Supervised Machine Learning: Tidy Modeling Approach

Mukti Subedi

25 December 2021

Contents

1 Introduction

As packages in R developed by many volunteers, most of these packages essentially inherit flavor of developers. There are several `R` Packages available in `CRAN`[https://cran.r-project.org/web/packages/available_packages_by_name.html] for supervised and unsupervised machine learning (ML). Most packages don't have unified interface, which, when present offers a consistent and easy-to-use machine learning workflow. **Tidy Modeling** approach which consists of package ecosystem that supports unified workflow in ML including **Data Pre-processing**, **Data Sampling**,**Model Building**, **Hyper Parameters Tuning**, and **Model Validation**.

This tutorial provides a basic introduction to "Supervised Machine Learning" (SML)using **tidymodels** meta-package. ML is a iterative process as modeling process (**Figure 1**).

Exploratory Data Analysis (EDA):

EDA involves general descriptive/inferential data analysis to capture main characteristics of input data, often employing data visualization methods. EDA offers insight as to how the data should be manipulated to achieve the data analysis goal.

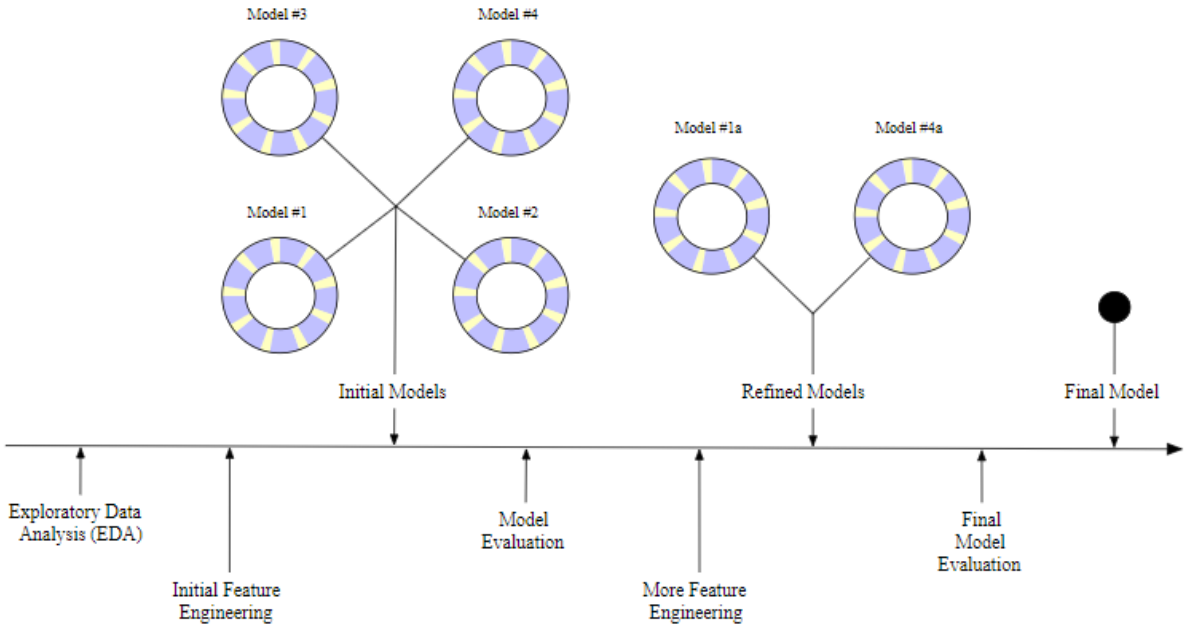

Figure 1: A schematic view of typical modeling process. [source:] (https://www.tmwr.org/premade/modeling-process.svg)

EDA tools include:

- Clustering (unsupervised ML), and dimension reduction (unsupervised ML: e.g., PCA)
- uni/bi variate visualization of variables with summary statistics
- multivariate visualizations.

Feature Engineering (FE):

Broadly speaking clustering and dimensions reduction(e.g. PCA), although falls under EDA it is essentially part of **Feature Engineering** in Machine learning framework. FE uses one or more variables/features to generate another set of variable(s)

Model Tuning and Selection (MTS):

Different ML models have different parameters inherent in their algorithms. Our data may best perform under different combinations of these parameters in specific ML model. MTS allows to find the best combinations of these parameters using either **grid** search, **random** search or other optimization techniques. Automatic finding of such combinations of parameters often called as **"Hyperparameter Tuning"**.

Model Evaluation (ME) :

Finally, ME allows us to judge the performance of ML model. ME has several metrics depending on the nature of ML algorithms. For example, in Regression problem "RMSE: Root Mean Square Error", "MAE: Mean Absolute Error" etc. can be used. Similarly, in classification problem "confusion matrix based metrics can be used", e.g. Kappa statistics, Overall Accuracy, Mathew's correlation coefficient (MCC). Moreover, the ME involves generation of plots "Residual vs fitted", "feature importance" and so on.

