# Applied Machine Learning - Backup Slides

Max Kuhn (RStudio)

# Outline

- Equivocal Zones
- Multiclass Model Statistics
- `tidypredict`

# tidymodels

```r
library(tidymodels)
```

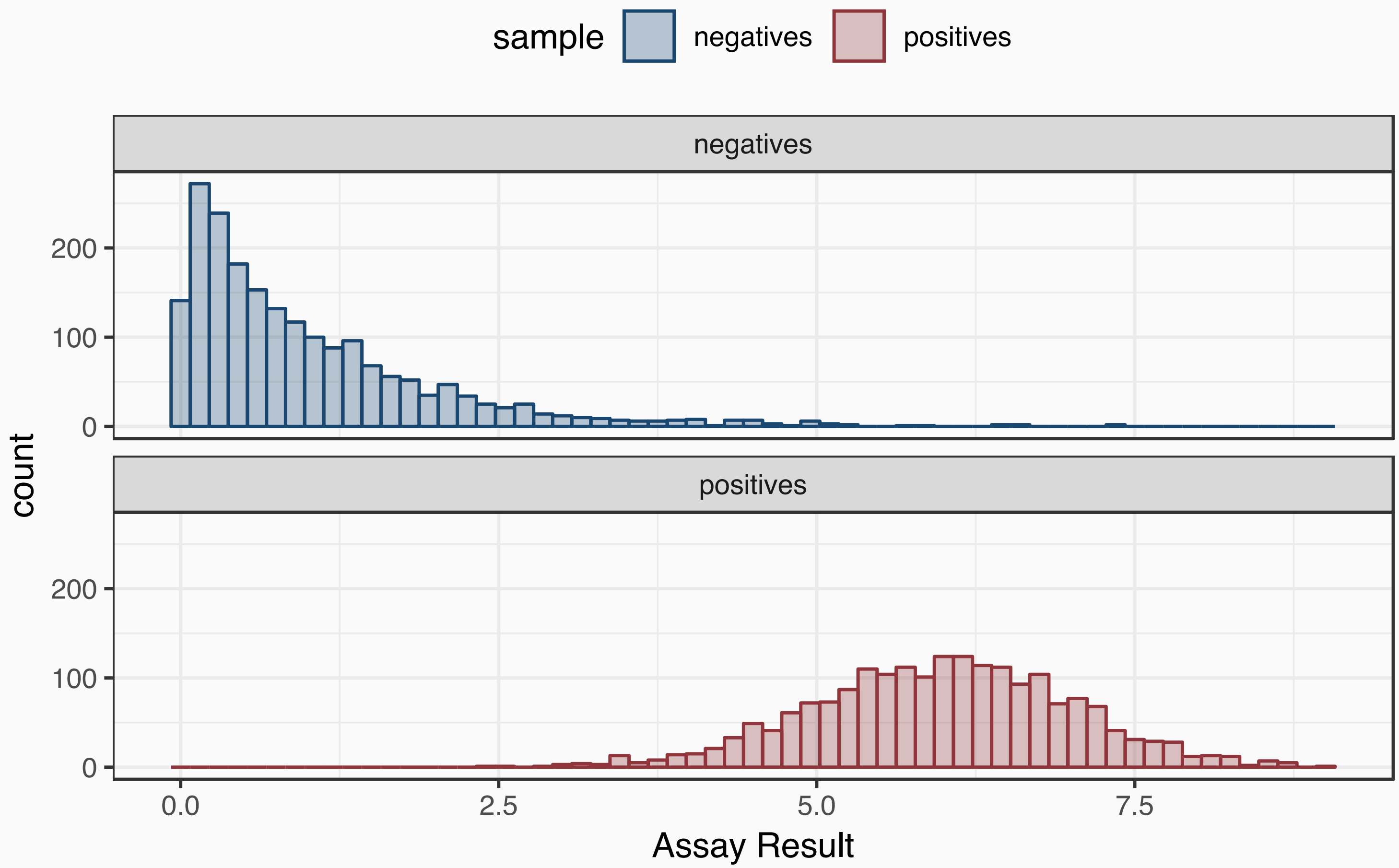
```
## ── Attaching packages ─────────────────────────────────── tidymodels 0.0.2 ──
```

```
## ✔ broom     0.5.1     ✔ purrr     0.2.5
## ✔ dials     0.0.2     ✔ recipes   0.1.4
## ✔ dplyr     0.7.8     ✔ rsample   0.0.4
## ✔ infer     0.4.0     ✔ tibble    2.0.0
## ✔ parsnip   0.0.1     ✔ yardstick 0.0.2
```

