Digital Soil Mapping Report

Amanda Cole

2023-12-19

5.1 Simple Model

After loading the data, I set the target to waterlog.100, a binary categorical variable which contains values of 0 and 1 indicating whether the soil is waterlogged at 100cm depth. I set the predictors to include all predictors in the covariates dataset. I then split the dataset into training and test sets.

```
## Specify target: Waterlog.100
target <- "waterlog.100"

## Specify predictors_all
predictors_all <- names(df_full)[14:ncol(df_full)]
```

```
## The target is: waterlog.100
## The predictors_all are: be_gwn25_hdist,  be_gwn25_vdist,  cindx10_25,  cindx50_25,  geo500h1id,  geo5
```

```
# Splitting dataset into training and testing sets
df_train <- df_full |> dplyr::filter(dataset == "calibration")
df_test  <- df_full |> dplyr::filter(dataset == "validation")

# Filtering out any NAs
df_train <- df_train |> tidyr::drop_na()
df_test <- df_test   |> tidyr::drop_na()
```

```
## For model training, we have a calibration / validation split of:  75 / 25 %
```

After splitting the data into training and test sets, I then trained a Random Forest model using the {ranger} package. Because we are predicting for a categorical variable, this will be a classification model so I set "classification" = TRUE.

```
rf_basic <- ranger::ranger(
  probability = FALSE,
  classification = TRUE,
  y = df_train[, target],     # Target variable
  x = df_train[, predictors_all], # Predictor variables
  seed = 42,                    # Specifying seed for randomization
  num.threads = parallel::detectCores() - 1)
```

```
## Ranger result
##
## Call:
##  ranger::ranger(probability = FALSE, classification = TRUE, y = df_train[,      target], x = df_trai
##
## Type:                             Classification
## Number of trees:                  500
## Sample size:                      605
## Number of independent variables:  91
## Mtry:                             9
## Target node size:                 1
## Variable importance mode:         none
## Splitrule:                        gini
## OOB prediction error:             21.32 %
```

I then evaluated the RF classification model on the testing set and saved the predictions to the test data.

```
# Load area to be predicted
raster_mask <- terra::rast(here::here("data-raw/geodata/study_area/area_to_be_mapped.tif"))
# Turn target raster into a dataframe, 1 px = 1 cell
df_mask <- as.data.frame(raster_mask, xy = TRUE)

# Filter only for area of interest
df_mask <- df_mask |>
  dplyr::filter(area_to_be_mapped == 1)

# Display df
head(df_mask) |>
  knitr::kable()
```

x	y	area_to_be_mapped
2587670	1219750	1
2587690	1219750	1
2587090	1219190	1
2587090	1219170	1
2587110	1219170	1
2587070	1219150	1

```
files_covariates <- list.files(
  path = here::here("data-raw/geodata/covariates/"),
  pattern = ".tif$",
  recursive = TRUE,
  full.names = TRUE
)

# Filter for variables used in RF
preds_all <- names((df_train[, predictors_all]))
files_selected <- files_covariates[apply(sapply(X = preds_all,
                                                FUN = grepl,
                                                files_covariates),
                                         MARGIN =  1,
                                         FUN = any)]
```

```r
# Load all rasters as a stack
raster_covariates <- terra::rast(files_selected)

# Get coordinates
df_locations <- df_mask |>
  dplyr::select(x, y)

# Extract data from covariate raster stack for all gridcells in the raster
df_predict <- terra::extract(
  raster_covariates,
  df_locations,
  ID = FALSE
)

df_predict <- cbind(df_locations, df_predict) |>
  tidyr::drop_na()
```

```r
# Make predictions for validation sites
prediction <- predict(
  rf_basic,           # RF model
  data = df_test,   # Predictor data
  num.threads = parallel::detectCores() - 1
)
```

```r
# Save predictions to validation df
df_test$pred <- prediction$predictions
```

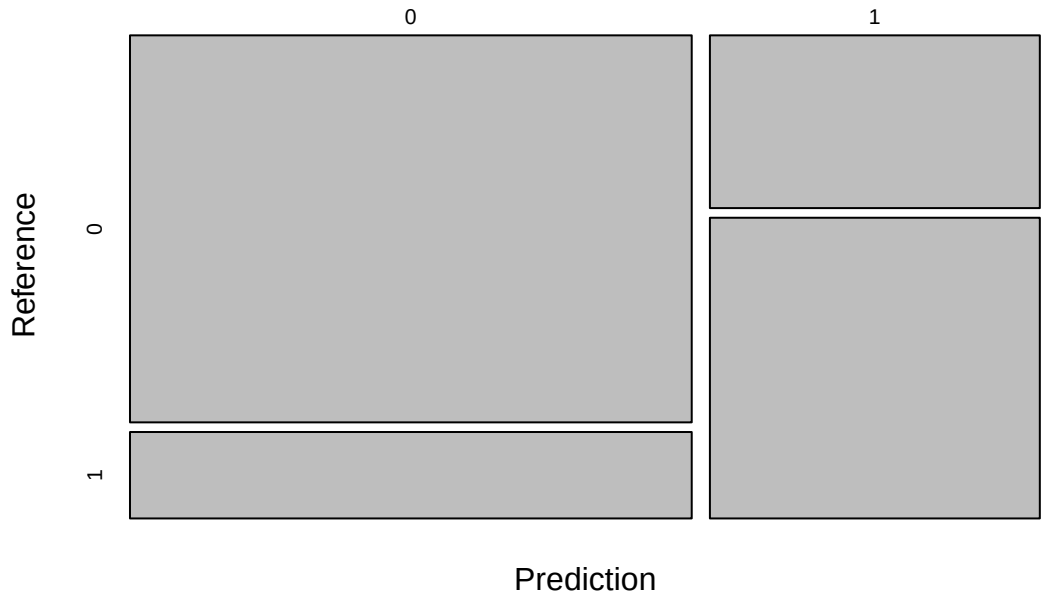
The model is fairly well balanced in terms of TRUE and FALSE values, with 73 "TRUE" values (36.5%) and 127 "FALSE" values (63.5%). This means that Accuracy, a measure of the proportion of outputs that were correctly classified[1], is an appropriate metric to use to measure the quality of our model. If the model was imbalanced, Accuracy would not be an appropriate metric to use.

