Decision trees

Jin Seo Jo

Cell images

We are going to look at an example where the data is an evaluation of automatic cell segmentation algorithms. For each cell, an automated agorithm is run to try and detect cell boundaries, and then a human looks at the output and declares the cell to be "well segmented" (`WS`) or "poorly segmented" (`PS`). More infomation about the data can be found here.

The human is expensive, so we want to see if there are measurable characteristics of the cell that can predicit whether or not a cell is well segmented. If this is accurate it can massively reduce the human cost to analysing this sort of data.

```
library(tidymodels)
```

```
## -- Attaching packages -------------------------------------- tidymodels 0.1.3 --

## v broom        0.7.6      v recipes      0.1.16
## v dials        0.0.9      v rsample      0.0.9
## v dplyr        1.0.5      v tibble       3.1.1
## v ggplot2      3.3.3      v tidyr        1.1.3
## v infer        0.5.4      v tune         0.1.5
## v modeldata    0.1.0      v workflows    0.2.2
## v parsnip      0.1.5      v workflowsets 0.0.2
## v purrr        0.3.4      v yardstick    0.0.8

## -- Conflicts ----------------------------------------- tidymodels_conflicts() --
## x purrr::discard() masks scales::discard()
## x dplyr::filter()  masks stats::filter()
## x dplyr::lag()     masks stats::lag()
## x recipes::step()  masks stats::step()
## * Use tidymodels_prefer() to resolve common conflicts.
```

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v readr   1.4.0     v forcats 0.5.1
## v stringr 1.4.0

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x readr::col_factor() masks scales::col_factor()
## x purrr::discard()    masks scales::discard()
## x dplyr::filter()     masks stats::filter()
```

```
## x stringr::fixed()    masks recipes::fixed()
## x dplyr::lag()        masks stats::lag()
## x readr::spec()       masks yardstick::spec()
```

```r
library(modeldata)
data(cells)
cells <- cells %>% select(-case)
cells
```

```
## # A tibble: 2,019 x 57
##    class angle_ch_1 area_ch_1 avg_inten_ch_1 avg_inten_ch_2 avg_inten_ch_3
##    <fct>      <dbl>     <int>          <dbl>          <dbl>          <dbl>
##  1 PS        143.         185           15.7           4.95           9.55
##  2 PS        134.         819           31.9         207.            69.9
##  3 WS        107.         431           28.0         116.            63.9
##  4 PS         69.2        298           19.5         102.            28.2
##  5 PS          2.89       285           24.3         112.            20.5
##  6 WS         40.7        172          326.          654.           129.
##  7 WS        174.         177          260.          596.           124.
##  8 PS        180.         251           18.3           5.73          17.2
##  9 WS         18.9        495           16.1          89.5           13.7
## 10 WS        153.         384           17.7          89.9           20.4
## # ... with 2,009 more rows, and 51 more variables: avg_inten_ch_4 <dbl>,
## #   convex_hull_area_ratio_ch_1 <dbl>, convex_hull_perim_ratio_ch_1 <dbl>,
## #   diff_inten_density_ch_1 <dbl>, diff_inten_density_ch_3 <dbl>,
## #   diff_inten_density_ch_4 <dbl>, entropy_inten_ch_1 <dbl>,
## #   entropy_inten_ch_3 <dbl>, entropy_inten_ch_4 <dbl>,
## #   eq_circ_diam_ch_1 <dbl>, eq_ellipse_lwr_ch_1 <dbl>,
## #   eq_ellipse_oblate_vol_ch_1 <dbl>, eq_ellipse_prolate_vol_ch_1 <dbl>,
## #   eq_sphere_area_ch_1 <dbl>, eq_sphere_vol_ch_1 <dbl>,
## #   fiber_align_2_ch_3 <dbl>, fiber_align_2_ch_4 <dbl>,
## #   fiber_length_ch_1 <dbl>, fiber_width_ch_1 <dbl>, inten_cooc_asm_ch_3 <dbl>,
## #   inten_cooc_asm_ch_4 <dbl>, inten_cooc_contrast_ch_3 <dbl>,
## #   inten_cooc_contrast_ch_4 <dbl>, inten_cooc_entropy_ch_3 <dbl>,
## #   inten_cooc_entropy_ch_4 <dbl>, inten_cooc_max_ch_3 <dbl>,
## #   inten_cooc_max_ch_4 <dbl>, kurt_inten_ch_1 <dbl>, kurt_inten_ch_3 <dbl>,
## #   kurt_inten_ch_4 <dbl>, length_ch_1 <dbl>, neighbor_avg_dist_ch_1 <dbl>,
## #   neighbor_min_dist_ch_1 <dbl>, neighbor_var_dist_ch_1 <dbl>,
## #   perim_ch_1 <dbl>, shape_bfr_ch_1 <dbl>, shape_lwr_ch_1 <dbl>,
## #   shape_p_2_a_ch_1 <dbl>, skew_inten_ch_1 <dbl>, skew_inten_ch_3 <dbl>,
## #   skew_inten_ch_4 <dbl>, spot_fiber_count_ch_3 <int>,
## #   spot_fiber_count_ch_4 <dbl>, total_inten_ch_1 <int>,
## #   total_inten_ch_2 <dbl>, total_inten_ch_3 <int>, total_inten_ch_4 <int>,
## #   var_inten_ch_1 <dbl>, var_inten_ch_3 <dbl>, var_inten_ch_4 <dbl>,
## #   width_ch_1 <dbl>
```

Look for imbalance in this data and prepare test and training sets taking to account any imbalance. Check if the test and training sets are balanced.