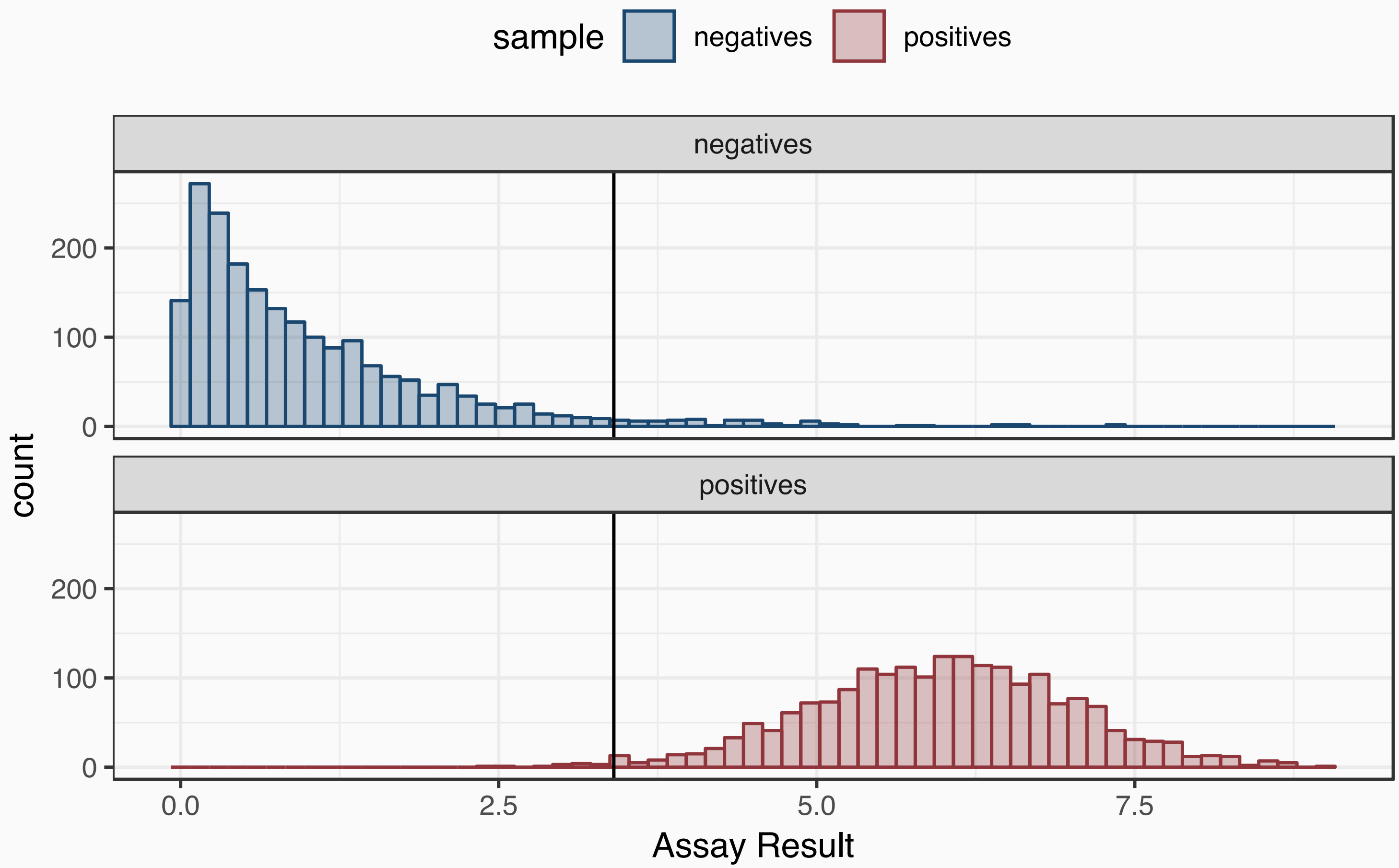
```
## ── Conflicts ──────────────────────────────────── tidymodels_conflicts() ──
## ✖ purrr::discard()    masks scales::discard()
## ✖ dplyr::filter()     masks stats::filter()
## ✖ dplyr::lag()        masks stats::lag()
## ✖ rsample::populate() masks Rcpp::populate()
## ✖ recipes::step()     masks stats::step()
```

