Supporting Information. Valavi, R., G. Guillera-Arroita, J. Lahoz-Monfort, and J. Elith. 2021. Predictive performance of presence-only species distribution models: a benchmark study with reproducible code. Ecological Monographs.

Appendix S2

Contents

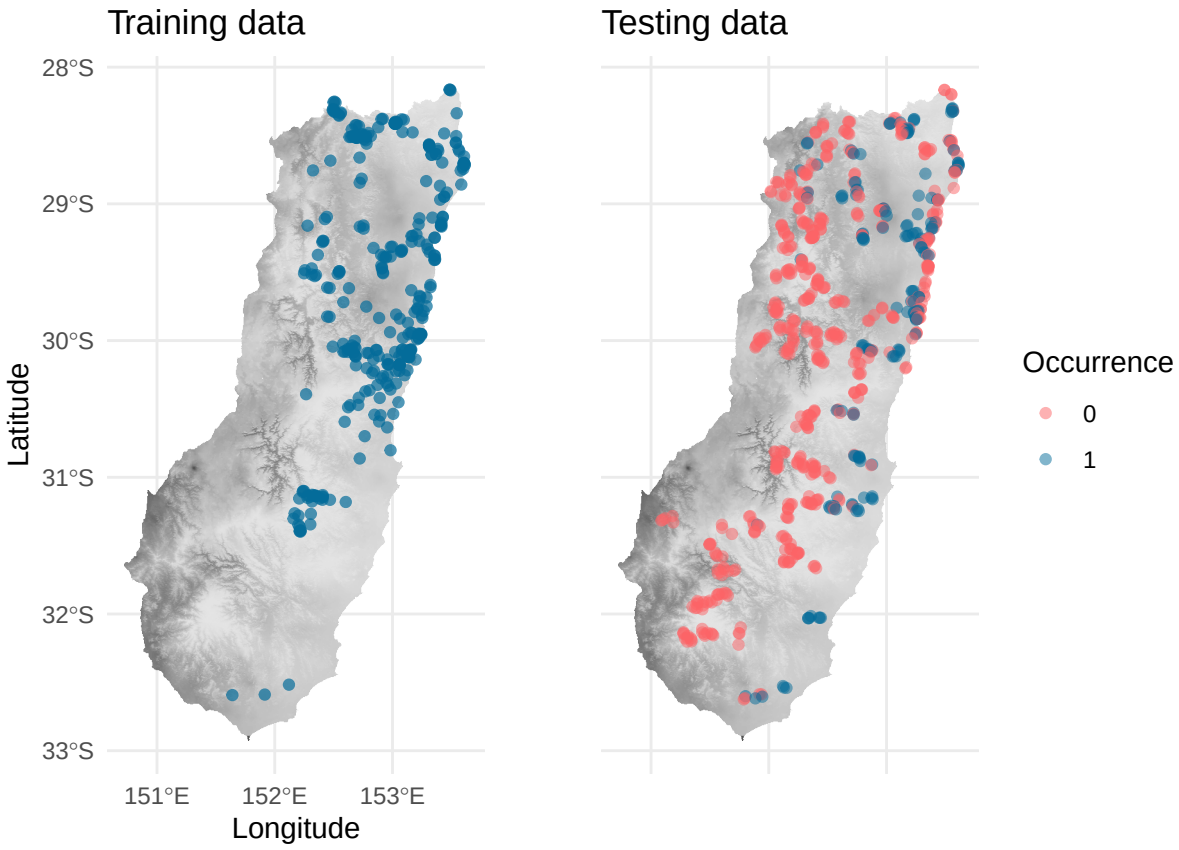
Introduction

In this document, we show how different **species distribution models (SDMs)** can be fitted to **presence-background** data in the R programming language. Several statistical and machine learning algorithms are presented here, covering all methods presented in the main text. Each model is fitted using the same code we used for producing the results in our paper. Some additional suggestions are also provided that could be useful for smaller modeling exercises, but that we could not apply due to the large number of species and computation limits. The code provided here can be used for modeling any binary response

Please cite us by **Valavi, R., Guillera-Arroita, G, Lahoz-Monfort, J.J. & Elith, J. (2021) Predictive performance of presence-only species distribution models: a benchmark study with reproducible code. Ecological Monographs.** if you use the materials in this document.

The example species

One of the species in New South Wales (NSW), Australia dataset ("nsw14") is used as an example. This species has 315 presence records (represented as 1) coupled with **10,000** randomly selected background points (represented as 0) from the landscape that make up the `training` data. The `evaluation` data for this species has 161 and 541 presences and absences, respectively. The geographic distribution of the species' occurrence data is shown in the following figure (the background points are ignored here for visualization purposes).



Loading species data

Here, code for loading the example species data from the `disdat` package is presented. You need to load the species *presences* and *background* for training the models and species *presence-absence* and the *environmental covariates* for testing the models. Notice the background data shown in the following code is **10,000** random background sample (from the NCEAS 2006 paper). For the our paper, **50,000** random samples were used. So, the model fitting is faster for this example compared to the modeling time reported in the main text. You can find the files for 50k background samples in the supplementary materials.

After loading the data, you need to convert the categorical covariates to *factor* (check Elith *et al.* 2020) or the documentation of the `disdat` package for the list of covariates and their types). In addition, some models require the covariates to be on the same scale e.g. SVM. So we normalized (scale and center to mean and standard deviation) the continuous covariate before model fitting (in the real modeling, the covariates were not normalized for Lasso GLM, Ridge regression, and MaxEnts as they have internal normalization). The same parameter used for transforming (mean and standard deviation of the training samples) the training data should be applied to the testing data and the raster files when one wants to predict to raster.

To install the `disdat` package use:

```r
install.packages("disdat")
```

```r
# loading the data package
library(disdat)

# specifying the species id
spID <- "nsw14"

# loading the presence-background data
pr <- disPo("NSW")
bg <- disBg("NSW") # this is 10,000 random backgrounds
pr <- pr[pr$spid == spID, ] # subset the target species
# combine the presence and background points
training <- rbind(pr, bg)

# loading the presence-absence testing data
# species 'nsw14' is in the 'db' group, see Elith et al. (2020) for details
testing_env <- disEnv("NSW", group = "db")
testing_pa <- disPa("NSW", group = "db")

# uncorrelated variables - see appendix Table 1 for variables used in each region
covars <- c("cti", "disturb", "mi", "rainann", "raindq",
            "rugged", "soildepth", "soilfert", "solrad",
            "tempann", "topo", "vegsys")
# subset uncorrelated covariates
training <- training[, c("occ", covars)]
testing_env <- testing_env[, covars]

# convert the categoricals to factor
training$vegsys <- as.factor(training$vegsys)
testing_env$vegsys <- as.factor(testing_env$vegsys)

# normalize the covariates (exept vegsys which is categorical)
# *notice: not all the models are fitted on normalized data in
# the main analysis! Please check the main text.
for(v in covars[covars!="vegsys"]){
  meanv <- mean(training[,v])
  sdv <- sd(training[,v])
  training[,v] <- (training[,v] - meanv) / sdv
  testing_env[,v] <- (testing_env[,v] - meanv) / sdv
}

# print the first few rows and columns
training[1:5, 1:5]
```

```
##      occ        cti    disturb        mi  rainann
## 1150   1  0.0444338 -0.1104779 0.9155303 1.268810
## 1151   1  0.5326107 -0.1104779 0.9155303 1.274938
## 1152   1  1.4601469 -0.1104779 1.0000868 1.281066
## 1153   1 -0.3461077 -1.1455181 1.2537564 1.238172
## 1154   1 -0.2484724 -1.1455181 1.1691998 1.201405
```

Modeling methods:

This section shows how the models are fitted in this study followed by code for calculating model performance metrics. In the final section, mapped distributions are produced. Codes are provided to facilitate predicting to rasters for some of the models for which the generic prediction function in the `raster` package is not working e.g. **SVM** and **glmnet**.

Most of the models in this study accept weights (see table 2 in the main text). There are two types of weights, *case weights* and *class weights*. In the case weights, there is a weight for every single observation of presence and background (i.e. 10315 number of weights in this example; 10000 backgrounds and 315 presences). But, the class weights has one weight per class (one for presences and one for backgrounds). Only RF down-sampled and SVM accepts class weights, the rest of the models allow only case weights. The weights are generated by giving a weight of 1 to every presence location and give the weights to the background in a way that the sum of the weights for the presence and background samples are equal i.e. *number of presences / number of background* (315 / 10000 here). For GLM and GAM, we also used the weighting scheme used in Infinitely Weighted Logistic Regression (IWLR) suggested by Fithian and Hastie (2013). We called them `IWLR-GLM` and `IWLR-GAM`. This IWLR weighting can be generated by:

```
# calculating the IWLR weights
iwp <- (10^6)^(1 - training$occ)
```

GAM

The **mgcv** package is used to fit a Generalized Additive Model (GAM). `mgcv` can estimate the level of smoothness and there is no need to pre-specify the *degreee of freedom (df)*. The package does that by allowing a very high value of df and then regularising it by internal validation methods (Wood, 2016).

Few parameters can set the model flexibility in `mgcv::gam`. First, it is recommended by Pedersen *et al.,* (2019) to use **REML (restricted maximum likelihood)** to estimate the coefficients and smoothers. Second, the k parameter which is the number of *basis functions* (for creating smoothing terms) specifies the possible maximum *Effective Degree of Freedom (EDF)*. This is the amount of wiggliness that each function can have. Thus **k** should be high enough to give sufficient flexibility and low enough to be computationally affordable (the higher the k, the longer it takes to fit the model). We used default k (`k = 10`) in our modeling procedure. You can use `mgcv::gam.check()` to check if the k is consistent with your data. This is related to the number of unique values in each covariate: k should not be higher than that. For choosing k, *the number of unique values* in the covariate should be considered i.e. the lower this number, the lower k should be. For more details read Pedersen *et al.,* (2019).

```
# loading the packages
library(mgcv)

# calculating the case weights (equal weights)
# the order of weights should be the same as presences and backgrounds in the training data
prNum <- as.numeric(table(training$occ)["1"]) # number of presences
bgNum <- as.numeric(table(training$occ)["0"]) # number of backgrounds
wt <- ifelse(training$occ == 1, 1, prNum / bgNum)

form <- occ ~ s(cti) + s(mi) + s(rainann) + s(raindq) + s(rugged) +
             s(soildepth) + s(solrad) + s(tempann) + s(topo) +
             disturb + soilfert + vegsys

tmp <- Sys.time()
set.seed(32639)
gm <- mgcv::gam(formula = as.formula(form),
                data = training,
```

