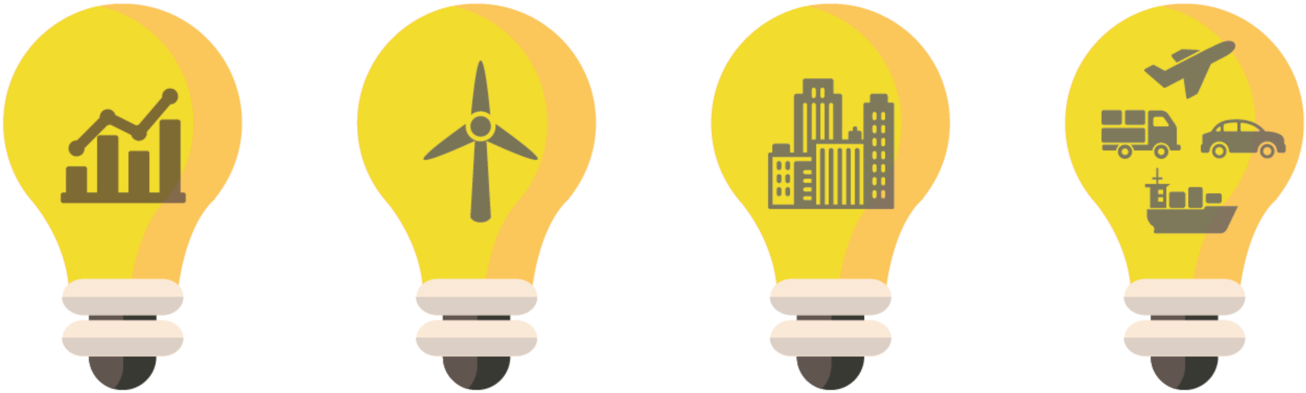
BENV0091 Lecture 9: Tidymodels

Patrick de Mars



Tidymodels

- Tidymodels is a collection of packages that applies tidyverse principles for modelling and machine learning
- We can easily create modular **pipelines** with steps for data pre-processing, resampling, model selection, model fitting and model evaluation
- Task: install tidymodels

Data: NILM

- We will return to the Voltaware dataset for non-intrusive load monitoring (NILM) used for the Kaggle competition
- Our task is to **classify appliances** based on voltage and current measurements and other derived data
- Task: read the data and apply the following initial processing steps:
  - Sample 10% of the data (purely to speed-up model fitting!)
  - Sort by timestamp
  - Add a week column
  - Drop any rows with NA in the appliances column

```r
df <- read_csv('data/vw_train.csv') %>%
  sample_frac(0.1, replace = FALSE) %>% # choose 10% of the data
  arrange(timestamp) %>% # sort data by timestamp
  mutate(week = week(timestamp)) %>% # add a week number column
  drop_na(appliances)
```

Splitting Data

- From `rsample`, we can split the data into train and test sets using two main functions:
  - `initial_split()`: data is split randomly. Can be **stratified**
  - `initial_time_split()`: data is split such that the first *prop* is used for training, remainder for testing. Suitable for time series data
- The train and test data frames can be retrieved from the split object with `training()` and `testing()`
- Task: split the data using `initial_time_split()` and retrieve the train and test data frames

```
set.seed(123)
data_split <- initial_time_split(df, prop = 3/4)

train <- training(data_split)
test <- testing(data_split)
```

Overview: Code Templates

- The code on the right shows a general template for a glmnet model (from the `**usemodels**` package)
- The main components are:
  - **Recipe:** defines data pre-processing steps
  - **Model specification:** defines the estimator, as well as hyperparameters (including those you want to tune)
  - **Workflow:** an object that combines a recipe and a model. The workflow can be fit to the data and used to make predictions
  - **Hyperparameter tuning:** we specify a grid of hyperparameters to try and fit the workflow with each unique combination to resampled data (such as in k-fold cross validation)

Model template for glmnet

Pre-processing

Model definition

Workflow

Hyperparameter tuning

```
> use_glmnet(appliances ~ ., data = train)
glmnet_recipe <-
  recipe(formula = appliances ~ ., data = train) %>%
  step_string2factor(one_of(appliances)) %>%
  step_zv(all_predictors()) %>%
  step_normalize(all_predictors(), -all_nominal())

glmnet_spec <-
  multinom_reg(penalty = tune(), mixture = tune()) %>%
  set_mode("classification") %>%
  set_engine("glmnet")

glmnet_workflow <-
  workflow() %>%
  add_recipe(glmnet_recipe) %>%
  add_model(glmnet_spec)

glmnet_grid <- tidyr::crossing(penalty = 10^seq(-6, -1, length.out = 20), mixture = c(0.05,
    0.2, 0.4, 0.6, 0.8, 1))

glmnet_tune <-
  tune_grid(glmnet_workflow, resamples = stop("add your rsample object"), grid = glmnet_grid)
```

