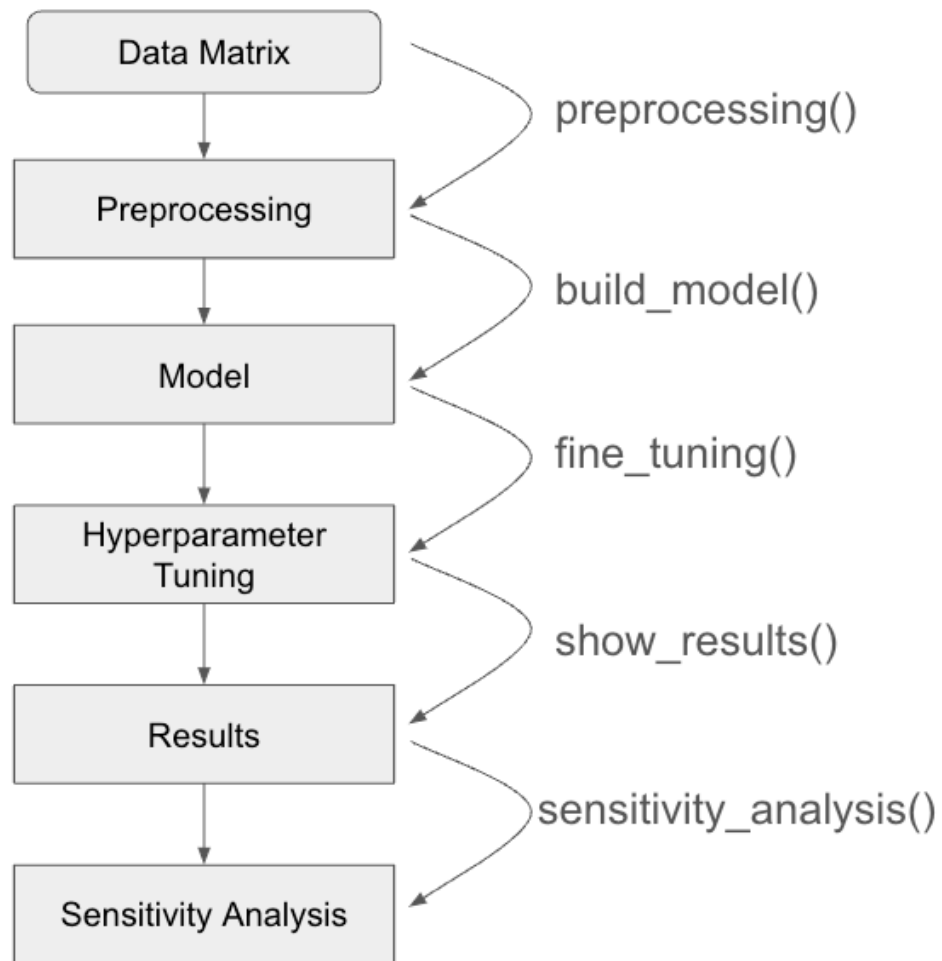


Tutorial TidyML

Introduction

TidyML is a minimal library focused on providing all the essential tools for the workflow of a machine learning modelling process. It divides the whole process into 5 sequential steps:

1. Preprocessing
2. Model Building
3. Fine Tuning
4. Computing Performance Metrics
5. Sensitivity Analysis / Interpretable ML



TidyML can be downloaded from github using the *`devtools::install_github()`* function.

```
#install.packages("devtools")
#devtools::install_github("JMartinezGarcia/TidyML")

library(dplyr)
```

Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

`filter`, `lag`

The following objects are masked from 'package:base':

