Lab6

Patty Park

2023-03-06

We are using the variables to predict presence or absence of the species (eel)

note on mtry:needs to be given the full set of features to see what it needs to train on

Case Study: Eel Distribution Modeling

This week's lab follows a project modeling the eel species Anguilla australis described by Elith et al. (2008). There are two data sets for this lab. You'll use one for training and evaluating your model, and you'll use your model to make predictions predictions on the other. Then you'll compare your model's performance to the model used by Elith et al.

```
#load libraries
library(tidyverse)
library(tidymodels)
library(xgboost)
library(janitor)
library(vip)
library(gt)
```

Data

Grab the training data sets (eel.model.data.csv, eel.eval.data.csv) from github here: https://github.com/MaRo406/eds-232-machine-learning/blob/main/data

```
eel_model <- read_csv(here::here("week_7", "data", "eel.model.data.csv")) %>%
  clean_names() %>%
  select(-site) %>%
  mutate(angaus = as.factor(angaus),
         method = as.factor(method))

eel_eval <- read_csv(here::here("week_7", "data", "eel.eval.data.csv")) %>%
    clean_names() %>%
  mutate(angaus = as.factor(angaus_obs),
         method = as.factor(method))
```

Split and Resample

Split the model data (eel.model.data.csv) into a training and test set, stratified by outcome score (Angaus). Use 10-fold CV to resample the training set.

```
#set seed for reproducibility
set.seed(50)

#split data into 80 20 split
eel_split <- initial_split(eel_model, prop = .8, strata = angaus) #split data stratified by survived

eel_train <- training(eel_split)#get training data
eel_test = testing(eel_split) #get testing data

#look at first 5 per training and testing data
#head(eel_train)
#head(eel_test)

#set folds to 5
cv_folds = vfold_cv(eel_train, v = 5)
```

Preprocess

Create a recipe to prepare your data for the XGBoost model

```
#create eel recipe
eel_recipe <- recipe(angaus ~ ., data = eel_train) %>% #create model recipe
  step_dummy(all_nominal_predictors()) %>% #create dummy variables from all factors
  step_normalize(all_numeric_predictors()) #normalize all numeric predictors
```

Tuning XGBoost

Tune Learning Rate

Following the XGBoost tuning strategy outlined in lecture, first we conduct tuning on just the learning rate parameter:

1. Create a model specification using {xgboost} for the estimation

- Only specify one parameter to tune()

```
#create model
eel_xgb_model <- boost_tree(learn_rate = tune()) %>% #tuning the learn_rate and trees for the parameter
  set_engine("xgboost") %>%  #nthread = 2
  set_mode("classification")

#create workflow
eel_xgb_workflow = workflow() %>% #create workflow
  add_model(eel_xgb_model) %>% #add boosted trees model
  add_recipe(eel_recipe) #add recipe

#look at workflow
#eel_xgb_workflow
```

2. Set up a grid to tune your model by using a range of learning rate parameter values: expand.grid(learn_rate = seq(0.0001, 0.3, length.out = 30))

- Use appropriate metrics argument(s) - Computational efficiency becomes a factor as models get more complex and data get larger. Record the time it takes to run. Do this for each tuning phase you run.You could use {tictoc} or system.time().

Note: - defining the grid: matrix of parameter values (use on difference cv folds) see what is the best set of values - giving a set of value to define

```
#create the grid to tune for the learning rate parameter
grid_1 <- expand.grid(learn_rate = seq(0.0001, 0.3, length.out = 30))

#tune the model using created grid
system.time(
  eel_xbg_tune <- eel_xgb_workflow %>%
    tune_grid(resamples = cv_folds, grid = grid_1)
)
```

