Notes on Chapter 9: Judging Model Effectiveness

The Caveman Coder

2025-08-20

Regression metrics

Previous code from Chapter 8:

```
library(tidymodels)
```

```
## -- Attaching packages -------------------------------------- tidymodels 1.3.0 --
```

```
## v broom        1.0.8     v recipes      1.3.1
## v dials        1.4.1     v rsample      1.3.1
## v dplyr        1.1.4     v tibble       3.3.0
## v ggplot2      3.5.2     v tidyr        1.3.1
## v infer        1.0.9     v tune         1.3.0
## v modeldata    1.5.0     v workflows    1.2.0
## v parsnip      1.3.2     v workflowsets 1.1.1
## v purrr        1.1.0     v yardstick    1.3.2
```

```
## -- 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()
```

```
tidymodels_prefer()

data(ames)
ames <- mutate(ames, Sale_Price = log10(Sale_Price))

set.seed(502)
ames_split <- initial_split(ames, prop = 0.80, strata = Sale_Price)
ames_train <- training(ames_split)
ames_test <- testing(ames_split)

ames_rec <-
  recipe(
    Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type + Longitude + Latitude,
    data = ames_train
  ) |>
  step_log(Gr_Liv_Area, base = 10) |>
  step_other(Neighborhood, threshold = 0.01) |>
  step_dummy(all_nominal_predictors()) |>
  step_interact( ~ Gr_Liv_Area:starts_with("Bldg_Type_")) |>
  step_ns(Latitude, Longitude, deg_free = 20)

lm_model <- linear_reg() |> set_engine("lm")
```

```r
lm_wflow <-
  workflow() |>
  add_model(lm_model) |>
  add_recipe(ames_rec)

lm_fit <- fit(lm_wflow, ames_train)
```

Predicting results on the test set:

```r
ames_test_res <- predict(lm_fit, new_data = ames_test)
ames_test_res
```

```
## # A tibble: 588 x 1
##    .pred
##    <dbl>
##  1  5.07
##  2  5.31
##  3  5.28
##  4  5.33
##  5  5.30
##  6  5.24
##  7  5.67
##  8  5.52
##  9  5.34
## 10  5.00
## # i 578 more rows
```

Matching the predicted outcome with the corresponding observed outcome:

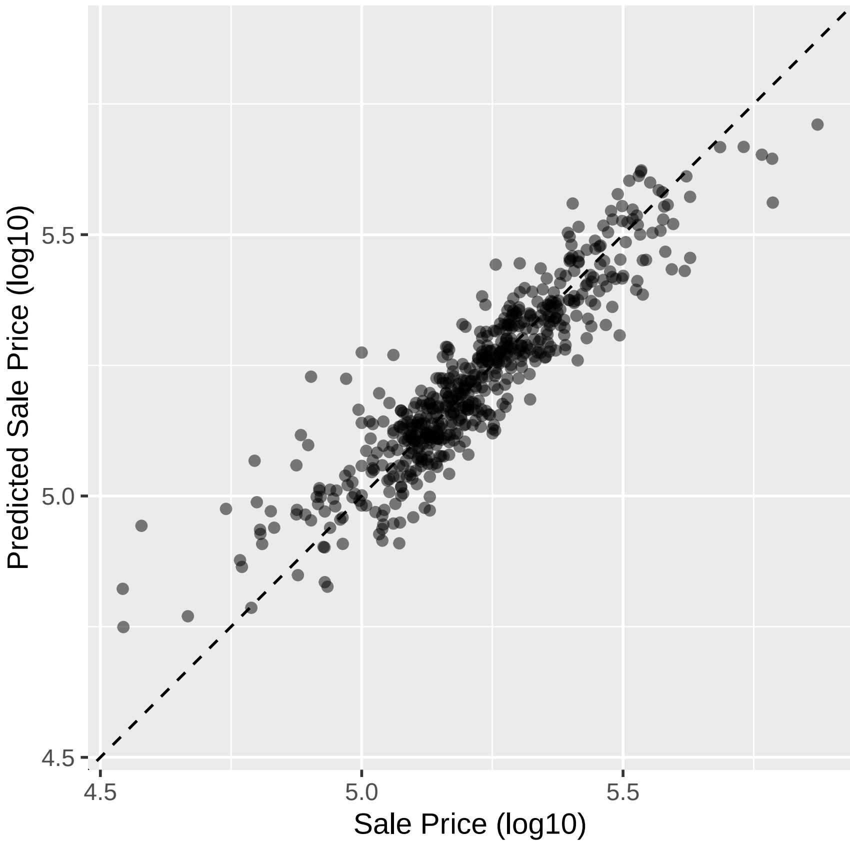
```r
ames_test_res <- bind_cols(ames_test_res, ames_test |> select(Sale_Price))

ames_test_res
```

```
## # A tibble: 588 x 2
##    .pred Sale_Price
##    <dbl>      <dbl>
##  1  5.07       5.02
##  2  5.31       5.39
##  3  5.28       5.28
##  4  5.33       5.28
##  5  5.30       5.28
##  6  5.24       5.26
##  7  5.67       5.73
##  8  5.52       5.60
##  9  5.34       5.32
## 10  5.00       4.98
## # i 578 more rows
```