```
## # A tibble: 2 x 3
##   class     n  prop
##   <fct> <int> <dbl>
## 1 PS     1300 0.644
## 2 WS      719 0.356
```

```
## # A tibble: 2 x 3
##   class     n  prop
##   <fct> <int> <dbl>
## 1 PS      325 0.645
## 2 WS      179 0.355
```

```
## # A tibble: 2 x 3
##   class     n  prop
##   <fct> <int> <dbl>
## 1 PS      975 0.644
## 2 WS      540 0.356
```

This data is all organized pretty nicely so we don't really need a recipe. This means that instead of an `add_recipe` step we can use `add_formula` when building our workflow.

A random forest model

Random forests are pretty charming because we don't really need to do much to tune them. In `R`, they can be fit using the `ranger` package. We set up the model like this.

```
rf_spec <- rand_forest(trees = 1000) %>%
  set_engine("ranger") %>%
  set_mode("classification")
```

Here the argument `trees` controls how many trees should be grown as part of the random forest. We can then build our workflow.

```
wf_rf <- workflow() %>% add_model(rf_spec ) %>% add_formula(class ~ .)
```

We can now fit the random forest.

Fit the model on the entire training data and estimate the `accuracy` and `roc_auc` on the *training* set. Compare that to these metrics on the *test* set.

Note: We will need to predict twice: once with `type = "class` and once with `type = "prob"`.

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc binary          1.00
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.993
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc binary         0.904
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.839
```

We should have noticed that the training error was too optimistic. We can fix that by using cross validation with `fit_resamples` to fit a separate random forest to each fold.

```
folds <- vfold_cv(test, v = 10)
fit_rf_resample <- wf_rf %>% fit_resamples(folds)
fit_rf_resample %>% collect_metrics()
```

```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.821    10  0.0139 Preprocessor1_Model1
## 2 roc_auc  binary     0.894    10  0.0113 Preprocessor1_Model1
```

Much better!

Now let's compare with decision trees

We can also fit the data using a single decision tree. But in this case we will need to do more work to tune it!