intersect, setdiff, setequal, union

```
library(TidyML)
```

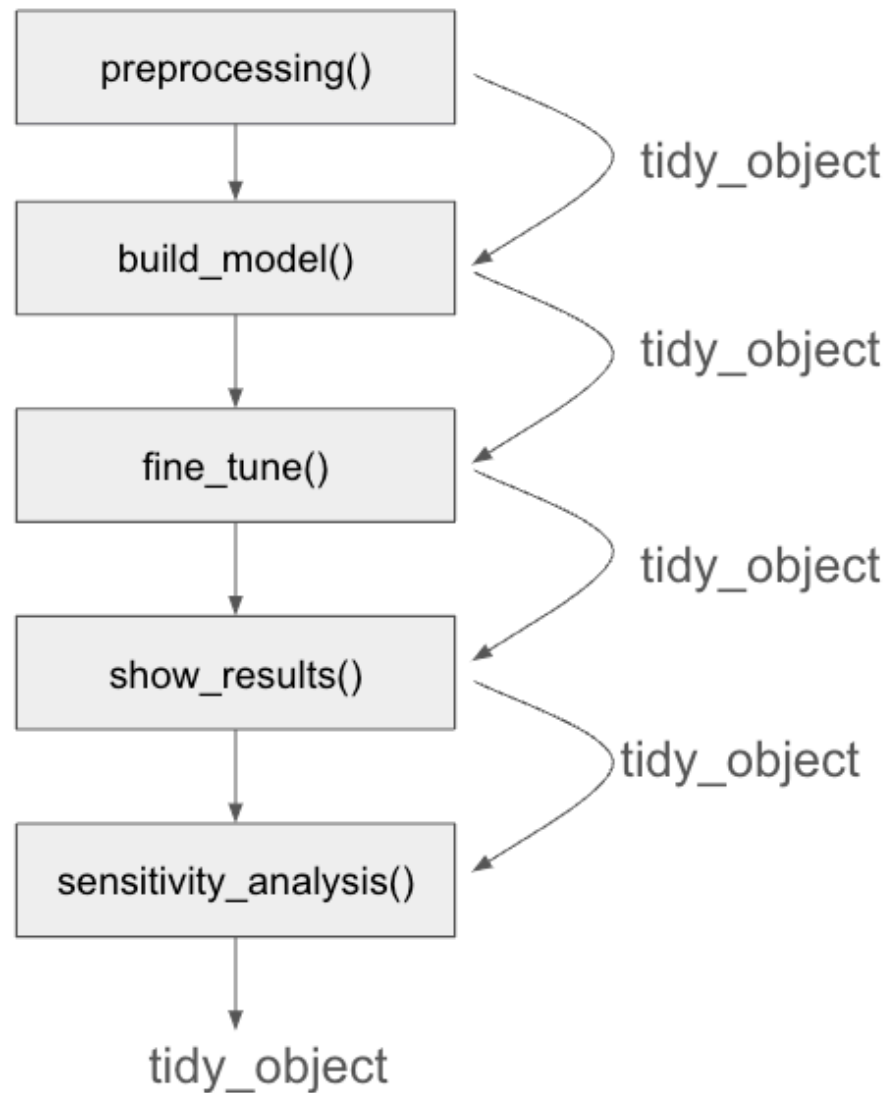
Loading required package: tidyverse

```
-- Attaching core tidyverse packages ----- tidyverse 2.0.0 --
v forcats   1.0.0      v readr     2.1.5
v ggplot2   3.5.2      v stringr   1.5.1
v lubridate 1.9.4      v tibble    3.2.1
v purrr     1.0.4      v tidyr     1.3.1
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()     masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```

Internally, due to the sequential nature of the workflow, each step stores new information of the analysis on an object called “tidy_object”. At any time during the process, the internal information of the analysis can be retrieved from the tidy_object using the “\$” operator. The implemented fields are:

- *formula*: Modelling formula
- *task*: task of analysis (“regression” or “classification”)
- *full_data*: Full data matrix
- *train_data*: Training data matrix
- *validation_data*: Validation data matrix (with Grid Search CV there is no validation data)
- *test_data*: Testing data matrix
- *outcome_levels*: Levels of target feature if task is classification
- *transformer*: Preprocessor object (recipe from recipe library)
- *models_names*: Name of model
- *models*: Model before fitting (from parsnip library)
- *hyperparameters*: Hyperparameters for tuning
- *metrics*: Metric used for tuning

- *tuner*: Name of tuner
- *tuner_fit*: Results from tuner search
- *workflow*: Whole workflow (from workflow library)
- *final_models*: Final trained model (from parsnip library)
- *fit_summary*: Performance summary results
- *predictions*: Predictions of the model
- *sensitivity_analysis*: Sensitivity analysis results



Binary Classification Example

We will start by creating a small binary classification dataset using the palmerpenguins data.

Create Dataset

```
df <- palmerpenguins::penguins %>%  
  na.omit() %>%  
  dplyr::select(-year) %>%  
  dplyr::filter(species == "Adelie" | species == "Gentoo") %>%  
  dplyr::mutate(species = droplevels(species))
```

Preprocessing Step

We will first preprocess the data set using the *preprocessing()* function. We will pass the dataset along with the formula for our problem. The preprocessing step requires to specify which columns are going to be preprocessed:

- Numerical columns will be normalized by z-score
- Categorical columns will be one-hot encoded

As well as the task to be performed: “regression” or “classification”.

In our case, we will preprocess all numerical columns and all categorical columns using the **all** keyword (by default):

```
formula = "species ~ ."  
  
tidy_object = preprocessing(df,  
                             formula = formula,  
                             norm_num_vars = "all",  
                             encode_cat_vars = "all",  
                             task = "classification"  
                             )
```

Registered S3 method overwritten by 'future':

