corresponding predictions. Pearson’s correlation coefficient r when applied to observed and predicted values is defined as:

$$r = \frac{\sum_{i=1}^n (obs_i - \overline{obs})(pred_i - \overline{pred})}{\sqrt{\sum_{i=1}^n (obs_i - \overline{obs})^2} \sqrt{\sum_{i=1}^n (pred_i - \overline{pred})^2}} \quad (3)$$

The R^2 measures the precision of the relationship (between observed and predicted). Concordance, or more formally — Lin’s concordance correlation coefficient (Lin, 1989), on the other hand is a single statistic that both

evaluates the accuracy and precision of the relationship. It is often referred to as the goodness of fit along a 45 degree line. Thus it is probably a more useful statistic than the R^2 alone. Concordance ρ_c is defined as:

$$\rho_c = \frac{2\rho\sigma_{pred}\sigma_{obs}}{\sigma_{pred}^2 + \sigma_{obs}^2 + (\mu_{pred} - \mu_{obs})^2} \quad (4)$$

where μ_{pred} and μ_{obs} are the means of the predicted and observed values respectively. σ_{pred}^2 and σ_{obs}^2 are the corresponding variances. ρ is the correlation coefficient between the predictions and observations.

1.1 Model goodness of fit

So lets fit a simple linear model. We will use the `soil.data` set used before in the introductory to R chapter. First load the data in. We then want to regress CEC content on clay (also be sure to remove as NAs).

```
library(ithir)
library(MASS)
data(USYD_soil1)
soil.data <- USYD_soil1
mod.data <- na.omit(soil.data[, c("clay", "CEC")])
mod.1 <- lm(CEC ~ clay, data = mod.data, y = TRUE, x = TRUE)
mod.1

##
## Call:
## lm(formula = CEC ~ clay, data = mod.data, x = TRUE, y = TRUE)
##
## Coefficients:
## (Intercept)      clay
##      3.7791      0.2053
```

You will recall that this is the same model that we fitted during the introduction to R chapter. What we now want to do is evaluate some of the model quality statistics that were just described. Conveniently, these are available in the `goof` function in the `ithir` package. We will use this function a lot during this chapter, so it might be useful to describe it. `goof` takes four inputs. A vector of **observed** values, a vector of **predicted** values, a logical choice of whether an output plot is required, and a character input of what type of output is required. There are number of possible goodness of fit statistics that can be requested, with only some being used frequently in digital soil mapping projects. Therefore setting the `type` parameter to ‘‘DSM’’ will output only the R^2 , RMSE, MSE, bias and concordance statistics as these are most most relevant to DSM. Additional statistics can be returned if ‘‘spec’’ is specified for the `type` parameter

```
goof(observed = mod.data$CEC, predicted = mod.1$fitted.values, type = "DSM")

##           R2 concordance      MSE      RMSE bias
## 1 0.4173582  0.5888521 14.11304 3.756733      0
```

You may wish to generate a plot in which case you would set the `plot.it` logical to `TRUE`.

This model `mod.1` does not seem to be too bad. On average the predictions are 3.75 cmol (+)/kg off the true value. The model on average is neither over- or under-predictive, but we can see that a few high CEC values are influencing the concordance and R^2 . This outcome may mean that there are other factors that influence the CEC, such as mineralogy type.

1.2 Model validation

Above we performed goodness of fit assessment of the `mod.1` model. Usually it is more appropriate however to validate a model using data that was not included for model fitting. Model validation has a few different forms. For completely unbiased assessments of model quality it is ideal to have an additional data set that is completely independent of the model data. It is recommended that a design based random sampling from the target area be conducted, to which there are a few types such as simple random sampling and stratified simple random sampling. Further information regarding sampling, sampling designs, their formulation and the relative advantages and constraints of each are described in de Gruijter et al. (2006). Usually from an operational perspective it is difficult to arrange the additional costs of organising and implementing some sort of probability sampling for determining unbiased model quality assessment. The alternative is to perform some sort of data sub-setting, such that with a data set we split it into a set for model calibration and another set for validation. This type of procedure can take different forms: the two main ones being random-hold back and leave-one-out-cross-validation (LOCV). Random-hold back (or sometimes k-fold validation) is where we may sample a data set of some pre-determined proportion (say 70%) for which is used for model calibration. We then validate the model using the other 30% of the data. For k-fold validation we divide the data set into equal sized partitions or folds, with all but one of the folds being used for the model calibration, the remaining fold is used for validation. We could repeat this k-fold process a number of times, each time using a different random sample from the data set for model calibration and validation. This allows one to efficiently derive distributions of the validation statistics as a means of assessing the stability and sensitivity of the models and parameters.

LOCV involves a little more computation such that if we had n number of data, we would subset $n-1$ of these data, and fit a model. Using this model we would make a prediction for the single data that was left out of the model (and save the residual). This is repeated for all n . LOCV would be undertaken when there are very few data to work with. When we can sacrifice a few data points, the random-hold back or k-fold cross-validation procedure would be acceptable.

When we are validating trained models with some sort of data sub-setting mechanism, always keep in mind that the validation statistics will be biased. As Brus et al. (2011) explains, the sampling from the target mapping area to

be used for DSM is more often than not from legacy soil survey, to which would not have been based on a probability sampling design. Therefore, that sample will be biased i.e not a true representation of the total population. Even though we may randomly select observations from the legacy soil survey sites, those validation points do not become a probability sample of the target area, and consequently will only provide biased estimates of model quality. Thus an independent probability sample is required. Further ideas on the statistical validation of models can be found in Hastie et al. (2001).

So lets implement some of the validation techniques in R. We will use the same data as before i.e regressing CEC with clay content. First we will do the random-back validation using 70% of the data for calibration. A random sample of the data will be performed using the `sample` function.

```
set.seed(123)
training <- sample(nrow(mod.data), 0.7 * nrow(mod.data))
training
```

##	[1]	42	115	59	127	134	7	74	125	77	63	131	62	91	138	14	118	32
##	[18]	6	146	122	113	87	80	123	124	86	66	71	35	18	112	104	79	90
##	[35]	3	54	84	24	136	25	16	44	105	38	106	15	109	47	27	110	5
##	[52]	43	76	12	52	19	93	68	114	33	58	9	139	23	67	37	65	143
##	[69]	135	34	121	48	53	1	108	102	98	95	100	8	17	145	70	50	141
##	[86]	64	60	140	92	10	82	36	142	72	120	57	40	96	83	107	28	101

These values correspond to row numbers which will correspond to the row which we will use for the calibration data. We subset these rows out of `mod.data` and fit a new linear model.

```
mod.rh <- lm(CEC ~ clay, data = mod.data[training, ], y = TRUE, x = TRUE)
```

So lets evaluate the calibration model with `goof`:

```
goof(predicted = mod.rh$fitted.values, observed = mod.data$CEC[training])
```

##		R2	concordance	MSE	RMSE	bias
## 1	0.4457907	0.6158071	12.31952	3.509917	0	

But we are more interested in how this model performs when we use the validation data. Here we use the `predict` function to predict upon this data.

```
mod.rh.V <- predict(mod.rh, mod.data[-training, ])
goof(predicted = mod.rh.V, observed = mod.data$CEC[-training])
```

##		R2	concordance	MSE	RMSE	bias
## 1	0.3591283	0.5208349	18.35828	4.284656	-0.5355242	

So the model is not as good as we first imagined. When we validate a model with an external data set, it is quite normal that the model will not perform nearly as well as when using calibration data. Set the `plot.it` parameter to `TRUE` and re-run the script above and you will see a plot like Figure 1.

In fact the `mod.rh` model does not appear to perform too bad after all. A few of the high observed values contribute greatly to the validation diagnostics. A

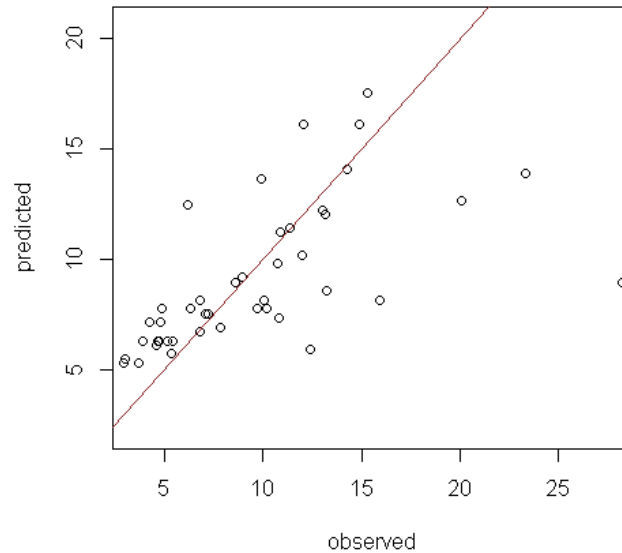


Figure 1: Observed vs. predicted plot of CEC model (validation data set) with line of concordance (red line).

couple of methods are available to assess the sensitivity of these results. The first is to remove what could potentially be outliers from the data. The second is to perform a sensitivity analysis such as bootstrapping where we iterate the data sub-setting procedure and evaluate the validation statistics each time to get a sense how much they vary.

At the most basic level, LOCV involves the use of a looping function or `for` loop. We have not really covered `for` loops yet, but essentially they can be used to great effect when we want to perform a particular analysis over-and-over. For example with LOCV, for each iteration or loop we take a subset of $n-1$ rows and fit a model to them, then use that model to predict for the point left out of the calibration. Computationally it will look something like this:

```
looPred <- numeric(nrow(mod.data))
for (i in 1:nrow(mod.data)) {
  looModel <- lm(CEC ~ clay, data = mod.data[-i, ], y = TRUE, x = TRUE)
  looPred[i] <- predict(looModel, newdata = mod.data[i, ])
}
```

The `i` here is the counter, so for each loop it increases by 1 until we get to the end of the data set. As you can see, we can index the `mod.data` using the `i`, meaning that for each loop we will have selected a different calibration set. On each loop, the prediction on the point left out of the calibration is made onto the corresponding row position of the `looPred` object. Again we can

assess the performance of the LOCV using the `goof` function.

```
goof(predicted = looPred, observed = mod.data$CEC)

##           R2 concordance           MSE           RMSE           bias
## 1 0.4025255 0.5790589 14.47653 3.804804 0.005758669
```

LOCV will generally be less sensitive to outliers, so overall these external validation results are not too different to those when we performed the internal validation. Make a plot of the LOCV results to visually compare against the internal validation.