Plotting the prediction vs observed outcome:

```r
ggplot(ames_test_res, aes(x = Sale_Price, y = .pred)) +
  # Create a diagonal line:
  geom_abline(lty = 2) +
  geom_point(alpha = 0.5) +
  labs(
    y = "Predicted Sale Price (log10)",
```

```r
    x = "Sale Price (log10)"
  ) +
  coord_obs_pred()
```



Computing for the root mean squared error using `rmse()`:

```r
rmse(ames_test_res, truth = Sale_Price, estimate = .pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard      0.0736
```

Computing for multiple metrics by creating a *metric set*:

```r
ames_metrics <- metric_set(rmse, rsq, mae)
ames_metrics(ames_test_res, truth = Sale_Price, estimate = .pred)
```

```
## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard      0.0736
## 2 rsq     standard      0.836
## 3 mae     standard      0.0549
```

The rmse and mae metrics are both on the scale of the outcome (`log10(Sale_Price)`). The rsq metric measures the squared correlation between the predicted and observed values, so the closer it is to 1, the better.

The **yardstick** package does not contain a function for *adjusted R^2* since this modified coefficient of determination commonly uses the same data for training and testing. It is always better to judge the performance of the models using a separate dataset than the one used to fit the model.

Binary classification metrics

Showing a model evaluation result from the **modeldata** package:

```
data("two_class_example")
tibble(two_class_example)
```

```
## # A tibble: 500 x 4
##    truth   Class1   Class2 predicted
##    <fct>    <dbl>    <dbl> <fct>
##  1 Class2 0.00359 0.996    Class2
##  2 Class1 0.679   0.321    Class1
##  3 Class2 0.111   0.889    Class2
##  4 Class1 0.735   0.265    Class1
##  5 Class2 0.0162  0.984    Class2
##  6 Class1 0.999   0.000725 Class1
##  7 Class1 0.999   0.000799 Class1
##  8 Class1 0.812   0.188    Class1
##  9 Class2 0.457   0.543    Class2
## 10 Class2 0.0976  0.902    Class2
## # i 490 more rows
```

In the example, there are two predicted class probabilities – `Class1` and `Class2`.

Creating a confusion matrix from the sample two-class prediction results:

```
conf_mat(two_class_example, truth = truth, estimate = predicted)
```

```
##           Truth
## Prediction Class1 Class2
##     Class1    227     50
##     Class2     31    192
```

Other **yardstick** "measures" of model effectiveness:

```
# Accuracy:
accuracy(two_class_example, truth, predicted)
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.838
```

```
# Matthews correlation coefficient:
mcc(two_class_example, truth, predicted)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 mcc     binary         0.677
```

```
# F1 metric:
f_meas(two_class_example, truth, predicted)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
```

```
##   <chr>   <chr>          <dbl>
## 1 f_meas  binary         0.849
```

Combining the three classification metrics in a metric set:

```
classification_metrics <- metric_set(accuracy, mcc, f_meas)
classification_metrics(two_class_example, truth = truth, estimate = predicted)
```

```
## # A tibble: 3 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.838
## 2 mcc      binary         0.677
## 3 f_meas   binary         0.849
```

The default behavior of these functions it to have the positive class, or the event of interest, as the first level of outcome, but this behavior can be changed, if desired, by specifying the `event_level`.