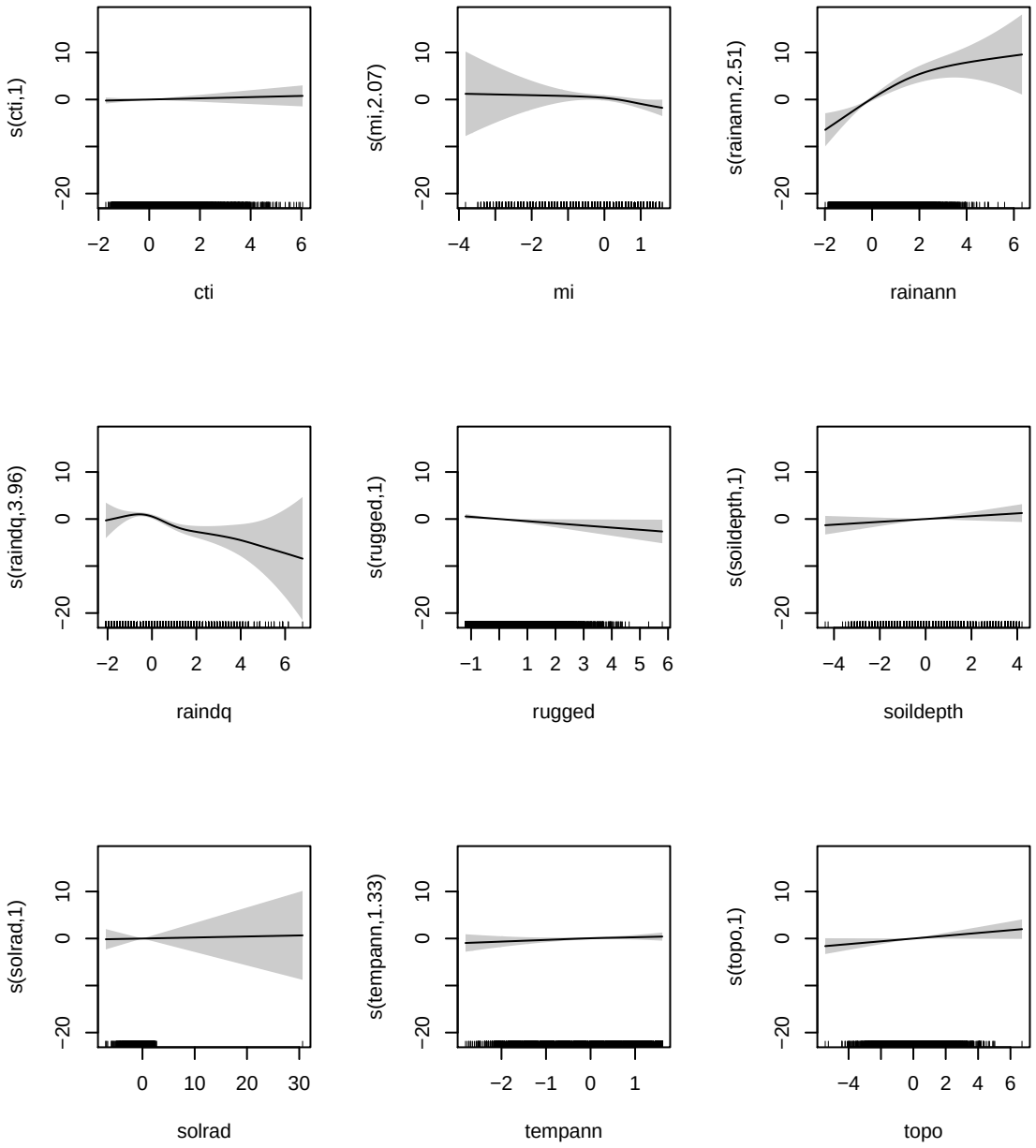
```
                family = binomial(link = "logit"),
                weights = wt,
                method = "REML")
Sys.time() - tmp
```

```
## Time difference of 1.385438 mins
```

```
# check the appropriateness of Ks
gam.check(gm)
```

```
##
## Method: REML   Optimizer: outer newton
## full convergence after 14 iterations.
## Gradient range [-2.758364e-05,3.141404e-06]
## (score 188.7573 & scale 1).
## Hessian positive definite, eigenvalue range [7.776258e-06,0.7727162].
## Model rank =  92 / 92
##
## Basis dimension (k) checking results. Low p-value (k-index<1) may
## indicate that k is too low, especially if edf is close to k'.
##
##                k'  edf k-index p-value
## s(cti)       9.00 1.00    0.83   0.245
## s(mi)        9.00 2.07    0.78  <2e-16 ***
## s(rainann)   9.00 2.51    0.76  <2e-16 ***
## s(raindq)    9.00 3.96    0.82   0.065 .
## s(rugged)    9.00 1.00    0.80  <2e-16 ***
## s(soildepth) 9.00 1.00    0.82   0.045 *
## s(solrad)    9.00 1.00    0.83   0.370
## s(tempann)   9.00 1.33    0.81   0.025 *
## s(topo)      9.00 1.00    0.82   0.060 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
plot(gm, pages = 1, rug = TRUE, shade = TRUE)
```



GLM

To fit GLMs, we used forward and backward model selection with options for dropping variables or including them, considering only linear or linear plus quadratic terms. No interaction terms were tested. Selection was based on change in AIC (Hastie *et al.*, 2009).
For this purpose, we use the `step.Gam` function in **gam** package. A model scope is needed to determine the possible terms to select from. We restricted the choices to excluding a variable, or fitting linear or *quadratic* functions.

```
# loading the packages
library(gam)

# calculating the weights
# the order of weights should be the same as presences and backgrounds in the training data
```

```r
prNum <- as.numeric(table(training$occ)["1"]) # number of presences
bgNum <- as.numeric(table(training$occ)["0"]) # number of backgrounds
wt <- ifelse(training$occ == 1, 1, prNum / bgNum)

# the base glm model with linear terms
lm1 <- glm(occ ~., data = training, weights = wt, family = binomial(link = "logit"))

summary(lm1)
```

```
##
## Call:
## glm(formula = occ ~ ., family = binomial(link = "logit"), data = training,
##     weights = wt)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -0.62671  -0.18315  -0.05279  -0.01257   2.87070
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept)    0.18367    0.54038   0.340 0.733936
## cti            0.06186    0.17826   0.347 0.728567
## disturb       -0.40902    0.16786  -2.437 0.014822 *
## mi             0.15917    0.36632   0.435 0.663917
## rainann        1.86001    0.35517   5.237 1.63e-07 ***
## raindq        -1.25517    0.28189  -4.453 8.48e-06 ***
## rugged        -0.55564    0.20841  -2.666 0.007672 **
## soildepth      0.31337    0.22166   1.414 0.157436
## soilfert      -0.15427    0.15619  -0.988 0.323280
## solrad         0.14789    0.14730   1.004 0.315368
## tempann        0.64806    0.21912   2.958 0.003101 **
## topo           0.33231    0.15135   2.196 0.028117 *
## vegsys2       -0.84141    0.51099  -1.647 0.099630 .
## vegsys3       -1.61942    0.55960  -2.894 0.003805 **
## vegsys4      -15.21785  888.05828  -0.017 0.986328
## vegsys5       -1.72746    0.77693  -2.223 0.026185 *
## vegsys6       -0.82025    1.15430  -0.711 0.477330
## vegsys7       -2.71096    0.80901  -3.351 0.000805 ***
## vegsys8      -17.60038 1597.10043  -0.011 0.991207
## vegsys9       -5.26220    0.84171  -6.252 4.06e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 873.37  on 10314  degrees of freedom
## Residual deviance: 448.52  on 10295  degrees of freedom
## AIC: 235.13
##
## Number of Fisher Scoring iterations: 15
```

```r
# model scope for subset selection
mod_scope <- list("cti" = ~1 + cti + poly(cti, 2),
                  "disturb" = ~1 + disturb + poly(disturb, 2),
```

```
                  "mi" = ~1 + mi + poly(mi, 2),
                  "rainann" = ~1 + rainann + poly(rainann, 2),
                  "raindq" = ~1 + raindq + poly(raindq, 2),
                  "rugged" = ~1 + rugged + poly(rugged, 2),
                  "soildepth" = ~1 + soildepth + poly(soildepth, 2),
                  "soilfert" = ~1 + soilfert + poly(soilfert, 2),
                  "solrad" = ~1 + solrad + poly(solrad, 2),
                  "tempann" = ~1 + tempann + poly(tempann, 2),
                  "topo" = ~1 + topo + poly(topo, 2),
                  "vegsys" = ~1 + vegsys)

tmp <- Sys.time()
set.seed(32639)
lm_subset <- step.Gam(object = lm1,
                      scope = mod_scope,
                      direction = "both",
                      data = training, # this is optional - see details below
                      trace = FALSE)
Sys.time() - tmp
```

```
## Time difference of 56.8871 secs
```

```
summary(lm_subset)
```

```
##
## Call:
## glm(formula = occ ~ poly(disturb, 2) + poly(mi, 2) + poly(rainann,
##     2) + raindq + rugged + vegsys, family = binomial(link = "logit"),
##     data = ..1, weights = wt, trace = FALSE)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -0.54153  -0.17830  -0.04768  -0.00912   2.13540
##
## Coefficients:
##                     Estimate Std. Error z value Pr(>|z|)
## (Intercept)          -0.8682     0.6372  -1.363 0.173034
## poly(disturb, 2)1   -84.1123    25.1332  -3.347 0.000818 ***
## poly(disturb, 2)2   -42.8549    19.8905  -2.155 0.031197 *
## poly(mi, 2)1        -10.0855    62.7815  -0.161 0.872374
## poly(mi, 2)2        -96.0440    41.8100  -2.297 0.021610 *
## poly(rainann, 2)1   337.6355    41.4932   8.137 4.05e-16 ***
## poly(rainann, 2)2   -68.1113    18.3056  -3.721 0.000199 ***
## raindq               -1.3751     0.2622  -5.244 1.57e-07 ***
## rugged               -0.6303     0.1739  -3.625 0.000289 ***
## vegsys2              -0.8490     0.5136  -1.653 0.098295 .
## vegsys3              -1.5931     0.5620  -2.835 0.004589 **
## vegsys4             -13.6715   778.5333  -0.018 0.985989
## vegsys5              -1.0845     0.7076  -1.533 0.125365
## vegsys6              -0.4391     1.1432  -0.384 0.700881
## vegsys7              -2.6227     0.8233  -3.186 0.001444 **
## vegsys8             -16.7025  1572.9139  -0.011 0.991528
## vegsys9              -4.0533     0.9151  -4.429 9.46e-06 ***
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 873.37  on 10314  degrees of freedom
## Residual deviance: 434.91  on 10298  degrees of freedom
## AIC: 221.49
##
## Number of Fisher Scoring iterations: 15
```

The `data` argument in the `step.Gam` function is optional. There is no option for it in the help file of this function. We found that when the `step.Gam` is running on a remote computer or when you are calling it by another user-defined function, you might receive an error with no clear explanation. You can fix this by adding the `data = training` argument.

Regularized regressions: lasso and ridge regression

To fit regularized regression, we used the `glmnet` package. There are a few parameters in this model that need to be set. The `alpha parameter` in this model ranges from **0** to **1**, where selecting an alpha of `0` leads to **ridge regression** and `1` to **lasso** and anything in between is a combination of both called **elastic-net**. The alpha parameter can be used as a tuning parameter. However, we chose to assess the performance of classic lasso and ridge regression (1 and 0 alpha respectively). The lambda parameter controls regularization – it is the penalty applied to the model's coefficients. To select the best **lambda**, internal cross-validation was used (in cv.glmnet function).