2 Multiple linear regression

Multiple linear regression (MLR) is where we regress a target variable against more than one covariate. In terms of soil spatial prediction functions, MLR is a least-squares model whereby we want to predict a continuous soil variable from a suite of covariates. There are a couple of ways to go about this. We could just put everything (all the covariates) in the model and then fit it (estimate the model parameters). We could perform a stepwise regression model where we only enter variables that are statistically significant, based on some selection criteria. Alternatively we could fit what could be termed, an “expert” model, such that based on some pre-determined knowledge of the soil variable we are trying to model, we include covariates that best describe this knowledge. In some ways this is a biased model because we really don’t know everything about (the spatial characteristics) the soil property under investigation. Yet in many situations it is better to rely on expert knowledge that is gained in the field as opposed to some other form.

So lets firstly get the data organized. Recall from before in the data preparatory exercises that we were working with the soil point data and environmental covariates for the Hunter Valley area. These data are stored in the `HV_subsoilpH` and `hunterCovariates_sub` objects from the `ithir` package. For the succession of models to be used, we will concentrate on modelling and mapping the soil pH for the 60-100cm depth interval. To refresh, lets load the data in, then intersect the data with the available covariates.

```
library(ithir)
library(raster)
library(rgdal)
library(sp)

# point data
data(HV_subsoilpH)

# Start afresh round pH data to 2 decimal places
HV_subsoilpH$pH60_100cm <- round(HV_subsoilpH$pH60_100cm, 2)

# remove already intersected data
```

```

HV_subsoilpH <- HV_subsoilpH[, 1:3]

# add an id column
HV_subsoilpH$id <- seq(1, nrow(HV_subsoilpH), by = 1)

# re-arrange order of columns
HV_subsoilpH <- HV_subsoilpH[, c(4, 1, 2, 3)]

# Change names of coordinate columns
names(HV_subsoilpH)[2:3] <- c("x", "y")

# grids (covariate raster)
data(hunterCovariates_sub)

Perform the covariate intersection.
coordinates(HV_subsoilpH) <- ~x + y

# extract
DSM_data <- extract(hunterCovariates_sub, HV_subsoilpH, sp = 1, method = "simple")
DSM_data <- as.data.frame(DSM_data)
str(DSM_data)

## 'data.frame': 506 obs. of  15 variables:
##  $ id                : num  1 2 3 4 5 6 7 8 9 10 ...
##  $ x                 : num  340386 340345 340559 340483 340734 ...
##  $ y                 : num  6368690 6368491 6369168 6368740 6368964 ...
##  $ pH60_100cm        : num  4.47 5.42 6.26 8.03 8.86 7.28 4.95 5.61 5.39 3.44 ...
##  $ Terrain_Ruggedness_Index: num  1.34 1.42 1.64 1.04 1.27 ...
##  $ AACN              : num  1.619 0.281 2.301 1.74 3.114 ...
##  $ Landsat_Band1      : num  57 47 59 52 62 53 47 52 53 63 ...
##  $ Elevation          : num  103.1 103.7 99.9 101.9 99.8 ...
##  $ Hillshading        : num  1.849 1.428 0.934 1.517 1.652 ...
##  $ Light_insolation    : num  1689 1701 1722 1688 1735 ...
##  $ Mid_Slope_Positon  : num  0.876 0.914 0.844 0.848 0.833 ...
##  $ MRVBF             : num  3.85 3.31 3.66 3.92 3.89 ...
##  $ NDVI              : num  -0.143 -0.386 -0.197 -0.14 -0.15 ...
##  $ TWI               : num  17.5 18.2 18.8 18 17.8 ...
##  $ Slope             : num  1.79 1.42 1.01 1.49 1.83 ...

```

Often it is handy to check to see whether there are missing values both in the target variable and of the covariates. It is possible that a point location does not fit within the extent of the available covariates. In these cases the data should be excluded. A quick way to assess whether there are missing or NA values in the data is to use the `complete.cases` function.

```

which(!complete.cases(DSM_data))

## integer(0)

DSM_data <- DSM_data[complete.cases(DSM_data), ]

```

There do not appear to be any missing data as indicated by the `integer(0)` output above i.e there are zero rows with missing information.

With the soil point data prepared, lets fit a model with everything in it (all covariates) to get an idea of how to parametise the MLR models in R. Remember the soil variable we are making a model for is soil pH for the 60-100cm depth interval.

```
hv.MLR.Full <- lm(pH60_100cm ~ +Terrain_Ruggedness_Index + AACN + Landsat_Band1 +
  Elevation + Hillshading + Light_insolation + Mid_Slope_Positon + MRVBF +
  NDVI + TWI + Slope, data = DSM_data)
summary(hv.MLR.Full)

##
## Call:
## lm(formula = pH60_100cm ~ +Terrain_Ruggedness_Index + AACN +
##     Landsat_Band1 + Elevation + Hillshading + Light_insolation +
##     Mid_Slope_Positon + MRVBF + NDVI + TWI + Slope, data = DSM_data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -3.2380 -0.7843 -0.1225  0.7057  3.4641
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)    5.372452    2.147673   2.502 0.012689 *
## Terrain_Ruggedness_Index 0.075084    0.054893   1.368 0.171991
## AACN           0.034747    0.007241   4.798 2.12e-06 ***
## Landsat_Band1  -0.037712    0.009355  -4.031 6.42e-05 ***
## Elevation      -0.013535    0.005550  -2.439 0.015079 *
## Hillshading     0.152819    0.053655   2.848 0.004580 **
## Light_insolation 0.001329    0.001178   1.127 0.260081
## Mid_Slope_Positon 0.928823    0.268625   3.458 0.000592 ***
## MRVBF          0.324041    0.084942   3.815 0.000154 ***
## NDVI           4.982413    0.887322   5.615 3.28e-08 ***
## TWI            0.085150    0.045976   1.852 0.064615 .
## Slope          -0.102262    0.062391  -1.639 0.101838
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.178 on 494 degrees of freedom
## Multiple R-squared:  0.2501, Adjusted R-squared:  0.2334
## F-statistic: 14.97 on 11 and 494 DF, p-value: < 2.2e-16
```

From the summary output above, it seems a few of the covariates are significant in describing the spatial variation of the target variable. To determine the most parsimonious model we could perform a stepwise regression using the `step` function. With this function we can also specify what direction we want step wise algorithm to proceed.

```
hv.MLR.Step <- step(hv.MLR.Full, trace = 0, direction = "both")
summary(hv.MLR.Step)
```

```
##
## Call:
## lm(formula = pH60_100cm ~ AACN + Landsat_Band1 + Elevation +
##      Hillshading + Mid_Slope_Positon + MRVBF + NDVI + TWI, data = DSM_data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -3.1202 -0.8055 -0.1286  0.7443  3.4407
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)    7.449369    0.930288   8.008 8.36e-15 ***
## AACN           0.037413    0.006986   5.356 1.31e-07 ***
## Landsat_Band1  -0.037795    0.009134  -4.138 4.12e-05 ***
## Elevation      -0.012042    0.005299  -2.273 0.023481 *
## Hillshading     0.089275    0.018576   4.806 2.04e-06 ***
## Mid_Slope_Positon 0.982066    0.263538   3.726 0.000216 ***
## MRVBF           0.307179    0.083361   3.685 0.000254 ***
## NDVI            5.111642    0.882036   5.795 1.21e-08 ***
## TWI             0.092169    0.045241   2.037 0.042149 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.179 on 497 degrees of freedom
## Multiple R-squared:  0.245, Adjusted R-squared:  0.2329
## F-statistic: 20.16 on 8 and 497 DF,  p-value: < 2.2e-16
```

Comparing the outputs of both the full and stepwise MLR models, there is very little difference in the model diagnostics such as the R^2 . Both models explain about 25% of variation of the target variable. Obviously the “full” model is more complex as it has more parameters than the “step” model. If we apply Occam’s Razor, the “step” model is preferable.

As described earlier, it is more acceptable to test the performance of a model based upon an external validation. Lets fit a new model using the covariates selected in the step wise regression to a random subset of the available data. We will sample 70% of the available rows for the model calibration data set.

```
set.seed(123)
training <- sample(nrow(DSM_data), 0.7 * nrow(DSM_data))
hv.MLR.rh <- lm(pH60_100cm ~ AACN + Landsat_Band1 + Elevation + Hillshading +
  Mid_Slope_Positon + MRVBF + NDVI + TWI, data = DSM_data[training, ])
# calibration predictions
hv.pred.rhC <- predict(hv.MLR.rh, DSM_data[training, ])

# validation predictions
hv.pred.rhV <- predict(hv.MLR.rh, DSM_data[-training, ])
```

Now we can evaluate the test statistics of the calibration model using the `goof` function.

```

# calibration
goof(observed = DSM_data$pH60_100cm[training], predicted = hv.pred.rhC)

##           R2 concordance      MSE      RMSE      bias
## 1 0.2437432  0.3936011 1.370124 1.170523 -6.217249e-15

# validation
goof(observed = DSM_data$pH60_100cm[-training], predicted = hv.pred.rhV)

##           R2 concordance      MSE      RMSE      bias
## 1 0.2248642  0.3798786 1.376643 1.173304 0.05121269

```

In this situation the calibration model does not appear to be over fitting because the test statistics for the validation are similar to those of the calibration data. While this is a good result, the prediction model performs only moderately well by the fact there is a noticeable deviation between observations and corresponding model predictions. Examining other candidate models is a way to try to improve upon this results.

2.1 Applying the model spatially

From a soil mapping perspective the important question to ask is: What does the map look like that results from a particular model? In practice this can be answered by applying the model parameters to the grids of the covariates that were used in the model. There are a few options on how to do this.

2.1.1 Covariate table

The traditional has been to collate a grid table where there would be two columns for the coordinates followed by other columns for each of the available covariates that were sourced. This was seen as an efficient way to organize all the covariate data as it ensured that a common grid was used which also meant that all the covariates are of the same scale in terms of resolution and extent. We can simulate the covariate table approach using the `hunterCovariates_sub` object as below.

```

data(hunterCovariates_sub)
tempD <- data.frame(cellNos = seq(1:ncell(hunterCovariates_sub)))
vals <- as.data.frame(getValues(hunterCovariates_sub))
tempD <- cbind(tempD, vals)
tempD <- tempD[complete.cases(tempD), ]
cellNos <- c(tempD$cellNos)
gXY <- data.frame(xyFromCell(hunterCovariates_sub, cellNos, spatial = FALSE))
tempD <- cbind(gXY, tempD)
str(tempD)

## 'data.frame': 33252 obs. of  14 variables:
##  $ x                : num  340935 340960 340985 341010 341035 ...
##  $ y                : num  6370416 6370416 6370416 6370416 6370416 ...
##  $ cellNos           : int   101 102 103 104 105 106 107 108 109 110 ...