Other useful metrics for classification include Precision, a measure of the proportion of predictions that were correct out of the total number of predictions[2]; Sensitivity, a measure of the proportion of real positives that the model correctly predicted[2], or the True Positive Rate (True Positives/True Positives + False Negatives); False Positive Rate, a measure of the proportion of real negatives that the model classified as positive (False Positives/False Positives + True Negatives)[1]; Specificity, which is a measure of the proportion of true negatives that the model was able to capture[2]; and and the F1 score which is defined as the harmonic mean of precision and sensitivity[1]. The Receiver Operating Characteristic (ROC) curve is also commonly used to evaluate classification models and is composed by plotting the True Positive Rate against the False Positive Rate. This allows you to determine trade-offs between sensitivity and specificity across various classification thresholds[2]. The Area Under the Curve (AUC), defined as the area between the ROC curve and the x-axis, can be used to assess the model performance across the thresholds with a range of 0 to 1, with 1 representing perfect classification and AUC values of closer to 0.5 representing essentially a random classifier[1,2].

I used the {caret} package to create a Confusion Matrix which includes the necessary metrics.

```
## Confusion Matrix and Statistics
##
##            Reference
## Prediction   0   1
##          0 103  23
##          1  27  47
```

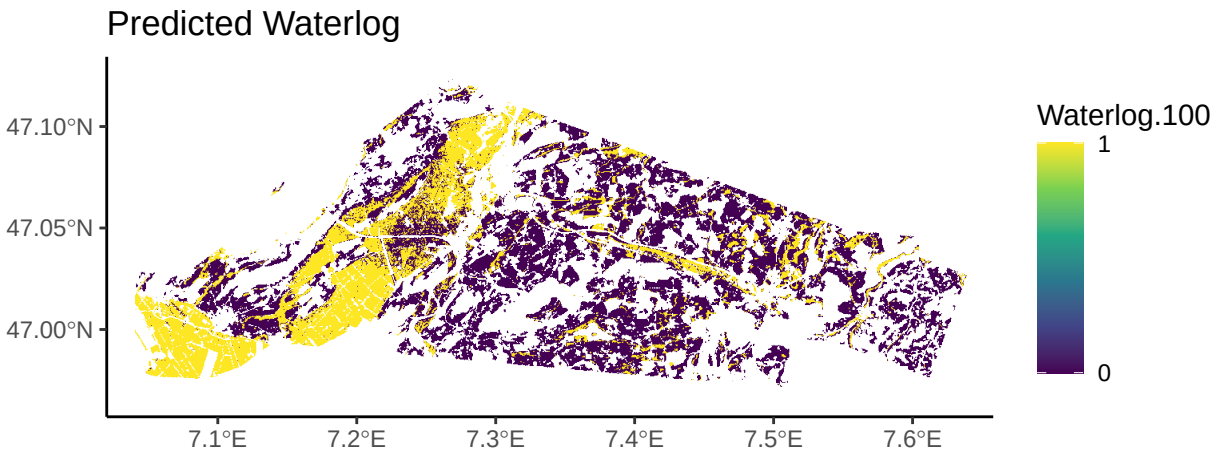
```
##
##                Accuracy : 0.75
##                  95% CI : (0.684, 0.8084)
##     No Information Rate : 0.65
##     P-Value [Acc > NIR] : 0.001542
##
##                   Kappa : 0.4577
##
##  Mcnemar's Test P-Value : 0.671373
##
##             Sensitivity : 0.6714
##             Specificity : 0.7923
##          Pos Pred Value : 0.6351
##          Neg Pred Value : 0.8175
##              Prevalence : 0.3500
##          Detection Rate : 0.2350
##    Detection Prevalence : 0.3700
##       Balanced Accuracy : 0.7319
##
##        'Positive' Class : 1
##
```



As reported by the confusion matrix above, our simple model using all predictors and pre-defined hyper-parameters produced an Accuracy of 0.75. The Sensitivity was 0.6714, meaning that our model correctly predicted ~67.14% of true positive and the Specificity was 0.7923, meaning that our model correctly predicted ~97.23 of true negatives.

Finally, I used the model to create predictions for the entire study area and mapped these. As you can see from the below map, the soil was predicted to be waterlogged in the south west and there are more areas predicted to be not waterlogged in the north east.

```
prediction <- predict(
  rf_basic,               # RF model
  data = df_predict,
  num.threads = parallel::detectCores() - 1)
```