Data preparation

Unlike many other packages, `glmnet` package does not accept the input data as a *data.frame* and you need to convert the *data.frame* to a *matrix* or *sparse matrix*.

To make the orthogonal quadratic features for `glmnet` package (see more detail in the supplementary material of Guillera-Arroita *et al.* 2014), the `make_quadratic` function in `myspatial` package is used. The object generated by this function can be used in the generic `predict()` function to apply the transformation on the training and testing datasets and later used in predicting to rasters. The package `myspatial` is archived in GitHub and can be installed using the following code. The function is also provided in the supplementary materials.

```
# installing the package from github
remotes::install_github("rvalavi/myspatial")
```

```
# loading the library
library(glmnet)
library(myspatial)

# generating the quadratic terms for all continuous variables
# function to creat quadratic terms for lasso and ridge
quad_obj <- make_quadratic(training, cols = covars)
# now we can predict this quadratic object on the training and testing data
# this make two columns for each covariates used in the  transformation
training_quad <- predict(quad_obj, newdata = training)
testing_quad <- predict(quad_obj, newdata = testing_env)

# convert the data.frames to sparse matrices
# select all quadratic (and non-quadratic) columns, except the y (occ)
```

```
new_vars <- names(training_quad)[names(training_quad) != "occ"]
training_sparse <- sparse.model.matrix(~. -1, training_quad[, new_vars])
testing_sparse <- sparse.model.matrix( ~. -1, testing_quad[, new_vars])

# calculating the case weights
prNum <- as.numeric(table(training_quad$occ)["1"]) # number of presences
bgNum <- as.numeric(table(training_quad$occ)["0"]) # number of backgrounds
wt <- ifelse(training$occ == 1, 1, prNum / bgNum)
```

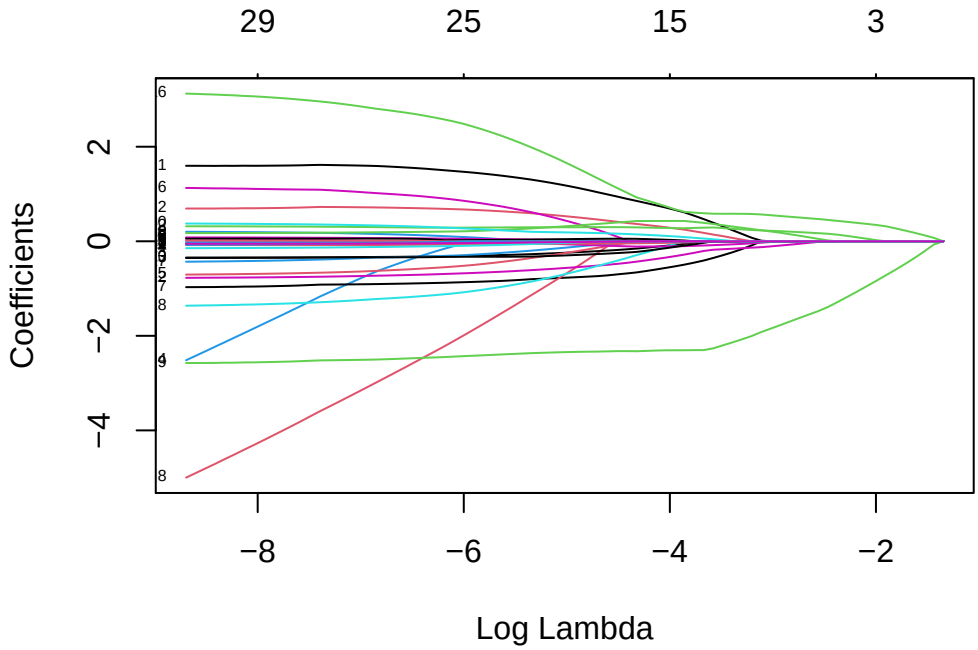
Fitting lasso and ridge

Now, we can fit *lasso GLM* and *ridge regression* with `glmnet` package.

```
# fitting lasso
tmp <- Sys.time()
set.seed(32639)
lasso <- glmnet(x = training_sparse,
                y = training_quad$occ,
                family = "binomial",
                alpha = 1, # here 1 means fitting lasso
                weights = wt)
Sys.time() - tmp
```

```
## Time difference of 1.940385 secs
```

```
# plot the regularization path and shrinkage in the coefficients
plot(lasso, xvar = "lambda", label = TRUE)
```

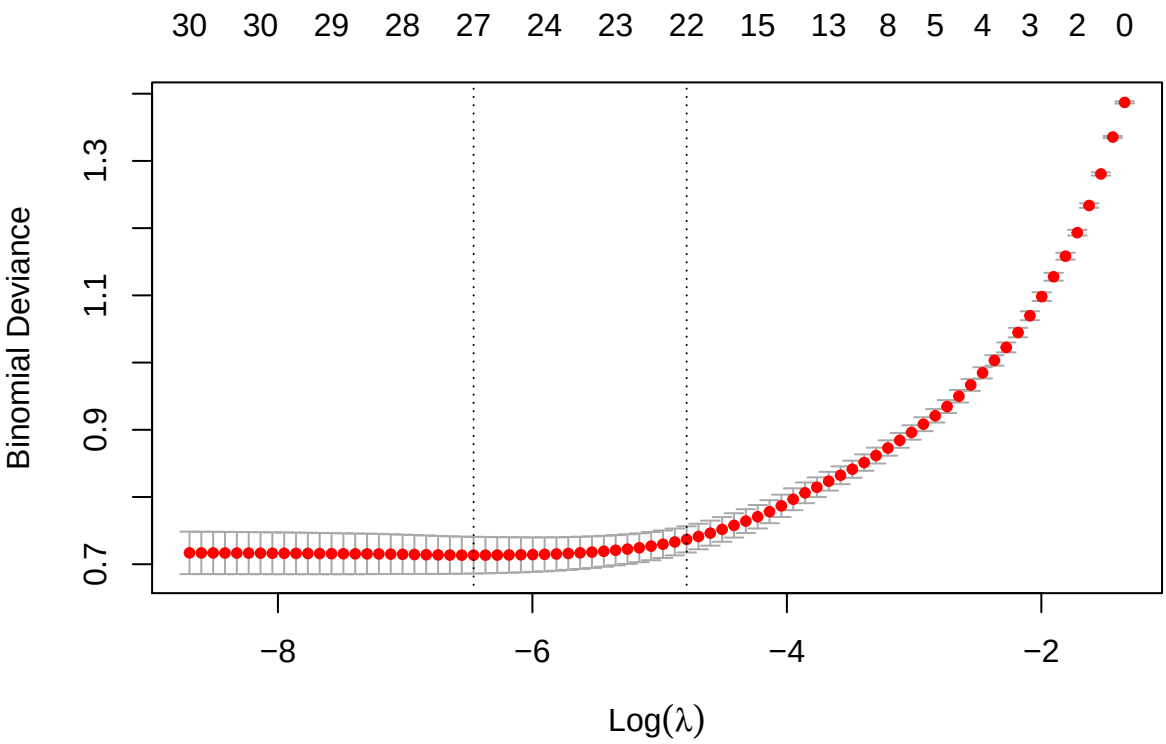


As you can see by changing (log) `Lambda` the coefficients shrink (the y-axes) and the number of covariates included in the model, decreases (x-axes, top) as the coefficients can be set to zeros in the lasso. To select the best `Lambda`, cross-validation is used. The following code shows how to do cross-validation to select this parameter in *lasso* (and *ridge*). This is the model object later used for predicting the testing data and on the raster files.

```
tmp <- Sys.time()
set.seed(32639)
lasso_cv <- cv.glmnet(x = training_sparse,
                      y = training_quad$occ,
                      family = "binomial",
                      alpha = 1, # fitting lasso
                      weights = wt,
                      nfolds = 10) # number of folds for cross-validation
Sys.time() - tmp
```

```
## Time difference of 15.77522 secs
```

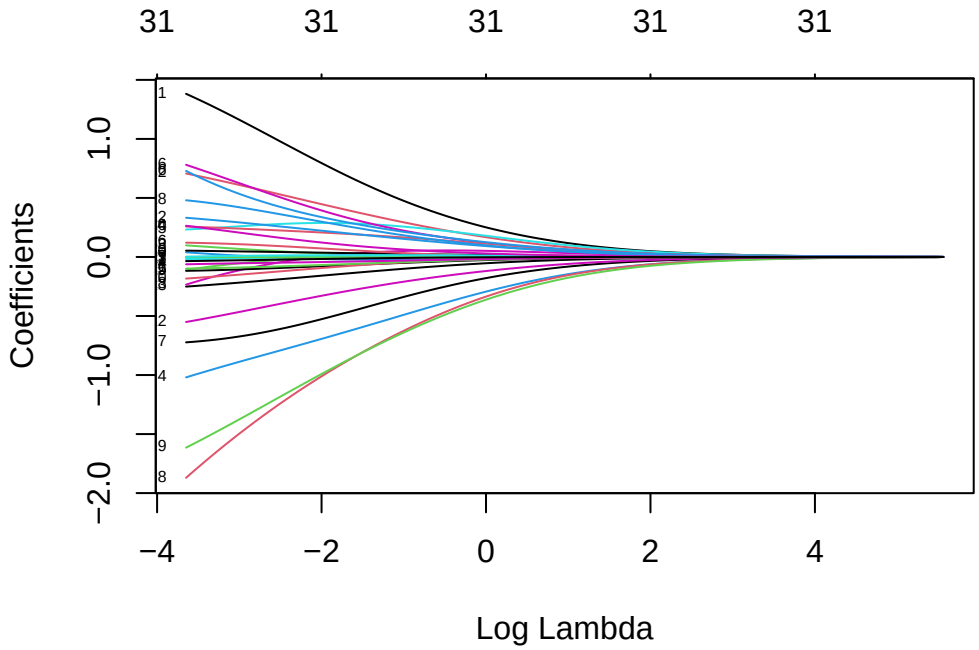
```
# the cross-validation result
plot(lasso_cv)
```



The two vertical dashed-lines show options for selecting the best `Lambda` parameter. The left one shows the `Lambda` that gives the minimum deviance and the right one shows the best `Lambda` within one standard deviation (1se) of the left dashed-line. One of these need to be selected when predicting with the model. As you can see from the top x-axes, the `Lambda` with the minimum deviance will select only 26 of the terms (quadratic and categorical covariates) and with the 1se one, 21 of the terms will be used for prediction.