```

```
## $ Terrain_Ruggedness_Index: num 0.745 0.632 0.535 0.472 0.486 ...
## $ AACN : num 9.78 9.86 10.04 10.27 10.53 ...
## $ Landsat_Band1 : num 68 63 59 62 56 54 59 62 54 56 ...
## $ Elevation : num 103 103 102 102 102 ...
## $ Hillshading : num 0.94 0.572 0.491 0.515 0.568 ...
## $ Light_insolation : num 1712 1706 1701 1699 1697 ...
## $ Mid_Slope_Positon : num 0.389 0.387 0.386 0.386 0.386 ...
## $ MRVBF : num 0.376 0.765 1.092 1.54 1.625 ...
## $ NDVI : num -0.178 -0.18 -0.164 -0.169 -0.172 ...
## $ TWI : num 16.9 17.2 17.2 17.2 17.2 ...
## $ Slope : num 0.968 0.588 0.503 0.527 0.581 ...
```

The result shown above is that the covariate table contains 33252 rows and has 14 variables. It is always necessary to have the coordinate columns, but some saving of memory could be earned if only the required covariates are appended to the table. It will quickly become obvious however that the covariate table approach could be limiting when mapping extents get very large or the grid resolution of mapping becomes more fine-grained, or both.

With the covariate table arranged it then becomes a matter of using the `MLR predict` function.

```
map.MLR <- predict(hv.MLR.rh, newdata = tempD)
map.MLR <- cbind(data.frame(tempD[, c("x", "y")])), map.MLR)
```

Now we can rasterise the predictions for mapping (Figure 2) and grid export. In the example below we set the CRS to WGS84 Zone 56 before exporting the raster file out as a Geotiff file.

```
map.MLR.r <- rasterFromXYZ(as.data.frame(map.MLR[, 1:3]))
plot(map.MLR.r, main = "MLR predicted soil pH (60-100cm)")

# set the projection
crs(map.MLR.r) <- "+proj=utm +zone=56 + south +
ellps=WGS84 +datum=WGS84 +units=m +no_defs"

writeRaster(map.MLR.r, "soilpH_60_100_MLR.tif", format = "GTiff",
datatype = "FLT4S", overwrite = TRUE)
# check working directory for presence of raster
```

Some of the parameters used within the `writeRaster` function that are worth noting include: `format`, which is the raster format that we want to write to. Here “GTiff” is being specified — use the `writeFormats` function to look at what other raster formats can be used. the parameter `datatype` is specified as “FLT4S” which indicates that a 4 byte, signed floating point values are to be written to file. Look at the function `dataTypes` to look at other alternatives, for example for categorical data where we may be interested in logical or integer values.

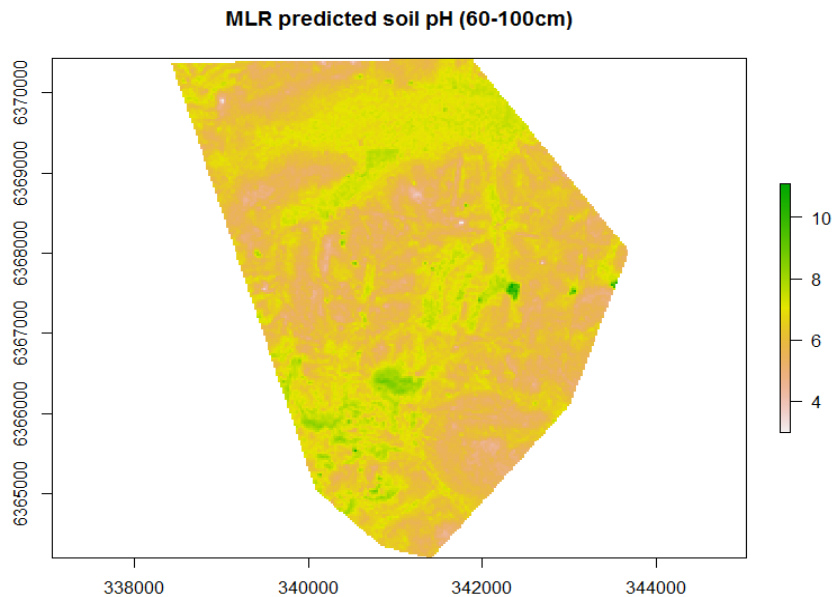


Figure 2: MLR predicted soil pH 60-100cm across the Hunter Valley

2.1.2 Raster predictions

Probably a more efficient way of applying the fitted model is to apply it directly to the rasters themselves. This avoids the step of arranging all covariates into table format. If multiple rasters are being used, it is necessary to have them arranged as a **rasterStack** object. This is useful as it also ensures all the rasters are of the same extent and resolution. Here we can use the **raster predict** function such as below using the **covStack** raster stack as input.

```
map.MLR.r1 <- predict(hunterCovariates_sub, hv.MLR.rh, "soilpH_60_100_MLR.tif",
  format = "GTiff", datatype = "FLT4S", overwrite = TRUE)
# check working directory for presence of raster
```

The prediction function is quite versatile. For example we can also map the standard error of prediction or the confidence interval or the prediction interval even. The script below is an example of creating maps of the 90% prediction intervals for the **hv.MLR.rh** model. We need to explicitly create a function called in this case **predfun** which will direct the raster **predict** function to output the predictions plus the upper and lower prediction limits. In the **predict** function we insert **predfun** for the **fun** parameter and control the output by changing the **index** value to either 1, 2, or 3 to request either the prediction, lower limit, upper limit respectively. Setting the **level** parameter to 0.90 indicates that we want to return the 90% prediction interval. The resulting plots are shown in Figure 3.

```

par(mfrow = c(3, 1))
predfun <- function(model, data) {
  v <- predict(model, data, interval = "prediction", level = 0.9)
}

map.MLR.r.low <- predict(hunterCovariates_sub, hv.MLR.rh,
  "soilPh_60_100_MLR_low.tif", fun = predfun, index = 2, format = "GTiff",
  datatype = "FLT4S", overwrite = TRUE)
plot(map.MLR.r.low, main = "MLR predicted soil pH (60-100cm) lower limit")

map.MLR.r.pred <- predict(hunterCovariates_sub, hv.MLR.rh,
  "soilPh_60_100_MLR_pred.tif", fun = predfun, index = 1, format = "GTiff",
  datatype = "FLT4S", overwrite = TRUE)
plot(map.MLR.r.pred, main = "MLR predicted soil pH (60-100cm)")

map.MLR.r.up <- predict(hunterCovariates_sub, hv.MLR.rh,
  "soilPh_60_100_MLR_up.tif", fun = predfun, index = 3, format = "GTiff",
  datatype = "FLT4S", overwrite = TRUE)
plot(map.MLR.r.up, main = "MLR predicted soil pH (60-100cm) upper limit")
# check working directory for presence of rasters

```

2.1.3 Directly to rasters using parallel processing

An extension of using the raster `predict` function is to apply the model again to the rasters, but to do it across multiple computer nodes. This is akin to breaking a job up into smaller pieces then processing the jobs in parallel rather than sequentially. The parallel component here is that the smaller pieces are passed to more than 1 compute nodes. Most desktop computers these days can have up to 8 compute nodes which can result in some excellent gains in efficiency when applying models across massive extents and or at fine resolutions. The **raster** package has some built in dependencies with other R packages that facilitate parallel processing options. For example the **raster** package ports with the **parallel** package for setting up and controlling the compute node processes. The script below is an example of using 4 compute nodes to apply the `hv.MLR.rh` model to the `hunterCovariates_sub` raster stack.

```

library(parallel)
beginCluster(4)
clusterMLR.pred <- clusterR(hunterCovariates_sub, predict,
  args = list(hv.MLR.rh), filename = "soilpH_60_100_MLR_pred.tif",
  format = "GTiff", progress = FALSE, overwrite = T)
endCluster()

```

To set up the compute nodes, you use the `beginCluster` function and inside it, specify how many compute nodes you want to use. If empty brackets are used, the function will use 100% of the compute resources. The `clusterR` function is the work horse function that then applies the model in parallel to the rasters. The parameters and subsequent options are similar to the raster