5.2 Variable Selection

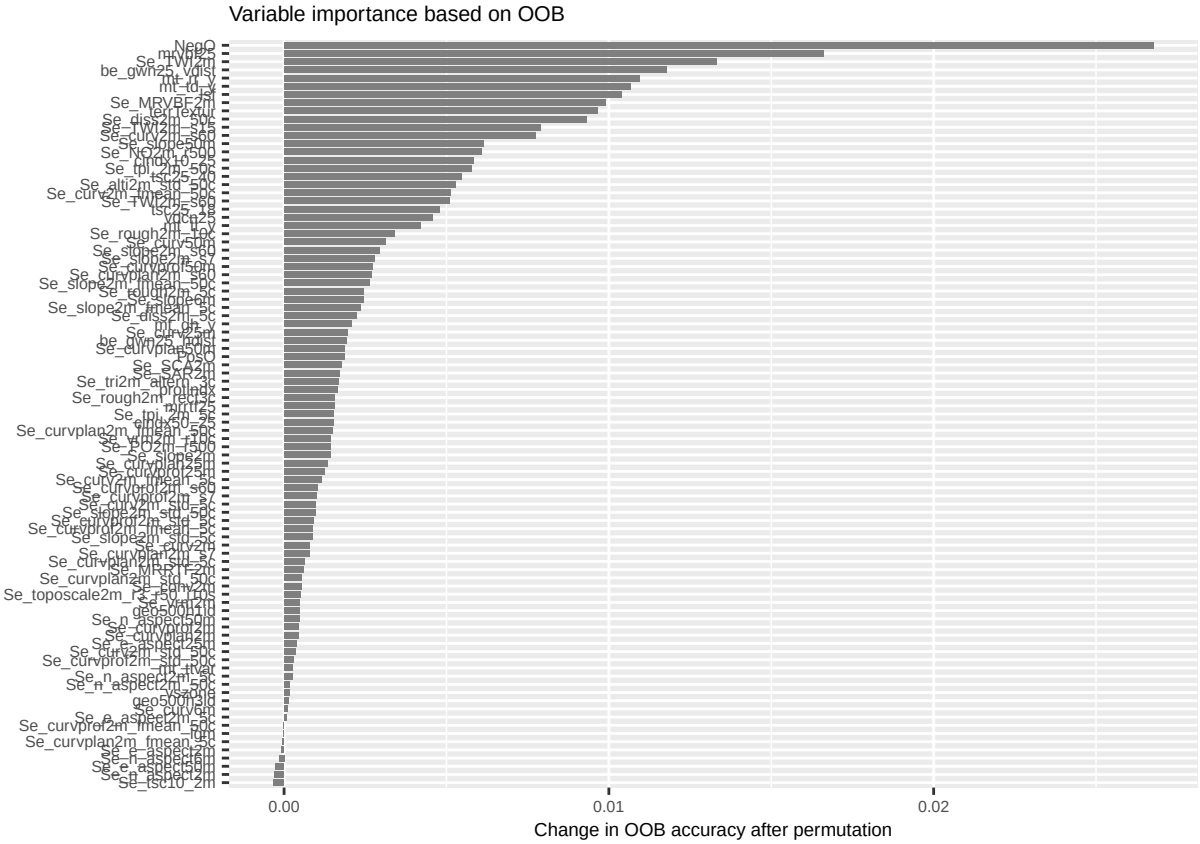
Using all available predictors to train a model increases the risk of overfitting, i.e. the model reflects not just the true underlying pattern or relationship in the data but also observation errors or random fluctuations in the data[1]. By identifying and selecting on the models that are most important, variance can be reduced and model generalisability (the model's ability to make predictions on unseen data) improves.

I first used the {ranger} package, setting "importance" = permutation to calculate variable importance. This will permute or randomly shuffle a predictors values and measure the impact on model performance to determine the impact on model performance.

```
### Variable Importance
rf_varimport <- ranger::ranger(
  probability = FALSE,
  y = df_train[, target],     # target variable
```

```
  x = df_train[, predictors_all],   # Predictor variables
  importance   = "permutation",
  classification = TRUE,
  seed = 42,                        # Specify seed for randomization
  num.threads = parallel::detectCores() - 1)
```

The below bar graph shows the order of importance of the variables. Higher values reflect a stronger the effect of permutation and higher importance of the variables.



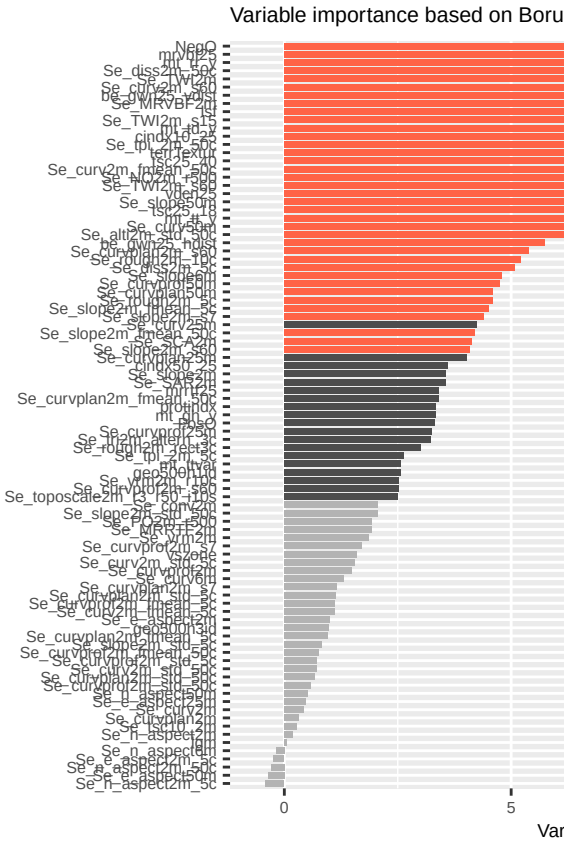
The five most important variables were as follows:

Importance	Variable	Value
1	NegO	0.0267649
2	mrvbf25	0.0166148
3	Se_TWI2m	0.0133032
4	be_gwn25_vdist	0.0117677
5	mt_rr_y	0.0109605

I then used the {Boruta} package to perform a permutation of the values to determine their importance.

```
# run the algorithm
bor <- Boruta::Boruta(
  y = df_train[, target],
  x = df_train[, predictors_all],
```

```
  maxRuns = 50,
  num.threads = parallel::detectCores()-1)
```



The variables deemed important "Confirmed" are shown in orange below.

The Boruta algorithm categorized 37 variables as "Confirmed". The variables deemed important by the Boruta algorithm were largely the same as those identified by the {ranger} package. I then retrained the Random Forest model using the 39 variables deemed important by the Boruta algorithm.

```
# get retained important variables
predictors_selected <- df_bor |>
  dplyr::filter(decision == "Confirmed") |>
  dplyr::pull(rowname)
length(predictors_selected)
```

```
## [1] 37
```

```
# re-train Random Forest model
rf_bor <- ranger::ranger(
  probability = FALSE,
  y = df_train[, target],              # target variable
  x = df_train[, predictors_selected], # Predictor variables
  classification = TRUE,
  seed = 42,                           # Specify the seed for randomization
  num.threads = parallel::detectCores() - 1)
```