```
method          from  
all.equal.connection parallelly
```

Model Definition

The function *build_model()* allows to create a ML model. Each model has it's own set of hyperparameters which we can choose to fine_tune by passing a range of values or to set to a specific value. By default each hyperparameter will be tuned within a given range. The ML models implemented are:

1. Neural Network:

1. *hidden_units* : number of hidden_units
2. *activation* : activation functions (“relu”, “sigmoid”, “tanh”)
3. *learn_rate* : learning rate

2. Support Vector Machine (“SVM”):

1. *cost* : regularization penalty
2. *margin* : margin of classifier
3. *type* : type of kernel (“linear”, “rbf”, “polynomial”)
4. *rbf_sigma* (rbf kernel only) : rbf kernel sigma
5. *degree* (polynomial kernel only) : polynomial kernel degree
6. *scale_factor* (polynomial kernel only) : polynomial kernel scale factor

3. Random Forest:

1. *mtry* : Size of feature sampling
2. *trees* : Number of trees
3. *min_n* : Minimum number of samples for splitting

4. XGBoost:

1. *mtry* : Size of feature sampling
2. *trees* : Number of trees
3. *min_n* : Minimum number of samples for splitting
4. *tree_depth* : Maximum tree depth
5. *learn_rate* : Learning rate
6. *loss_reduction* : Loss reduction

```
tidy_object <- build_model(tidy_object,
                           model_names = "Random Forest",
                           hyperparameters =
                             list(
                               mtry = c(2,3),
                               trees = 15
                             )
                           )
```

Hyperparameter Tuning

Once the model has been defined, we can fine tune the hyperparameters using the ***fine_tune()*** function. There are 2 different hyperparameter tuning strategies:

1. *Bayesian Optimization*
2. *Grid Search CV*

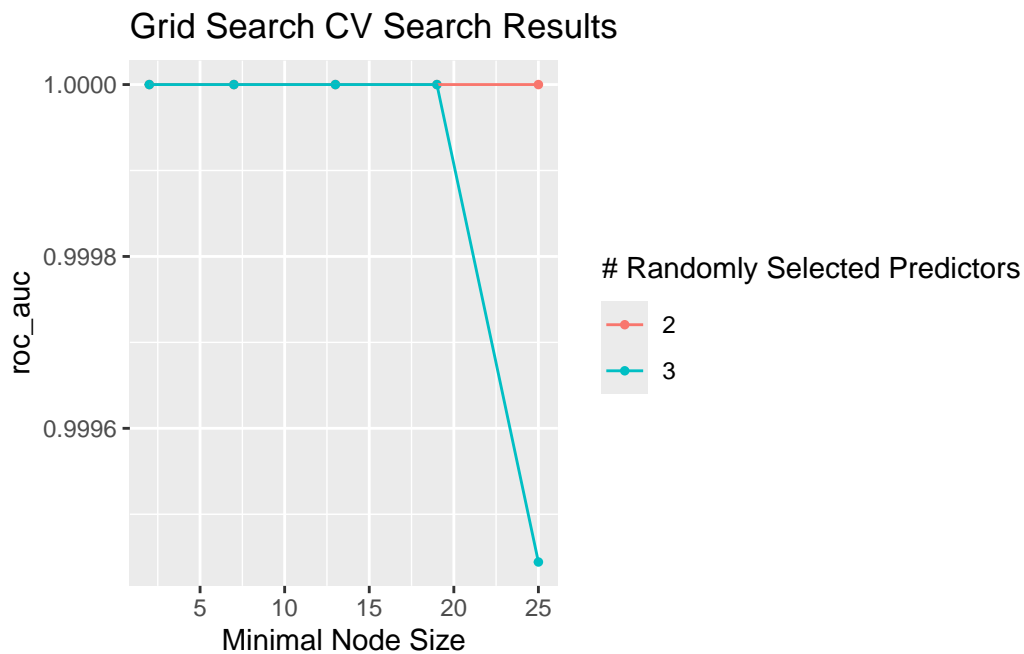
Additionally, we will specify the metric used to select the best performing hyperparameters:

Regression	Classification
rmse	accuracy
mae	bal_accuracy
mpe	precision
mape	recall
ccc	specificity
smape	sensitivity
rpiq	kap
rsq	f_meas
	mcc
	detection_prevalence
	j_index
	roc_auc
	pr_auc
	gain_capture
	brier_class
	roc_aunp

We can visualize the tuning results by setting the *plot_results* parameter to TRUE:

```
tidy_object <- fine_tuning(tidy_object,
                          tuner = "Grid Search CV",
                          metrics = "roc_auc",
                          plot_results = T
                          )
```

```
[1] "Commencing Tuning..."
[1] "Tuning Finalized"
[1] "##### Hyperparameter Tuning Results"
```