In this tutorial, I will be using classification based problem in Land Use/Land Cover Classification of remotely sensed data employing tidymodeling approach

1.1 What is Tidymodels

If you are using R for sometimes, I assume you must be familiar with `tidyverse`. Tidyverse consists of many packages such as `dplyr` `ggplot2`, `tidyr`, `readr`. Tidymodels consists of several packages that shares common syntax in modeling process. If you haven't already you can install these packages together using `install.packages("tidymodels")` command, and load these packages using `library(tidymodels)`. What packages comes with `tidymodels`? you can use `tidymodels_packages()` function to see a list of packages.

```
##  [1] "broom"        "cli"          "conflicted"   "dials"        "dplyr"
##  [6] "ggplot2"      "hardhat"      "infer"        "modeldata"    "parsnip"
## [11] "purrr"        "recipes"      "rlang"        "rsample"      "rstudioapi"
## [16] "tibble"       "tidyr"        "tune"         "workflows"    "workflowsets"
## [21] "yardstick"    "tidymodels"
```

1.2 Load libraries

For ML we need some libraries, for now let's load `tidymodels`, `tidyverse`, `rgdal`, and `sf` packages.

```
# load tidymodels and tidyverse
library(tidymodels) # several modeling packages
library(tidyverse)  # several data manipulating, and graphics packages

# load simple feature (sf), packages to read
# spatial data

library(rgdal)  # geographic data abstraction library
library(sf)     # for loading spatial data
```

1.3 Read Data

I am using point (`shapefile`) data based on training [heads- up digitization] samples created in `ArcGIS` environment. I could have used `arcgisbinding` package to read data directly from geodatabase. If you have ArcGIS desktop or ArcGIS pro you can use `arcgisbinding` package to read your data into R directly from geodatabase (GDB).

```
# load Data
train<- st_read(dsn = "C:/NRM5404/TidyModels/crockettTrainData.shp")
```

```
## Reading layer 'crockettTrainData' from data source
##   'C:\NRM5404\TidyModels\crockettTrainData.shp' using driver 'ESRI Shapefile'
## Simple feature collection with 4084 features and 41 fields
## Geometry type: POINT
## Dimension:     XYZ
## Bounding box:  xmin: 178808.9 ymin: 3352309 xmax: 312624.9 ymax: 3444039
## z_range:       zmin: 0 zmax: 0
## Projected CRS: NAD83 / UTM zone 14N
```

We can now examine the data. Before that we can remove the spatial coordinates, spatial data handling process is slightly different from regular tabular data. To avoid any unwanted consequences, we can remove geometry from the data. However, first lets plot the spatial data then remove the geometry information.

```
# plot spatial data

lulcdata<- train %>%
  ggplot()+
  geom_sf(aes(color = factor(Class_name)))+
  theme_bw()+
```

```
  xlab("Longitude")+
  ylab("Latitude")+
  guides(color = guide_legend(title = "LULC Class"))

lulcdata
```