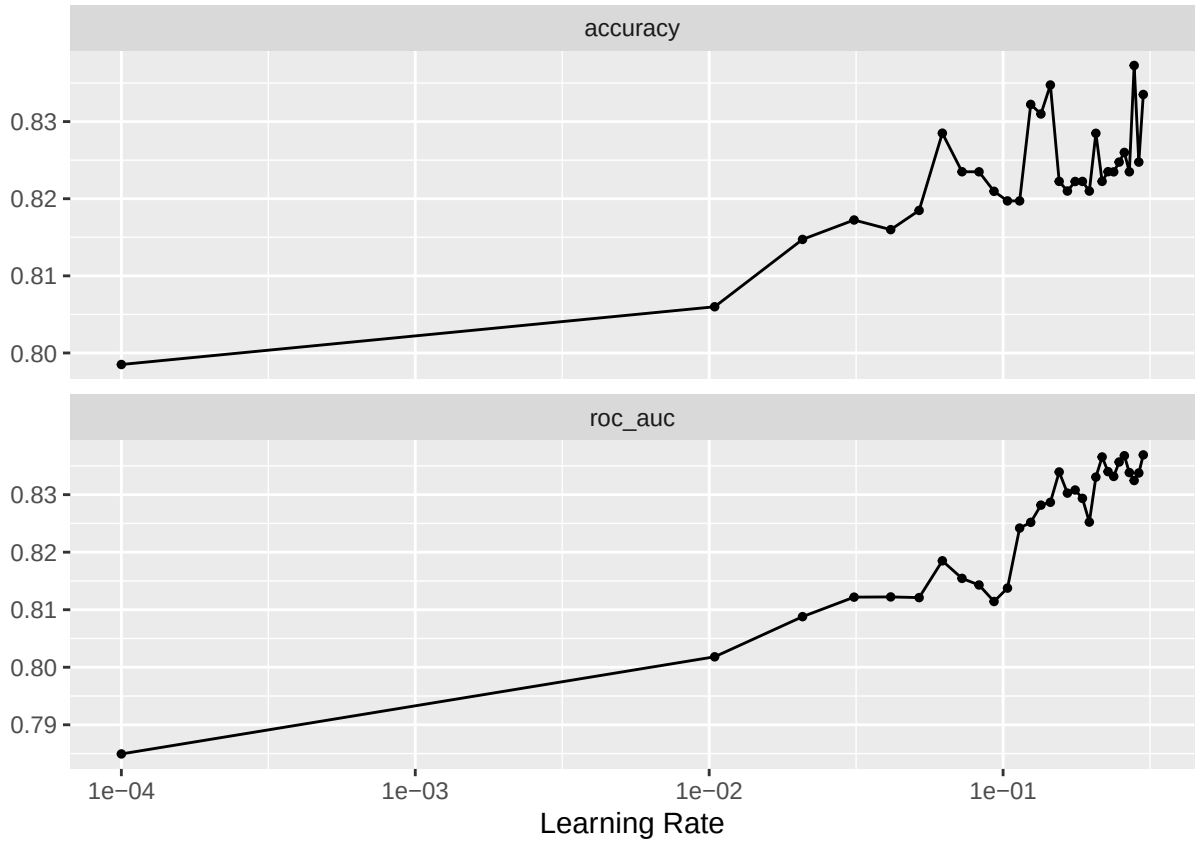
```
##    user  system elapsed
## 157.717   0.491  12.981
```

```
#save(eel_xbg_tune, file = "rda/eel_xbg_tune.rda")

#load save rda file
load(file = here::here("week_7", "rda", "eel_xbg_tune.rda"))

#view tuned table dataset to see if we have metrics
#eel_xbg_tune
```

Answer: To tune the learning grid, the time it took for this to run was about 12 seconds.

3. Show the performance of the best models and the estimates for the learning rate parameter values associated with each.

```
#look at performance of model
autoplot(eel_xbg_tune)
```




```
#find the best model from the tuned models
tree_learn <- show_best(eel_xbg_tune, n = 1)

#print out the best model
tree_learn %>%
  gt()
```

learn_rate	.metric	.estimator	mean	n	std_err	.config
0.3	roc_auc	binary	0.8369089	5	0.01299633	Preprocessor1_Model30

Answer: Here, the show_best() function shows that the best metrics is the roc_auc model. Looking at the graph comparing the accuracy and roc_auc models, the roc_auc shows a slightly larger number than the accuracy curve.

Tune Tree Parameters

1. Create a new specification where you set the learning rate (which you already optimized) and tune the tree parameters.

```
#create a new variable that stores the learning rate
learn_rate <- tree_learn$learn_rate

#create model setting the learning rate
```

```
eel_xgb_model_learn <- boost_tree(learn_rate = learn_rate,
                                  trees = 3000,
                                  tree_depth = tune(),
                                  min_n = tune(),
                                  loss_reduction = tune()) %>% #tuning the learn_rate and trees for the
  set_engine("xgboost") %>%  #nthread = 2
  set_mode("classification")

#create the workflow
eel_xgb_workflow_learn <- workflow() %>% #create workflow
  add_model(eel_xgb_model_learn) %>% #add boosted trees model
  add_recipe(eel_recipe)

#look at the workflow
#eel_xgb_workflow_learn
```

2. Set up a tuning grid. This time use grid_latin_hypercube() to get a representative sampling of the parameter space

```
#set up tuning grid using the `grid_latin_hypercube()`

grid_2 <- grid_latin_hypercube(tree_depth(), min_n(), loss_reduction())

#tune the grid for tree_depth, min_n, and loss_reduction
system.time(
  eel_xbg_tune_latin <- eel_xgb_workflow_learn %>%
    tune_grid(resamples = cv_folds, grid = grid_2)
)
```