```
[1] "##### Best Hyperparameters Found:"
# A tibble: 1 x 8
  mtry min_n .metric .estimator mean     n std_err .config
<int> <int> <chr>   <chr>     <dbl> <int>   <dbl> <chr>
1     2     2 roc_auc binary      1     5     0 Preprocessor1_Model01
```

Results

Once we have found the best hyperparameter configuration we can compute the performance metrics of our model based on the test data using the *show_results()* function. There are different options for the results depending on whether we are doing a regression task or a classification task:

- **Regression:**

- *summary*
- *scatter_residuals*
- *scatter_predictions*
- *residuals_dist*

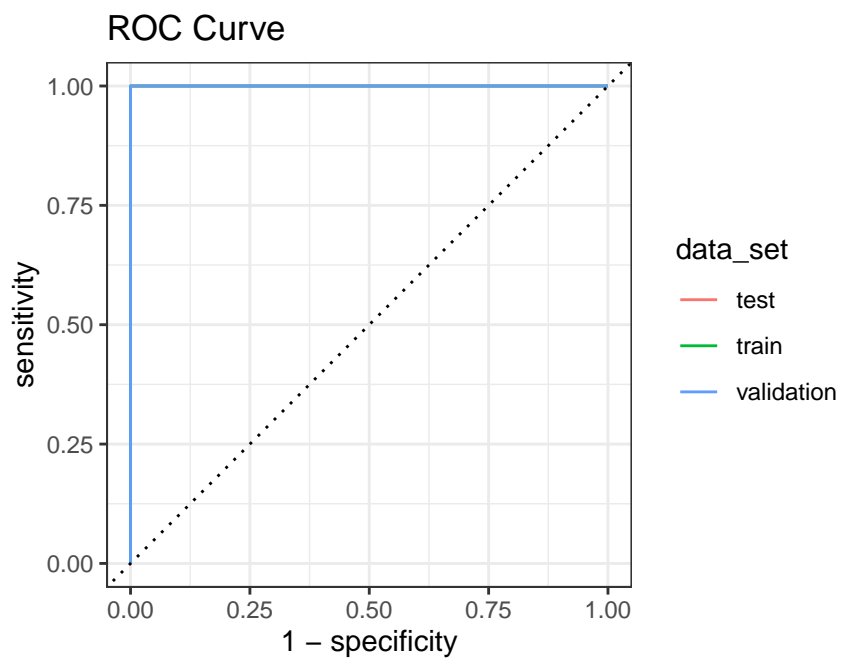
- **Classification:**

- *summary*
- *roc_curve*
- *pr_curve*
- *gain_curve*
- *lift_curve*
- *dist_by_class*
- *reliability_plot*
- *confusion_matrix*

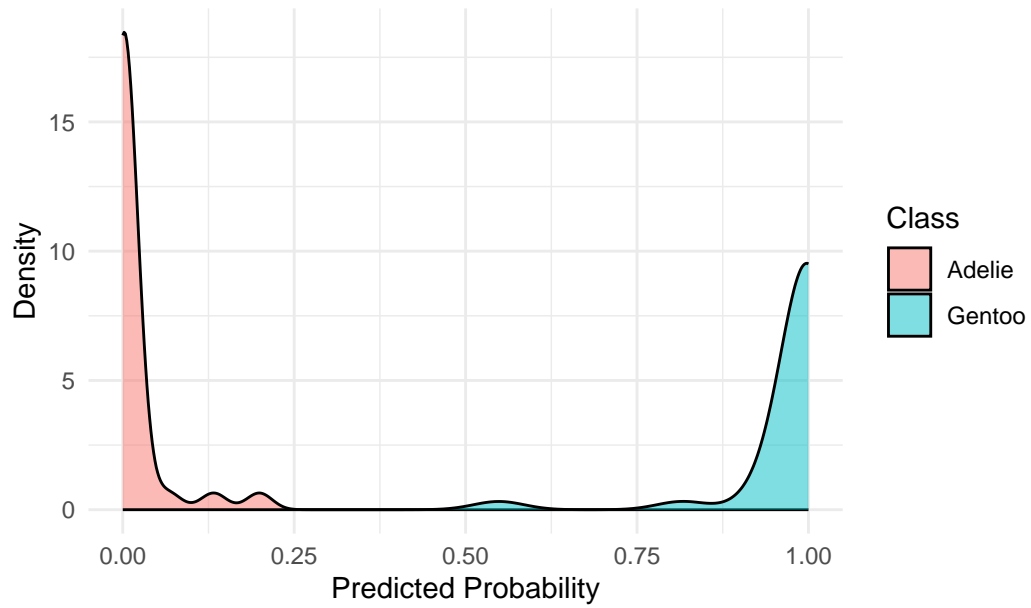
```
tidy_object <- show_results(tidy_object,  
                             summary = TRUE,  
                             roc_curve = TRUE,  
                             confusion_matrix = TRUE,  
                             reliability_plot = TRUE,  
                             dist_by_class = TRUE)
```