For fitting ridge regression, the same code is used except the `alpha` parameter is set to `0`.

```
# fitting ridge resgression
set.seed(32639)
ridge <- glmnet(x = training_sparse,
                y = training_quad$occ,
                family = "binomial",
                alpha = 0, # here 1 means fitting ridge regression
                weights = wt)
# plot the regularization path and shrinkage in the coefficients
plot(ridge, xvar = "lambda", label = TRUE)
```



The shrinkage in ridge regression does not lead to removing the coefficients. So, all the variables will stay in the model (x-axes, top). Similar to lasso, a value of `Lambda` should be selected that gives the lowest error or deviance. This is done with `cv.glmnet` function again. We do not repeat it here.

Predicting on test data

For predicting with *ridge* or *lasso*, you only need their `cv.glmnet` objects. A value of `Lambda` must be selected by the `s` argument. You should either select the minimum cv deviance (left dashed-line in the `cv.glmnet` plot) or the one-standard-deviation (right dashed-line) by setting the `s = "lambda.min"` or `s = "lambda.1se"` respectively. We used the minimum cv deviance for our analysis, although the package authors recommended to used one-standard-deviation.

```
# predicting with lasso model
lasso_pred <- predict(lasso_cv, testing_sparse, type = "response", s = "lambda.min")
head(lasso_pred)
```

```
##           1
## 1 0.8076853
## 2 0.8095829
## 3 0.9305091
## 4 0.9161369
## 5 0.9758157
## 6 0.9669616
```

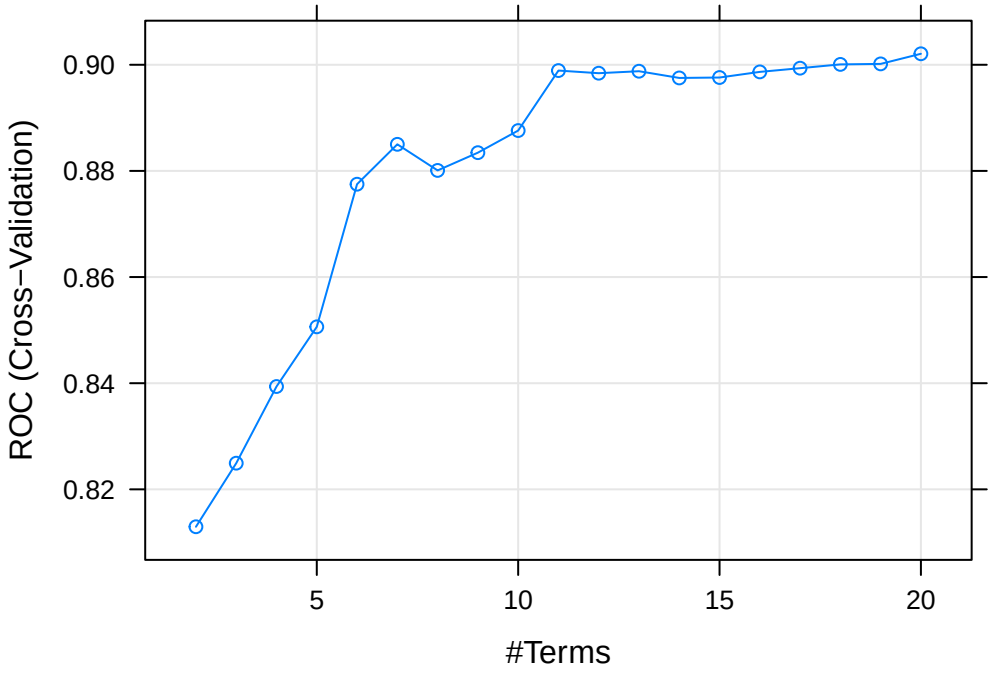
MARS

To fit MARS models, we used the `earth` package. The parameter tuning for this model was done through the `caret` package. There are two tuning parameters in MARS, *interaction level* and *n prone* (the maximum number of model terms). We chose the MARS model with *no interaction* as it showed by Elith 2006 the interaction terms decrease the accuracy of the model. So, we only tune the `nrpone` parameter. Parallel processing is used to make the model tuning process faster.

```r
library(caret)
library(earth)
library(doParallel)

# change the response to factor variables with non-numeric levels
training$occ <- as.factor(training$occ)
levels(training$occ) <- c("C0", "C1")

mytuneGrid <- expand.grid(nprune = 2:20,
                          degree = 1) # no interaction

mytrControl <- trainControl(method = "cv",
                            number = 5, # 5-fold cross-validation
                            classProbs = TRUE,
                            summaryFunction = twoClassSummary,
                            allowParallel = TRUE)


tmp <- Sys.time()

cluster <- makeCluster(6) # you can use all cores of your machine instead e.g. 8
registerDoParallel(cluster)
set.seed(32639)
mars_fit <- train(form = occ ~ .,
                  data = training,
                  method = "earth",
                  metric = "ROC",
                  trControl = mytrControl,
                  tuneGrid = mytuneGrid,
                  thresh = 0.00001)
stopCluster(cluster)
registerDoSEQ()

Sys.time() - tmp
```

```
## Time difference of 1.060728 mins
```

```r
plot(mars_fit)
```



MaxEnt and MaxNet

The *MaxEnt* model is fitted with `dismo` package. There are two implementations of the MaxEnt model in the paper, *MaxEnt* and *MaxEnt tuned*. The latter was tested because some studies showed that MaxEnt can perform better when its **regularization multiplier** and **feature classes** (i.e., L, Q, H and P for *linear*, *quadratic*, *hinge* and *product*, respectively) is tuned (e.g. see Muscarella *et al.*, 2014; Radosavljevic & Anderson, 2014). These parameters can control the *complexity* or *generality* of the model. The *default* value of regularization multiplier is `1` and the lower the value, the more complex model can be fitted (Elith *et al.*, 2011). Feature types are selected based on the number of presence records by default (see Elith *et al.*, 2011). We simultaneously tuned regularization multiplier and feature classes using a 5-fold cross-validation (CV) on presence-background training data. We used five different regularization multipliers (`0.5, 1, 2, 3` and `4`) in combination with different features (L, LQ, H, LQH, LQHP) to find the best parameters that maximizes the average AUC_{ROC} in CV. We used stratified CV to have equal number of background and presences in the CV folds. You can use

There are some packages that tune MaxEnt. However, we wrote the following code to do that.

```
# function for simultaneous tuning of maxent regularization multiplier and features
maxent_param <- function(data, y = "occ", k = 5, folds = NULL, filepath){
  require(dismo)
  require(caret)
  require(precrec)
  if(is.null(folds)){
    # generate balanced CV folds
    folds <- caret::createFolds(y = as.factor(data$occ), k = k)
  }
  names(data)[which(names(data) == y)] <- "occ"
  covars <- names(data)[which(names(data) != y)]
  # regularization multipliers
  ms <- c(0.5, 1, 2, 3, 4)
  grid <- expand.grid(
    regmult = paste0("betamultiplier=", ms),
```

```
    features = list(
      c("noautofeature", "nothreshold"), # LQHP
      c("noautofeature", "nothreshold", "noproduct"), # LQH
      c("noautofeature", "nothreshold", "nohinge", "noproduct"), # LQ
      c("noautofeature", "nothreshold", "nolinear", "noquadratic", "noproduct"), # H
      c("noautofeature", "nothreshold", "noquadratic", "nohinge", "noproduct")), # L
    stringsAsFactors = FALSE
    )
  AUCs <- c()
  for(n in seq_along(grid[,1])){
    full_pred <- data.frame()
    for(i in seq_len(length(folds))){
      trainSet <- unlist(folds[-i])
      testSet <- unlist(folds[i])
      if(inherits(try(
        maxmod <- dismo::maxent(x = data[trainSet, covars],
                                p = data$occ[trainSet],
                                removeDuplicates = FALSE,
                                path = filepath,
                                args = as.character(unlist(grid[n, ]))
        )
      ), "try-error")){
        next
      }
      modpred <- predict(maxmod, data[testSet, covars], args = "outputformat=cloglog")
      pred_df <- data.frame(score = modpred, label = data$occ[testSet])
      full_pred <- rbind(full_pred, pred_df)
    }
    AUCs[n] <- precrec::auc(precrec::evalmod(scores = full_pred$score,
                                             labels = full_pred$label))[1,4]
  }
  best_param <- as.character(unlist(grid[which.max(AUCs), ]))
  return(best_param)
}
```

Now, we use the function to tune MaxEnt.

```
# load the package
library(dismo)

# number of folds
nfolds <- ifelse(sum(training$occ) < 10, 2, 5)

tmp <- Sys.time()
set.seed(32639)
# tune maxent parameters
param_optim <- maxent_param(data = training,
                            k = nfolds,
                            filepath = "output/maxent_files")
# fit a maxent model with the tuned parameters
maxmod <- dismo::maxent(x = training[, covars],
                        p = training$occ,
                        removeDuplicates = FALSE,
                        path = "output/maxent_files",
```

```
                        args = param_optim)
Sys.time() - tmp
```

Fitting `MaxNet` is very similar to the `MaxEnt` model. For MaxNet, we kept the default parameters.

```
library(maxnet)

presences <- training$occ # presence (1s) and background (0s) points
covariates <- training[, 2:ncol(training)] # predictor covariates

tmp <- Sys.time()
set.seed(32639)
mxnet <- maxnet::maxnet(p = presences,
                        data = covariates,
                        regmult = 1, # regularization multiplier
                        maxnet.formula(presences, covariates, classes = "default"))
Sys.time() - tmp
```

```
## Time difference of 16.59158 secs
```

```
# predicting with MaxNet
maxnet_pred <- predict(mxnet, testing_env, type = c("cloglog"))
head(maxnet_pred)
```

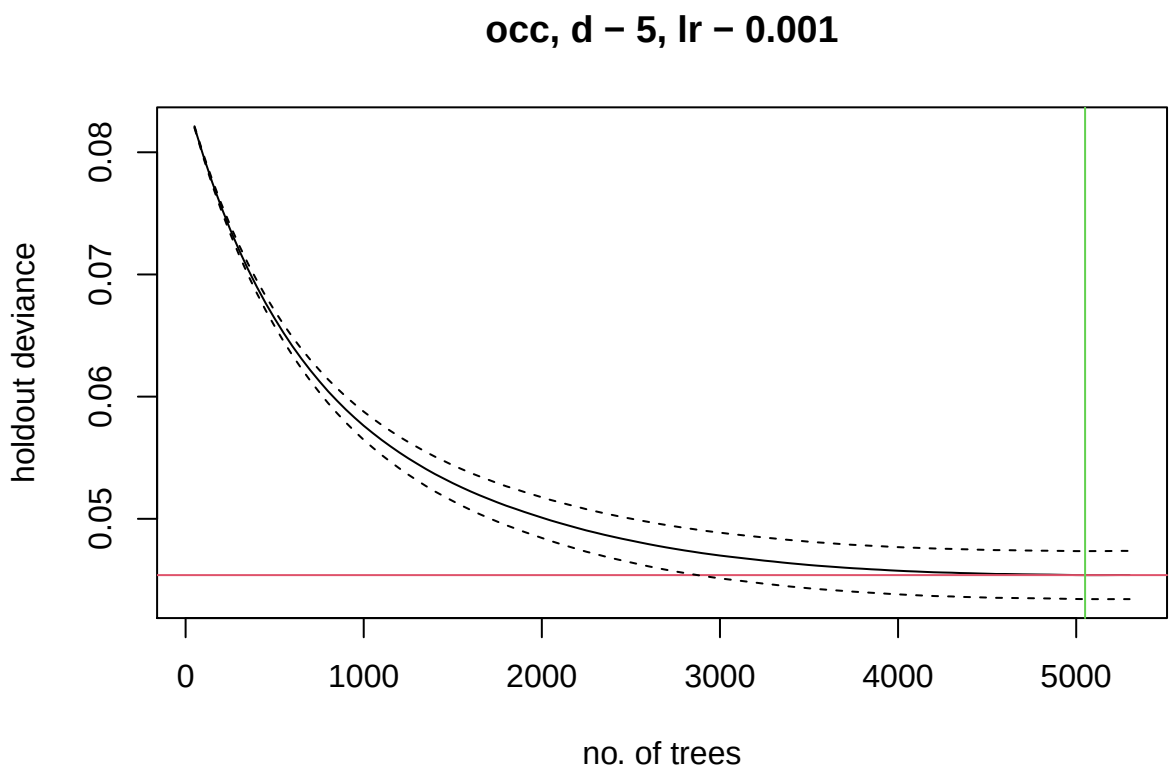
```
##        [,1]
## 1 0.6075664
## 2 0.4931290
## 3 0.8779032
## 4 0.8878335
## 5 0.9998653
## 6 1.0000000
```