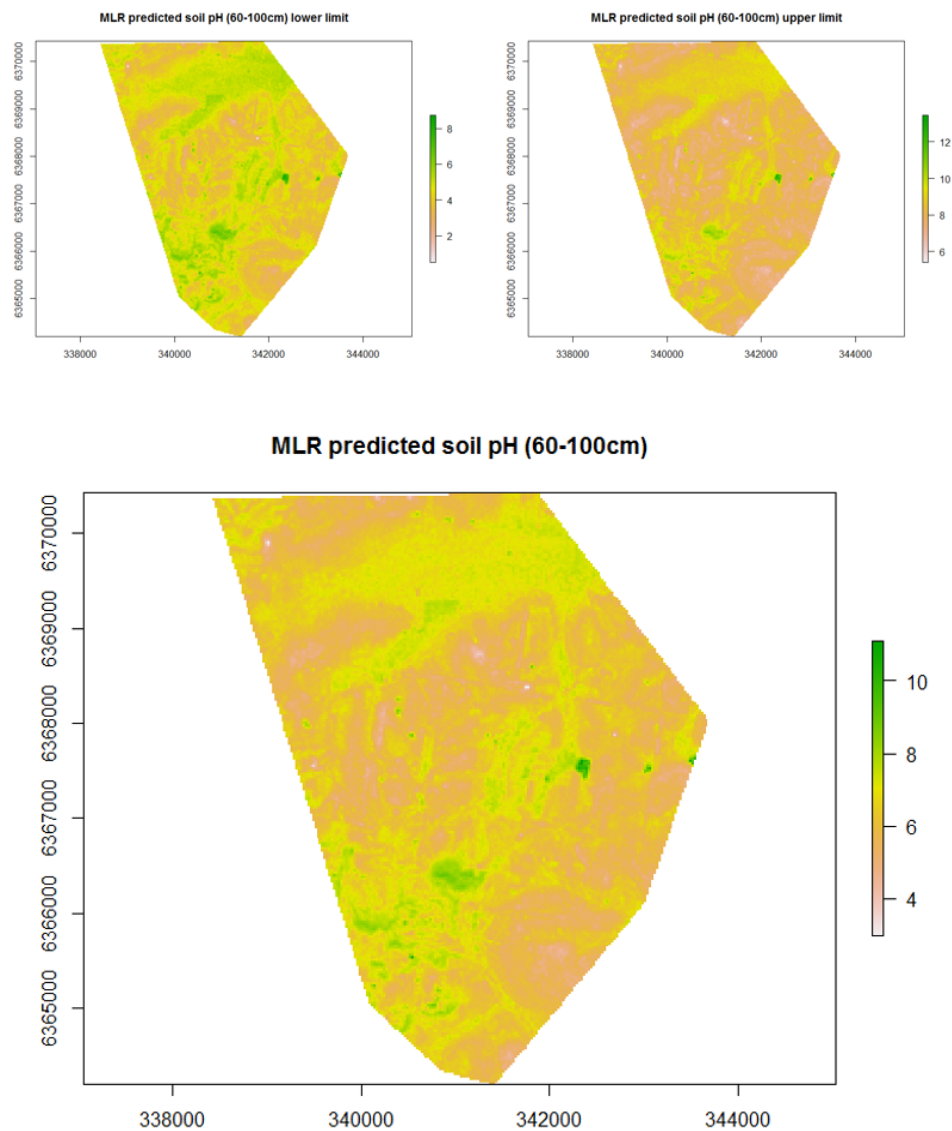


Figure 3: MLR predicted soil pH (60-100cm) across the Hunter Valley with associated lower and upper prediction limits.

`predict` function, although it would help to look at the help files on this function for more detailed explanations. It is always important after the prediction is completed to shutdown the nodes using the `endCluster` function.

The relative ease in setting up the parallel processing for our mapping needs has really opened up the potential for performing DSM using very large data

sets and rasters. Moreover, using the parallel processing together with the file pointing ability (that was discussed earlier) **raster** has made the possibility of big DSM a reality, and importantly- practicable.

3 Decision Trees

Linear regression is a global model, where there is a single predictive formula holding over the entire data space. With a linear model we therefore make some assumptions about how our target variable relates to the covariates. These may often hold, however, it is models that allow one the flexibility of modelling non-linearity that are increasingly popular in the DSM community. One of these model structures are classification and regression trees (CART). These models are a non-parametric decision tree learning techniques that produces either classification or regression trees. In this section we will concentrate on regression trees because our target variable is numeric i.e. a continuous variable. Later we will look at classification trees for categorical variables. Decision trees (either regression or classification) are formed by a collection of rules based on variables in the modeling data set:

- Rules based on variables' values are selected to get the best split to differentiate observations based on the dependent variable.
- Once a rule is selected and splits a node into two, the same process is applied to each subsequent node (i.e. it is a recursive procedure).
- Splitting stops when CART detects no further gain can be made, or some pre-set stopping rules are met. Alternatively, the data are split as much as possible and then the tree is later pruned.

Each branch of the tree ends in a terminal node. Each observation falls into one and exactly one terminal node, and each terminal node is uniquely defined by a set of rules. For a regression tree, the terminal node is a single value, or could be a regression model (which is the case for Cubist models which we will look at later). Implementation of regression trees in R is provided both through the **rpart** and **party** packages. We will use the **rpart** package and its **rpart** function. However, the **party** package through the **ctree** function offers more functionality, and implements the partitioning in a more statistical robust fashion. Both functions however can handle both continuous and categorical predictor variables.

Fitting a decision tree in R is quite similar to that for linear models:

```
library(rpart)
set.seed(123)
training <- sample(nrow(DSM_data), 0.7 * nrow(DSM_data))
hv.RT.Exp <- rpart(pH60_100cm ~ AACN + Landsat_Band1 + Elevation + Hillshading +
  Mid_Slope_Positon + MRVBF + NDVI + TWI,
data = DSM_data[training, ], control = rpart.control(minsplit = 50))
```

It is worthwhile to look at the help file for **rpart** particularly those aspects regarding the **rpart.control** parameters which control the **rpart** fit. Often it

is helpful to just play around with the parameters to get a sense of what does what. Here for the `minsplit` parameter within `rpart.control` we are specifying that we want at least 50 observations in a node in order for a split to be attempted.

Detailed results of the model fit can be provided via the `summary` and `printcp` functions.

```
summary(hv.RT.Exp)
```

The `summary` output provides detailed information of the data splitting as well as information as to the relative importance of the covariates.

```
printcp(hv.RT.Exp)
```

The `printcp` function provides the useful output of indicating which covariates were included in the final model. For the visually inclined, a plot of the tree assists a lot to interpret the model diagnostics and assessing the important covariates too(Figure 4).

```
plot(hv.RT.Exp)
```

```
text(hv.RT.Exp)
```

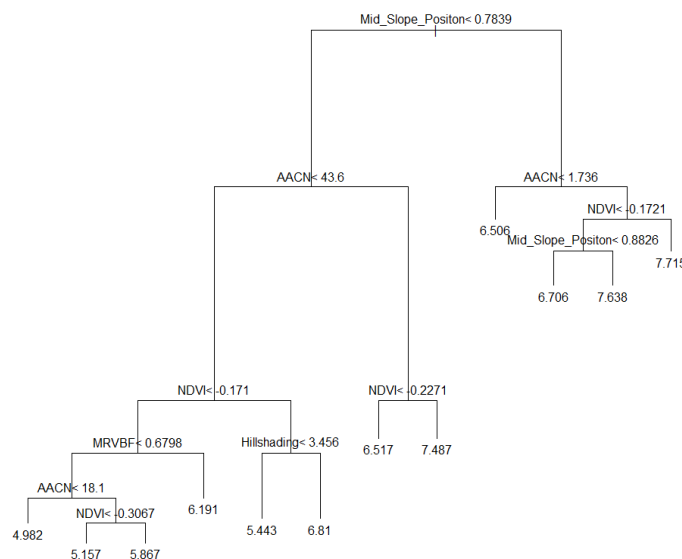


Figure 4: Decision tree of Hunter Valley soil pH (60-100cm).

As before, we can use the `goof` function to test the performance of the model fit both internally and externally.

```

# Internal validation
RT.pred.C <- predict(hv.RT.Exp, DSM_data[training, ])
goof(observed = DSM_data$PH60_100cm[training], predicted = RT.pred.C)

##           R2 concordance      MSE      RMSE      bias
## 1 0.3627782  0.5328417 1.154466 1.074461 -8.881784e-16

# External validation
RT.pred.V <- predict(hv.RT.Exp, DSM_data[-training, ])
goof(observed = DSM_data$PH60_100cm[-training], predicted = RT.pred.V)

##           R2 concordance      MSE      RMSE      bias
## 1 0.05587206  0.2131035 1.858148 1.363139 0.03135219

```

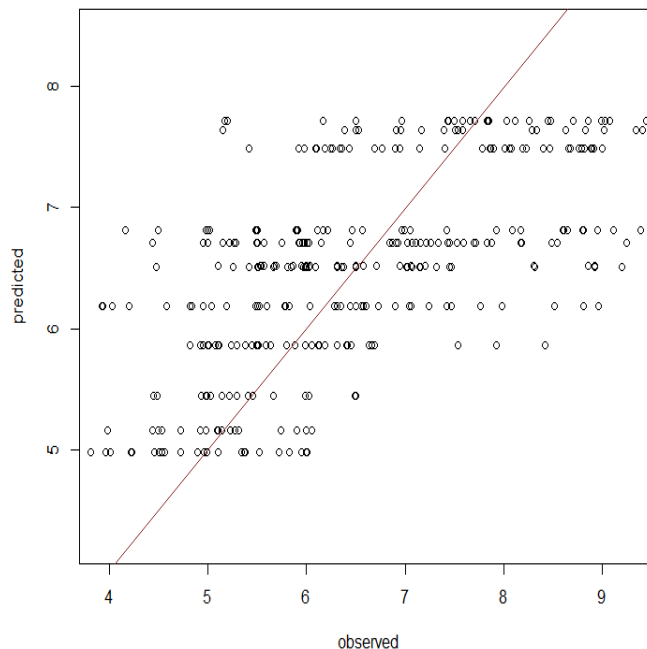


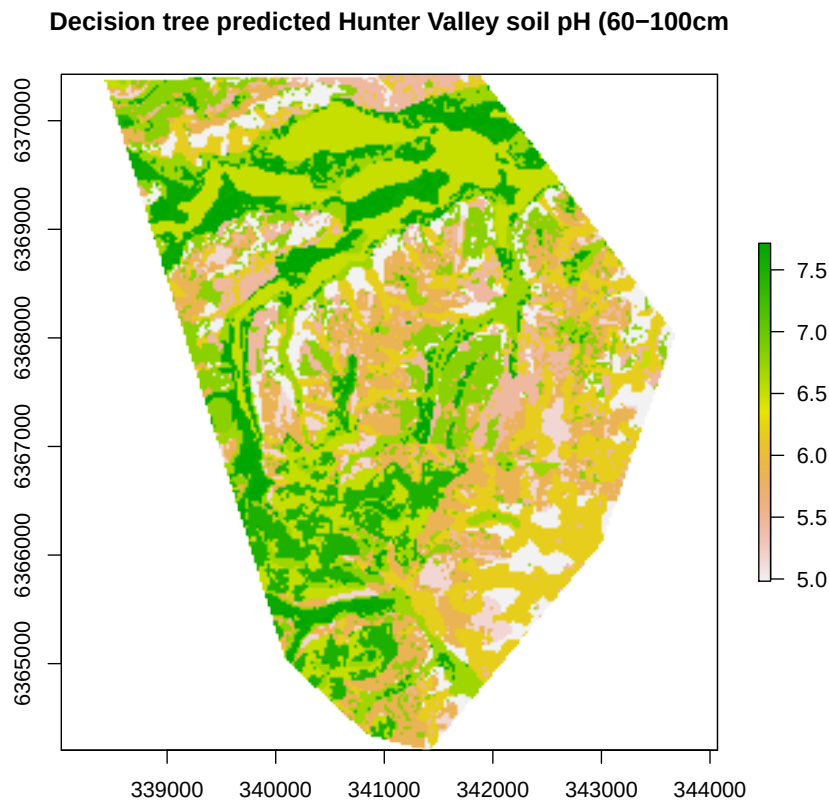
Figure 5: Decision tree xy-plot plot of predicted soil pH (60-100cm) (validation data set).