Example showing different settings for `event_level`:

```
# event_level = "first"
f_meas(two_class_example, truth, predicted)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 f_meas  binary         0.849
```

```
# event_level = "second"
f_meas(two_class_example, truth, predicted, event_level = "second")
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 f_meas  binary         0.826
```

Calculating the ROC and ROC-AUC for the sample result:

```
# ROC
two_class_curve <- roc_curve(two_class_example, truth, Class1)
two_class_curve
```

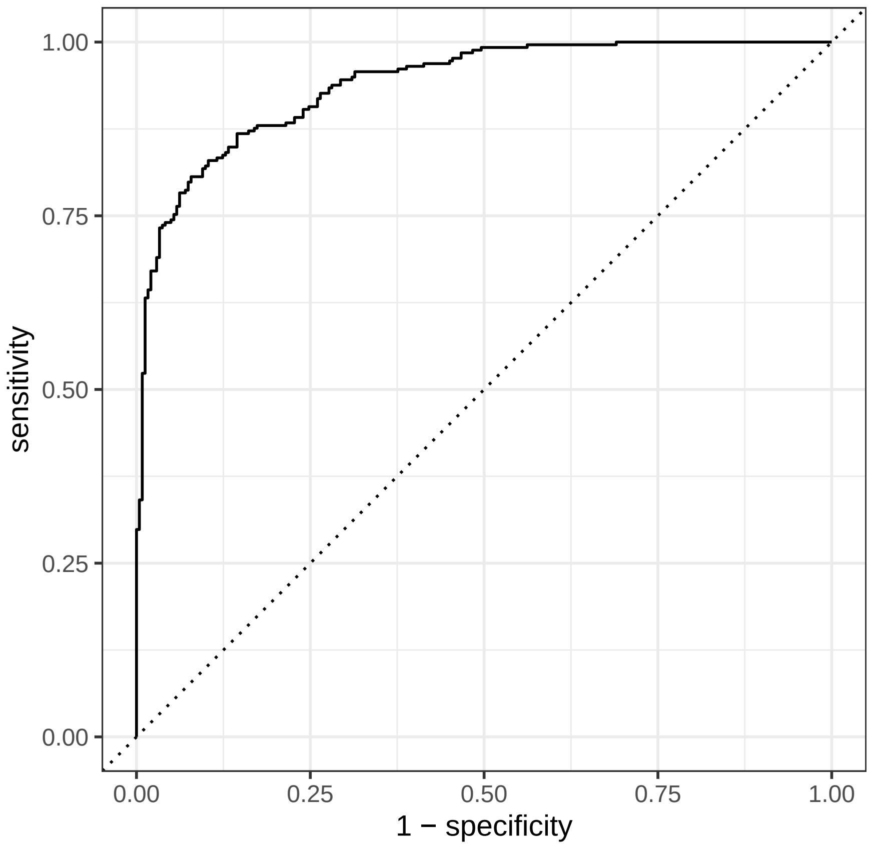
```
## # A tibble: 502 x 3
##    .threshold specificity sensitivity
##         <dbl>       <dbl>       <dbl>
##  1 -Inf           0                 1
##  2    1.79e-7     0                 1
##  3    4.50e-6     0.00413           1
##  4    5.81e-6     0.00826           1
##  5    5.92e-6     0.0124            1
##  6    1.22e-5     0.0165            1
##  7    1.40e-5     0.0207            1
##  8    1.43e-5     0.0248            1
##  9    2.38e-5     0.0289            1
## 10    3.30e-5     0.0331            1
## # i 492 more rows
```

```
# ROC-AUC
roc_auc(two_class_example, truth, Class1)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc binary         0.939
```

Plotting the ROC:

```
# ROC plot
autoplot(two_class_curve)
```



Since the plot is not close to the diagonal line, we can say that our model performs well at different thresholds.