BRT (GBM)

Boosted regression trees (BRT) or *Gradient Boosting Machine (GBM)* applies a boosting algorithm to decision trees. We fit the BRT model with a *forward stepwise cross-validation* approach (Elith *et al.,* 2008) implemented in the `dismo` package. BRT has a stagewise procedure, meaning that every tree is built on the residual of the previously fitted trees. For model tuning, in each round (stage) of model fitting, a specified number of trees (here `n.trees = 50`) are added to the previously grown trees and, as the ensemble grows, k-fold cross-validation (here, `n.folds = 5`) is used to estimate the optimal number of trees while keeping the other parameters constant. Other settings were selected according to the recommendations in the literature (see Elith *et al.,* 2008); we aimed to fit final models with at least `1000` trees, but did not specifically test / iterate for that.

```
library(dismo)

# calculating the case weights
prNum <- as.numeric(table(training$occ)["1"]) # number of presences
bgNum <- as.numeric(table(training$occ)["0"]) # number of backgrounds
wt <- ifelse(training$occ == 1, 1, prNum / bgNum)

tmp <- Sys.time()
set.seed(32639)
brt <- gbm.step(data = training,
                gbm.x = 2:ncol(training), # column indices for covariates
```

```
                gbm.y = 1, # column index for response
                family = "bernoulli",
                tree.complexity = ifelse(prNum < 50, 1, 5),
                learning.rate = 0.001,
                bag.fraction = 0.75,
                max.trees = 10000,
                n.trees = 50,
                n.folds = 5, # 5-fold cross-validation
                site.weights = wt,
                silent = TRUE) # avoid printing the cv results
```




```
Sys.time() - tmp
```

```
## Time difference of 8.800018 mins
```

The vertical green line shows the best number of trees, where the holdout deviance converges (the horizontal red line).

```
# the optimal number of trees
brt$gbm.call$best.trees
```

```
## [1] 5050
```

```
# predicting with the best trees
brt_pred <- predict(brt, testing_env, n.trees = brt$gbm.call$best.trees, type = "response")
head(brt_pred)
```

```
## [1] 0.8931857 0.8908768 0.8874179 0.8697575 0.9221985 0.9200913
```

XGBoost

We fitted the `XGBoost` model with `XGBoost` and `caret` packages. `XGBoost` has many parameters that need model tuning (see Muñoz-Mas, R., *et al.,* 2019 for more details and the ranges of suitable values for model tuning). Due to computational limitations, we set all the parameters the same as selected for **BRT** and only tuned the *nround* parameter (the number of fitted trees). We also add an extra source of variability (compared to BRT) in the trees fitting by only using 80% of the covariates in building each tree. This is done by *colsample_bytree* argument.

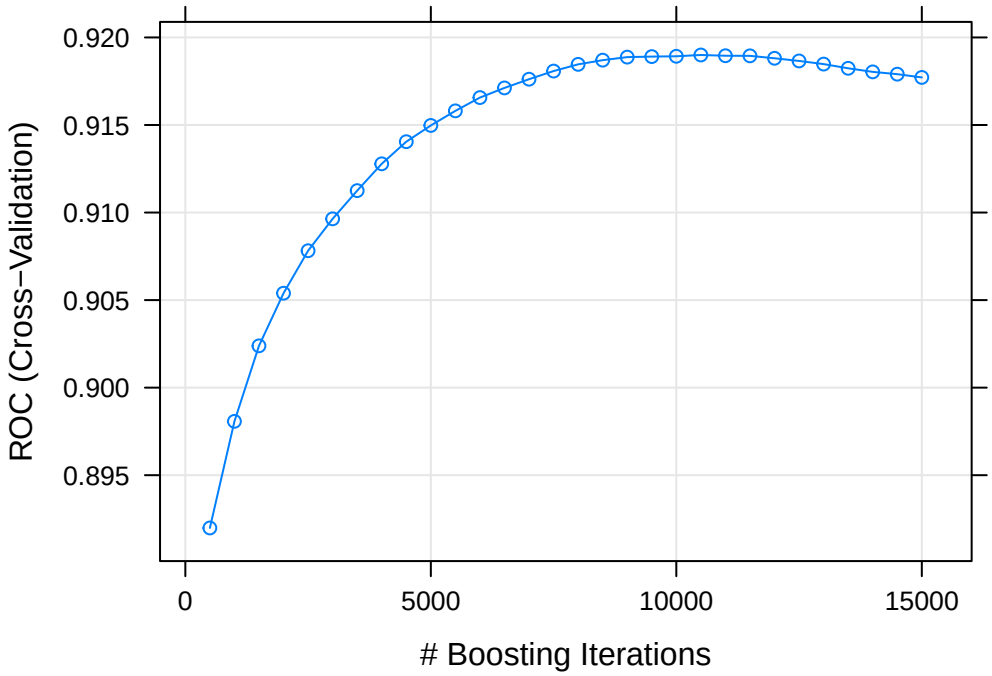
You can tune all the parameters in a *grid search* hyper-parameter tuning. In this way, the models are run with all combination of the parameters. This may take a very long time depending on your data and your machine. Parallel processing can be used to speed up the computation. We also tried to tune XGBoost using *random search* tuning with 50 & 500 tune length. The result was not better than the default parameters.

```
# loading required libraries
library(caret)
library(xgboost) # just to make sure this is installed
library(doParallel)

# change the response to factor variables with non-numeric levels
training$occ <- as.factor(training$occ)
levels(training$occ) <- c("C0", "C1") # caret does not accept 0 and 1 as factor levels

# train control for cross-validation for model tuning
mytrControl <- trainControl(method = "cv",
                            number = 5, # 5 fold cross-validation
                            summaryFunction = twoClassSummary,
                            classProbs = TRUE,
                            allowParallel = TRUE)

# setting the range of parameters for grid search tuning
mytuneGrid <- expand.grid(
  nrounds = seq(from = 500, to = 15000, by = 500),
  eta = 0.001,
  max_depth = 5,
  subsample = 0.75,
  gamma = 0,
  colsample_bytree = 0.8,
  min_child_weight = 1
)

tmp <- Sys.time()

cluster <- makeCluster(6) # you can use all cores of your machine instead e.g. 8
registerDoParallel(cluster)

set.seed(32639)
xgb_fit <- train(form = occ ~ .,
                 data = training,
                 method = "xgbTree",
                 metric = "ROC",
                 trControl = mytrControl,
                 tuneGrid = mytuneGrid,
                 verbose = TRUE)
```

```
stopCluster(cluster)
registerDoSEQ()

Sys.time() - tmp
```

```
## Time difference of 8.504614 mins
```

```
plot(xgb_fit)
```




```
print(xgb_fit$bestTune)
```

```
##    nrounds max_depth   eta gamma colsample_bytree min_child_weight subsample
## 21   10500         5 0.001     0              0.8                1      0.75
```

Model tuning for the same species with 50,000 background points took around 1 hour, although we only have one parameter to tune (`nrounds`).

cforest

The `party` package was used fotr fitting `cforest` models (and `cforest weighted`). The `cforest` algorithm is designed for *revealing the most important variables*. This needs relatively high `mtry` (the number of random variables that can be selected on each split) to give enough opportunity for variables to be selected on each split (the default is `mtry = 5`). However, high `mtry` makes *more similar trees* leading to *higher variance* in the RF model. As our purpose is higher accuracy of *prediction* rather than variable importance, we used lower mtry (`sqrt` of the number of predictors, similar to `randomForest` default) to achieve higher prediction accuracy. Other `mtry` values can be tested to improve the accuracy.

```
library(party)

# convert the response to factor for producing class probabilities
training$occ <- as.factor(training$occ)

# the number of predictors and mtry
```

```
nvars <- ncol(training) - 1 # excluding the response
mtry <-  round(sqrt(nvars))

# calculate weights for each class
prNum <- as.numeric(table(training$occ)["1"]) # number of presences
bgNum <- as.numeric(table(training$occ)["0"]) # number of backgrounds
wt <- ifelse(training$occ == 1, 1, prNum / bgNum)

tmp <- Sys.time()
set.seed(32639)
cfparty <- cforest(formula = occ ~.,
                   data = training,
                   weights = wt,
                   controls = cforest_unbiased(ntree = 500,
                                               mtry = mtry))
Sys.time() - tmp
```

```
## Time difference of 2.125252 mins
```

The process of fitting unbiased trees is costly and as a result creating ensembles of many `ctrees` in `cforest` is computationally extensive. This is also true when the fitted model is used to predict new data. It almost takes the same time as the model fitting process.

```
# predict with cforest
cfpred <- predict(cfparty, newdata = testing_env, type = "prob", OOB = TRUE)
```

RF and RF down-sampled

To fit *Random Forest (RF)* models and RF with **down-sampling**, the `randomForest` package was used. RF can be run either in regression or classification. With binary data, classification is commonly used. To use the RF with classification, the response should be converted to factors. We used the default `mtry` parameter for both RF and RF down-sampled. We used higher number of bootstrap samples (iterations = final number of trees in the model; the default = 500) for RF down-sampled. For predicting continuous values rather than classes, use `type = "prob"`. For presence-background data the output will be the relative likelihood of both classes (0 and 1).