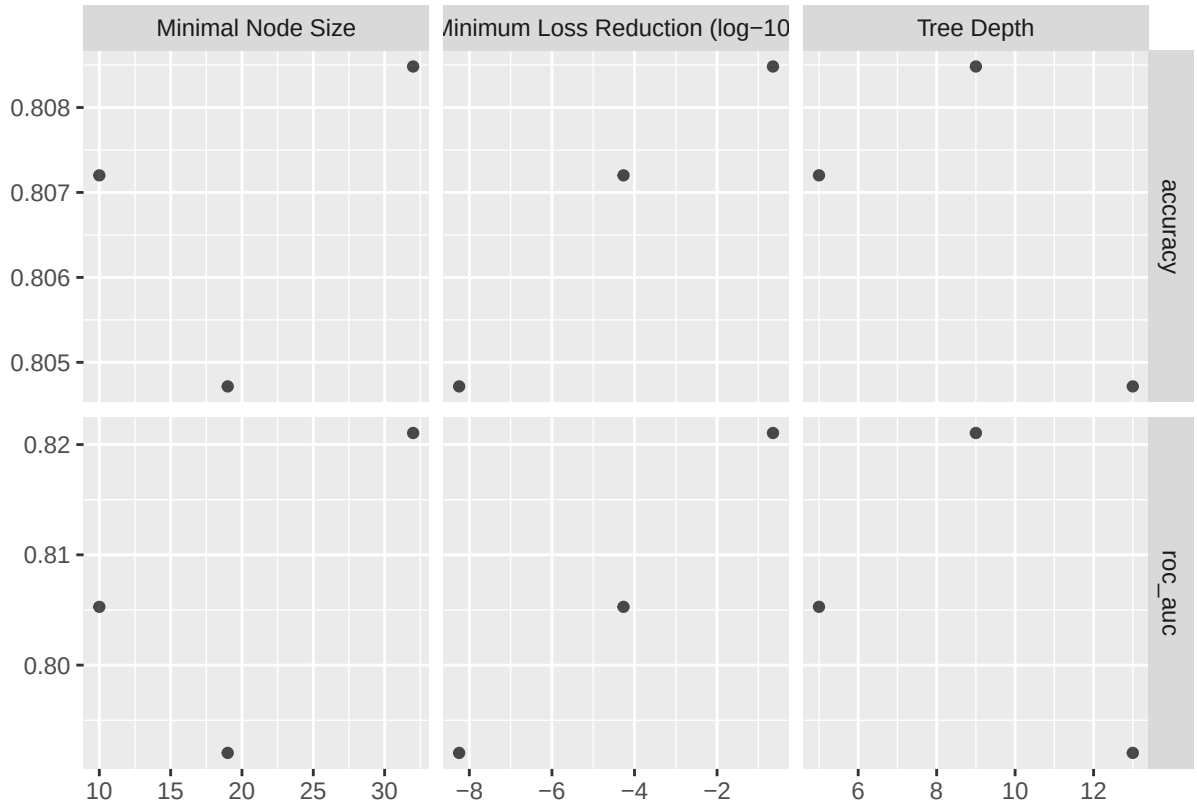
```
##    user  system elapsed
##  41.932   0.413  24.704
```

```
#save the file
#save(eel_xbg_tune_latin, file = "rda/eel_xbg_tune_latin.rda")

#load save rda file
load(file = here::here("week_7", "rda", "eel_xbg_tune_latin.rda"))

#look at tuned model to make sure it has metrics
#eel_xbg_tune_latin
```

Answer: To tune these parameters, it took about 24 seconds to tune all tree_depth, min_n, and loss_reduction.

3. Show the performance of the best models and the estimates for the tree parameter values associated with each.

```
#graph the performance of the tuned model
autoplot(eel_xbg_tune_latin)
```




```
#find the best model for the tree parameters tuned grid
tree_param_2 <- show_best(eel_xbg_tune_latin, n = 1)

#look at what was the best model
tree_param_2 %>%
  gt()
```

min_n	tree_depth	loss_reduction	.metric	.estimator	mean	n	std_err	.config
32	9	0.2262885	roc_auc	binary	0.8210468	5	0.01928177	Preprocessor1_Model3

```
#set new variable to have the most optimal parameters
min_n <- tree_param_2$min_n
tree_depth <- tree_param_2$tree_depth
loss_reduction <- tree_param_2$loss_reduction
```

Answer: Here, our best model comes from the roc_auc metric. Listed here, our most optimal min_n is 32, most optimal tree_depth is 9, and most optimal loss_reduction is around 0.226.