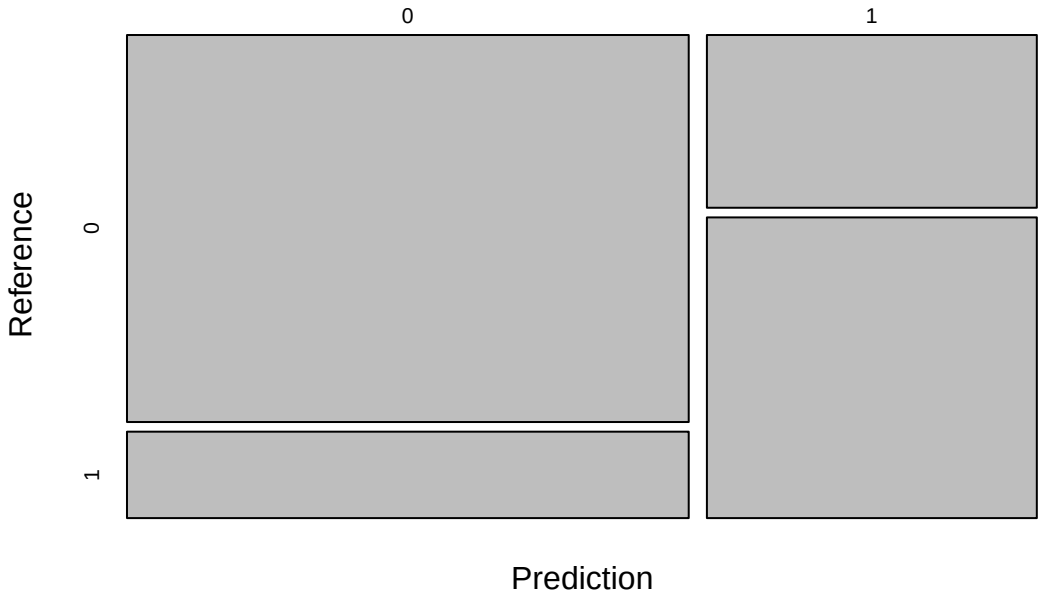
```
## Ranger result
```

```
##
## Call:
##  ranger::ranger(probability = FALSE, y = df_train[, target], x = df_train[,      predictors_selected]
##
## Type:                             Classification
## Number of trees:                  500
## Sample size:                      605
## Number of independent variables:  37
## Mtry:                             6
## Target node size:                 1
## Variable importance mode:         none
## Splitrule:                        gini
## OOB prediction error:             20.99 %
```

I then evaluated the model on the testing subset of the data and created a confusion matrix with the relevant metrics.

```r
# Make predictions for validation sites
prediction <- predict(
  rf_bor,           # RF model
  data = df_test,   # Predictor data
  num.threads = parallel::detectCores() - 1
)
```

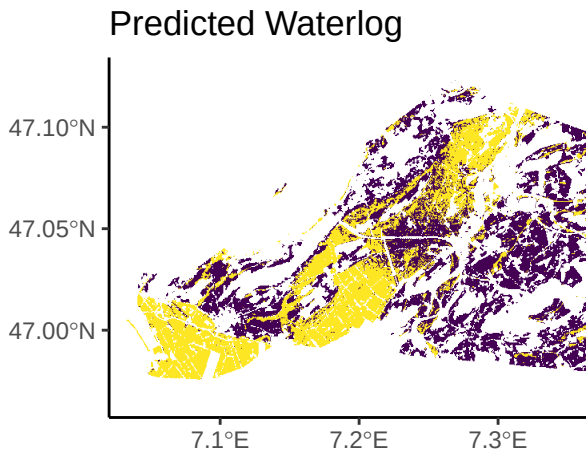
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 107  19
##          1  23  51
##
##                Accuracy : 0.79
##                  95% CI : (0.7269, 0.8443)
##     No Information Rate : 0.65
##     P-Value [Acc > NIR] : 1.134e-05
##
##                   Kappa : 0.5445
##
##  Mcnemar's Test P-Value : 0.6434
##
##             Sensitivity : 0.7286
##             Specificity : 0.8231
##          Pos Pred Value : 0.6892
##          Neg Pred Value : 0.8492
##              Prevalence : 0.3500
##          Detection Rate : 0.2550
##    Detection Prevalence : 0.3700
##       Balanced Accuracy : 0.7758
##
##        'Positive' Class : 1
##
```



As reported by the confusion matrix above, our model using predictors selected by the Boruta algorithm and pre-defined hyperparameters produced an Accuracy of 0.79. The Sensitivity was 0.7286, meaning that our model correctly predicted 72.86% of true positive and the Specificity was 0.8231, meaning that our model correctly predicted 82.31% of true negatives.

The model trained using the predictors selected by the Boruta algorithm had a higher accuracy when evaluated on the testing subset than the simple model (0.79 vs 0.75 respectively. Therefore, the new model trained on the selected variables generalises better to unseen data than the simple model.

However, when considering the OOB prediction error reported as part of the trained model object, the simple model actually has a better OOB prediction error than that of the model trained on the Boruta selected predictors (21.32% vs 20.99% respectively).



The Predicted Waterlog map using the selected predictors is shown below.

5.3 Model Optimization

For Random Forest algorithms, we can control the complexity and randomness of the RF through hyperparameters in the {ranger} package such as min.node.size, num.trees, and mtry. Hyperparameters have a large impact on model performance and sub-optimal hyperparameters can lead to over or under fitting or low generalisability[1]. While the Random Forest models above have fairly good accuracy with the pre-defined hyperparameters, I tuned the hyperparameters to try to improve model performance. I used the {caret} library to implement a 5-fold cross-validation to optimise hyperparameters m.try, min.node.size, and splitrule, using a greedy hyperparameter tuning approach and a grid hyperparameter approach.

After setting the possible mtry, min.node.size, and splitrule values, I optimized using a greedy hyperparameter approach first starting by optmizing min.node.size, keeping mtry constant at 6 (as the default value for mtry for a classification problem is the square root of the number of predictors) and splitrule constant to gini.

```
### Greedy Hyperparameter Tuning
mtry_values <- c(2,3,4,5,6,7,8,9,10,12,14,16)
min.node.size_values <- c(2,5,10,20,25)
splitrule_values <- c("gini", "extratrees")

set.seed(42)
mod <- caret::train(
  pp,
  data = df_train %>%
    drop_na(),
```

```
  method = "ranger",
  trControl = trainControl(method = "cv", number = 5, savePredictions = "final"),
  tuneGrid = expand.grid( .mtry = 9,
                          .min.node.size = min.node.size_values,
                          .splitrule = "gini"),
  metric = "Accuracy",
  replace = FALSE,
  sample.fraction = 0.5,
  num.trees = 50,
  seed = 42
)
```

```
## Random Forest
##
## 605 samples
## 103 predictors
##   2 classes: '0', '1'
##
## Recipe steps:
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 483, 485, 483, 485, 484
## Resampling results across tuning parameters:
##
##   min.node.size  Accuracy   Kappa
##    2             0.8032538  0.5923368
##    5             0.7900300  0.5646531
##   10             0.7950300  0.5765381
##   20             0.7850027  0.5524783
##   25             0.7900576  0.5659257
##
## Tuning parameter 'mtry' was held constant at a value of 9
## Tuning
##  parameter 'splitrule' was held constant at a value of gini
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were mtry = 9, splitrule = gini
##  and min.node.size = 2.
```

Then I optimized mtry while keeping min.node.size and splitrule constant.