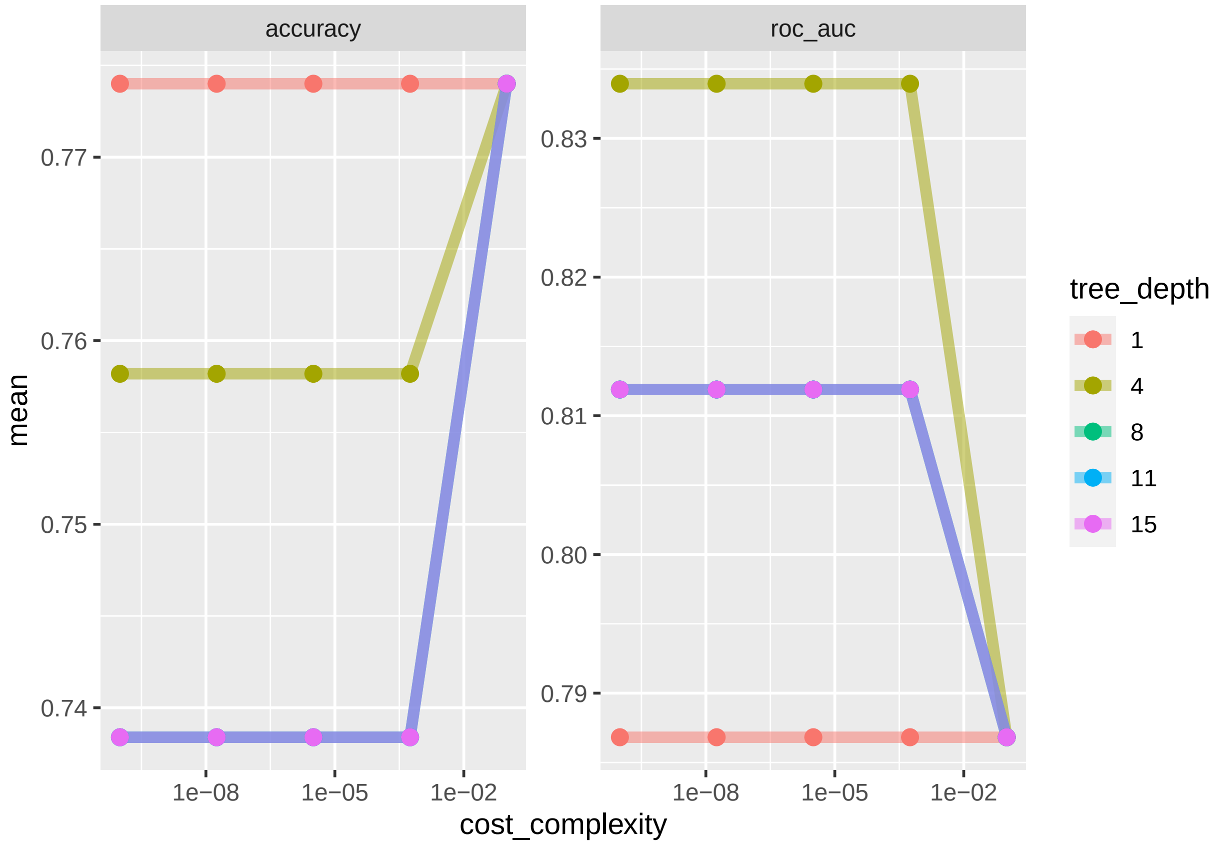
Parsnip has a `decision_tree` function, which is driven by the `rpart` package. There are several parameters to tune, but the key ones are the `cost_complexity`, which controls the pruning, and `tree_depth` which controls how deep the initial tree is grown before being pruned.

```
spec_dt <- decision_tree(
  cost_complexity = tune(),
  tree_depth = tune()
  ) %>%
  set_engine("rpart") %>%
  set_mode("classification")

grid_dt <- grid_regular(cost_complexity(), tree_depth(), levels = 5)
grid_dt
```

```
## # A tibble: 25 x 2
##    cost_complexity tree_depth
##              <dbl>      <int>
##  1    0.0000000001          1
##  2    0.0000000178          1
##  3    0.00000316            1
##  4    0.000562              1
##  5    0.1                   1
##  6    0.0000000001          4
##  7    0.0000000178          4
##  8    0.00000316            4
##  9    0.000562              4
## 10    0.1                   4
## # ... with 15 more rows
```