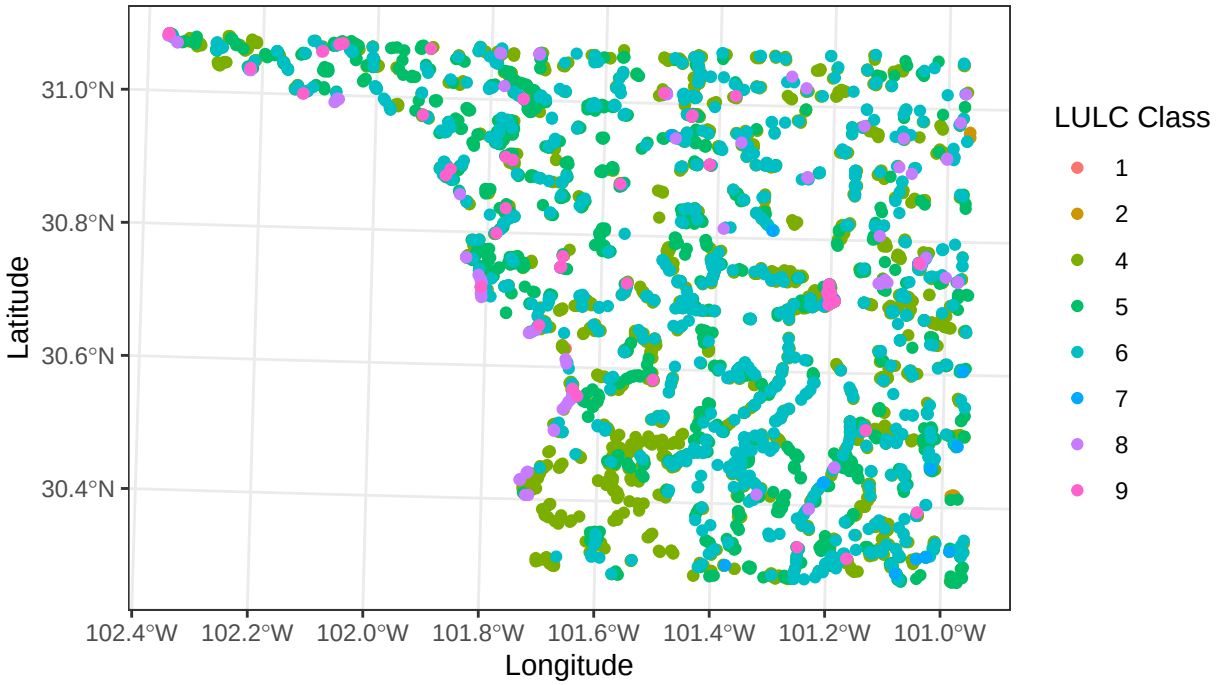

Figure 2: Plot of trainiing data

```
# remove  geometry of data

train<-st_drop_geometry(train)

# Head
head.data<- (train[1:5,1:6])

kbl(head.data, caption =  "First five observations of first six variables",booktabs = TRUE) %>%
  kable_styling(latex_options = c("striped", "hold_position"))
```

1.3.1 Categorical Response Variable

In this example response variable [`Class_name`] is actually categorical. For classification problem, we need to change this variable into factor. For now let's look at how to convert numeric data into factor. Later, I will show how to subset certain classes.

```
# check the data type first
str(train) # the data is numeric
```

Table 1: First five observations of first six variables

Class_name	ASYM	BDRI	COMPT	DENS	MASM
4	0.8526849	2.059259	2.092408	1.541494	0.0836357
4	0.3783444	1.848101	2.422074	2.008263	0.1236753
4	0.7178056	2.094340	1.952358	1.816636	0.1015442
4	0.7422005	3.850427	2.901449	1.393453	0.0910174
4	0.8287658	1.326316	1.444705	1.727289	0.1580023

```
## 'data.frame':    4084 obs. of  41 variables:
##  $ Class_name: num  4 4 4 4 4 4 4 4 4 4 ...
##  $ ASYM      : num  0.853 0.378 0.718 0.742 0.829 ...
##  $ BDRI      : num  2.06 1.85 2.09 3.85 1.33 ...
##  $ COMPT     : num  2.09 2.42 1.95 2.9 1.44 ...
##  $ DENS      : num  1.54 2.01 1.82 1.39 1.73 ...
##  $ MASM      : num  0.0836 0.1237 0.1015 0.091 0.158 ...
##  $ MAXD      : num  3.07 2.98 3.01 2.95 3.13 ...
##  $ MBLU      : num  144 125 131 126 132 ...
##  $ MDIS      : num  1.57 1.1 1.24 1.47 0.8 ...
##  $ MENT      : num  2.72 2.41 2.55 2.67 2.14 ...
##  $ MGRN      : num  162 146 151 142 167 ...
##  $ MHOM      : num  0.489 0.612 0.576 0.539 0.69 ...
##  $ MNDSI     : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ MNDVI     : num  -0.0146 0.0301 0.016 0.0153 -0.4905 ...
##  $ MNDWI     : num  0.00267 -0.03061 -0.02346 -0.01395 0.49753 ...
##  $ MNIR      : num  161 155 158 146 59 ...
##  $ MPC1      : num  316 286 296 277 256 ...
##  $ MPC2      : num  105 103 103 106 106 ...
##  $ MPC23     : num  173.8 185.8 182.7 181.3 94.2 ...
##  $ MRED      : num  166 146 153 142 163 ...
##  $ MSAVI     : num  -0.0309 0.055 0.0292 0.026 -2.1704 ...
##  $ MSTD      : num  1.736 1.386 1.555 1.789 0.996 ...
##  $ RECT      : num  0.73 0.707 0.79 0.551 0.882 ...
##  $ ROUND     : num  1.245 1.471 1.295 1.84 0.767 ...
##  $ SASM      : num  0.0429 0.0798 0.0564 0.054 0.0968 ...
##  $ SAVI      : num  0.0515 0.0801 0.066 0.0859 1.0891 ...
##  $ SBLU      : num  14.87 11.12 13.58 11.45 7.49 ...
##  $ SDIS      : num  1.108 0.916 1.018 1.088 0.705 ...
##  $ SENT      : num  0.439 0.59 0.5 0.489 0.586 ...
##  $ SGRN      : num  13.14 10.5 13.55 12.11 6.28 ...
##  $ SHOM      : num  0.177 0.187 0.177 0.186 0.191 ...
##  $ SHPI      : num  2.28 1.89 2.24 4.29 1.51 ...
##  $ SNDSI     : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ SNDVI     : num  0.0251 0.0432 0.0346 0.0483 0.1566 ...
##  $ SNDVI_1   : num  0.0239 0.0339 0.0307 0.0411 0.1628 ...
##  $ SNIR      : num  11.42 9.18 11.33 14.23 27.41 ...
##  $ SPC1      : num  24.3 18.8 24.6 23.4 20.4 ...
##  $ SPC2      : num  6.93 9.96 8.16 10.1 21.11 ...
##  $ SPC3      : num  4.51 2.73 3.42 3.61 2.69 ...
##  $ SRED      : num  12.08 12.52 14.12 14.02 8.45 ...
##  $ SSTD      : num  1.186 1.244 1.281 1.248 0.769 ...
```

```
# train %>% mutate(Class_name = factor(Class_name, levels = c(4, 6, 8, 5, 1, 7, 9, 2),
#                 labels = c("Grassland", "Built-up1", "Water", "Shrubland", "Cropland", "Built-up2", "
```