Tune Stochastic Parameters

1. Create a new specification where you set the learning rate and tree parameters (which you already optimized) and tune the stochastic parameters.

```
#create model to tune for mtry and sample_size
eel_xgb_model_stoch <- boost_tree(learn_rate = learn_rate,
                                  trees = 3000,
                                  tree_depth = tree_depth,
                                  min_n = min_n,
                                  loss_reduction = loss_reduction,
                                  mtry = tune(),
                                  sample_size = tune()) %>% #tuning the learn_rate and trees for the pa
  set_engine("xgboost") %>%  #nthread = 2
  set_mode("classification")

#create workflow
eel_xgb_workflow_stoch <- workflow() %>% #create workflow
  add_model(eel_xgb_model_stoch) %>% #add boosted trees model
  add_recipe(eel_recipe) #add recipe

#look at workflow
#eel_xgb_workflow_stoch
```

2. Set up a tuning grid. Use grid_latin_hypercube() again.

```
#create grid for sample_prop() and mtry()
grid_3 <- grid_latin_hypercube(
  sample_size = sample_prop(),
  finalize(mtry(), eel_train)
)

# how you can set range for any of the parameters. ex can look at certain range for range.
# sample_size() %>% range_set()

# tune grid from the workflow
system.time(
  eel_xbg_tune_stoch <- eel_xgb_workflow_stoch %>%
    tune_grid(resamples = cv_folds, grid = grid_3)
)
```

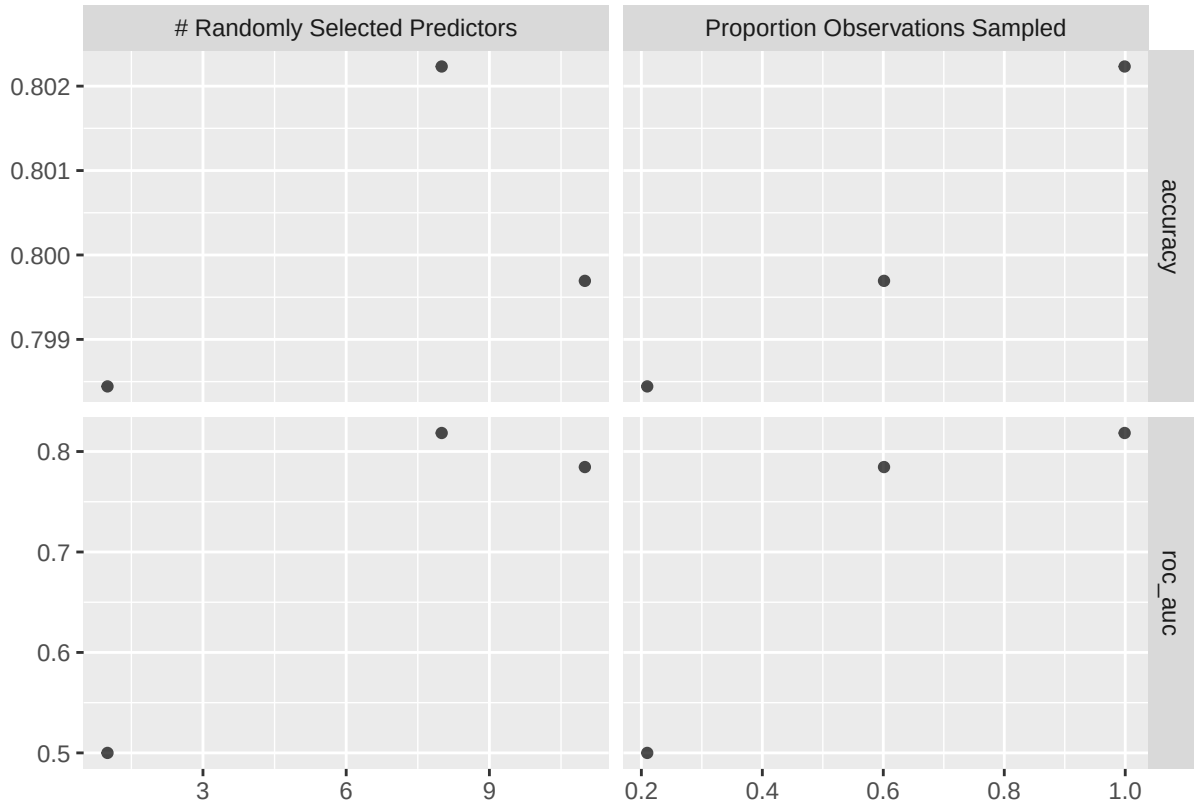
```
##    user  system elapsed
##  32.024   0.176  18.828
```

```
# look at tuned grid
#eel_xbg_tune_stoch
```

Answer: To tune the stochastic parameters, including mtry() and sample_size(), it took about 20 seconds.