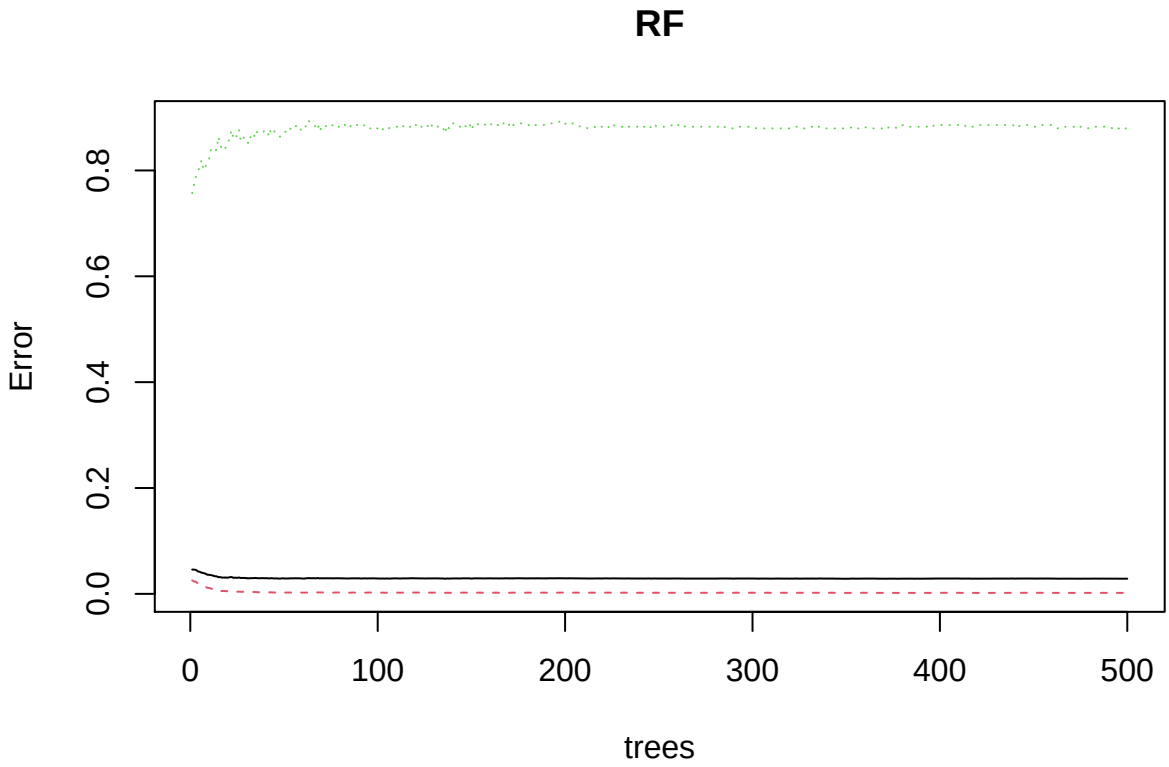
```
# loading the package
library(randomForest)

# convert the response to factor for producing class relative likelihood
training$occ <- as.factor(training$occ)

tmp <- Sys.time()
set.seed(32639)
rf <- randomForest(formula = occ ~.,
                   data = training,
                   ntree = 500) # the default number of trees
Sys.time() - tmp
```

```
## Time difference of 11.14968 secs
```

```
# predict with RF and RF down-sampled
rfpred <- predict(rf, testing_env, type = "prob")
head(rfpred)
```

```
##       0     1
## 1 0.290 0.710
## 2 0.404 0.596
## 3 0.916 0.084
## 4 0.866 0.134
## 5 0.606 0.394
## 6 0.398 0.602
```

```
plot(rf, main = "RF")
```

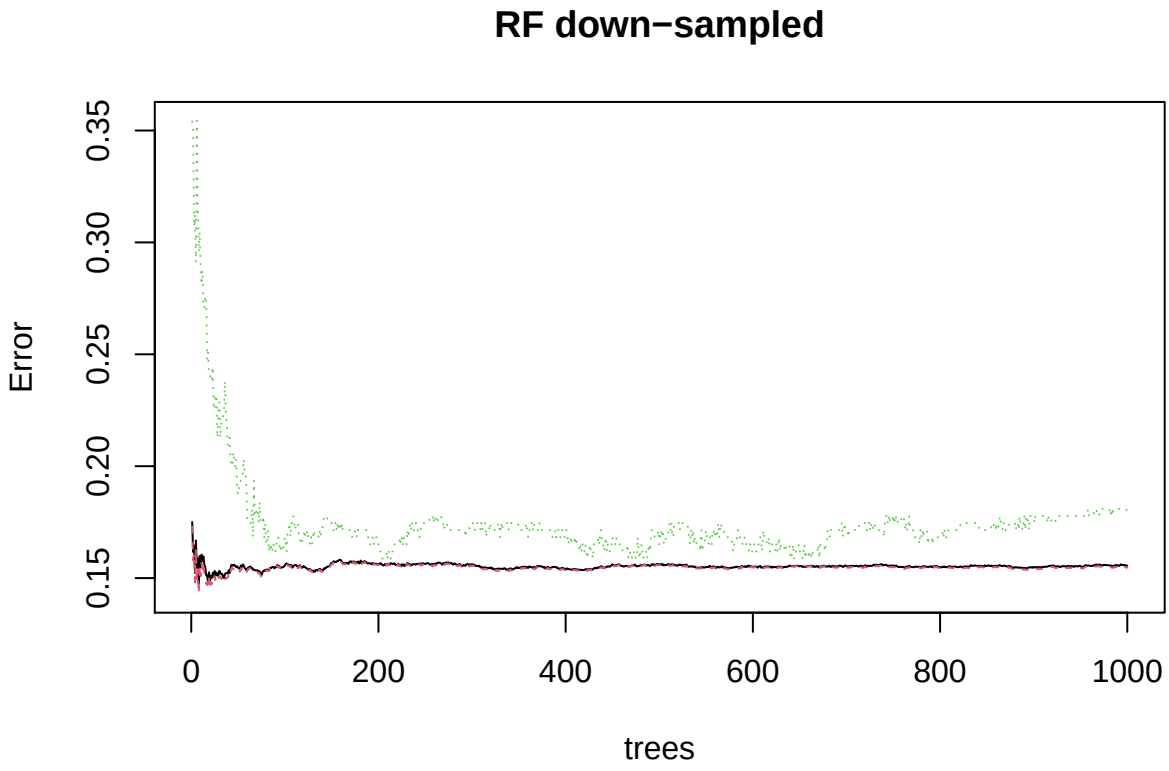


For down-sampling, the number of background (0s) in each bootstrap sample should the same as presences (1s). For this, we use `sampsize` argument to do this.

```
prNum <- as.numeric(table(training$occ)["1"]) # number of presences
bgNum <- as.numeric(table(training$occ)["0"]) # number of backgrounds
# the sample size in each class; the same as presence number
smpsize <- c("0" = prNum, "1" = prNum)

tmp <- Sys.time()
set.seed(32639)
rf_downsample <- randomForest(formula = occ ~.,
                              data = training,
                              ntree = 1000,
                              sampsize = smpsize,
                              replace = TRUE)
Sys.time() - tmp
```

```
## Time difference of 6.173524 secs
```

```
plot(rf_downsample, main = "RF down-sampled")
```



SVM

There are two packages to fit *SVM* with, `e1071` and `kernlab`. We used the `e1071` for our study, but the `kernlab` is easier when you want to predict to rasters. Their results are quite similar.

Similar to `randomForest`, the `e1071::svm` accepts class weights rather than case weights. We used class weights (1 and the ratio of presence/background for presences and backgrounds, respectively) to increase the cost of miss-classification rate on the presence points rather than the background. Parameters `c` and `gamma` are two tuning parameters that can further improve the model's performance. Due to computation limits, we chose to not tune them in our study. For more detail about how to tune them see James *et al.,* (2013) pages 363-364.

```
library(e1071)

# change the response to factor
training$occ <- as.factor(training$occ)

# calculate weights the class weight
prNum <- as.numeric(table(training$occ)["1"]) # number of presences
bgNum <- as.numeric(table(training$occ)["0"]) # number of backgrounds
cwt <- c("0" = prNum / bgNum, "1" = 1)

tmp <- Sys.time()
set.seed(32639)
svm_e <- e1071::svm(formula = occ ~ .,
                    data = training,
                    kernel = "radial",
```

```
                    class.weights = cwt,
                    probability = TRUE)
Sys.time() - tmp
```

```
## Time difference of 55.65968 secs
```

```
# predicting on test data
svm_pred <- predict(svm_e, testing_env, probability = TRUE)
svm_prob <- attr(svm_pred, "probabilities")[,"1"]

# see the first few predictions
head(svm_prob)
```

```
##         1         2         3         4         5         6
## 0.1304283 0.1465849 0.1706320 0.1032008 0.2616781 0.1783190
```

biomod

`biomod` is a package specifically written for modeling species distributions in R (Thuiller *et al.,* 2009), allowing ensembles across several modeling methods. Many users apply this model with default parameters (Hao *et al.* 2019). You can change the parameters of each model separately by `BIOMOD_ModelingOptions()` function.

To avoid GAM models producing errors when a term has fewer unique covariate combinations than the default maximum degrees of freedom (`K`), we used our own formula for `GAM` (`myFormula = as.formula(form)` see GAM section). This (and the other methods) were otherwise fitted with the *default parameters* to test common usage of `biomod`. We needed to specify that `maxent` use all **50,000** background points (`maximumbackground = 50000`), so it used all provided data (see below; we used 10,000 in this example).