1.3.2 Subset variables

Out of 41 variables two (MNDSI, and SNDSI) have constant values. Usually these type of variables are automatically removed as from model building process as they don't add any information to the model. However, we can remove unwanted variables or variables that are not important for the model. Let's get the vector of variables `names ()`

```
# select variables to be used in the training process
var.sel<- c("ASYM", "BDRI", "COMPT", "DENS", "MASM", "MAXD", "MBLU", "MDIS",
            "MENT", "MGRN", "MHOM", "MNDVI", "MNDWI", "MNIR", "MPC1", "MPC2",
            "MPC23", "MRED", "MSAVI", "MSTD", "RECT", "ROUND", "SASM", "SAVI",
            "SBLU", "SDIS", "SENT", "SGRN", "SHOM", "SHPI", "SNDVI", "SNDVI_1",
            "SNIR", "SPC1", "SPC2", "SPC3", "SRED", "SSTD",
           "Class_name")
# filter data
train<- train %>% select(var.sel)

train %>% group_by(Class_name) %>% summarise(frequ = n())
```

```
## # A tibble: 8 x 2
##   Class_name frequ
##        <dbl> <int>
## 1          1     4
## 2          2    15
## 3          4  1918
## 4          5   969
## 5          6   903
## 6          7   113
## 7          8    89
## 8          9    73
```

Now we have our data ready, but still Class_name 1, and 2 have fewer samples. Let's remove them and change the data type of "Class_name" into factor.

```
##
## Grassland Built-up1     Water Shrubland Built-up2    Shadow
##      1918       903        89       969       113        73
```

2 Data Spliting: Train and Test Set

There are several ways to test the models, however, splitting data into certain percentages into training and testing is one of the standard procedure. When data set is small then **leave-one-out**, is one of the way to assess the performance of the model.

Here, we will split data into Training (80 %), and remaining data (20%) will be hold for testing the models, where the former data set will be used to train the model as its name suggests. Test data will be used to evaluate model's performance.

We need access to spatial package for this i.e., `rsample`. In `caret` package similar task is performed using `CreateDataPartition()` function. For reproducibility let's make use of `set.seed()` function.

```
# define set.seed
```

```
set.seed(1318) # any number is fine
# split data into train 80% and test (20%)
train.split <- initial_split(train, prop = 8/10)

# examine
train.split
```

```
## <Analysis/Assess/Total>
## <3252/813/4065>
```

train.split prints `train/test/total` observations. Now training and testing sets can be extracted from the `train.split` object using the `training()` and `testing()` functions.

```
# extract training and testing data
#-- train
dt.train<- training(train.split)

#-- test
dt.test<- testing(train.split)


# for cross validation of results using re-sampling

dt.train.cv<- vfold_cv(dt.train,v = 5)
```

3 Defining a Recipe

When the training and testing data sets are ready, think of this act as an chopping vegetables. Now, different combinations of these vegetables may be required depending on what we want to prepare. This step is considered as recipe in `tidymodeling`. What exactly is the recipe? well in simplest term this allows to define role of variables/features in your data set. One may ask how does the pre-processing fit here or is pre-processing is part of this? the answers is yes. One may run dimension reduction (e.g., PCA), data normalization or imputation. In general recipe creation is two step process or two-layered process.

1. Formula Specification. This is accomplished using `receipe()` to define, response/outcome/dependent variable and predictor/independent variables.
2. Specify pre-processing steps. `step_xxx()` functions.

In this example we don't need a pre-processing. However, for the sake of example. Lets normalize numeric variables

```
# define recipe
classi.recipe<- recipe(Class_name~., data = dt.train) %>%
  step_normalize(all_numeric_predictors())
```

In the above formula `Class_name` is dependent variable (factor), and tilde and period "~." represents the short hand indicating model building using all variables (columns). `all_numeric()` function as a argument in the `step_normalize()` function to pre-process all numeric variables/columns.