```
[1] "##### Showing Results"
```

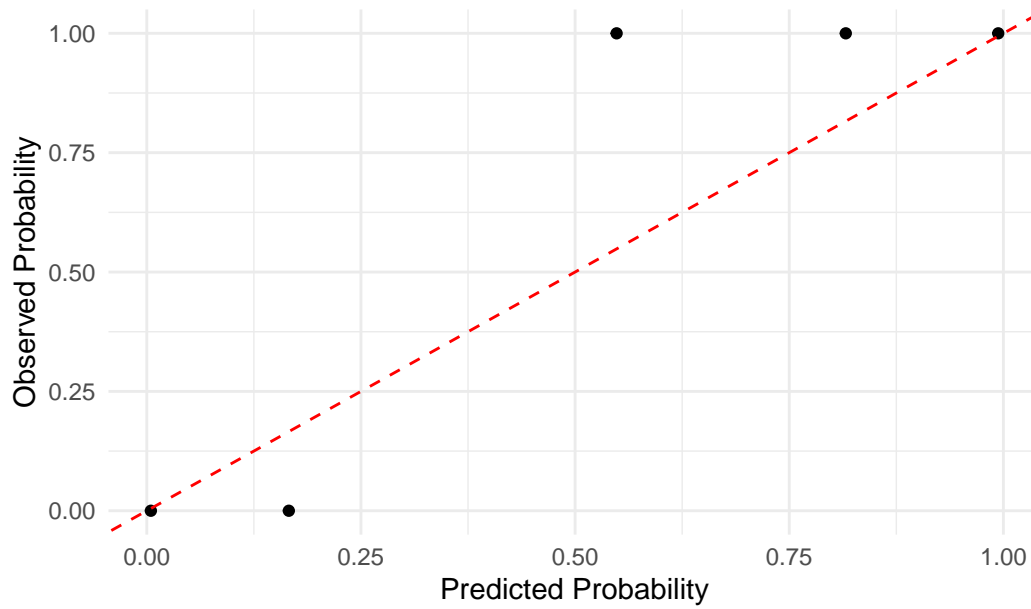
Accuracy	1.000
Balanced_Accuracy	1.000
Precision	1.000
Recall	1.000
Specificity	1.000
Sensitivity	1.000
Kappa	1.000
F1_score	1.000
MCC	1.000
J_index	1.000
Detection_Prevalence	0.507
AUC_ROC	1.000
AUC_PR	1.000
Gain_Capture	1.000

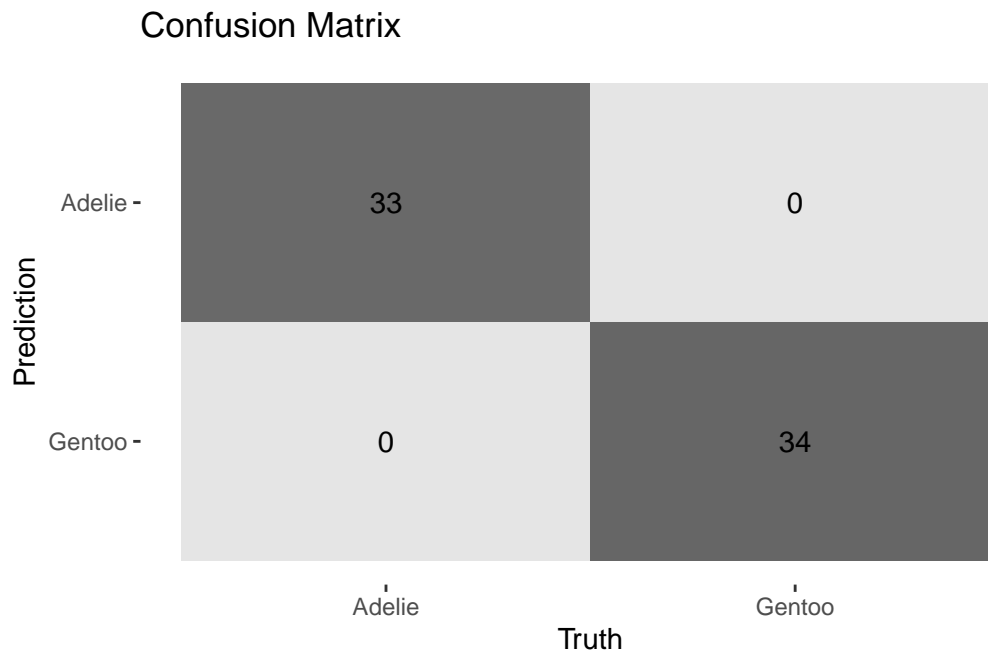


Probability Distribution by Class



Reliability Plot



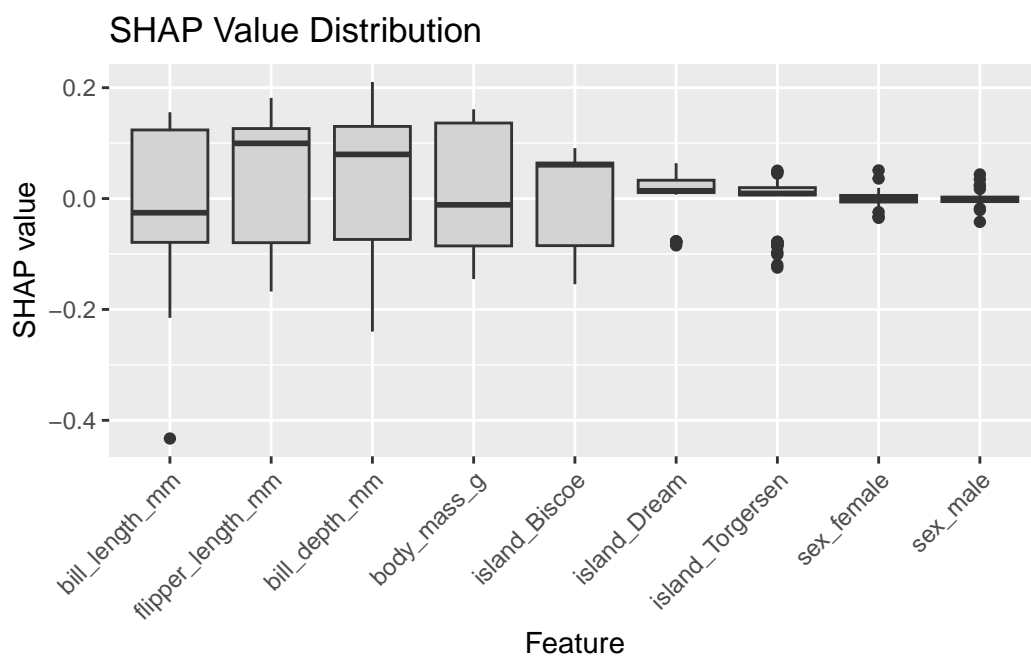
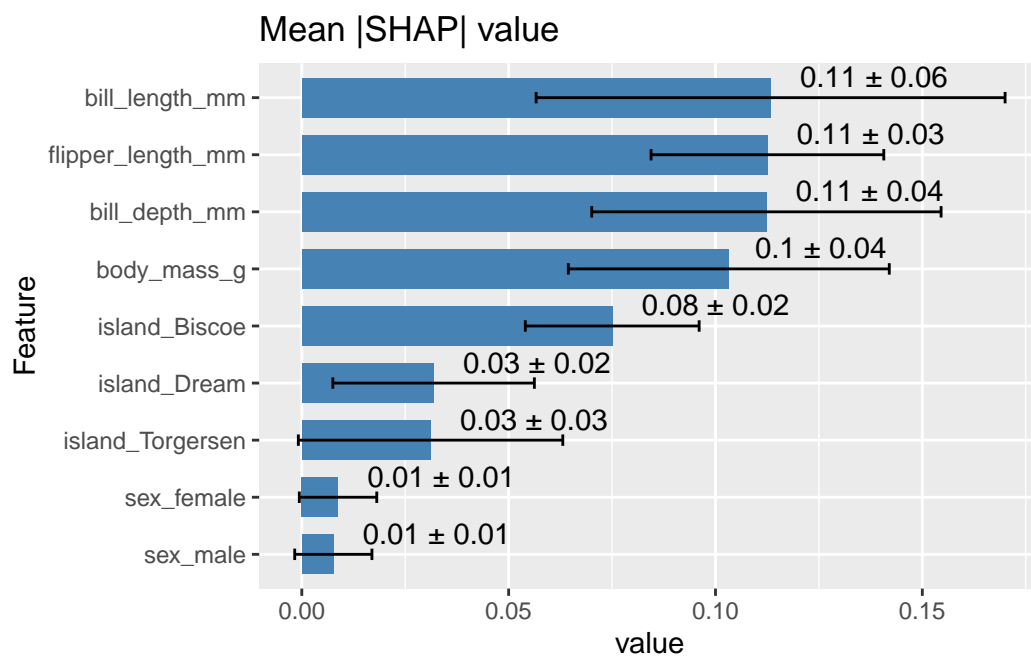