For creating `biomodDataFormat` the `x` and `y` spatial coordinates are required. So, we need to reload the species data as we filtered out the coordinates for the other models. Here we used the `data.frame` of the training set, although `biomod2` allows for raster files as an input and can select the background sample for you. To have a fair comparison with the other models, we used the same set of 50,000 background samples (10,000 in this example).

```
library(biomod2)

# specifying the species id
spID <- "nsw14"

# re-loading the species data
pr <- disPo("NSW")
bg <- disBg("NSW")
pr <- pr[pr$spid == spID, ] # subset the target species
training <- rbind(pr, bg)
training$vegsys <- as.factor(training$vegsys)

myRespName <- "occ"
myResp <- as.numeric(training[, myRespName])
myResp[which(myResp == 0)] <- NA
myExpl <- data.frame(training[, covars])
myRespXY <- training[, c("x", "y")]

# create biomod data format
myBiomodData <- BIOMOD_FormatingData(resp.var = myResp,
                                     expl.var = myExpl,
```

```
                                    resp.name = myRespName,
                                    resp.xy = myRespXY,
                                    PA.nb.absences = 10000,
                                    PA.strategy = 'random',
                                    na.rm = TRUE)

# using the default options
# you can change the mentioned parameters by changes this
myBiomodOption <- BIOMOD_ModelingOptions()

# models to predict with
mymodels <- c("GLM","GBM","GAM","CTA","ANN","FDA","MARS","RF","MAXENT.Phillips")

# model fitting
tmp <- Sys.time()
set.seed(32639)
myBiomodModelOut <- BIOMOD_Modeling(myBiomodData,
                                    models = mymodels,
                                    models.options = myBiomodOption,
                                    NbRunEval = 1,
                                    DataSplit = 100, # use all the data for training
                                    models.eval.meth = c("ROC"),
                                    SaveObj = TRUE,
                                    rescal.all.models = FALSE,
                                    do.full.models = TRUE,
                                    modeling.id = paste(myRespName,"NCEAS_Modeling", sep = ""))
# ensemble modeling using mean probability
myBiomodEM <- BIOMOD_EnsembleModeling(modeling.output = myBiomodModelOut,
                                      chosen.models = 'all',
                                      em.by = 'all',
                                      eval.metric = c("ROC"),
                                      eval.metric.quality.threshold = NULL, # since some species's auc
                                      prob.mean = TRUE,
                                      prob.cv = FALSE,
                                      prob.ci = FALSE,
                                      prob.median = FALSE,
                                      committee.averaging = FALSE,
                                      prob.mean.weight = FALSE)
```

```
Sys.time() - tmp
```

```
## Time difference of 4.348362 mins
```

```
# project single models
myBiomodProj <- BIOMOD_Projection(modeling.output = myBiomodModelOut,
                                  new.env = as.data.frame(testing_env[, covars]),
                                  proj.name = "nceas_modeling",
                                  selected.models = "all",
                                  binary.meth = "ROC",
                                  compress = TRUE,
                                  clamping.mask = TRUE)

# project ensemble of all models
myBiomodEnProj <- BIOMOD_EnsembleForecasting(projection.output = myBiomodProj,
```

```
                                                EM.output = myBiomodEM,
                                                selected.models = "all")
```

```
# extracting the values for ensemble prediction
myEnProjDF <- as.data.frame(get_predictions(myBiomodEnProj))

# see the first few pridictions
# the prediction scale of biomod is between 0 and 1000
head(myEnProjDF)
```

```
##   occ_EMmeanByROC_mergedAlgo_mergedRun_mergedData
## 1                                             234
## 2                                             234
## 3                                             236
## 4                                             236
## 5                                             235
## 6                                             235
```

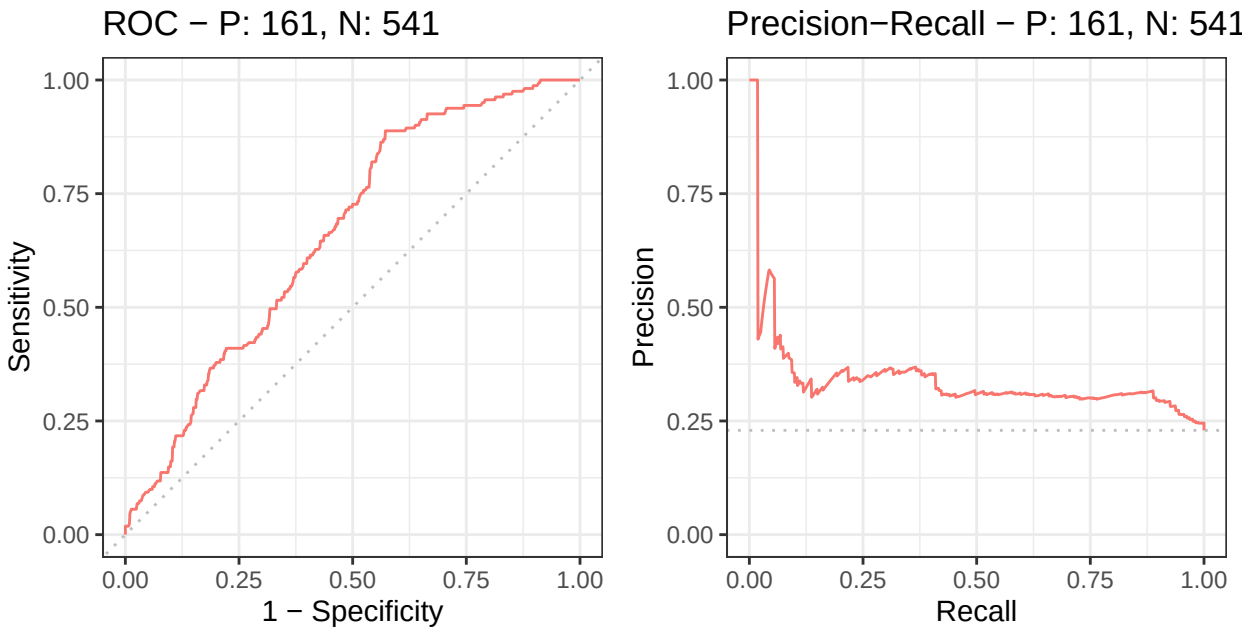
Generating ROC and Precision-Recall Gain (PRG) curves

To plot and calculate the area under the ROC and PR curves automatically, the `precrec` package can be used. Here, we show the performance of the RF down-sampled on our test dataset. You need the relative likelihood predicted on the test data by a model, and the corresponding value of presence/absence (0 for absences and 1 for presences) to calculate the ROC and PR curves.

```
# laod the packages
library(precrec)
library(ggplot2) # for plotting the curves

# predict RF down-sample on the test data
prediction <- predict(rf_downsample, testing_env, type = "prob")[, "1"]

# calculate area under the ROC and PR curves
precrec_obj <- evalmod(scores = prediction, labels = testing_pa[,spID])
print(precrec_obj)
```

```
##
##     === AUCs ===
##
##      Model name Dataset ID Curve type       AUC
##    1         m1          1        ROC 0.6612094
##    2         m1          1        PRC 0.3407269
##
##
##     === Input data ===
##
##      Model name Dataset ID # of negatives # of positives
##    1         m1          1             541            161
```

```
# plot the ROC and PR curves
autoplot(precrec_obj)
```

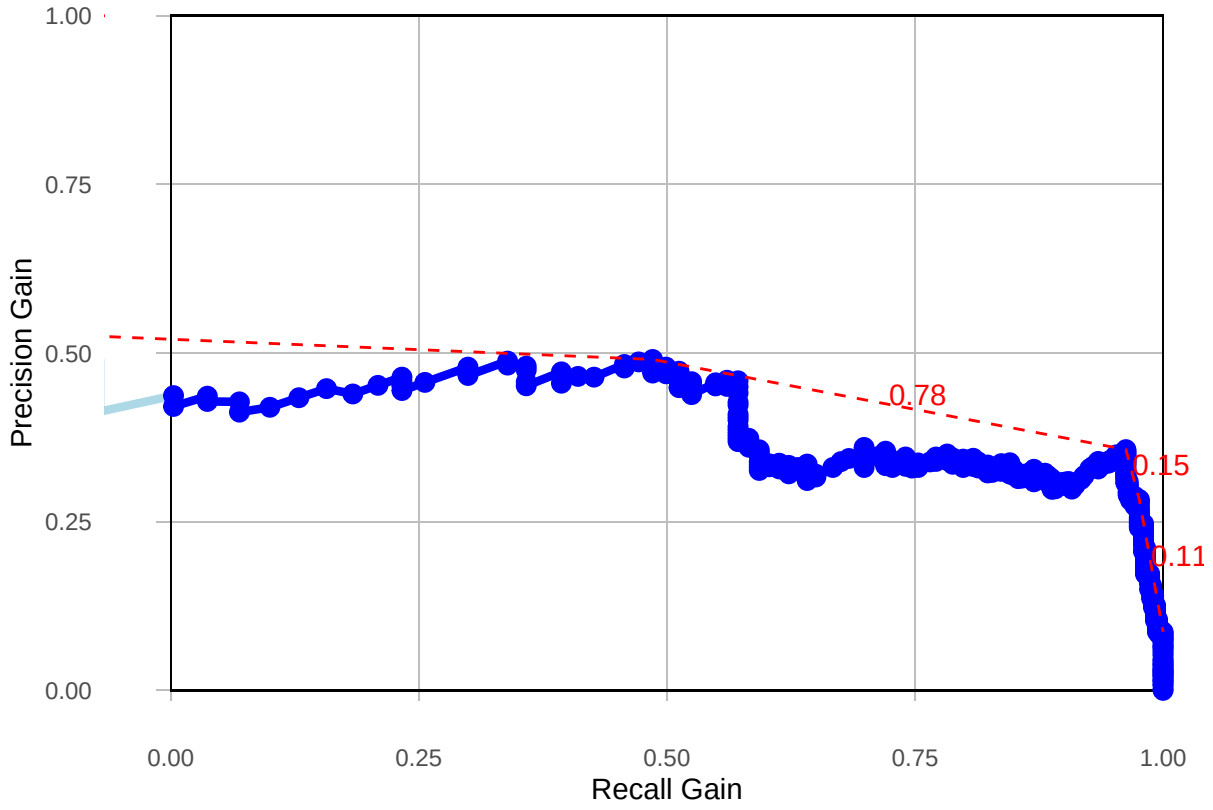


For calculating the **Precision-Recall Gain** curve (Flach & Kull, 2015), the `prg` package is used. This package is available here: https://github.com/meeliskull/prg/tree/master/R_package.

```
# install the prg package
# devtools::install_github("meeliskull/prg/R_package/prg")
library(prg)

# calculate the PRG curve for RF down-sampled
prg_curve <- create_prg_curve(labels = testing_pa[,spID], pos_scores = prediction)
# calculate area under the PRG cure
au_prg <- calc_auprg(prg_curve)
print(au_prg)
```

```
## [1] 0.3970376
```

```
# plot the PRG curve
plot_prg(prg_curve)
```