The decision tree model performance is not too dissimilar to the MLR model. Looking at the xy-plot from the external validation (Figure 5) and the decision tree (Figure 4), it becomes clear that a potential issue is apparent. This is: there are only a finite number of possible outcomes in terms of the predictions.

This finite property becomes obviously apparent once we make a map by applying the fitted model to the covariates (using the **raster** predict function and **hunterCovariates_sub** object). (Figure 6).

```
map.RT.r1 <- predict(hunterCovariates_sub, hv.RT.Exp, "soilpH_60_100_RT.tif",
  format = "GTiff", datatype = "FLT4S", overwrite = TRUE)

plot(map.RT.r1, main = "Decision tree predicted Hunter Valley soil pH (60-100cm)")
```



4 Cubist Models

The Cubist model is currently a very popular model structure used within the DSM community. Its popularity is due to its ability to “mine” non-linear relationships in data, but does not have the issues of finite predictions that occur for other decision and regression tree models. In similar vain to regression trees however, Cubist models also are a data partitioning algorithm. The Cubist model is based on the M5 algorithm of Quinlan (1992), and is implemented in the R Cubist package.

The Cubist model first partitions the data into subsets within which their characteristics are similar with respect to the target variable and the covariates. A series of rules (a decision tree structure may also be defined if

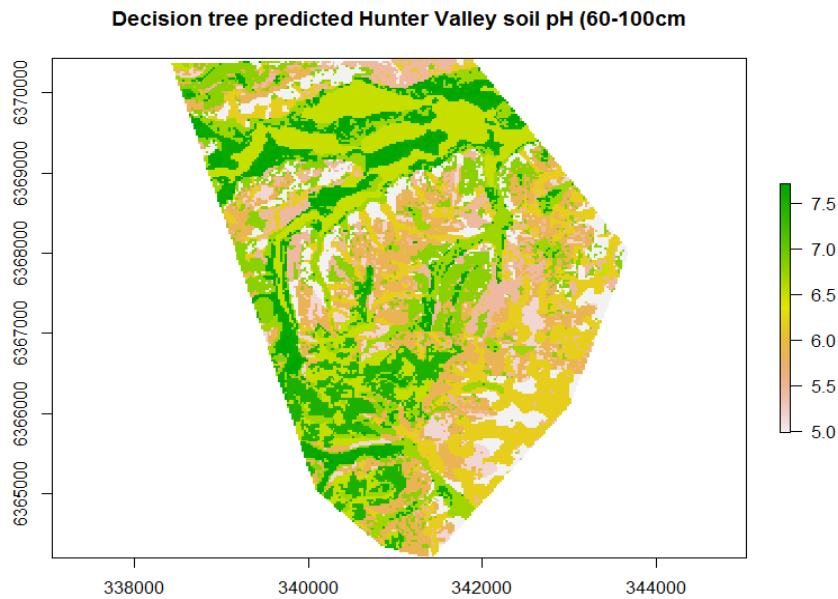


Figure 6: Decision tree predicted Hunter Valley soil pH (60-100cm).

requested) defines the partitions, and these rules are arranged in a hierarchy. Each rule takes the form:

```
if [condition is true]
then [regress]
else [apply the next rule]
```

The condition may be a simple one based on one covariate or, more often, it comprises a number of covariates. If a condition results in being true then the next step is the prediction of the soil property of interest by ordinary least-squares regression from the covariates within that partition. If the condition is not true then the rule defines the next node in the tree, and the sequence of *if*, *then*, *else* is repeated. The result is that the regression equations, though general in form, are local to the partitions and their errors smaller than they would otherwise be. More details of the Cubist model can be found in the `Cubist` help files or Quinlan (1992).

Luckily, fitting a Cubist model in `R` is not too difficult — although it will be useful to spend some time playing around with many of the controllable parameters the function has. In the example below we can control the number of potential rules that could potentially partition the data (note this limits the number of possible rules, and does not necessarily mean that those number of rules will actually be realised i.e. the outcome is internally optimised). We can also limit the extrapolation of the model predictions, which is a useful model constraint feature. These various control parameters plus others can be

adjusted within the `cubistControl` parameter. The `committees` parameter is specified as an integer of how many committee models (e.g., boosting iterations) are required. Here we just set it to 1, but naturally it is possible to perform some sort of sensitivity analysis when this committee model option is set to greater than 1. In terms of specifying the target variable and covariates, we do not define a formula as we did earlier for the MLR model. Rather we specify the columns explicitly — those that are the target variable (`x`), and those that are the covariates (`y`).

```
library(Cubist)
set.seed(875)
training <- sample(nrow(DSM_data), 0.7 * nrow(DSM_data))
mDat <- DSM_data[training, ]

# fit the model
hv.cub.Exp <- cubist(x = mDat[, c("AACN", "Landsat_Band1", "Elevation", "Hillshading",
                                "Mid_Slope_Positon", "MRVBF", "NDVI", "TWI")],
y = mDat$pH60_100cm,
cubistControl(rules = 100, extrapolation = 15), committees = 1)
```

The output generated from fitting a Cubist model can be retrieved using the `summary` function. This provides information about the conditions for each rule, the regression models for each rule, and information about the diagnostics of the model fit, plus the frequency of which the covariates were used as conditions and/or within a model.

```
summary(hv.cub.Exp)

##
## Call:
## cubist.default(x = mDat[, c("AACN", "Landsat_Band1",
## "NDVI", "TWI")], y = mDat$pH60_100cm, committees = 1, control
## = cubistControl(rules = 100, extrapolation = 15))
##
##
## Cubist [Release 2.07 GPL Edition] Tue Jan 10 13:12:13 2017
## -----
##
## Target attribute `outcome'
##
## Read 354 cases (9 attributes) from undefined.data
##
## Model:
##
## Rule 1: [148 cases, mean 6.175, range 3.92 to 9.74, est err 0.798]
##
## if
## Elevation > 99.47475
## MRVBF > 0.004846
## NDVI <= -0.192982
## then
```

```

## outcome = 4.895 + 0.56 MRVBF + 0.0224 AACN + 0.093 Hillshading
##           + 0.92 Mid_Slope_Positon + 1.5 NDVI - 0.014 Landsat_Band1
##           + 0.024 TWI - 0.0025 Elevation
##
## Rule 2: [148 cases, mean 6.500, range 3 to 9.41, est err 1.122]
##
##     if
## Elevation > 99.47475
## MRVBF > 0.004846
## NDVI > -0.192982
##     then
## outcome = 7.116 + 0.0398 AACN + 0.18 TWI - 0.074 Landsat_Band1
##           + 0.12 MRVBF + 1.6 NDVI + 0.025 Hillshading - 0.0028 Elevation
##           + 0.25 Mid_Slope_Positon
##
## Rule 3: [37 cases, mean 7.042, range 5 to 9.25, est err 0.893]
##
##     if
## Elevation <= 99.47475
##     then
## outcome = 1.488 + 0.0582 Elevation + 0.0081 AACN + 0.07 MRVBF + 1.1 NDVI
##           + 0.023 TWI + 0.017 Hillshading - 0.009 Landsat_Band1
##           + 0.17 Mid_Slope_Positon
##
## Rule 4: [21 cases, mean 7.878, range 6.26 to 9.45, est err 0.654]
##
##     if
## MRVBF <= 0.004846
##     then
## outcome = 8.122 + 422.73 MRVBF - 0.0118 AACN + 0.3 NDVI
##           - 0.0007 Elevation + 0.005 TWI + 0.004 Hillshading
##           - 0.002 Landsat_Band1
##
##
## Evaluation on training data (354 cases):
##
##     Average |error|           1.128
##     Relative |error|         0.99
##     Correlation coefficient    0.35
##
##
## Attribute usage:
##     Conds  Model
##
##     94%    100%    Elevation
##     90%    100%    MRVBF
##     84%    100%    NDVI
##           100%    AACN
##           100%    Landsat_Band1

```

```
##          100%    Hillshading
##          100%    TWI
##          94%     Mid_Slope_Positon
##
##
## Time: 0.0 secs
```

It appears the `hv.cub.Exp` model contains 4 rules in this case. The useful feature of the Cubist model is that it does not unnecessarily overfit and partition the data, although you do need to keep an eye on the number of observations within each ruleset. Lets see how well the model validates.

```
# Internal validation
Cubist.pred.C <- predict(hv.cub.Exp, newdata = DSM_data[training, ])
goof(observed = DSM_data$PH60_100cm[training], predicted = Cubist.pred.C)

##          R2 concordance      MSE      RMSE      bias
## 1 0.3227766  0.4884487 1.305012 1.142371 -0.1572845

# External validation
Cubist.pred.V <- predict(hv.cub.Exp, newdata = DSM_data[-training, ])
goof(observed = DSM_data$PH60_100cm[-training], predicted = Cubist.pred.V)

##          R2 concordance      MSE      RMSE      bias
## 1 0.2995264  0.460038 1.137435 1.066506 -0.1578419
```

The calibration model validates quite well, but its performance against the validation is comparable. From Figure 7 it appears a few observations are underpredicted (negative bias), otherwise the Cubist model performs reasonably well.