3. Show the performance of the best models and the estimates for the tree parameter values associated with each.

```
#graph the performance of the tuned model
autoplot(eel_xbg_tune_stoch)
```




```
#find best model from the tuned grid
tree_stoch_3 <- show_best(eel_xbg_tune_stoch, n = 1)

#look at the best model
tree_stoch_3 %>%
  gt()
```

mtry	sample_size	.metric	.estimator	mean	n	std_err	.config
8	0.9989948	roc_auc	binary	0.8184371	5	0.01995319	Preprocessor1_Model2

Answer: Here, our best model metric is the roc_auc mode. From this model, it states that the most optimal mtry() is 8 while the most optimal sample_size is almost close to 1.

Finalize workflow and make final prediction

1. How well did your model perform? What types of errors did it make?

```
#finalize work flow for our show best model
rf_final_roc <- finalize_workflow(eel_xgb_workflow_stoch, select_best(eel_xbg_tune_stoch, metric = "roc

#print out results
#rf_final_roc
```

```
#fit the finalized workflow
fit_final_model <- fit(rf_final_roc, eel_train)

#view fitted model
#fit_final_roc

set.seed(50)
# look at the predicted info for training data for final fitted dataset
test_predict_model <- predict(object = fit_final_model, new_data = eel_test) %>% # predict the training
  bind_cols(eel_test)
#view results
#test_predict_model

#find metrics of the predicted train data
test_metrics_model <- test_predict_model %>%
  metrics(angaus, .pred_class) # get testing data metrics
#view metrics results
test_metrics_model %>%
  gt()
```

.metric	.estimator	.estimate
accuracy	binary	0.8059701
kap	binary	0.2935929

```
#look at sens. and spec.
sensitivity(test_predict_model, truth = angaus, estimate = .pred_class) %>%
  gt()
```

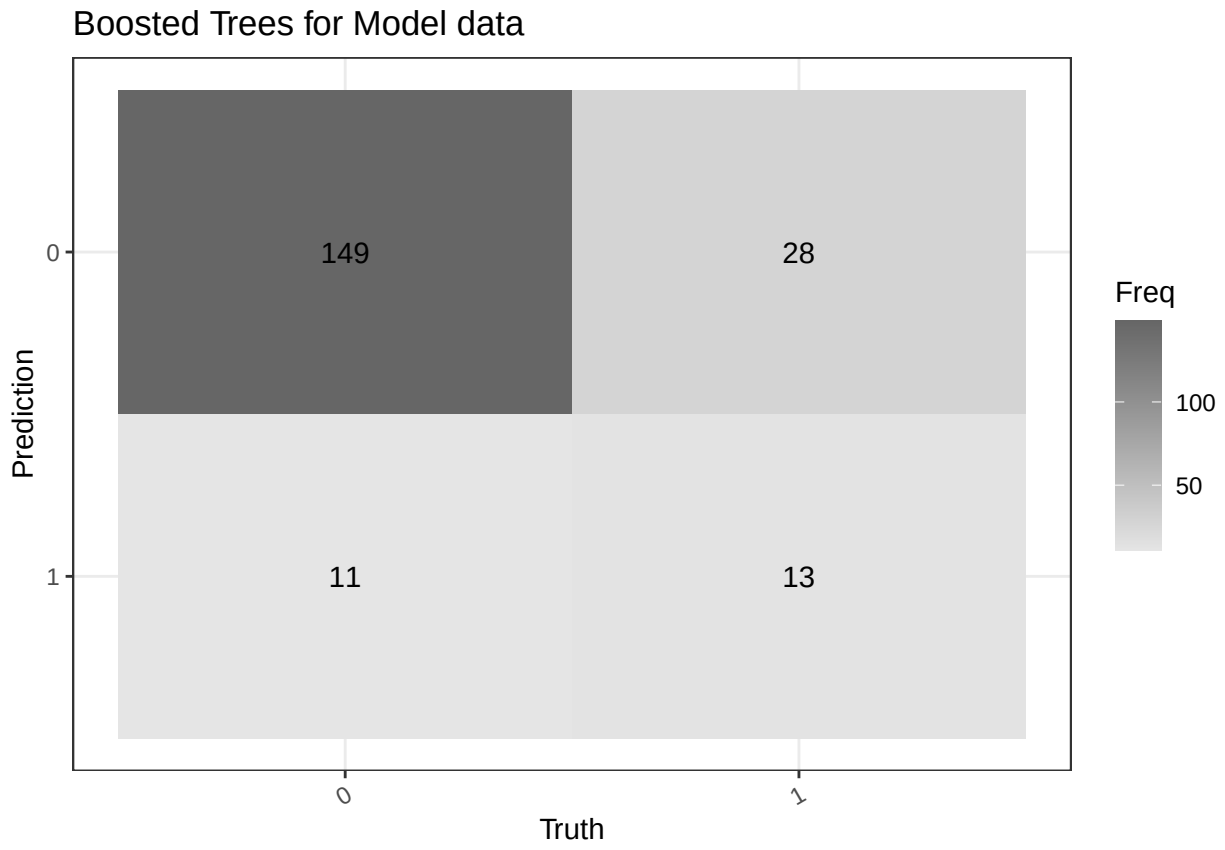
.metric	.estimator	.estimate
sensitivity	binary	0.93125

```
specificity(test_predict_model, truth = angaus, estimate = .pred_class) %>%
  gt()
```