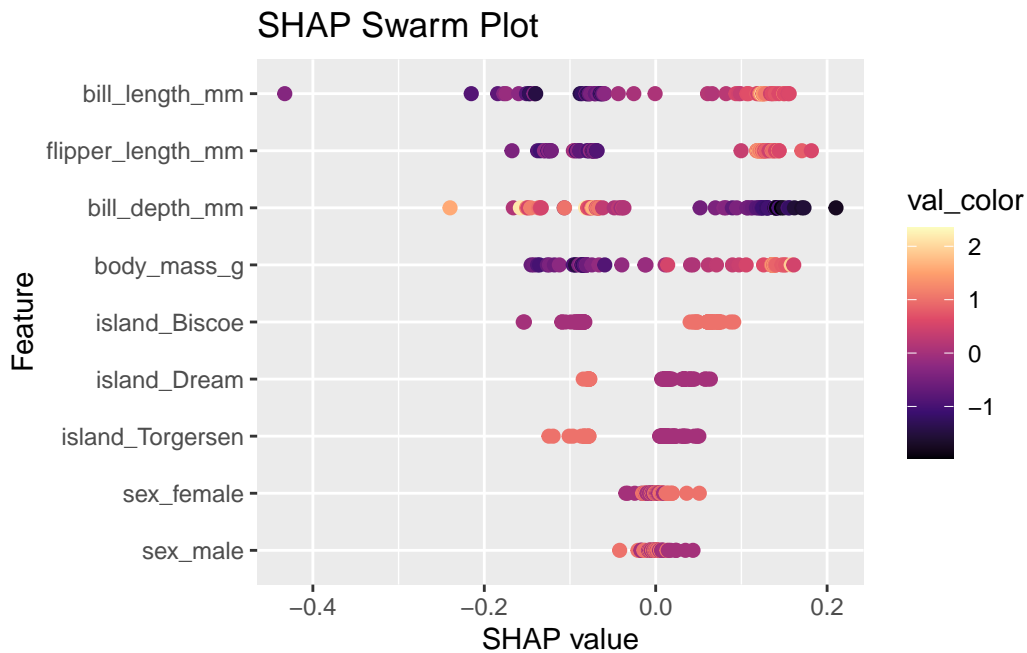
Sensitivity Analysis

We can also perform sensitivity analysis for our fitted model using the ***sensitivity_analysis()*** function. There are currently different methods implemented, some of these methods only work for particular models:

1. Permutation Feature Importance (“PFI”)
2. SHAP
3. Integrated Gradients (Neural Network only)
4. Olden method (Neural Network only)
5. **TO DO:** SOBOL? Shapley Effects?:

```
tidy_object <- sensitivity_analysis(tidy_object, type = "SHAP")
```





Regression Example

Create Dataset

We will again use the palmerpenguins dataset but choose a different formula for a regression task:

```
df <- palmerpenguins::penguins %>%
  na.omit() %>%
  dplyr::select(-year)
```

The Pipe (%>%) Operator

Due to the sequential nature of the processing, we can concatenate all the modelling steps using the %>% (pipe) operator without expliciting passing the tidy_object each time:

```
formula = "bill_length_mm ~ ."

tidy_object = preprocessing(
  df,
  formula = formula,
```

```

        norm_num_vars = "all",
        encode_cat_vars = "all",
        task = "regression"
    ) %>%

    build_model(
        model_names = "Neural Network",
        hyperparameters =
            list(
                hidden_units = c(3,10),
                activation = c("relu", "tanh")
            )
    ) %>%

    fine_tuning(
        tuner = "Bayesian Optimization",
        metrics = "rmse",
        plot_results = T
    ) %>%

    show_results(
        summary = TRUE,
        scatter_residuals = TRUE,
        scatter_predictions = TRUE
    ) %>%

    sensitivity_analysis(
        type = "Integrated Gradients"
    ) %>%

    sensitivity_analysis(
        type = "Olden"
    )

```

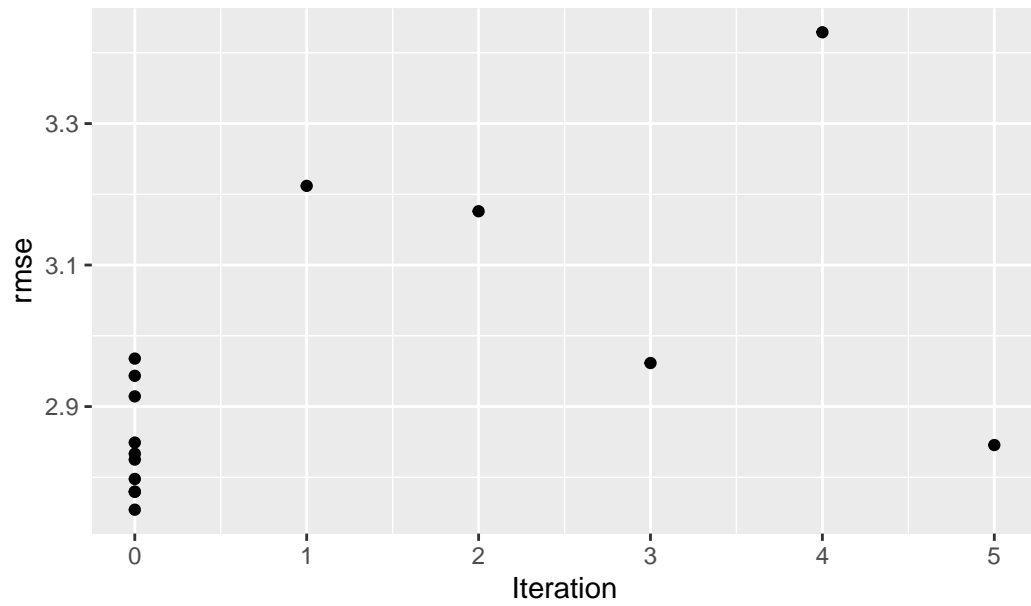
```
[1] "Commencing Tuning..."
```