```
set.seed(42)
mod <- caret::train(
  pp,
  data = df_train %>%
    drop_na(),
  method = "ranger",
  trControl = trainControl(method = "cv", number = 5, savePredictions = "final"),
  tuneGrid = expand.grid( .mtry = mtry_values,
                          .min.node.size = 5,
                          .splitrule = "gini"),
  metric = "Accuracy",
  replace = FALSE,
  sample.fraction = 0.5,
  num.trees = 50,
```

```
  seed = 42
)
```

```
## Random Forest
##
## 605 samples
## 103 predictors
##   2 classes: '0', '1'
##
## Recipe steps:
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 483, 485, 483, 485, 484
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##    2    0.8000571  0.5868651
##    3    0.7899756  0.5639508
##    4    0.7783909  0.5404693
##    5    0.7949892  0.5784724
##    6    0.7916696  0.5697465
##    7    0.7900165  0.5668478
##    8    0.7866421  0.5580205
##    9    0.7949616  0.5740301
##   10    0.7999070  0.5865636
##   12    0.7866965  0.5597747
##   14    0.7950438  0.5773864
##   16    0.7916967  0.5681074
##
## Tuning parameter 'splitrule' was held constant at a value of gini
##
## Tuning parameter 'min.node.size' was held constant at a value of 5
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were mtry = 2, splitrule = gini
##  and min.node.size = 5.
```

Then, used the best value for mtry and the best value for min.node.size obtained from the previous run and optimised splitrule.

```
set.seed(42)
mod <- caret::train(
  pp,
  data = df_train %>%
    drop_na(),
  method = "ranger",
  trControl = trainControl(method = "cv", number = 5, savePredictions = "final"),
  tuneGrid = expand.grid( .mtry = 12,
                          .min.node.size = 10,
                          .splitrule = splitrule_values),
  metric = "Accuracy",
  replace = FALSE,
  sample.fraction = 0.5,
  num.trees = 50,
```

```
  seed = 42
)
```

```
## Random Forest
##
## 605 samples
## 103 predictors
##   2 classes: '0', '1'
##
## Recipe steps:
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 483, 485, 483, 485, 484
## Resampling results across tuning parameters:
##
##   splitrule   Accuracy   Kappa
##   gini        0.7883909  0.5627873
##   extratrees  0.7883363  0.5619095
##
## Tuning parameter 'mtry' was held constant at a value of 12
## Tuning
##  parameter 'min.node.size' was held constant at a value of 10
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were mtry = 12, splitrule = gini
##  and min.node.size = 10.
```

I then retrained the model using the optimized hyperparameters.

```
mod_greedy <- caret::train(
  pp,
  data = df_train |>
    drop_na(),
  method = "ranger",
  trControl = trainControl(method = "cv", number = 5, savePredictions = "final"),
  tuneGrid = expand.grid( .mtry = 12,
                          .min.node.size = 10,
                          .splitrule = "gini"),
  metric = "Accuracy",
  replace = FALSE,
  sample.fraction = 0.5,
  num.trees = 100,
  seed = 42
)
```

```
## Random Forest
##
## 605 samples
## 103 predictors
##   2 classes: '0', '1'
##
## Recipe steps:
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 483, 484, 485, 485, 483
## Resampling results:
```

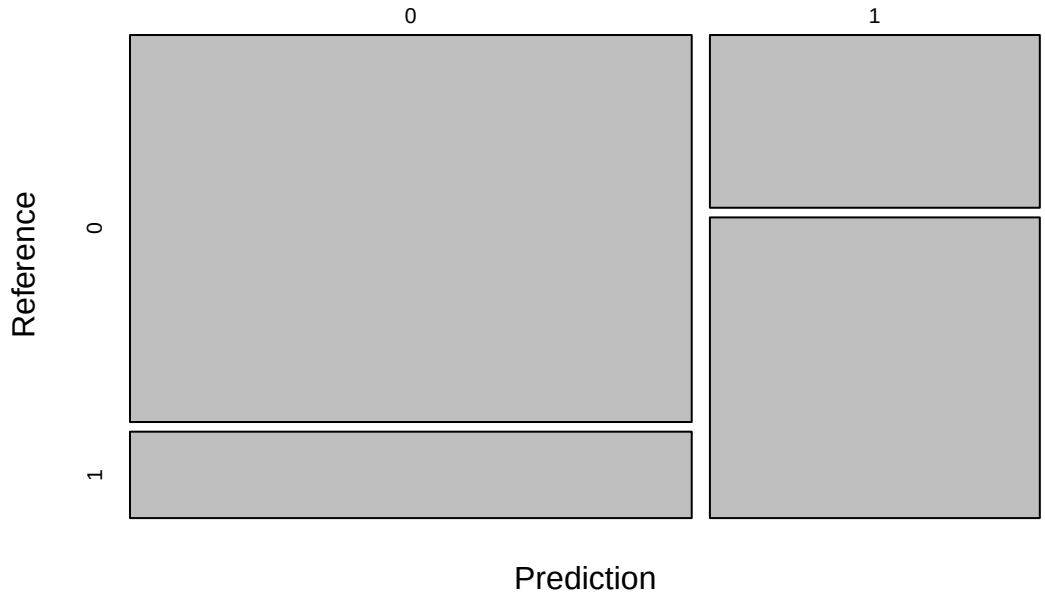
```
##
##   Accuracy   Kappa
##   0.7851127  0.5565504
##
## Tuning parameter 'mtry' was held constant at a value of 12
## Tuning
##  parameter 'splitrule' was held constant at a value of gini
## Tuning
##  parameter 'min.node.size' was held constant at a value of 10
```

I then evaluated the model optimised using the greedy hyperparameter tuning approach on the testing subset of the data.