We can print the recipe to understand the nature of the recipe. We haven't yet run the model

```
classi.recipe
```

```
## Recipe
##
## Inputs:
##
```

```
##       role #variables
##    outcome          1
##  predictor         38
##
## Operations:
##
## Centering and scaling for all_numeric_predictors()
```

If we want to extract the pre-processed data set, we can first `prep()` the recipe for a specific data set and `juice()` the recipe to extract the pre-processed data. Extracting the pre-processed data isn't actually necessary in data processing pipeline, tidymodels does it for us under the hood when the model is fit. This is just for an example that we can extract recipe if we want.

```
# prep recipe and extract pre-processed data

classi.train.preprocessed<- classi.recipe %>%
  prep(dt.train) %>%
  juice()
classi.train.preprocessed[1:5,1:5]
```

```
## # A tibble: 5 x 5
##     ASYM   BDRI   COMPT    DENS   MASM
##    <dbl>  <dbl>   <dbl>   <dbl>  <dbl>
## 1  0.444  0.420 -0.0527 -0.184  -0.594
## 2 -0.790 -0.773 -0.321   1.09    0.199
## 3 -0.338  2.49   0.178   0.0204 -0.391
## 4 -1.39  -0.717 -0.378   1.44   -0.344
## 5  0.232 -0.358 -0.326   0.553  -0.960
```

4 Model Specification

Now we have ready recipe to make a model. In this example, we will build the popular tree based ensemble machine learning model called 'Random Forest'. We need `parsnip` package for this task.

Usually, four sub-staks are required in the model specification process.

1. Model type: Type of model you want to fit, set using a different function depending on the model, such as `rand_forest()` for random forest, `logistic_reg()` for logistic regression etc.
2. Arguments: the model parameter values (now consistently named across different model, using `set_args()`.
3. Engine: select the package that has the model you want to run e.g., `randomm Forest` we can either use `ranger` or `randomForest` forest. We can use `set_engine()`.
4. Mode: Type of model / prediction e.g. regression, categorical. Make use of `set_engine()` function.

Let's use `rand_forest` (Model), and tune `m_try` (Argument), using `ranger` (engine), for our classification problem (land use/land cover with six classes)

```
rf.model <- rand_forest() %>%
  set_args(mtry = tune(), trees = tune()) %>%
  set_engine("ranger", importance = "impurity") %>%
  set_mode("classification") # regression
```

5 Put Classification in a Workflow

We are now ready to put the model and recipes together into a workflow. Let's initiate a workflow using `workflow()` (package = `workflows`) and then we can add a `recipe` and add a `model` to it.

```
# setting workflows
# set the workflow
rf.workflow <- workflow() %>%
  # add the recipe
  add_recipe(classi.recipe) %>%
  # add the model
  add_model(rf.model)
```

6 Tuning Hyperparameters

We selected the `mtry` parameter to be tuned, we need to tune it before fitting our model. When we don't have any parameters to tune, we can skip tuning process.

Note that we will do our tuning using the hold-out cross-validation object (dt.test). To do this, we specify the range of mtry values we want to try, and then we add a tuning layer to our workflow using `tune_grid()` function (package = tune). Note that we focus on two metrics: ***accuracy and roc_auc*** (package = yardstick).

Multiple parameters can be tuned using `expand.grid()` function. In random forest number of trees, and mtry can be tuned together[see code chunk below]. Also notice that process will run in parallel processing setting using all cores.[I used 12 core Lenovo thinkpad x1 extreme computer]

```
all_cores <- parallel::detectCores(logical = FALSE)

library(doParallel)
```

```
## Loading required package: foreach
```

```
##
## Attaching package: 'foreach'
```

```
## The following objects are masked from 'package:purrr':
##
##     accumulate, when
```

```
## Loading required package: iterators
```

```
## Loading required package: parallel
```

```
cl <- makePSOCKcluster(all_cores)
registerDoParallel(cl)

# specify parameters to be tuned

rf.grid<- expand.grid(mtry= c(1:(ncol(dt.train)-1)), trees = c(500))

# extract results

rf.results<- rf.workflow %>%
  tune_grid(resamples = dt.train.cv,
            grid = rf.grid,
            metrics = metric_set(accuracy,roc_auc))
```

```
# print results

rf.results %>%
  collect_metrics()
```

```
## # A tibble: 76 x 8
##     mtry trees .metric  .estimator  mean     n  std_err .config
##    <int> <dbl> <chr>    <chr>      <dbl> <int>    <dbl> <chr>
##  1     1   500 accuracy multiclass 0.980     5 0.00128  Preprocessor1_Model01
##  2     1   500 roc_auc  hand_till  0.997     5 0.000491 Preprocessor1_Model01
##  3     2   500 accuracy multiclass 0.981     5 0.00164  Preprocessor1_Model02
##  4     2   500 roc_auc  hand_till  0.997     5 0.000474 Preprocessor1_Model02
##  5     3   500 accuracy multiclass 0.981     5 0.00172  Preprocessor1_Model03
##  6     3   500 roc_auc  hand_till  0.997     5 0.000534 Preprocessor1_Model03
##  7     4   500 accuracy multiclass 0.979     5 0.00125  Preprocessor1_Model04
##  8     4   500 roc_auc  hand_till  0.997     5 0.000600 Preprocessor1_Model04
##  9     5   500 accuracy multiclass 0.980     5 0.00188  Preprocessor1_Model05
## 10     5   500 roc_auc  hand_till  0.997     5 0.000573 Preprocessor1_Model05
## # ... with 66 more rows
```