Creating the map resulting from the `hv.cub.Exp` model can be implemented as before using the raster `predict` function (Figure 8).

```
map.cubist.r1 <- predict(hunterCovariates_sub, hv.cub.Exp,
  "soilpH_60_100_cubist.tif", format = "GTiff", datatype = "FLT4S",
  overwrite = TRUE)

plot(map.cubist.r1, main = "Cubist model predicted Hunter Valley soil pH (60-100cm)")
```

5 Random Forests

An increasingly popular data mining algorithm in DSM and the soil sciences, and even in applied sciences in general is the Random Forests model. This algorithm is provided in the `randomForest` package and can be used for both regression and classification purposes. Random Forests are a boosted decision tree model. Further, Random Forests are an ensemble learning method for classification (and regression) that operate by constructing a multitude of decision trees at training time, which are later aggregated to give one single prediction for each observation in a data set. For regression the prediction is the average of the individual tree outputs, whereas in classification the trees

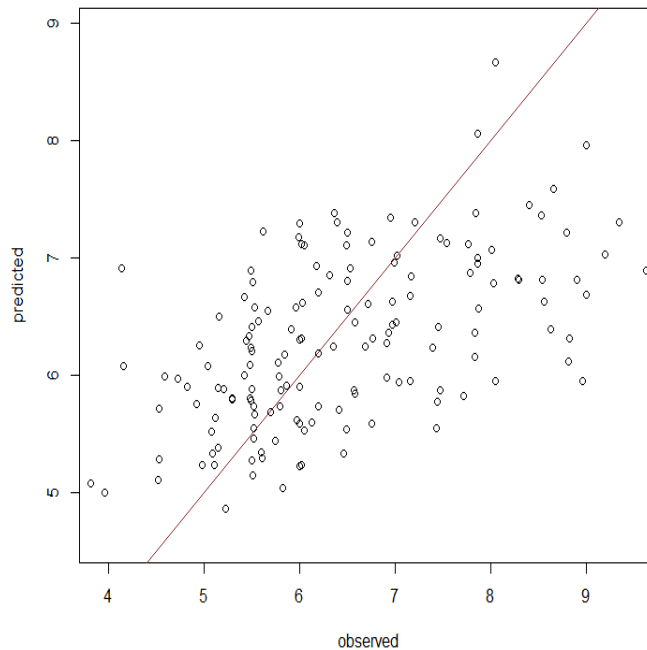


Figure 7: Cubist model xy-plot plot of predicted soil pH (60-100cm) (validation data set).

vote by majority on the correct classification (mode). For further information regarding Random Forest and their underlying theory it is worth consulting Breiman (2001) and Grimm et al. (2008) as an example of its application in DSM studies.

Fitting a Random Forest model in R is relatively straightforward. It is worth consulting the the richly populated help files regarding the `randomForest` package and its functions. We will be using the `randomForest` function and a couple of extractor functions to tease out some of the model fitting diagnostics. Familiar will be the formula structure of the model. As for the Cubist model, there are many model fitting parameters to consider such as the number of trees to build (`ntree`) and the number of variables (covariates) that are randomly sampled as candidates at each decision tree split (`mtry`), plus many others. The `print` function allows one to quickly assess the model fit. The `importance` parameter (logical variable) used within the `randomForest` function specifies that we want to also assess the importance of the covariates used.

```
library(randomForest)
set.seed(123)
training <- sample(nrow(DSM_data), 0.7 * nrow(DSM_data))
```

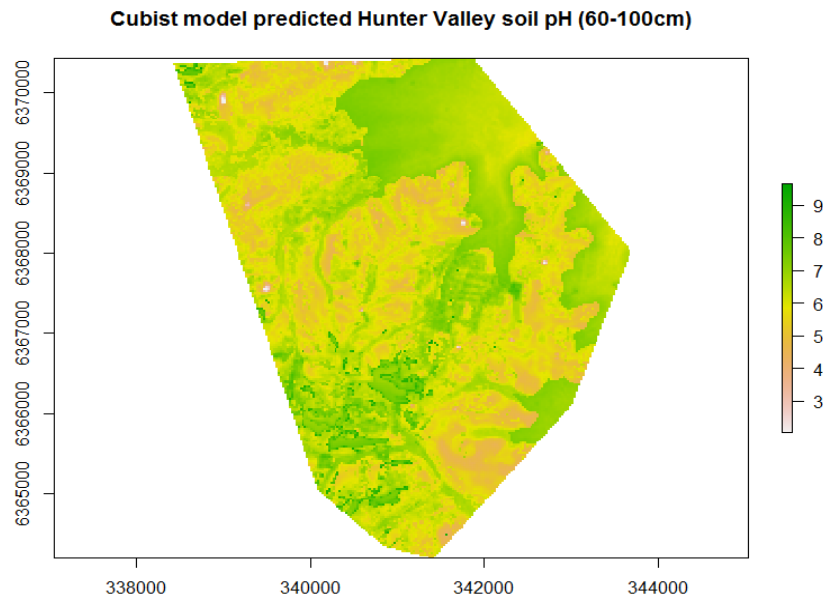


Figure 8: Cubist model predicted Hunter Valley soil pH (60-100cm.

```
# fit the model
hv.RF.Exp <- randomForest(pH60_100cm ~ AACN + Landsat_Band1 + Elevation + Hillshading +
  Mid_Slope_Positon + MRVBF + NDVI + TWI,
  data = DSM_data[training, ], importance = TRUE, ntree = 1000)

print(hv.RF.Exp)

##
## Call:
## randomForest(formula = pH60_100cm ~ AACN + Landsat_Band1 + Elevation +
## Hillshading + Mid_Slope_Positon + MRVBF + NDVI + TWI,
## data = DSM_data[training, ],
## importance = TRUE, ntree = 1000)
##              Type of random forest: regression
##              Number of trees: 1000
## No. of variables tried at each split: 2
##
##              Mean of squared residuals: 1.433532
##              % Var explained: 21.1
```

Using the `varImpPlot` function allows one to visualize which covariates are of most importance to the prediction of our soil variable (Figure 9).

```
varImpPlot(hv.RF.Exp)
```

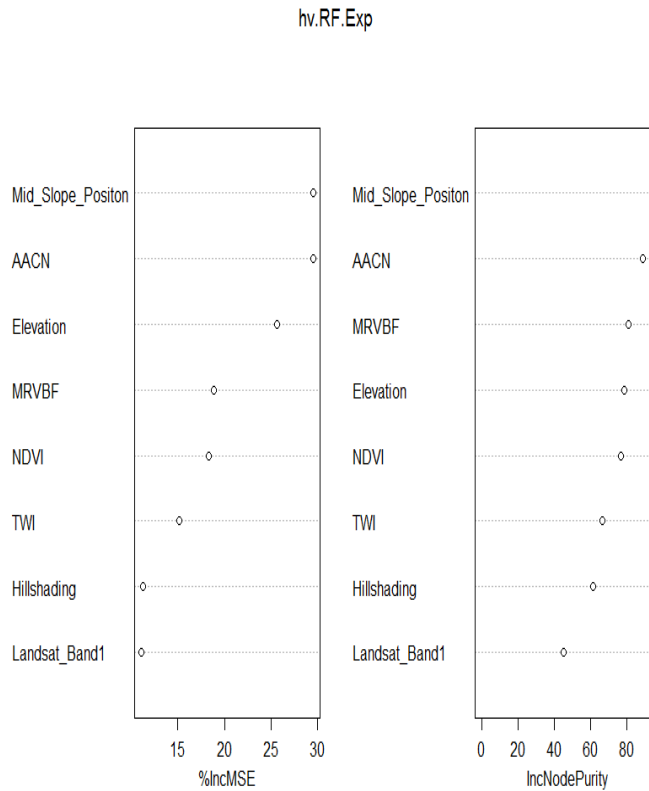


Figure 9: Covariate importance (ranking of predictors) from Random Forest model fitting.

There is a lot of talk about how the variable importance is measured in Random Forest models. So it is probably best to quote from the source:

“Here are the definitions of the variable importance measures. For each tree, the prediction accuracy on the out-of-bag portion of the data is recorded. Then the same is done after permuting each predictor variable. The difference between the two accuracy measurements are then averaged over all trees, and normalized by the standard error. For regression, the MSE is computed on the out-of-bag data for each tree, and then the same computed after permuting a variable. The differences are averaged and normalized by the standard error. If the standard error is equal to 0 for a variable, the division is not done (but the measure is almost always equal to 0 in that case). For the node purity, it is the total decrease in node impurities from splitting on the variable, averaged over all trees. For classification, the node impurity is measured by the Gini index. For regression, it is measured by residual sum of squares.”

The important detail in the quotation above is the the variable importance is measured based on “out-of-bag” samples. Which means observation not

included in the model. Another detail is that they are based on an MSE accuracy measure; in this case the difference, when a covariate is included and excluded in a tree model. The value is averaged over all trees.

So lets validate the `hv.RF.Exp` model both internally and externally.

```
# Internal validation
RF.pred.C <- predict(hv.RF.Exp, newdata = DSM_data[training, ])
goof(observed = DSM_data$PH60_100cm[training], predicted = RF.pred.C)

##           R2 concordance      MSE      RMSE      bias
## 1 0.9087286 0.8847666 0.2998826 0.5476153 0.003947595

# External validation
RF.pred.V <- predict(hv.RF.Exp, newdata = DSM_data[-training, ])
goof(observed = DSM_data$PH60_100cm[-training], predicted = RF.pred.V)

##           R2 concordance      MSE      RMSE      bias
## 1 0.1408058 0.2876967 1.534712 1.238835 0.04706017
```

Essentially these results are quite similar to those of the validations from the other models. One needs to be careful about accepting very good model fit results without a proper external validation. This is particularly poignant for Random Forest models which have a tendency to provide excellent calibration results which can often give the wrong impression about its suitability for a given application. Therefore when using random forest models it pays to look at the validation on the out-of-bag samples when evaluating the goodness of fit of the model.