Fit and tune this model and plot the metrics (x-axis: cost_complexity, y-axis: metric, colour: tree_depth).




```
## Warning: No value of 'metric' was given; metric 'roc_auc' will be used.

## # A tibble: 5 x 8
##   cost_complexity tree_depth .metric .estimator  mean     n std_err .config
##             <dbl>      <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1    0.0000000001          4 roc_auc binary     0.834    10  0.0177 Preprocesso~
## 2    0.0000000178          4 roc_auc binary     0.834    10  0.0177 Preprocesso~
## 3    0.00000316            4 roc_auc binary     0.834    10  0.0177 Preprocesso~
## 4    0.000562              4 roc_auc binary     0.834    10  0.0177 Preprocesso~
## 5    0.0000000001          8 roc_auc binary     0.812    10  0.0166 Preprocesso~
```

Once we finalize the workflow we can look at the best tree!

```
wf_dt_final <- wf_dt %>% finalize_workflow(best_tree)
final_tree <- wf_dt_final %>% fit(train)
final_tree
```

```
## == Workflow [trained] ==========================================================
## Preprocessor: Formula
## Model: decision_tree()
##
## -- Preprocessor ----------------------------------------------------------------
## class ~ .
##
```

```
## -- Model ------------------------------------------------------------------------
## n= 1515
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 1515 540 PS (0.64356436 0.35643564)
##    2) total_inten_ch_2< 43111.5 654  42 PS (0.93577982 0.06422018)
##      4) avg_inten_ch_2< 126.9228 510  13 PS (0.97450980 0.02549020) *
##      5) avg_inten_ch_2>=126.9228 144  29 PS (0.79861111 0.20138889)
##       10) total_inten_ch_1< 30993.5 128  19 PS (0.85156250 0.14843750)
##         20) skew_inten_ch_1>=-0.03644029 121  14 PS (0.88429752 0.11570248) *
##         21) skew_inten_ch_1< -0.03644029 7   2 WS (0.28571429 0.71428571) *
##       11) total_inten_ch_1>=30993.5 16   6 WS (0.37500000 0.62500000) *
##    3) total_inten_ch_2>=43111.5 861 363 WS (0.42160279 0.57839721)
##      6) fiber_width_ch_1< 11.35657 395 155 PS (0.60759494 0.39240506)
##       12) kurt_inten_ch_1>=-0.3452187 262  72 PS (0.72519084 0.27480916)
##         24) var_inten_ch_1< 214.8773 247  59 PS (0.76113360 0.23886640) *
##         25) var_inten_ch_1>=214.8773 15   2 WS (0.13333333 0.86666667) *
##       13) kurt_inten_ch_1< -0.3452187 133  50 WS (0.37593985 0.62406015)
##         26) total_inten_ch_1< 13594 25   6 PS (0.76000000 0.24000000) *
##         27) total_inten_ch_1>=13594 108  31 WS (0.28703704 0.71296296) *
##      7) fiber_width_ch_1>=11.35657 466 123 WS (0.26394850 0.73605150)
##       14) convex_hull_area_ratio_ch_1>=1.070151 165  67 WS (0.40606061 0.59393939)
##         28) total_inten_ch_2>=145395.5 32   8 PS (0.75000000 0.25000000) *
##         29) total_inten_ch_2< 145395.5 133  43 WS (0.32330827 0.67669173) *
##       15) convex_hull_area_ratio_ch_1< 1.070151 301  56 WS (0.18604651 0.81395349) *
```