.metric	.estimator	.estimate
specificity	binary	0.3170732

```
#another way to find the metrics for accuracy model
#accuracy(train_predict_acc, truth = angaus, estimate = .pred_class)



m2_model <- test_predict_model %>%
  conf_mat(truth = angaus, estimate = .pred_class) %>% #create confusion matrix
  autoplot(type = "heatmap") + #plot confusion matrix with heatmap
  theme_bw() + #change theme
  theme(axis.text.x = element_text(angle = 30, hjust=1)) +
```

```
  #rotate axis labels
  labs(title = "Boosted Trees for Model data")

m2_model
```



Answer: When looking at the performance of my model, I think my model performed very well, as the estimate from the testing data was very close to the estimates to the training data. In order to see what types of errors my model made, I created a confusion matrix. I can see that there are many true negatives that the model marked. However, there are very few true positives that the model marked. It should also be noted that more false positive have been marked verses the false negatives. When also looking at my sensitivity and specificity, I get a very high sensitivity number and a very low specificity number. Because we get a high sensitivity, we know that our model is better at identifying positive results, meaning that the model is well predicting if there is the presence or absence of a eel in that ecosystem. The low specificity number tells me that the model is incorrectly labeling many of the negative results as positives. However, comparing this to the confusion matrix, I can see that there are not very many that have been labeled as false positives. However, it is also good to note that there were not that many true negatives that were reported in the confusion matrixs

Fit your model the evaluation data and compare performance

1. Now used your final model to predict on the other dataset (eval.data.csv)

```
#set seed for reproducibility
set.seed(50)
```