Templates available from usemodels

"use_cubist"	"use_earth"	"use_glmnet"
"use_kknn"	"use_ranger"	"use_xgboost"

Pre-Processing: Recipes

- The `recipes` package is designed to help rigorously pre-process your data before feeding it to a model: **both during training and testing**
- Its capabilities include (see here for complete reference):
  - Defining predictors and response variables
  - Creating dummy variables
  - Normalising features
  - Creating date-time features
  - Cleaning text for categorical variables
  - Imputing missing data

```
rec <-
  recipe(appliances ~ ., data = train) %>% # define formula
  update_role(id, timestamp, new_role = "ID") # set timestamp as ID
```

- The recipe must define which variables will be response and predictors
- Further steps can optionally be created using `step_*()` functions
- Task: create the recipe shown
  - Note: update_role() makes timestamp an ID variable: it will not be used as a predictor
- Later, we will add this recipe to a **workflow**

*Remember: a **workflow** combines a recipe and a model*

Further Steps

- The recipe below expands the previous recipe with the following steps:
  - Make timestamp an ”ID” variable (will not be used as a predictor)
  - Create day of week dummy variables from timestamp
  - Create dummy variables for all categorical variables
  - Remove predictors with zero variance
  - Impute missing values for all numeric predictors using a K-nearest neighbours algorithm
  - Assign an “other” category to appliances if they make up <1% of total
- Task: update your recipe with the steps below

```r
rec <-
  recipe(appliances ~ ., data = train) %>% # define formula
  update_role(id, timestamp, new_role = "ID") %>% # set timestamp as ID
  step_mutate(power = current * voltage) %>% # create a new variable: power
  step_other(appliances, threshold = 0.01, skip = TRUE) %>%
  step_dummy(all_nominal_predictors()) %>% # create dummy vars
  step_zv(all_predictors()) %>% # remove zero variance predictors
  step_impute_knn(all_numeric_predictors())  # impute missing values
```

Calling a recipe prints the pre-processing steps

```
> rec
Data Recipe

Inputs:

      role #variables
        ID          2
   outcome          1
 predictor         28

Operations:

Variable mutation for power
Collapsing factor levels for appliances
Dummy variables from all_nominal_predictors()
Zero variance filter on all_predictors()
K-nearest neighbor imputation for all_numeric_predictors()
```

parsnip

- The parsnip package provides a standard standard interface for fitting and evaluating models
- Currently parsnip offers 30 different models: see here for a full list
- A model typically requires:
  - An **engine**: the ‘backend’ software used to fit the model
  - A **mode**: regression or classification
  - **Hyperparameters:** model-specific parameters such as number of trees
- Task: use `show_engine()` to see the available engines for the following models:
  - ”rand_forest”
  - “boost_tree”
  - “mlp”
  - “logistic_reg”

Defining a Model

- Now we will specify a model (in this case **gradient boosting trees**)
  - The **engine** is set to "xgboost"
  - The **mode** is set to "classification"
- Tidymodels allows you to set hyperparameters to **`tune()`:** this means we will tune these variables during cross-validation (or another model selection method)
- Task: define the gradient boosting tree model below

```
model_spec <- boost_tree(mtry = 5,
                         min_n = tune(),
                         tree_depth = tune(),
                         trees = 500) %>%
  set_engine('xgboost') %>%
  set_mode('classification')
```

Workflows

- A workflow combines a **recipe** (for pre-processing) and a **model specification.** This ensures that pre-processing steps are robustly followed both when fitting and evaluating the model
- Task: create a workflow combining the recipe with the model specification
- We can use \`fit()\` and \`predict()\` with a workflow. However:
  - *All hyperparameters must be specified in order to \`fit()\` - **we will complete this after hyperparameter tuning***
  - *The workflow must be fitted before we can use \`predict()\`*

Adding a recipe and a model

```
wflow <- workflow() %>%
  add_recipe(rec) %>%
  add_model(model_spec)
```

Fitting and predicting

```
fit(wflow, train)
predict(wflow, test)
```

Hyperparameter Tuning: Resampling

- Using `rsample`, we can easily create fit/validation data sets using a variety of resampling methods such as:
  - K-fold CV
  - Bootstrapping
  - Leave-one-out CV
  - Monte Carlo CV
- Here we will use <u>grouped</u> k-fold CV, with data grouped by <u>week:</u>
  - Data points from the same week will not be split up across different weeks, thus reducing risks of data leakage
- Task: create 5 data splits using `group_vfold_cv()`
  - Try using `bootstraps()` to create bootstrapped resamples