Multiclass classification metrics

A prediction result with four classes:

```
data("hpc_cv")
tibble(hpc_cv)
```

```
## # A tibble: 3,467 x 7
##    obs   pred     VF      F       M          L Resample
##    <fct> <fct> <dbl>  <dbl>   <dbl>      <dbl> <chr>
##  1 VF    VF    0.914 0.0779 0.00848 0.0000199  Fold01
##  2 VF    VF    0.938 0.0571 0.00482 0.0000101  Fold01
##  3 VF    VF    0.947 0.0495 0.00316 0.00000500 Fold01
##  4 VF    VF    0.929 0.0653 0.00579 0.0000156  Fold01
##  5 VF    VF    0.942 0.0543 0.00381 0.00000729 Fold01
##  6 VF    VF    0.951 0.0462 0.00272 0.00000384 Fold01
##  7 VF    VF    0.914 0.0782 0.00767 0.0000354  Fold01
```

```
##  8 VF    VF    0.918 0.0744 0.00726 0.0000157  Fold01
##  9 VF    VF    0.843 0.128  0.0296  0.000192   Fold01
## 10 VF    VF    0.920 0.0728 0.00703 0.0000147  Fold01
## # i 3,457 more rows
```

Metrics for discrete class predictions are identical to their binary counterparts:

```
# Accuracy
accuracy(hpc_cv, obs, pred)
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.709
```

```
# MCC
mcc(hpc_cv, obs, pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 mcc     multiclass     0.515
```

Othe methods that can be used to apply sensitivity (i.e. the true positive rate):

- macro-averaging: computes the average of a set of one-versus-all metrics using the standard two-class statistics.
- macro-weighted averaging: does the same as macro-averaging, but the average is weighted by the number of samples in each class.
- micro-averaging: computes for the contribution for each class, then aggregates them, then computes a single metric from the aggregates.

Manually calculating these averaging methods:

```
class_totals <-
  count(hpc_cv, obs, name = "totals") |>
  mutate(class_wts = totals / sum(totals))

class_totals
```

```
##   obs totals  class_wts
## 1  VF   1769 0.51023940
## 2   F   1078 0.31093164
## 3   M    412 0.11883473
## 4   L    208 0.05999423
```

```
cell_counts <-
  hpc_cv |>
  group_by(obs, pred) |>
  count() |>
  ungroup()

cell_counts
```

```
## # A tibble: 16 x 3
##    obs   pred      n
##    <fct> <fct> <int>
##  1 VF    VF     1620
```

```
##   2 VF    F       141
##   3 VF    M         6
##   4 VF    L         2
##   5 F     VF      371
##   6 F     F       647
##   7 F     M        24
##   8 F     L        36
##   9 M     VF       64
##  10 M     F       219
##  11 M     M        79
##  12 M     L        50
##  13 L     VF        9
##  14 L     F        60
##  15 L     M        28
##  16 L     L       111
```

```
# Compute the four sensitivities using 1-vs-all:
one_versus_all <-
  cell_counts |>
  filter(obs == pred) |>
  full_join(class_totals, by = "obs") |>
  mutate(sens = n / totals)

one_versus_all
```

```
## # A tibble: 4 x 6
##   obs   pred      n totals class_wts  sens
##   <fct> <fct> <int>  <int>     <dbl> <dbl>
## 1 VF    VF     1620   1769    0.510  0.916
## 2 F     F       647   1078    0.311  0.600
## 3 M     M        79    412    0.119  0.192
## 4 L     L       111    208    0.0600 0.534
```

```
# Three different estimates:
one_versus_all |>
  summarize(
    macro = mean(sens),
    macro_wts = weighted.mean(sens, class_wts),
    micro = sum(n) / sum(totals)
  )
```

```
## # A tibble: 1 x 3
##   macro macro_wts micro
##   <dbl>     <dbl> <dbl>
## 1 0.560     0.709 0.709
```

Whew! Thankfully, the **yarstick** functions can automatically compute for these metrics using the `estimator` argument:

```
sensitivity(hpc_cv, obs, pred, estimator = "macro")
```

```
## # A tibble: 1 x 3
##   .metric     .estimator .estimate
##   <chr>       <chr>          <dbl>
## 1 sensitivity macro          0.560
```

```r
sensitivity(hpc_cv, obs, pred, estimator = "macro_weighted")
```

```
## # A tibble: 1 x 3
##   .metric     .estimator     .estimate
##   <chr>       <chr>              <dbl>
## 1 sensitivity macro_weighted     0.709
```

```r
sensitivity(hpc_cv, obs, pred, estimator = "micro")
```

```
## # A tibble: 1 x 3
##   .metric     .estimator .estimate
##   <chr>       <chr>          <dbl>
## 1 sensitivity micro          0.709
```

Calculating the roc-auc for the multiclass case:

```r
roc_auc(hpc_cv, obs, VF, F, M, L)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.829
```