7 Final Model Based on Tuned Parameter(s)

let's extract the best parameter (mtry and trees) and use them to prepare final model to be used for hold-out validation.

```
# rf final parameters
rf.params<- rf.results %>%
  select_best(metric = "accuracy")

# print parameters
rf.params
```

```
## # A tibble: 1 x 3
##    mtry trees .config
##   <int> <dbl> <chr>
## 1     2   500 Preprocessor1_Model02
```

we know now the best mtry = 2, and trees = `rf.params[[1,2]]`

```
rf.workflow<- rf.workflow %>%
  finalize_workflow(rf.params)
```

8 Evaluate the Model: Using Hold-out Test Data Set

So far we have defined recipe, specified model, tuned model's parameters. This allows to check our model, however, how our model behaves on test data set (hold-out data) is not evaluated thus far. To test model's performance, we can use `last_fit()` function on our workflow and train/test split object. Trained model from the workflow and produce performance metrics based on the test set.

```
rf.trainTest.perf <- rf.workflow %>%
  # fit on the training set and evaluate on test set
  last_fit(train.split)
rf.trainTest.perf
```

```
## # Resampling results
```

```
## # Manual resampling
## # A tibble: 1 x 6
##   splits             id               .metrics   .notes  .predictions  .workflow
##   <list>             <chr>            <list>     <list>  <list>        <list>
## 1 <split [3252/813]> train/test split <tibble [~ <tibbl~ <tibble [813~ <workflo~
```

In the above snippet, we supplied the **train/test** object (train.split) when we fit the workflow, the metrics are evaluated on the test set. Now when we use the `collect_metrics()` function, it extracts the performance of the final model applied to the test set. `rf.test.perf`

```
test.perf <- rf.trainTest.perf %>% collect_metrics()
test.perf
```

```
## # A tibble: 2 x 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy multiclass     0.979 Preprocessor1_Model1
## 2 roc_auc  hand_till      0.997 Preprocessor1_Model1
```

`test.per` object suggests that overall performance is excellent ! with a accuracy of 97.909 % and area under receiver operating characteristic curve (roc_auc) is 0.9967

As per the heading, have we done something wrong? No. however, we haven't extracted the test set predictions. This can be achieved using using the `ollect_predictions()` function. Note that there are 813 rows in the predictions object (**rf.trainTest.perf$.predictions**) which matches the number of test set observations `nrow(dt.test)`.

```
# generate predictions from the test set
test.pred <- rf.trainTest.perf %>% collect_predictions()

kbl(test.pred[1:10,1:6], caption =  "Performance of RF on the test data set",
    booktabs = TRUE)%>%
  kable_styling(latex_options = c("striped", "hold_position"))
```

Table 2: Performance of RF on the test data set

id	.pred_Grassland	.pred_Built-up1	.pred_Water	.pred_Shrubland	.pred_Built-up2
train/test split	1.0000000	0.0000000	0.0000000	0.0000000	0.0000000
train/test split	1.0000000	0.0000000	0.0000000	0.0000000	0.0000000
train/test split	0.9982222	0.0017778	0.0000000	0.0000000	0.0000000
train/test split	0.0241587	0.0348944	0.8425206	0.0156119	0.0598056
train/test split	0.9871444	0.0065556	0.0000000	0.0063000	0.0000000
train/test split	0.7556357	0.1994571	0.0062778	0.0277413	0.0070222
train/test split	0.8849698	0.0900508	0.0004444	0.0188556	0.0047079
train/test split	0.9976000	0.0000000	0.0000000	0.0024000	0.0000000
train/test split	0.7454992	0.2308079	0.0057889	0.0075159	0.0061722
train/test split	0.7332254	0.2053516	0.0027222	0.0354627	0.0214810

```
# generate a confusion matrix
test.pred %>%
  conf_mat(truth = Class_name, estimate = .pred_class)
```

```
##            Truth
## Prediction  Grassland Built-up1 Water Shrubland Built-up2 Shadow
##   Grassland       377         1     0         7         0      0
```

```
##   Built-up1         2       182     0         0         1       0
##   Water             0         1    25         0         0       0
##   Shrubland         0         2     0       176         0       0
##   Built-up2         0         2     0         0        20       0
##   Shadow            0         0     1         0         0      16
```

8.1 Variable Importance of the Model

We can use `purrr` functions to extract the predictions column using `pull()` function. `collect_predictions()` function also does similar job extracting prediction from the `.metrics` column.

```
test.prediction <- rf.trainTest.perf %>% pull(.predictions)

kbl(test.prediction[[1]][1:10,c(".row", ".pred_class" ,"Class_name")], caption =  "Pulled prediction in
    booktabs = TRUE)%>%
  kable_styling(latex_options = c("striped", "hold_position"))
```

Table 3: Pulled prediction information

.row	.pred_class	Class_name
21	Grassland	Grassland
27	Grassland	Grassland
29	Grassland	Grassland
31	Water	Water
33	Grassland	Grassland
36	Grassland	Grassland
40	Grassland	Grassland
41	Grassland	Grassland
45	Grassland	Grassland
47	Grassland	Grassland

8.2 How does Final Model Look Like?

Following the usual modeling process we have fitted model using training data, and evaluated using testing data. Once the final model is determined, we can train the model using full data set and use it to predict the response to new data. However, we usually have training data collected from on-screen digitization. In this case, our usual approach of dividing data into training, and testing and prediction to whole area is usually the norm. Nevertheless, we can train final model on your full data set and then use it to predict the response for new data. In this case new data set could be data for whole study area.