```
# Make predictions for validation sites using model optimised through greedy approach
prediction <- predict(
  mod_greedy,           # RF model
  newdata = df_test,   # Predictor data
  num.threads = parallel::detectCores() - 1
)

# Save predictions to validation df
df_test$predcv <- prediction
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 102  26
##          1  28  44
##
##                Accuracy : 0.73
##                  95% CI : (0.6628, 0.7902)
##     No Information Rate : 0.65
##     P-Value [Acc > NIR] : 0.009749
##
##                   Kappa : 0.4105
##
##  Mcnemar's Test P-Value : 0.891756
##
##             Sensitivity : 0.6286
##             Specificity : 0.7846
##          Pos Pred Value : 0.6111
##          Neg Pred Value : 0.7969
##              Prevalence : 0.3500
##          Detection Rate : 0.2200
##    Detection Prevalence : 0.3600
##       Balanced Accuracy : 0.7066
##
##        'Positive' Class : 1
##
```



The model accuracy obtained with the hyperparameters optimized using the greedy approach was 0.73. Therefore, this model does not generalise better to unseen data than the initial model which had an accuracy of 0.79.

I then used a grid hyperparameter tuning approach as below.

```
## Grid Hyperparameter Tuning
mod_grid <- caret::train(
  pp,
  data = df_train|>
    drop_na(),
  method = "ranger",
  trControl = trainControl(method = "cv", number = 5, savePredictions = "final"),
  tuneGrid = expand.grid( .mtry = mtry_values,
                          .min.node.size = min.node.size_values,
                          .splitrule = splitrule_values),
  metric = "Accuracy",
  replace = FALSE,
  sample.fraction = 0.5,
  num.trees = 100,
  seed = 42
)
print(mod_grid)
```

```
## Random Forest
##
```

```
## 605 samples
##  35 predictor
##   2 classes: '0', '1'
##
## Recipe steps:
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 483, 485, 483, 485, 484
## Resampling results across tuning parameters:
##
##   mtry  min.node.size  splitrule   Accuracy   Kappa
##    2     2             gini        0.7900438  0.5659249
##    2     2             extratrees  0.7900165  0.5644143
##    2     5             gini        0.7883634  0.5632131
##    2     5             extratrees  0.7800849  0.5457687
##    2    10             gini        0.7883089  0.5636351
##    2    10             extratrees  0.7966012  0.5780266
##    2    20             gini        0.7999618  0.5867508
##    2    20             extratrees  0.7900576  0.5646738
##    2    25             gini        0.7932952  0.5737877
##    2    25             extratrees  0.7883092  0.5604662
##    3     2             gini        0.7884047  0.5633048
##    3     2             extratrees  0.7915739  0.5686203
##    3     5             gini        0.7751260  0.5342422
##    3     5             extratrees  0.7917513  0.5688521
##    3    10             gini        0.7818198  0.5488750
##    3    10             extratrees  0.7933500  0.5714150
##    3    20             gini        0.7867240  0.5600339
##    3    20             extratrees  0.7899621  0.5632809
##    3    25             gini        0.7917105  0.5709882
##    3    25             extratrees  0.7932954  0.5703374
##    4     2             gini        0.7834182  0.5520585
##    4     2             extratrees  0.7999207  0.5861208
##    4     5             gini        0.7834318  0.5527861
##    4     5             extratrees  0.7916012  0.5688435
##    4    10             gini        0.7801260  0.5453276
##    4    10             extratrees  0.8048934  0.5964036
##    4    20             gini        0.7917105  0.5695932
##    4    20             extratrees  0.7899618  0.5652061
##    4    25             gini        0.7900987  0.5658564
##    4    25             extratrees  0.7883498  0.5627276
##    5     2             gini        0.7950029  0.5780257
##    5     2             extratrees  0.7949343  0.5744185
##    5     5             gini        0.7983360  0.5842330
##    5     5             extratrees  0.8032403  0.5930796
##    5    10             gini        0.7966421  0.5805743
##    5    10             extratrees  0.7884045  0.5637376
##    5    20             gini        0.7916696  0.5683211
##    5    20             extratrees  0.7866287  0.5571371
##    5    25             gini        0.7983225  0.5844457
##    5    25             extratrees  0.7883365  0.5610867
##    6     2             gini        0.7866969  0.5587346
##    6     2             extratrees  0.7900025  0.5668352
##    6     5             gini        0.7900029  0.5642350
##    6     5             extratrees  0.7867376  0.5582096
```

```
##    6    10            gini        0.7834182  0.5529286
##    6    10            extratrees  0.7949481  0.5759274
##    6    20            gini        0.7916285  0.5685283
##    6    20            extratrees  0.7948525  0.5744511
##    6    25            gini        0.7933363  0.5727812
##    6    25            extratrees  0.7916150  0.5687309
##    7     2            gini        0.7866967  0.5610504
##    7     2            extratrees  0.8016147  0.5921333
##    7     5            gini        0.7851120  0.5578469
##    7     5            extratrees  0.7950574  0.5772623
##    7    10            gini        0.7785002  0.5449794
##    7    10            extratrees  0.7966558  0.5798828
##    7    20            gini        0.7900576  0.5678639
##    7    20            extratrees  0.7866969  0.5588135
##    7    25            gini        0.7950576  0.5773266
##    7    25            extratrees  0.7900029  0.5656219
##    8     2            gini        0.7817516  0.5502218
##    8     2            extratrees  0.7850438  0.5563039
##    8     5            gini        0.7800576  0.5460496
##    8     5            extratrees  0.7735413  0.5307672
##    8    10            gini        0.7883634  0.5635866
##    8    10            extratrees  0.7866832  0.5589575
##    8    20            gini        0.7933907  0.5744801
##    8    20            extratrees  0.7866014  0.5571130
##    8    25            gini        0.7983498  0.5834049
##    8    25            extratrees  0.7866016  0.5581093
##    9     2            gini        0.7949889  0.5749076
##    9     2            extratrees  0.7883638  0.5636532
##    9     5            gini        0.7883909  0.5625072
##    9     5            extratrees  0.7883225  0.5611159
##    9    10            gini        0.7884453  0.5622120
##    9    10            extratrees  0.7833638  0.5528783
##    9    20            gini        0.7900711  0.5658615
##    9    20            extratrees  0.7850440  0.5549110
##    9    25            gini        0.7966423  0.5806705
##    9    25            extratrees  0.7899756  0.5651793
##   10     2            gini        0.7866967  0.5591711
##   10     2            extratrees  0.7950165  0.5782711
##   10     5            gini        0.8015736  0.5906713
##   10     5            extratrees  0.7966285  0.5803598
##   10    10            gini        0.7817924  0.5490346
##   10    10            extratrees  0.7850849  0.5537484
##   10    20            gini        0.7884182  0.5623556
##   10    20            extratrees  0.7965192  0.5788000
##   10    25            gini        0.7883771  0.5617819
##   10    25            extratrees  0.7916558  0.5687441
##   12     2            gini        0.7851258  0.5560038
##   12     2            extratrees  0.7835413  0.5519535
##   12     5            gini        0.7866558  0.5605496
##   12     5            extratrees  0.7900849  0.5643874
##   12    10            gini        0.7850578  0.5557719
##   12    10            extratrees  0.7900711  0.5663302
##   12    20            gini        0.7916834  0.5697938
##   12    20            extratrees  0.7883365  0.5610362
```