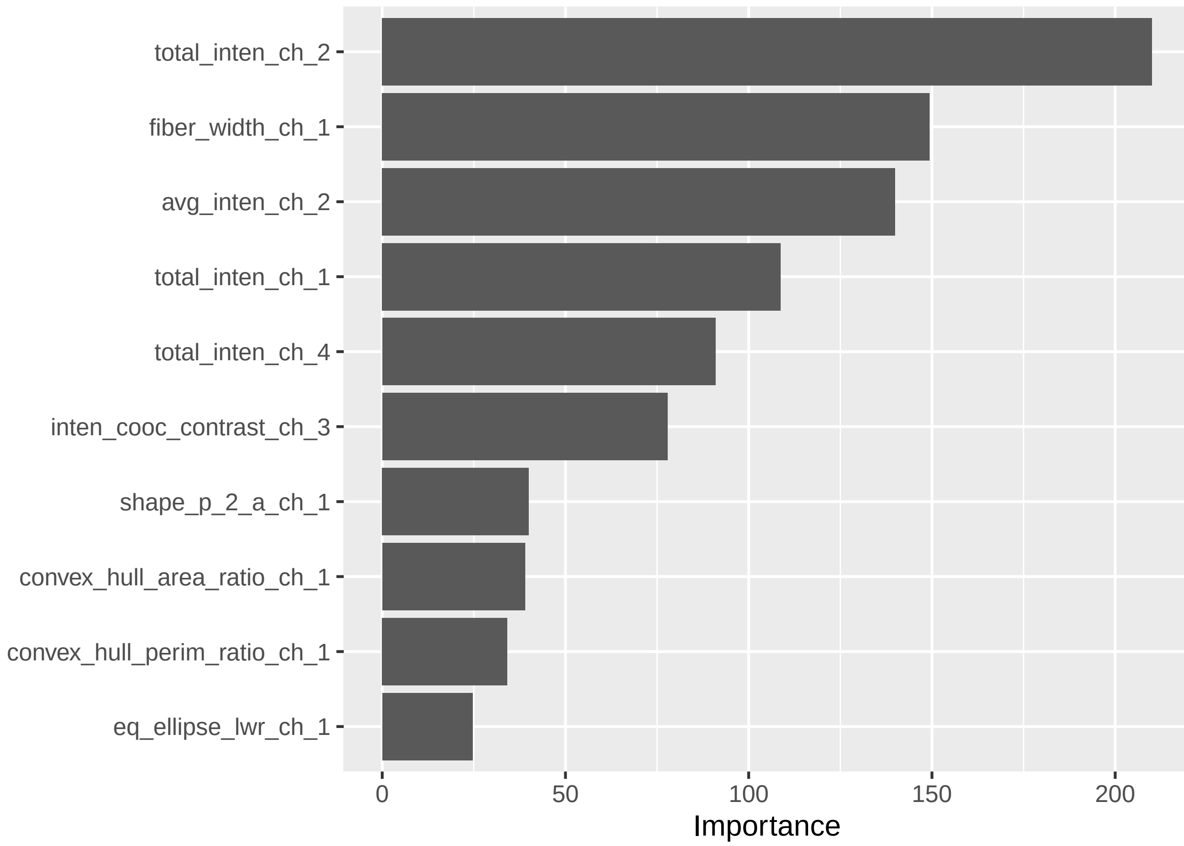
We can also look at variable imporance plots using the `vip` package.

```r
library(vip)
```

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

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

```r
final_tree %>% pull_workflow_fit() %>% vip()
```


total_inten_ch_2
fiber_width_ch_1
avg_inten_ch_2
total_inten_ch_1
total_inten_ch_4
inten_cooc_contrast_ch_3
shape_p_2_a_ch_1
convex_hull_area_ratio_ch_1
convex_hull_perim_ratio_ch_1
eq_ellipse_lwr_ch_1
0
50
100
150
200
Importance