We need to use the `fit()` function on workflow and the full data set (trian + test) on which we want to fit the final model on.

```
rf.final.model <- fit(rf.workflow, train)

# let's examine what rf.final.model contains

rf.final.model
```

```
## == Workflow [trained] ==========================================================
## Preprocessor: Recipe
## Model: rand_forest()
##
## -- Preprocessor ----------------------------------------------------------------
```

```
## 1 Recipe Step
##
## * step_normalize()
##
## -- Model ---------------------------------------------------------------------
## Ranger result
##
## Call:
##  ranger::ranger(x = maybe_data_frame(x), y = y, mtry = min_cols(~2L,      x), num.trees = ~500, impo
##
## Type:                             Probability estimation
## Number of trees:                  500
## Sample size:                      4065
## Number of independent variables:  38
## Mtry:                             2
## Target node size:                 10
## Variable importance mode:         impurity
## Splitrule:                        gini
## OOB prediction error (Brier s.):  0.02501494
```

8.2.1 Variable Importance

We have already examine confusion matrix, and we can calculate various model performance matrices using confusion matrix. Another important step that is often reported is variable importance. Random forest has two types of variable importance (gini, and accuracy). variable importance can be extracted and plot using **vip** package. Here, we can extract `fit()` object from final model `rf.final.model`, for which `extract_fit_parsnip()`function should be used to extract fit object.

```
library(vip)
```

```
##
## Attaching package: 'vip'
```

```
## The following object is masked from 'package:utils':
##
##     vi
```

```
# pull_workflow_fit(rf.final.model)$fit %>%
# vip(geom = "point")

vimp<- extract_fit_parsnip(rf.final.model)$fit %>%
  vip(geom = "col")+
  theme_bw(base_size = 12)
```

In the above code chunk, `pull_workflow_fit()`function returns message that this function is deprecated in version 0.2.3. However, the function as of today is working just fine.

To sum up, in this example, we used spatial training data of Crockett County of Texas in multiclass land use land cover classification, we use 5-fold cross validation in random forest model building using parallel processing. We tuned few parameters, and evaluated model creating confusion matrix on hold-out (20 % of the total data) data set. Finally, we used vip package to plot and display importance variable.

The End

```
sessionInfo()
```

```
## R version 4.1.2 (2021-11-01)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
```