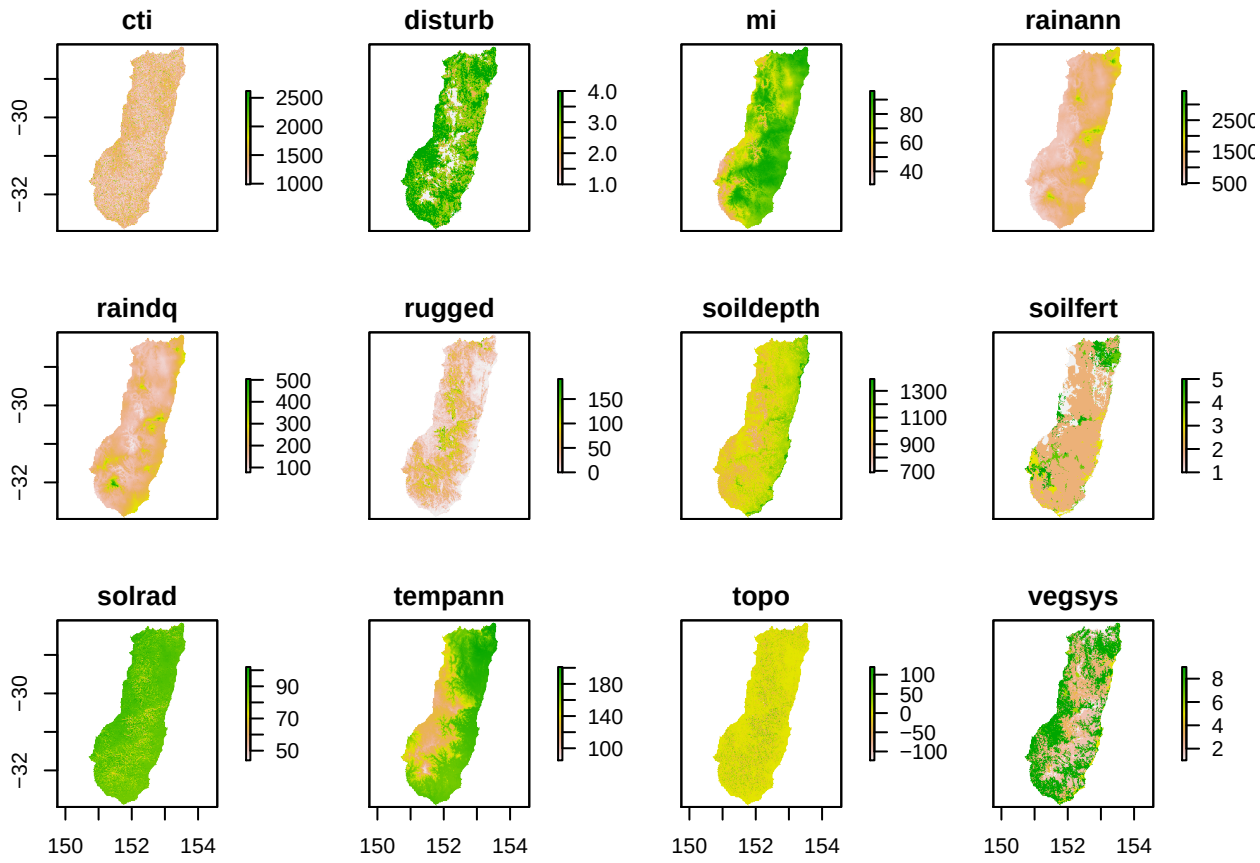


Making prediction maps

Here we demonstrate how to predict the fitted models on rasters with the `raster` package. Most of the models can predict to rasters with generic `predict()` function in the `raster` package. However, the prediction of some models is not very straight forward. We provide code in `myspatial` package for predicting `glmnet` and `svm` models. These codes are also provided in the supplementary materials. The `predict_glmnet_raster()` and `predict_svm_raster()` functions are used for predicting with `cv.glmnet` object and SVM from the `e1071` package. See the examples section in the help file for the `predict` function in the `raster` package to see how to predict the model fitted with `cforest` on rasters (use `?raster::predict`).

The rasters should be loaded first, then they should be normalized in the same way the training data was normalized. The *mean* and *standard deviation* of the training data should be used for normalising the rasters as well. The raster covariates for all the regions are provided in Elith *et al.,* (2020).

```
# laod the packages
library(raster)
library(myspatial)

# load the raster files
li <- list.files("data/grids", pattern = ".tif$", full.names = TRUE)
r <- stack(li)
plot(r)
```

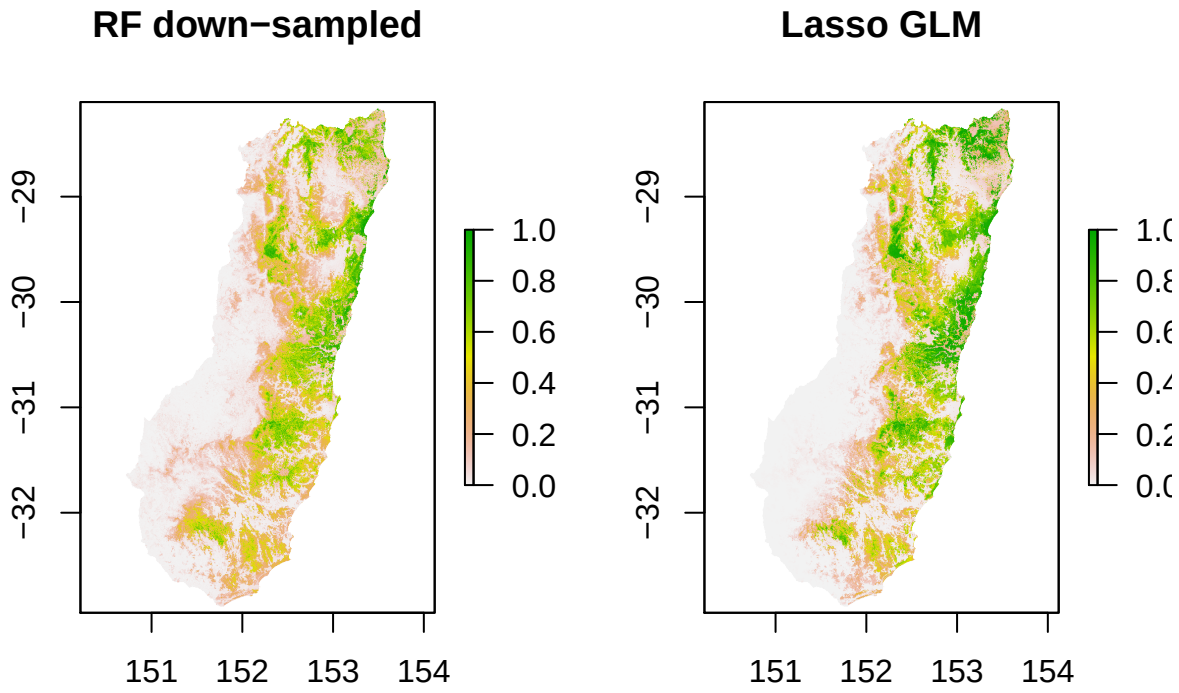



```
# the rasters should be normalized with the mean and sd of the training data
# re-loading the training data to get the mean and standard deviation
pr <- disPo("NSW")
bg <- disBg("NSW")
pr <- pr[pr$spid == spID, ] # subset the target species
training <- rbind(pr, bg)

# normalize the covariates (exept vegsys which is categorical)
normr <- stack()
for(v in covars[covars!="vegsys"]){
  meanv <- mean(training[,v])
  sdv <- sd(training[,v])
  normr <- stack(normr, (r[[v]] - meanv) / sdv)
}
# add the categorical covariates
normr <- stack(normr, r[["vegsys"]])

# providing the factor covariates for prediction
facts <- list(vegsys = levels(as.factor(training$vegsys)))

# predicting RF down-sampled on rasters with raster package
# use index = 2 for the likelihood of presence (1s)
rf_map <- predict(object = normr,
                  model = rf_downsample,
                  type = "prob",
                  index = 2,
                  factors = facts)
```

```
# predicting glment on rasters with myspatial package
lasso_map <- predict_glmnet_raster(r = normr,
                                   model = lasso_cv, # the lasso cv object
                                   quadraticObj = quad_obj, # make_quadratic object
                                   type = "response",
                                   slambda = "lambda.min",
                                   factors = facts)
```

```
## Preparation is done...
## Finalising...
```

```
# plot the result
par(mfrow = c(1,2))
plot(rf_map, zlim = c(0,1), main = "RF down-sampled")
plot(lasso_map, zlim = c(0,1), main = "Lasso GLM")
```



References

Elith, J., Graham, C.H., Anderson, R.P., Dudík, M., Ferrier, S., Guisan, A., J Hijmans, R., Huettmann, F., R Leathwick, J., Lehmann, A., Li, J., G Lohmann, L., Overton, J.McC.M., Peterson, A.T., Phillips, S.J., Richardson, K., Scachetti-Pereira, R., Schapire, R.E., Soberón, J., Williams, S., Wisz, M.S. & Zimmermann, N.E. (2006) Novel methods improve prediction of species' distributions from occurrence data. *Ecography*, 29, 129–151.

Elith, J., Leathwick, J.R. & Hastie, T. (2008) A Working Guide to Boosted Regression Trees. *Journal of Animal Ecology*, 77, 802–813.

Elith, J., Phillips, S.J., Hastie, T., Dudík, M., Chee, Y.E. & Yates, C.J. (2011) A statistical explanation of MaxEnt for ecologists. *Diversity and distributions*, 17, 43–57.

Elith, J., Graham, C.H., Valavi, R., Abegg, M., Bruce, C., Ferrier, S., Ford, A., Guisan, A., Hijmans, R.J., Huettmann, F., Lohmann, L.G., Loiselle, B.A., Moritz, C., Overton, J.McC.,

Peterson, A.T., Phillips, S., Richardson, K., Williams, S., Wiser, S.K., Wohlgemuth, T. & Zimmermann, N.E., (2020). Presence-only and presence-absence data for comparing species distribution modeling methods. *Biodiversity informatics* 15:69-80.

Fithian, W. & Hastie, T. (2013) Finite-sample equivalence in statistical models for presence-only data. *The Annals of Applied Statistics*, 7, 1917–1939.

Flach, P. & Kull, M. (2015) Precision-Recall-Gain Curves: PR Analysis Done Right. *Advances in Neural Information Processing Systems*, pp. 838–846. Curran Associates, Inc.

Guillera-Arroita, G., Lahoz-Monfort, J.J. & Elith, J. (2014) Maxent is not a presence–absence method: a comment on Thibaud et al. *Methods in Ecology and Evolution*, 5, 1192–1197.

Hastie, T., Tibshirani, R. & Friedman, J. (2009) The elements of statistical learning: Data Mining, Inference, and Prediction, 2nd edn. *Springer series in statistics New York*.

Hao, T., Elith, J., Guillera-Arroita, G. & Lahoz-Monfort, J.J. (2019) A review of evidence about use and performance of species distribution modelling ensembles like BIOMOD. *Diversity and Distributions*.

James, G., Witten, D., Hastie, T. & Tibshirani, R. (2013) An introduction to statistical learning, *Springer*.

Muñoz-Mas, R., Gil-Martínez, E., Oliva-Paterna, F.J., Belda, E.J. & Martínez-Capel, F. (2019) Tree-based ensembles unveil the microhabitat suitability for the invasive bleak (Alburnus alburnus L.) and pumpkinseed (Lepomis gibbosus L.): Introducing XGBoost to eco-informatics. *Ecological Informatics*, 53, 100974.

Muscarella, R., Galante, P.J., Soley-Guardia, M., Boria, R.A., Kass, J.M., Uriarte, M. & Anderson, R.P. (2014) ENMeval: an R package for conducting spatially independent evaluations and estimating optimal model complexity for Maxent ecological niche models. *Methods in Ecology and Evolution*, 5, 1198–1205.

Pedersen, E.J., Miller, D.L., Simpson, G.L. & Ross, N. (2019) Hierarchical generalized additive models in ecology: an introduction with mgcv. *PeerJ*, 7, e6876.

Radosavljevic, A. & Anderson, R.P. (2014) Making better Maxent models of species distributions: complexity, overfitting and evaluation. *Journal of biogeography*, 41, 629–643.

Thuiller, W., Lafourcade, B., Engler, R. & Araújo, M.B. (2009) BIOMOD–a platform for ensemble forecasting of species distributions. *Ecography*, 32, 369–373.

Wood, S.N., Pya, N. & Säfken, B. (2016) Smoothing Parameter and Model Selection for General Smooth Models. *Journal of the American Statistical Association*, 111, 1548–1563.