```
##   12    25             gini        0.7916696  0.5709248
##   12    25             extratrees  0.7899481  0.5657777
##   14     2             gini        0.7883771  0.5632697
##   14     2             extratrees  0.7934045  0.5726505
##   14     5             gini        0.7834456  0.5527872
##   14     5             extratrees  0.7917242  0.5701205
##   14    10             gini        0.7900436  0.5657630
##   14    10             extratrees  0.7950027  0.5774241
##   14    20             gini        0.7866829  0.5573537
##   14    20             extratrees  0.7899754  0.5653659
##   14    25             gini        0.7883498  0.5616439
##   14    25             extratrees  0.7933909  0.5738327
##   16     2             gini        0.7883498  0.5617264
##   16     2             extratrees  0.7735551  0.5321089
##   16     5             gini        0.7866832  0.5592643
##   16     5             extratrees  0.7966965  0.5788833
##   16    10             gini        0.7900849  0.5663894
##   16    10             extratrees  0.7751398  0.5358596
##   16    20             gini        0.7933498  0.5717438
##   16    20             extratrees  0.7850849  0.5550985
##   16    25             gini        0.7867107  0.5572940
##   16    25             extratrees  0.7900300  0.5649750
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were mtry = 4, splitrule = extratrees
##  and min.node.size = 10.
```

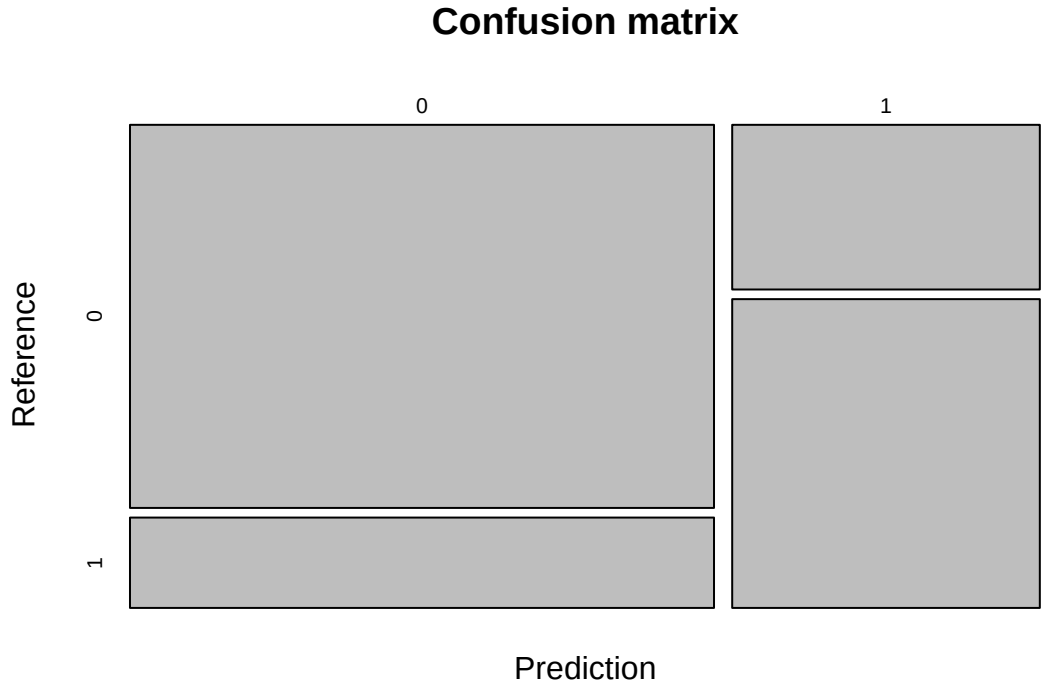
The final values obtained using a grid hyperparameter tuning approach are reported above. I evaluated the model optimised using the grid hyperparameter tuning approach on the testing subset of the data.

```
### Evaluation
# Make predictions for validation sites using greedy model
set.seed(42)
prediction <- predict(
  mod_grid,
  newdata = df_test,
  seed = 42,
  num.threads = parallel::detectCores() - 1
)

# Save predictions to validation df
df_test$predcv <- prediction
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 106  25
##          1  24  45
##
##                Accuracy : 0.755
##                  95% CI : (0.6894, 0.8129)
##     No Information Rate : 0.65
```