So lets look at the map resulting from applying the `hv.RF.Exp` models to the covariates (Figure 10).

```
map.RF.r1 <- predict(hunterCovariates_sub, hv.RF.Exp, "soilpH_60_100_RF.tif",
  format = "GTiff", datatype = "FLT4S", overwrite = TRUE)
plot(map.RF.r1,
  main = "Random Forest model predicted Hunter Valley soil pH (60-100cm)")
```

6 Advanced Work: Model fitting with Caret Package

It becomes quickly apparent that there are many variants of prediction functions that could be used for DSM. As was observed, each of the models used have their relative advantages and disadvantages. Each also has their own specific parameterisations and quirks for fitting. Sometimes for the various parameters that are used for model training are chosen without any sort of optimisation, even due consideration sometimes. Sometimes we might be confronted with many possible model structures to use, it is often difficult to make a choice what to use, and just default with a model we know well or have used often without considering alternatives. This is where the `caret` R package <http://topepo.github.io/caret/index.html> comes into its own in terms of efficiency and streamlining the workflow for fitting models and

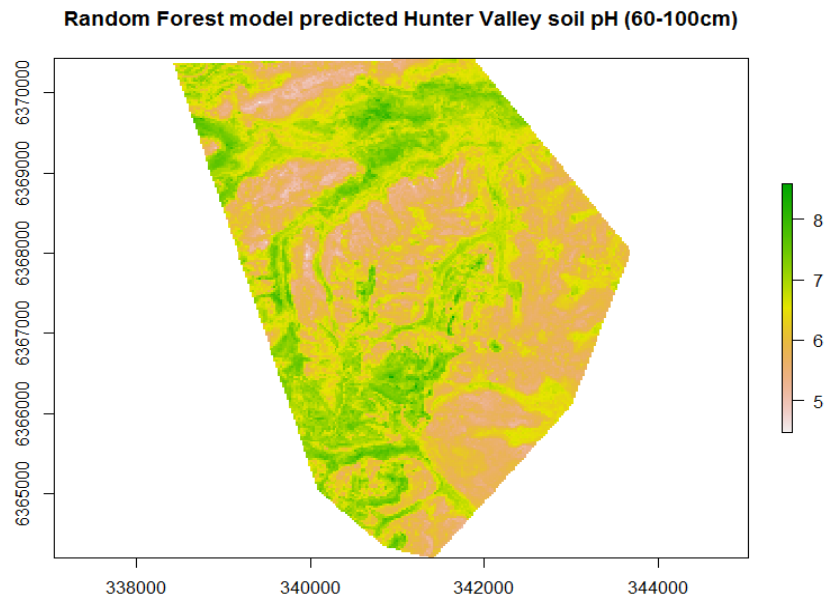


Figure 10: Random Forest model predicted Hunter Valley soil pH (60-100cm).

optimising some of those parameter variables. As the dedicated website indicates (<http://topepo.github.io/caret/index.html>), the **caret** package (short for Classification And REgression Training) is a set of functions that attempt to streamline the process for creating predictive models. As we have seen, there are many different modeling functions in R. Some have different syntax for model training and/or prediction. The **caret** package provides a uniform interface to the various functions themselves, as well as a way to standardize common tasks (such parameter tuning and variable importance). There are currently nearly 400 model functions that the **caret** package interfaces with.

To begin, we first need to load the package into R:

```
library(caret)
```

The workhorse of the **caret** package is the **train** function. We can specify the model to be fitted in two ways:

```
# 1.
```

```
fit <- train(form = pH60_100cm ~ AACN + Landsat_Band1 + Elevation + Hillshading +  
             Mid_Slope_Positon + MRVBF + NDVI + TWI, data = DSM_data, method = "lm")
```

```
# or 2.
```

```
fit <- train(x = DSM_data[, c(6, 7, 8, 9, 11, 12, 13, 14)], y = DSM_data$pH60_100cm,  
            method = "lm")
```

Using the **summary(fit)** command brings up the model parameter estimates,

while the object `fit` also contains a summary of some useful model goodness of fit diagnostics such as the RMSE and R^2 statistics. You can control how model validation is done where options include simple goodness of fit calibration, k-fold cross-validation, and leave-one-out cross-validation. This option is controlled using the parameter `trControl` in the `train` function. The example below illustrates a 5-fold cross validation of a linear regression model with 10 repetitions.

```
fit <- train(x = DSM_data[, c(6, 7, 8, 9, 11, 12, 13, 14)],
y = DSM_data$pH60_100cm, method = "lm",
trControl = trainControl(method = "repeatedcv", number = 5, repeats = 10))

fit

## Linear Regression
##
## 506 samples
## 8 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 10 times)
## Summary of sample sizes: 406, 404, 404, 404, 406, 405, ...
## Resampling results
##
## RMSE      Rsquared  RMSE SD      Rsquared SD
## 1.185794  0.2278382  0.06378662  0.04951837
##
##
```

There are a lot of potential models that you could consider too for DSM. Check out <http://topepo.github.io/caret/modelList.html> or print them as below:

```
list_of_models <- modelLookup()
head(list_of_models)

##      model parameter      label forReg forClass probModel
## 1      ada      iter      #Trees  FALSE      TRUE      TRUE
## 2      ada maxdepth Max Tree Depth  FALSE      TRUE      TRUE
## 3      ada      nu  Learning Rate  FALSE      TRUE      TRUE
## 4  AdaBag      mfinal      #Trees  FALSE      TRUE      TRUE
## 5  AdaBag maxdepth Max Tree Depth  FALSE      TRUE      TRUE
## 6 AdaBoost.M1      mfinal      #Trees  FALSE      TRUE      TRUE

# The number of models caret interfaces with
nrow(list_of_models)

## [1] 393
```

You can choose which model to use in the `train` function with the `method` option. You will note that the fitting of the Cubist and Random Forest models below automatically attempt to optimise some of the fitting parameters, for example the `mtry` parameter for Random Forest. To look at what parameters

can optimised for each model in caret we can use the `modelLookup` function.

```
# Cubist model
modelLookup(model = "cubist")

##      model parameter      label forReg forClass probModel
## 1 cubist committees #Committees  TRUE  FALSE  FALSE
## 2 cubist  neighbors #Instances  TRUE  FALSE  FALSE

fit_cubist <- train(x = DSM_data[, c(6, 7, 8, 9, 11, 12, 13, 14)], y = DSM_data$pH60_100cm,
  method = "cubist", trControl = trainControl(method = "cv", number = 5))

# random forest model
modelLookup(model = "rf")

##      model parameter      label forReg forClass probModel
## 1      rf          mtry #Randomly Selected Predictors  TRUE  TRUE  TRUE

fit_rf <- train(x = DSM_data[, c(6, 7, 8, 9, 11, 12, 13, 14)],
  y = DSM_data$pH60_100cm, method = "rf",
  trControl = trainControl(method = "cv", number = 5))
```

Using the fitted model, predictions can be achieved with the `predict` function:

```
# Cubist model
pred_cubist <- predict(fit_cubist, DSM_data)

# To raster data
pred_cubistMap <- predict(hunterCovariates_sub, fit_cubist)
```

There is plenty of other added functionality of the `caret` package. In addition to the detailed resources mentioned above, it always pays to look over the help files that are associated with each function.

7 Regression kriging

In the previous sections we looked at a few soil spatial prediction functions which at the most fundamental level, target the correlation between the target soil variable and the available covariate information. We fitted a number of models which included simple linear functions to non-linear functions such as regression trees to other more complicated data mining techniques (Cubist and Random Forest). In this section we will extend upon this DSM approach from what are called deterministic models to also include the spatially correlated residuals that result from fitting these models.

The approach we will now concentrate is a hybrid approach to modelling, whereby the predictions of the target variable are made via a deterministic method (regression model with covariate information) and a stochastic method where we determine the spatial auto-correlation of the model residuals with a variogram. The deterministic model essentially “detrends” the data, leaving

behind the residuals for which we need to investigate whether there is additional spatial structure which could be added to the regression model predictions. These residuals are the random component of the *scorpan + emodel*. This method is described as regression kriging and has formally been described in Odeh et al. (1995) and is synonymous with universal kriging (Hengl et al., 2007), which is the formal linear model procedure to this soil spatial modeling approach. The purpose of this exercise is to introduce some basic concepts of regression kriging. You will have already had some experience in regression models. We have also investigated briefly the fundamental concepts of kriging for which the variogram is central to.

7.1 Universal kriging

The universal kriging function in R is found in the **gstat** package. It is useful from the view that both the regression model and variogram modeling of the residuals are handled together. Using universal kriging, one can efficiently derive prediction uncertainties by way of the kriging variance. A limitation of universal kriging in the true sense of the model parameter fitting is that the model is linear. The general preference in DSM studies is to use non-linear and recursive models that do not require strict model assumptions and assume a linear relationship between target variable and covariates.

One of the strict requirements of universal kriging in **gstat** is that the CRS (coordinate reference system) of the point data and covariates must be exactly the same. First we will take a subset of the data to use for an external validation. Unfortunately some of our data has to be sacrificed for this.

```
set.seed(123)
training <- sample(nrow(DSM_data), 0.7 * nrow(DSM_data))
cDat <- DSM_data[training, ]

coordinates(cDat) <- ~x + y
crs(cDat) <- "+proj=utm +zone=56 +south +ellps=WGS84 +datum=WGS84 +units=m +no_defs"
crs(hunterCovariates_sub) = crs(cDat)

# check
crs(cDat)

## CRS arguments:
## +proj=utm +zone=56 +south +ellps=WGS84 +units=m +no_defs
crs(hunterCovariates_sub)

## CRS arguments:
## +proj=utm +zone=56 +south +ellps=WGS84 +units=m +no_defs
```

Now let's parametrise the universal kriging model, and we will use selected covariates as we were using before.

```
library(gstat)
vgm1 <- variogram(pH60_100cm ~ AACN + Landsat_Band1 + Elevation + Hillshading +
```

```

Mid_Slope_Positon + MRVBF + NDVI + TWI, cDat, width = 200, cutoff = 3000)
mod <- vgm(psill = var(cDat$pH60_100cm), "Exp", range = 3000, nugget = 0)
model_1 <- fit.variogram(vgm1, mod)
model_1

##      model      psill      range
## 1      Nug 0.8140845    0.0000
## 2      Exp 0.6168554 426.2951

# Universal kriging model
gUK <- gstat(NULL, "pH", pH60_100cm ~ AACN + Landsat_Band1 + Elevation + Hillshading +
Mid_Slope_Positon + MRVBF + NDVI + TWI, cDat, model = model_1)

```

Using the validation data we can assess the performance of universal kriging using the `goof` function.

```

vDat <- DSM_data[-training, ]
coordinates(vDat) <- ~x + y
crs(vDat) <- "+proj=utm +zone=56 +south +ellps=WGS84 +datum=WGS84 +units=m +no_defs"

crs(vDat)

## CRS arguments:
## +proj=utm +zone=56 +south +ellps=WGS84 +units=m +no_defs

# make the predictions
UK.preds.V <- as.data.frame(krige(pH60_100cm ~ AACN + Landsat_Band1 + Elevation +
Hillshading + Mid_Slope_Positon + MRVBF + NDVI + TWI, cDat, model = model_1,
newdata = vDat))

## [using universal kriging]
goof(observed = DSM_data$pH60_100cm[-training], predicted = UK.preds.V[, 3])

##           R2 concordance      MSE      RMSE      bias
## 1 0.399111 0.5490607 1.091746 1.044867 0.1559361

```

The universal kriging model performs a little better than the MLR model that was fitted earlier.