Output of grouped k-fold CV resampling

```
> resamples
# Group 5-fold cross-validation
# A tibble: 5 × 2
  splits             id
  <list>             <chr>
1 <split [3896/2409]> Resample1
2 <split [4063/2242]> Resample2
3 <split [5928/377]>  Resample3
4 <split [5267/1038]> Resample4
5 <split [6066/239]>  Resample5
```

```
n_resamples <- 5
resamples <- group_vfold_cv(train, week, n_resamples)
```

Hyperparameter Tuning: Grid Search

- We have now set up our model specification and created splits in the data to use for cross validation. Next, we should define the hyperparameter settings we want to try
- In this example we have 3 settings for each parameter: `min_n` and tree_depth`, so **9 combinations in total.**
- As we have split the training data into **5 folds**, tuning the model will requires us to fit **9*5 = 45 models**. We will then choose the parameter combination which has the highest accuracy over the 5 folds
- Task: use `tune_grid()` to fit models for each of the parameter combinations using k-fold CV
  - Note: you may want to remove one or two parameters from the grid to speed things up!

```
grid <- expand.grid(min_n = c(20, 50, 100), tree_depth = c(10, 20, 50))
tune_output <-
  tune_grid(wflow,
            resamples = resamples,
            grid = grid)
```

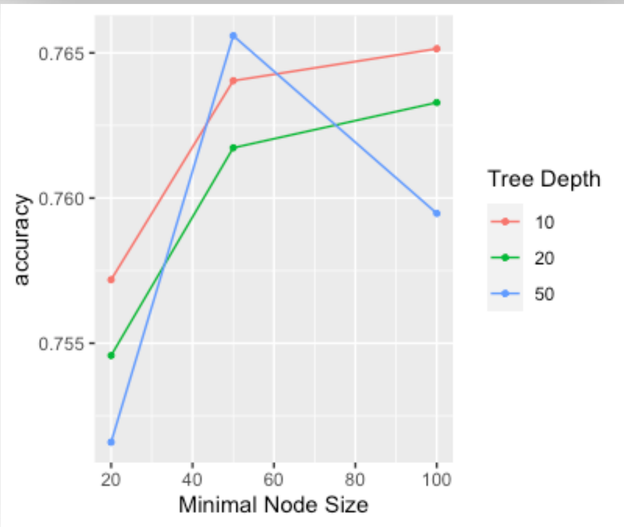
```
> grid
  min_n tree_depth
1    20         10
2    50         10
3   100         10
4    20         20
5    50         20
6   100         20
7    20         50
8    50         50
9   100         50
```

For tune_grid(), the parameters should be passed as a data frame with a column for each parameter to tune

Hyperparameter Tuning: Finalising the Model

- Now that we have fitted the models, we can retrieve the best parameter combination with `show_best()`
- We can also plot how the accuracy varies with the parameter settings using `autoplot()`
  - You may want to consider rerunning your hyperparameter tuning with a different selection of parameters
- Finally, we can finalise the workflow with the best parameters
- Task: finalise the workflow

```
> show_best(bt_tune, metric = 'accuracy')
# A tibble: 5 × 8
  min_n tree_depth .metric  .estimator  mean
  <dbl>      <dbl> <chr>    <chr>      <dbl>
1    50         50 accuracy multiclass 0.766
2   100         10 accuracy multiclass 0.765
3    50         10 accuracy multiclass 0.764
4   100         20 accuracy multiclass 0.763
5    50         20 accuracy multiclass 0.762
```




```
autoplot(bt_tune, metric = 'accuracy')
```

```
final_wflow <- wflow %>%
  finalize_workflow(select_best(tune_output, metric = 'accuracy'))
```

Evaluating the Model

- Now we can finally fit the model to the **entire** training data and make predictions for the test set
- Task:
  - Use `fit()` to fit the workflow to your training data
  - Use `predict()` to make predictions for the testing data
  - What is the accuracy of your model on the test set?
  - To what extent has your model overfitted to the training data? How can this be avoided?

```
final_fit <- fit(final_wflow, train)
preds <- predict(final_fit, test)
```

Recap

- To sum up, we have:
  - Created a recipe for data pre-processing
  - Specified a model
  - Created a workflow combining the model specification and recipe
  - Use grid search and k-fold CV to find a good set of hyperparameters
  - Finalised the workflow by updating the model with the tuned hyperparameters
  - Fitted the finalised workflow to the training data
  - Evaluated the model on the held-out set

Improving the Model

- Despite some hyper-parameter tuning, the gradient boosting trees model is still prone to overfitting
- Some ideas for improving the performance of your models:
  - Use a different model such as a linear model with regularisation
  - Further hyperparameter tuning:
    - More trees
    - Shallower trees
    - Fewer variables selected for each tree
  - Feature engineering
- **You are still able to make a late submission to the Kaggle competition if you're interested in how your model performs on truly unseen data!**

Further Reading

- The Tidy Modeling with R book is a great resource and while help you go deeper with tidymodels
- The author, Julia Silge, also posts excellent tutorial videos on her Youtube channel