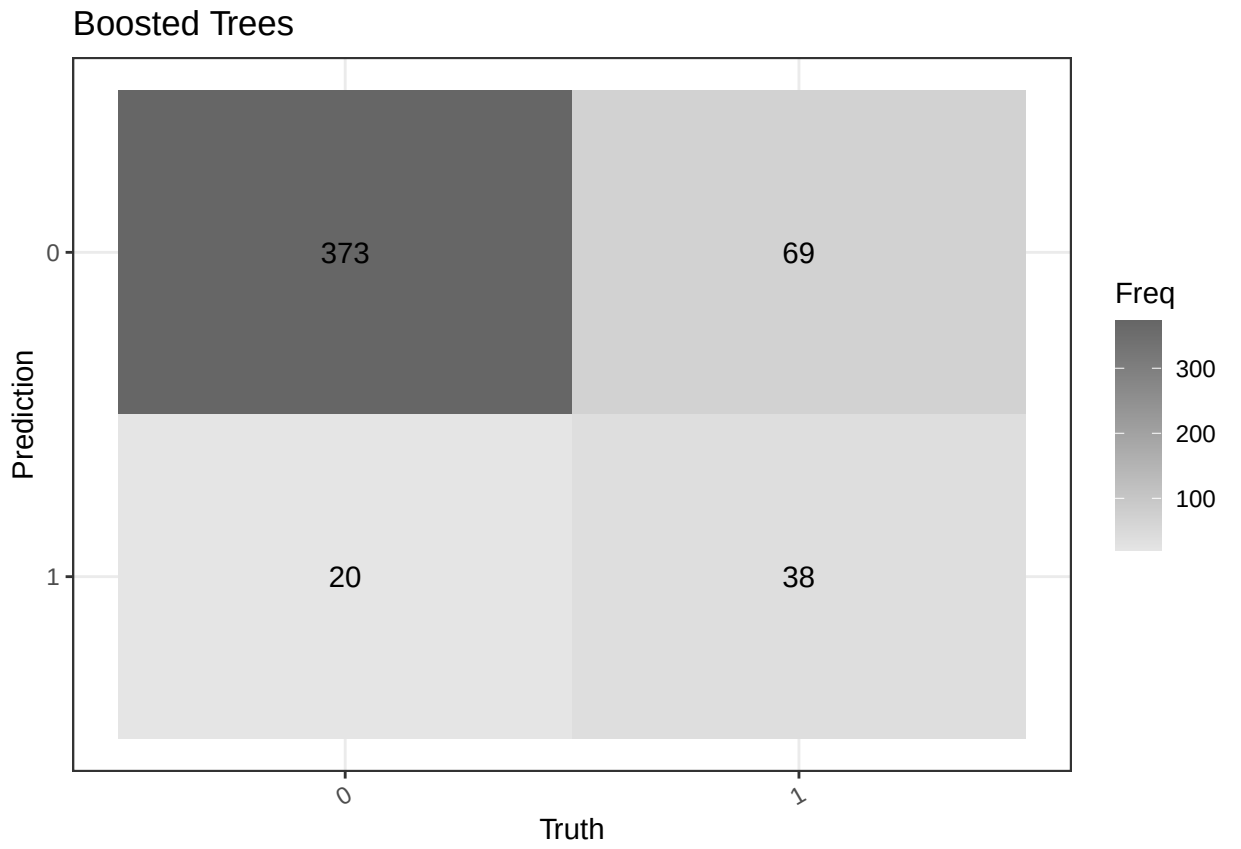
```
#split data into 80 20 split
# eel_split_eval <- initial_split(eel_eval, prop = .8, strata = angaus) #split data stratified by survi
#
# eel_train_eval <- training(eel_split_eval)#get training data
# eel_test_eval = testing(eel_split_eval) #get testing data


#fit final model on the eel_eval data
fit_final_eval <- fit(rf_final_roc, eel_eval)
#print results
#fit_final_eval


test_predict_eval <- predict(object = fit_final_eval, new_data = eel_eval) %>% # predict the testing se
  bind_cols(eel_eval)
#view results
#test_predict_eval
```

2. How does your model perform on this data?

```
#another way to find the metrics for accuracy model
#accuracy(train_predict_acc, truth = angaus, estimate = .pred_class)

#another way to find metrics on the testing data
#fit on both training and testing data, get results from just testing data
# final_eel <- last_fit(fit_final_roc, eel_split)
# #look at metrics from final fitted model on testing data
# collect_metrics(final_eel)

#find metrics of the predicted train data
test_metrics_eval <- test_predict_eval %>%
  metrics(angaus, .pred_class) # get testing data metrics
#view metrics results
test_metrics_eval %>%
  gt()
```

.metric	.estimator	.estimate
accuracy	binary	0.8220000
kap	binary	0.3650839

```
m2 <- test_predict_eval %>%
  conf_mat(truth = angaus, estimate = .pred_class) %>% #create confusion matrix
  autoplot(type = "heatmap") + #plot confusion matrix with heatmap
  theme_bw() + #change theme
  theme(axis.text.x = element_text(angle = 30, hjust=1)) +
  #rotate axis labels
  labs(title = "Boosted Trees")

#show confusion matrix
m2
```




```
#look at sens. and spec.
sensitivity(test_predict_eval, truth = angaus, estimate = .pred_class) %>%
  gt()
```

.metric	.estimator	.estimate
sensitivity	binary	0.9491094

```
specificity(test_predict_eval, truth = angaus, estimate = .pred_class) %>%
  gt()
```

.metric	.estimator	.estimate
specificity	binary	0.3551402

Answer: The model performed pretty well on the eval data. For the roc_auc metrics, I get around the same estimate number compared to the other dataset, meaning that my model holds up well. Looking at the confusion matrix, I see that the majority is true positives and very few are true negatives. Comparing that with the results I get for my sensitivity and specificity, I get a very high sensitivity number, but a very low specificity number. I can interpret this that if the model detects that there was an eel present, then the model accurately labeled the eel being present. However, because of the low specificity, I interpret this as if there was not an eel present in the area, the model will accidently label it as being present instead.