Applying the macro-weighted average to the multiclass roc-auc outcome:

```r
roc_auc(hpc_cv, obs, VF, F, M, L, estimator = "macro_weighted")
```

```
## # A tibble: 1 x 3
##   .metric .estimator     .estimate
##   <chr>   <chr>              <dbl>
## 1 roc_auc macro_weighted     0.868
```

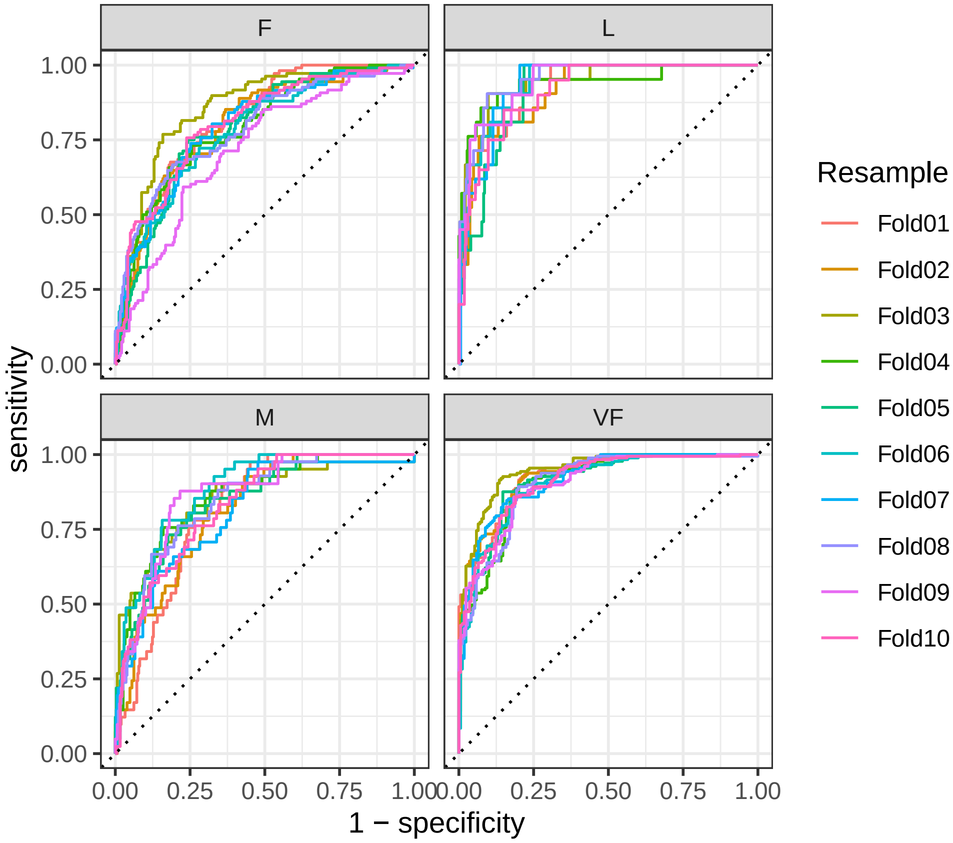
Calculating performance metrics on a per-fold basis (note: in the example, the result was generated using a k-fold cross-validation technique):

```r
hpc_cv |>
  group_by(Resample) |>
  accuracy(obs, pred)
```

```
## # A tibble: 10 x 4
##    Resample .metric  .estimator .estimate
##    <chr>    <chr>    <chr>          <dbl>
##  1 Fold01   accuracy multiclass     0.726
##  2 Fold02   accuracy multiclass     0.712
##  3 Fold03   accuracy multiclass     0.758
##  4 Fold04   accuracy multiclass     0.712
##  5 Fold05   accuracy multiclass     0.712
##  6 Fold06   accuracy multiclass     0.697
##  7 Fold07   accuracy multiclass     0.675
##  8 Fold08   accuracy multiclass     0.721
##  9 Fold09   accuracy multiclass     0.673
## 10 Fold10   accuracy multiclass     0.699
```

Plotting the ROC curves for each fold:

```r
hpc_cv |>
  group_by(Resample) |>
  roc_curve(obs, VF, F, M, L) |>
  autoplot()
```



Based on the results, the `VF` class is predicted better than the other classes.