```
##     P-Value [Acc > NIR] : 0.0009148
##
##                   Kappa : 0.4598
##
##  Mcnemar's Test P-Value : 1.0000000
##
##             Sensitivity : 0.6429
##             Specificity : 0.8154
##          Pos Pred Value : 0.6522
##          Neg Pred Value : 0.8092
##              Prevalence : 0.3500
##          Detection Rate : 0.2250
##    Detection Prevalence : 0.3450
##       Balanced Accuracy : 0.7291
##
##        'Positive' Class : 1
##
```



Neither the grid nor the greedy hyperparameter tuning approaches resulted in a model that had a higher accuracy than the boruta algorithm trained with the default hyperparameters of mtry=6, min.node.size=1, and splitrule=gini. The optimal min.node.size found by both the greedy and grid hyperparameter tuning approaches were higher than the default min.node.size of 1 and as higher min.node.sizes should increase model generalisability resulting in a model that performs better on the new testing data[1], we would expect a higher accuracy when evaluating the model on the testing subset of the data, however this was not the case.

5.4 Probabalistic Predictions

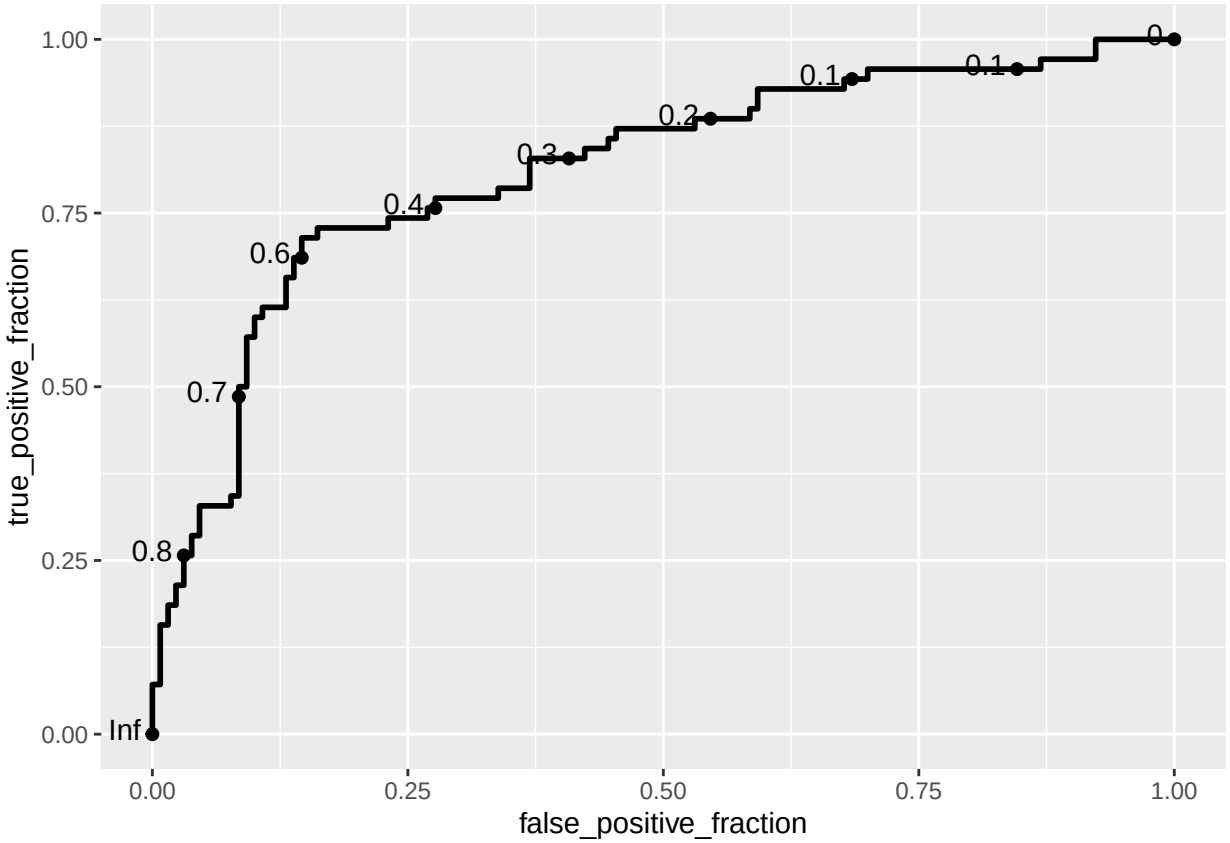
To predict the probability of the soil to be waterlogged, I trained a probabalistic random forest model by setting probabilty=TRUE as below and made predictions for the validation sites.

```
rf_prob <- ranger::ranger(
probability = TRUE,
classification = TRUE,
y = df_train[, "waterlog.100"],     # target variable
x = df_train[, predictors_selected], # Predictor variables
seed = 42,                     # Specify seed
num.threads = parallel::detectCores() - 1)
```

```
## Ranger result
##
## Call:
##  ranger::ranger(probability = TRUE, classification = TRUE, y = df_train[,      "waterlog.100"], x = d
##
## Type:                             Probability estimation
## Number of trees:                  500
## Sample size:                      605
## Number of independent variables:  35
## Mtry:                             5
## Target node size:                 10
## Variable importance mode:         none
## Splitrule:                        gini
## OOB prediction error (Brier s.):  0.1434481
```

```
# Make predictions for validation sites
prediction <- predict(
  rf_prob,           # RF model
  data = df_test,   # Predictor data
  type="response",
  num.threads = parallel::detectCores() - 1
)
```

To assess the performance of the model across various thresholds, I created the below Reciever Operating Curve with the True Positive Rate plotted against the False Positive Rate. Lower thresholds have higher TPR and FPRs.



In cases where waterlogged soils severely jeopardize the stability of the building I would set a high threshold because we want to ensure that soils that have a possibility of being waterlogged as classified as waterlogged and that the highest number of true positives are captured even at the expense of a high number of false positives. In cases where waterlogged soils are unwanted but not critical, we can set a lower threshold to avoid unnecessary delays that could result from false positives. A similar analogy to another field is that of medical testing. In medical testing if a disease is life-threatening, you would want to set a high threshold to ensure as many true positives are captured to be able to act early and this is worth it even if the patient recieves a false alarm. If the disease is less critical, the threshold can be lower as the consequences of not capturing as many true positives are less severe.

References

[1] Benjamin Stocker, Koen Hufkens, Pepa Arán, & Pascal Schneider. Applied Geodata Science (v1.0), Zenodo, 2019. https://geco-bern.github.io/agds/.

[2] Kuhn, Max and Johnson, Kjell. Feature Engineering and Selection: A Practical Approach for Predictive Models, Taylor & Francis, 2019. https://bookdown.org/max/FES/.