Applying the universal kriging model spatially is facilitated through the `interpolate` function from `raster`. One can also use the `clusterR` function used earlier in order to speed things up a bit by applying the model over multiple compute nodes. Kriging results in two main outputs: the prediction and the prediction variance. When using the `interpolate` function we can control the output by changing the `index` parameter. The below script results in the maps on Figure 11.

```

par(mfrow = c(1, 2))
# predictions
UK.P.map <- interpolate(hunterCovariates_sub, gUK, xyOnly = FALSE, index = 1)
plot(UK.P.map, main = "Universal kriging predictions")

# prediction variance
UK.Pvar.map <- interpolate(hunterCovariates_sub, gUK, xyOnly = FALSE, index = 2)

```

```
plot(UK.Pvar.map, main = "Universal kriging prediction variance")
```

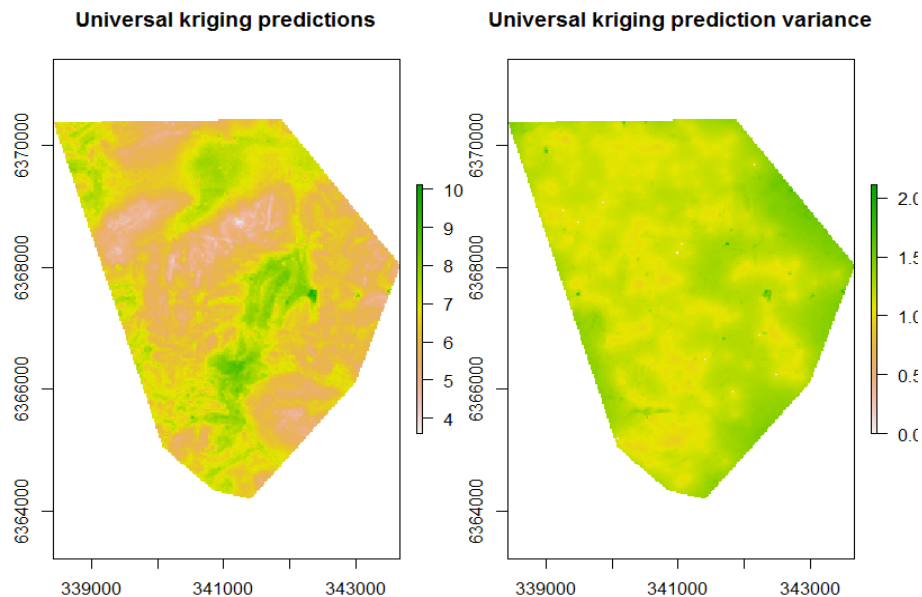


Figure 11: Universal kriging prediction and prediction variance of Hunter Valley soil pH (60-100cm).

7.2 Regression kriging with Cubist models

The following example will provide the steps one would use to perform regression kriging that incorporates a complex model structure such as a data mining algorithm. Here we will use the Cubist model that was used earlier. Lets start from the beginning.

```
library(Cubist)
set.seed(875)
training <- sample(nrow(DSM_data), 0.7 * nrow(DSM_data))
mDat <- DSM_data[training, ]

# fit the model
hv.cub.Exp <- cubist(x = mDat[, c("AACN", "Landsat_Band1", "Elevation",
"Hillshading", "Mid_Slope_Positon", "MRVBF", "NDVI", "TWI")],
y = mDat$pH60_100cm,
cubistControl(rules = 100, extrapolation = 15), committees = 1)
```

Now derive the model residual which is the model prediction subtracted from the residual.

```
mDat$residual <- mDat$pH60_100cm - predict(hv.cub.Exp, newdata = mDat)
mean(mDat$residual)
```

```
## [1] 0.1572845
```

If you check the histogram of these residuals you will find that the mean is around zero and the data seems normally distributed. Now we can assess the residuals for any model structure,

```
coordinates(mDat) <- ~x + y
crs(mDat) <- "+proj=utm +zone=56 +south +ellps=WGS84 + datum=WGS84 +units=m +no_defs"
```

```
vgm1 <- variogram(residual ~ 1, mDat, width = 200, cutoff = 3000)
mod <- vgm(psill = var(mDat$residual), "Sph", range = 3000, nugget = 0)
model_1 <- fit.variogram(vgm1, mod)
model_1
```

```
##      model      psill      range
## 1      Nug 0.6948288    0.0000
## 2      Sph 0.5827945  856.4408
```

```
# Residual kriging model
```

```
gRK <- gstat(NULL, "RKresidual", residual ~ 1, mDat, model = model_1)
```

With the two model components together, we can now compare the external validation statistics of using the Cubist model only and with the Cubist model and residual variogram together.

```
# Cubist model only
```

```
Cubist.pred.V <- predict(hv.cub.Exp, newdata = DSM_data[-training, ])
```

```
# Cubist model with residual variogram
```

```
vDat <- DSM_data[-training, ]
coordinates(vDat) <- ~x + y
crs(vDat) <- "+proj=utm +zone=56 +south +ellps=WGS84 + datum=WGS84 +units=m +no_defs"
```

```
# make the residual predictions
```

```
RK.preds.V <- as.data.frame(krige(residual ~ 1, mDat, model = model_1,
newdata = vDat))
```

```
## [using ordinary kriging]
```

```
# Sum the two components together
```

```
RK.preds.fin <- Cubist.pred.V + RK.preds.V[, 3]
```

```
# validation cubist only
```

```
goof(observed = DSM_data$pH60_100cm[-training], predicted = Cubist.pred.V)
```

```
##           R2 concordance           MSE           RMSE           bias
## 1 0.2995264      0.460038 1.137435 1.066506 -0.1578419
```

```
# validation regression kriging with cubist model
```

```
goof(observed = DSM_data$pH60_100cm[-training], predicted = RK.preds.fin)
```

```
##           R2 concordance           MSE           RMSE           bias
```

```
## 1 0.3899789 0.6098103 1.025823 1.012829 -0.04931782
```

These results confirm that there to be some advantage in performing regression kriging with this particular data. In any case, to apply the regression kriging model here, it requires three steps: First apply the Cubist model, then apply the residual kriging, then finally add both maps together. The script below illustrates how this is done, and the resulting maps are shown on Figure 12.

```
par(mfrow = c(3, 1))
map.RK1 <- predict(hunterCovariates_sub, hv.cub.Exp,
  filename = "soilpH_60_100_cubistRK.tif", format = "GTiff",
  datatype = "FLT4S", overwrite = TRUE)
plot(map.RK1, main = "Cubist model predicted soil pH")

map.RK2 <- interpolate(hunterCovariates_sub, gRK, xyOnly = TRUE, index = 1,
  filename = "soilpH_60_100_residualRK.tif", format = "GTiff",
  datatype = "FLT4S", overwrite = TRUE)
plot(map.RK2, main = "Kriged residual")

# Stack prediction and kriged residuals
pred.stack <- stack(map.RK1, map.RK2)

map.RK3 <- calc(pred.stack, fun = sum, filename = "soilpH_60_100_finalPredRK.tif",
  format = "GTiff", progress = "text", overwrite = T)
plot(map.RK3, main = "Regression kriging prediction")
```

References

- Breiman, L.
2001. Random forests. *Machine Learning*, 41:5–32.
- Brus, D., B. Kempen, and G. Heuvelink
2011. Sampling for validation of digital soil maps. *European Journal of Soil Science*, 62(3):394–407.
- de Gruijter, J., D. Brus, M. Bierkens, and M. Knotters
2006. *Sampling for Natural Resource Monitoring*. Berlin Heidelberg: Springer-Verlag.
- Grimm, R., T. Behrens, M. Marker, and H. Elsenbeer
2008. Soil organic carbon concentrations and stocks on barro colorado island: Digital soil mapping using random forests analysis. *Geoderma*, 146:102–113.
- Hastie, T., R. Tibshirani, and J. Friedman
2001. *The Elements of Statistical Learning*. New York, NY: Springer.
- Hengl, T., G. B. M. Heuvelink, and D. G. Rossiter
2007. About regression kriging: From equations to case studies. *Computers & Geosciences*, 33:1301–1315.

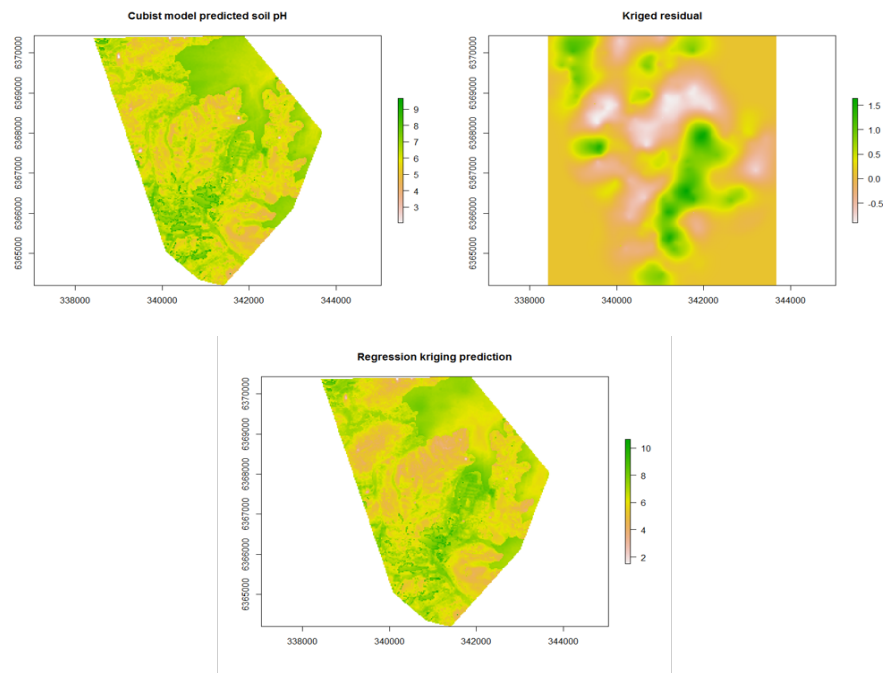


Figure 12: Regression kriging predictions with cubist models. Hunter Valley soil pH (60-100cm).

Lin, L. I.

1989. A concordance correlation coefficient to evaluate reproducibility. *Biometrics*, 45:255–268.

Odeh, I. O. A., A. B. McBratney, and D. J. Chittleborough

1995. Further results on prediction of soil properties from terrain attributes: heterotopic co-kriging and regression kriging. *Geoderma*, 67:215–226.

Quinlan, J. R.

1992. *Proceedings of AI92, 5th Australian Conference on Artificial Intelligence, Singapore*, chapter Learning with continuous classes, Pp. 343–348. World Scientific.