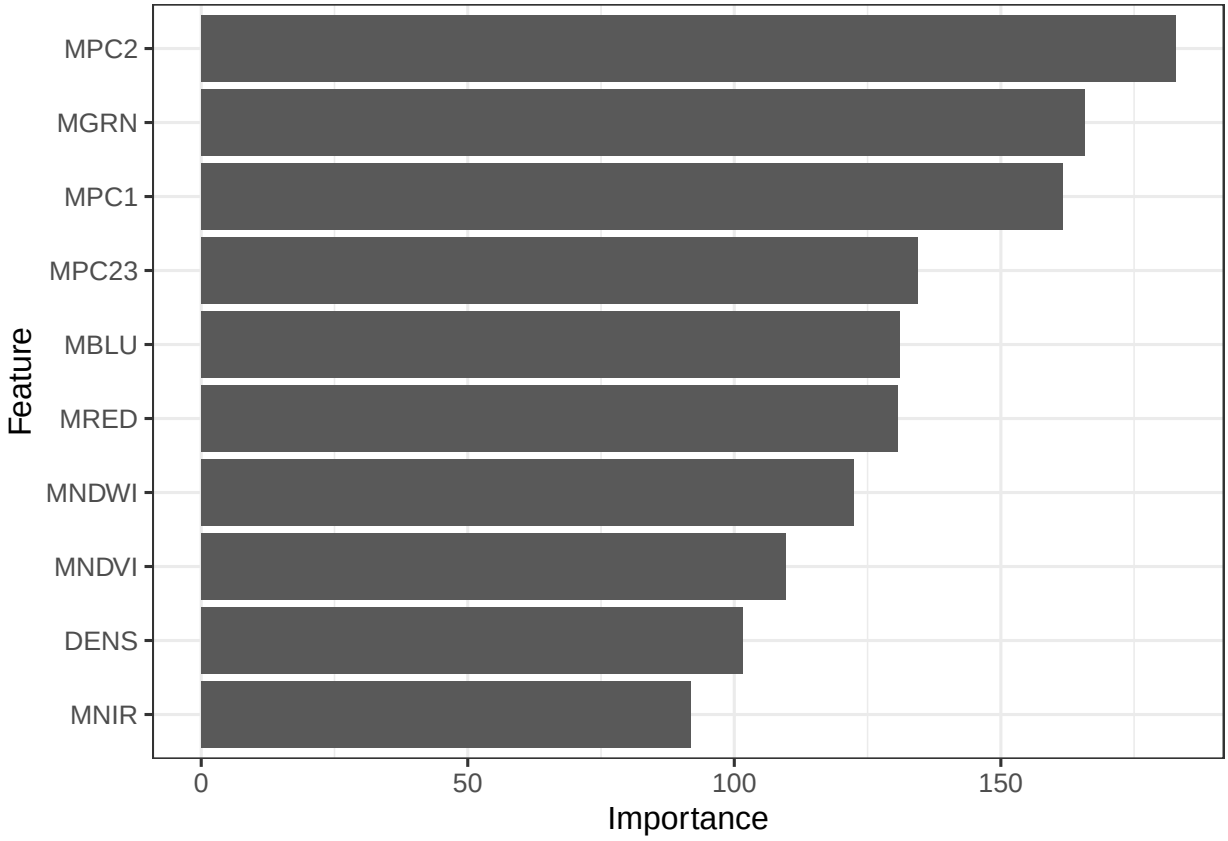

Figure 3: Feature Importance

```
## Running under: Windows 10 x64 (build 19042)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=English_United States.1252
## [2] LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## attached base packages:
## [1] parallel  stats     graphics  grDevices utils     datasets  methods
## [8] base
##
## other attached packages:
##  [1] vip_0.3.2          doParallel_1.0.16  iterators_1.0.13   foreach_1.5.1
##  [5] kableExtra_1.3.4   sf_1.0-5           rgdal_1.5-28       sp_1.4-6
##  [9] forcats_0.5.1      stringr_1.4.0      readr_2.1.1        tidyverse_1.3.1
## [13] yardstick_0.0.9    workflowsets_0.1.0 workflows_0.2.4    tune_0.1.6
## [17] tidyr_1.1.4        tibble_3.1.6       rsample_0.1.1      recipes_0.1.17
## [21] purrr_0.3.4        parsnip_0.1.7      modeldata_0.1.1    infer_1.0.0
## [25] ggplot2_3.3.5      dplyr_1.0.7        dials_0.0.10       scales_1.1.1
## [29] broom_0.7.10       tidymodels_0.1.4   knitr_1.37
##
## loaded via a namespace (and not attached):
##  [1] colorspace_2.0-2   ellipsis_0.3.2     class_7.3-19       fs_1.5.2
##  [5] rstudioapi_0.13    proxy_0.4-26       farver_2.1.0       listenv_0.8.0
##  [9] furrr_0.2.3        prodlim_2019.11.13 fansi_0.5.0        lubridate_1.8.0
## [13] ranger_0.13.1      xml2_1.3.3         codetools_0.2-18   splines_4.1.2
## [17] jsonlite_1.7.2     pROC_1.18.0        dbplyr_2.1.1       compiler_4.1.2
## [21] httr_1.4.2         backports_1.4.1    assertthat_0.2.1   Matrix_1.3-4
## [25] fastmap_1.1.0      cli_3.1.0          htmltools_0.5.2    tools_4.1.2
## [29] gtable_0.3.0       glue_1.6.0         Rcpp_1.0.7         cellranger_1.1.0
## [33] DiceDesign_1.9     vctrs_0.3.8        svglite_2.0.0      timeDate_3043.102
## [37] gower_0.2.2        xfun_0.29          globals_0.14.0     rvest_1.0.2
## [41] lifecycle_1.0.1    future_1.23.0      MASS_7.3-54        ipred_0.9-12
## [45] hms_1.1.1          yaml_2.2.1         gridExtra_2.3      rpart_4.1-15
## [49] stringi_1.7.6      e1071_1.7-9        lhs_1.1.3          hardhat_0.1.6
## [53] lava_1.6.10        systemfonts_1.0.3  rlang_0.4.12       pkgconfig_2.0.3
## [57] evaluate_0.14      lattice_0.20-45    labeling_0.4.2     tidyselect_1.1.1
## [61] parallelly_1.30.0  plyr_1.8.6         magrittr_2.0.1     R6_2.5.1
## [65] generics_0.1.1     DBI_1.1.1          pillar_1.6.4       haven_2.4.3
## [69] withr_2.4.3        units_0.7-2        survival_3.2-13    nnet_7.3-16
## [73] future.apply_1.8.1 modelr_0.1.8       crayon_1.4.2       KernSmooth_2.23-20
## [77] utf8_1.2.2         tzdb_0.2.0         rmarkdown_2.11     grid_4.1.2
## [81] readxl_1.3.1       webshot_0.5.2      reprex_2.0.1       digest_0.6.29
## [85] classInt_0.4-3     GPfit_1.0-8        munsell_0.5.0      viridisLite_0.4.0
```