```
! No improvement for 5 iterations; returning current results.
```

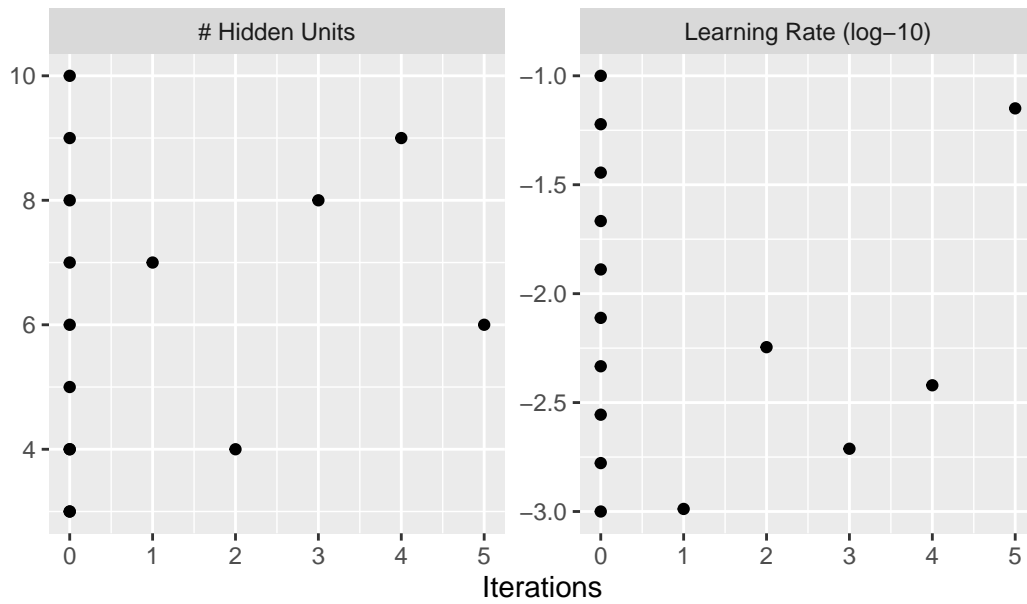
```
[1] "Tuning Finalized"
```

```
[1] "##### Hyperparameter Tuning Results"
```

Bayesian Optimization Iteration Loss



Bayesian Optimization Iteration Results



Bayesian Optimization Search Results



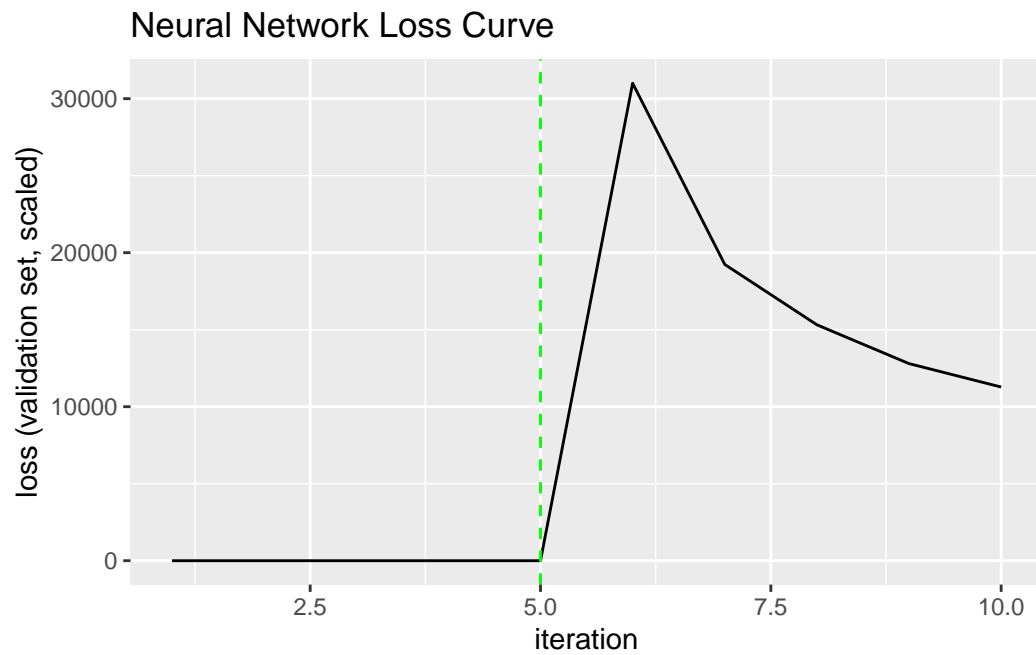
[1] "##### Best Hyperparameters Found:"

A tibble: 1 x 10

	hidden_units	activation	learn_rate	.metric	.estimator	mean	n	std_err
	<int>	<chr>	<dbl>	<chr>	<chr>	<dbl>	<int>	<dbl>
1	9	relu	0.00167	rmse	standard	2.75	1	NA

i 2 more variables: .config <chr>, .iter <int>

[1] "##### Loss Curve"



[1] "##### Showing Results"

Metric	Value
RMSE	2.710
MAE	2.190
MAPE	4.930
MPE	0.639
CCC	0.829
SMAPE	4.960
RPIQ	3.470
RSQ	0.798