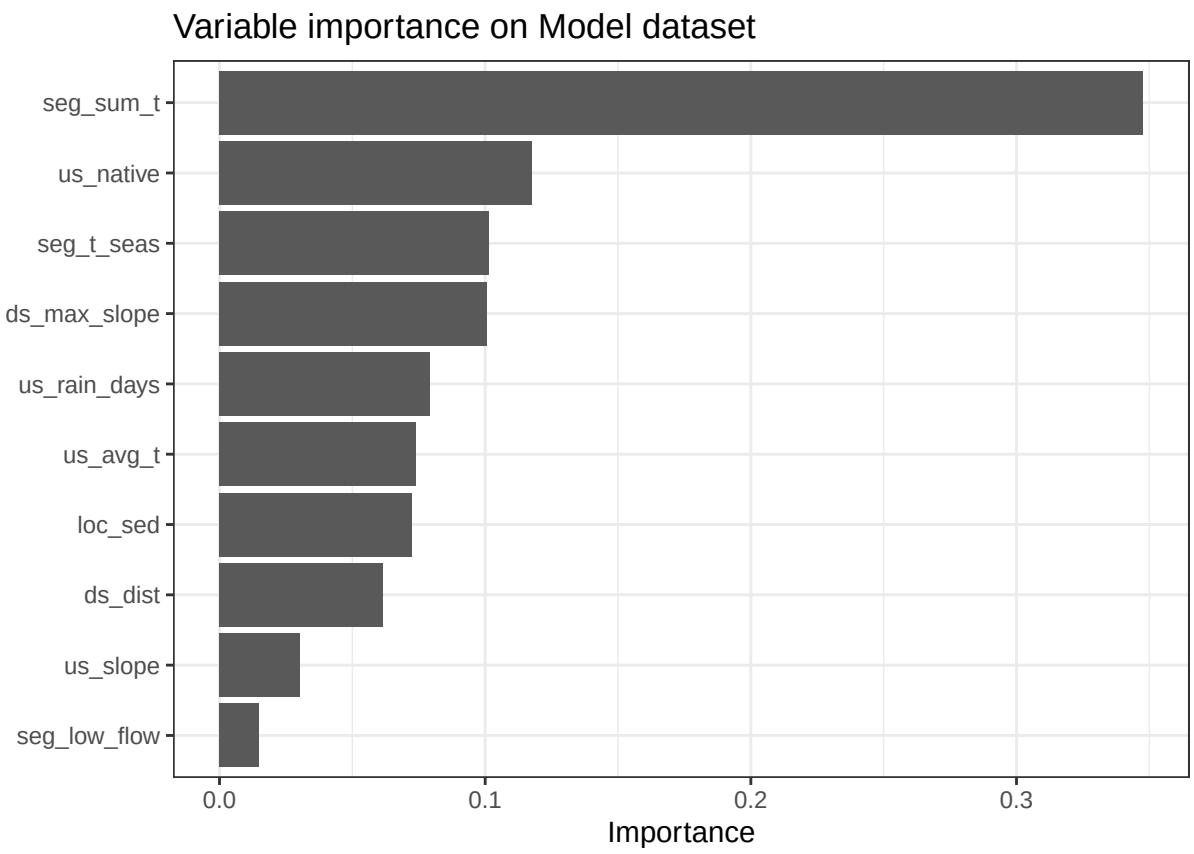
3. How do your results compare to those of Elith et al.?

Answer: When looking at the variable importance on both datasets, the most important variables that are the same between the two datasets are `seg_sum_t`, `us_native`, and `ds_max_slope`. After that, the order of variable importance changes between the two datasets. If more of these variable are featured in the area, there is a higher chance that an eel will be present in that area. Another way I interpret the top two important variable is that the summer air temperature and areas with indigenous forest are very important to see if an eel will be found in that area. It is interesting to note that for most of the variables for the model dataset, most of them are of somewhat importance. But in the eval dataset, most of the variable drop down in importance much more significantly. From this, my interpretation is that for the model data, more variables will help detect if there is a eel present or absent, while for the eval data, only a few variable are of importance in order to detect the presense or absence of an eel.

Now comparing it to the Elith et al. paper, I believe my model pretty well against the paper. Comparing what I got for the variable importance to the paper's variable importance, most of my variables got similar numbers to the paper's predictor numbers. Also looking at table 3, it shows that the higher temperatures give much higher marginal effects, which is what I concluded with my own set of data.

- Use {vip} to compare variable importance
- What do your variable importance results tell you about the distribution of this eel species?

```
#for model dataset
fit_final_model %>%
  extract_fit_parsnip() %>%
  vip() +
  theme_bw() +
  labs(title = "Variable importance on Model dataset")
```



```r
#for eval dataset
fit_final_eval %>%
  extract_fit_parsnip() %>%
  vip() +
  theme_bw() +
  labs(title = "Variable importance on Eval dataset")
```