# Applicability and Equivocals

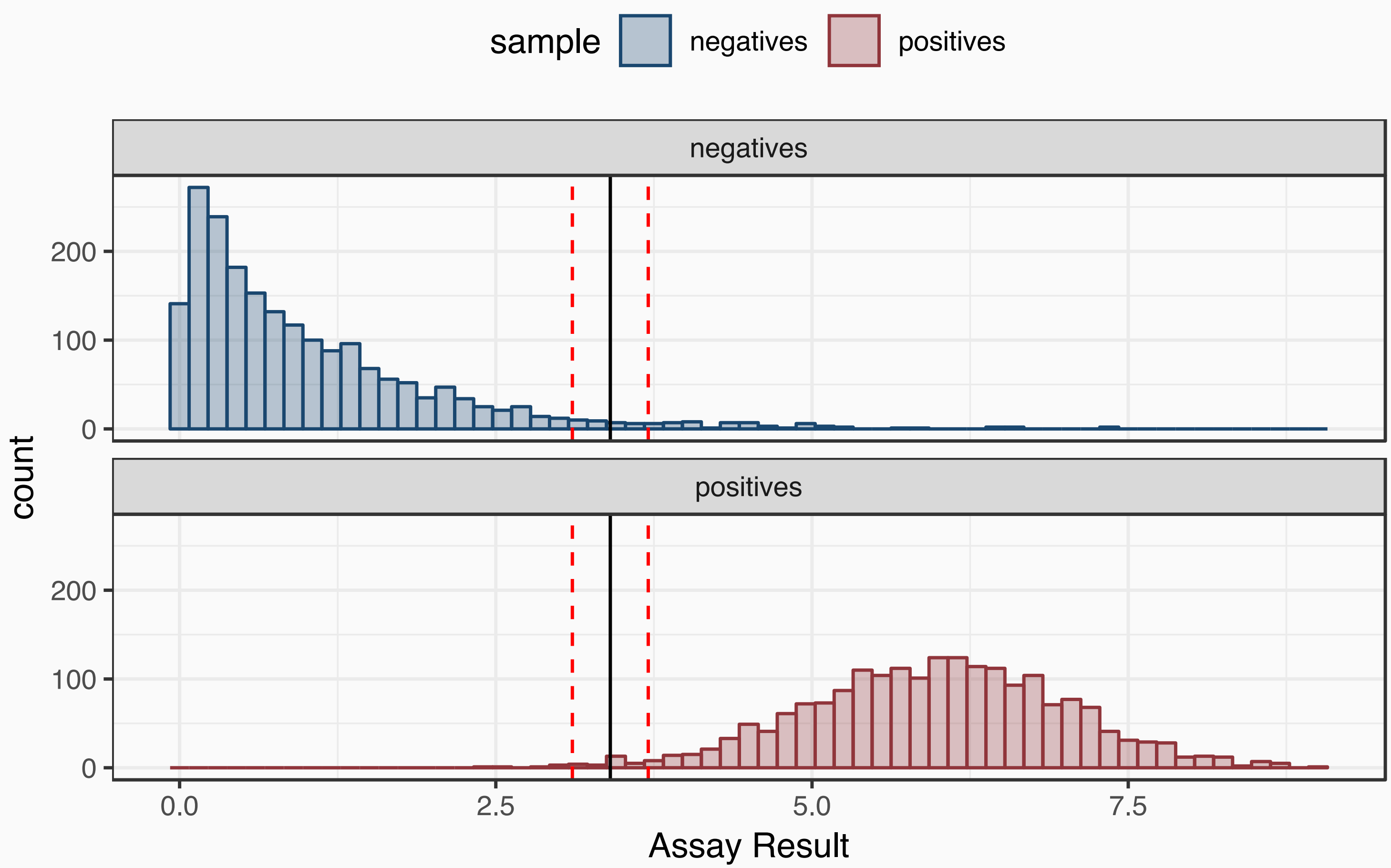
# Medical diagnostic analysis of assay results



# Add a cutoff via ROC curve



# Mandated buffer of equivocal results



# Equivocals and applicability domains

Are there times when we should *not* report a model result?

Just because we get a predicted value, it should not be assumed to be appropriate or *applicable*.

There is a modeling sub-field for determining the *applicability domain* of a model and deciding when to report a prediction.

As a real example, in drug discovery, computational chemistry models are built to predict various types of drug toxicity.

- Using assay results for existing compounds, predictions can be made on *proposed* compounds prior to their synthesis.
- Some models attempt to predict when prospective compounds are *toxic*.
- If a prediction were to be called **equivocal** or **unapplicable** (but interesting), the medicinal chemist and/or project biologist could then review the chemical structure and other properties in more detail (or send to a definitive screening assay).

# OkC Data

As a simple example, let's use the OkCupid data set with a reduced set of predictors.

A Bayesian logistic regression model with diffuse Gaussian priors ( $\beta_j \sim N(0, 10)$ ) was fit the data to make model predictions.

From this, we can get predictions of the probability of STEM as well as posterior *distribution* estimates.

A quasi standard error of fit was computed using the standard deviation of the posterior distribution.

- Recall that the standard error of the simple binomial rate $p$ is $\sqrt{p(1-p)/n}$.

10-fold cross-validation was used to compute out-of-sample predictions of each profile.

Other models, notably random forest, can compute uncertainty measures for prediction.

# Helper Functions

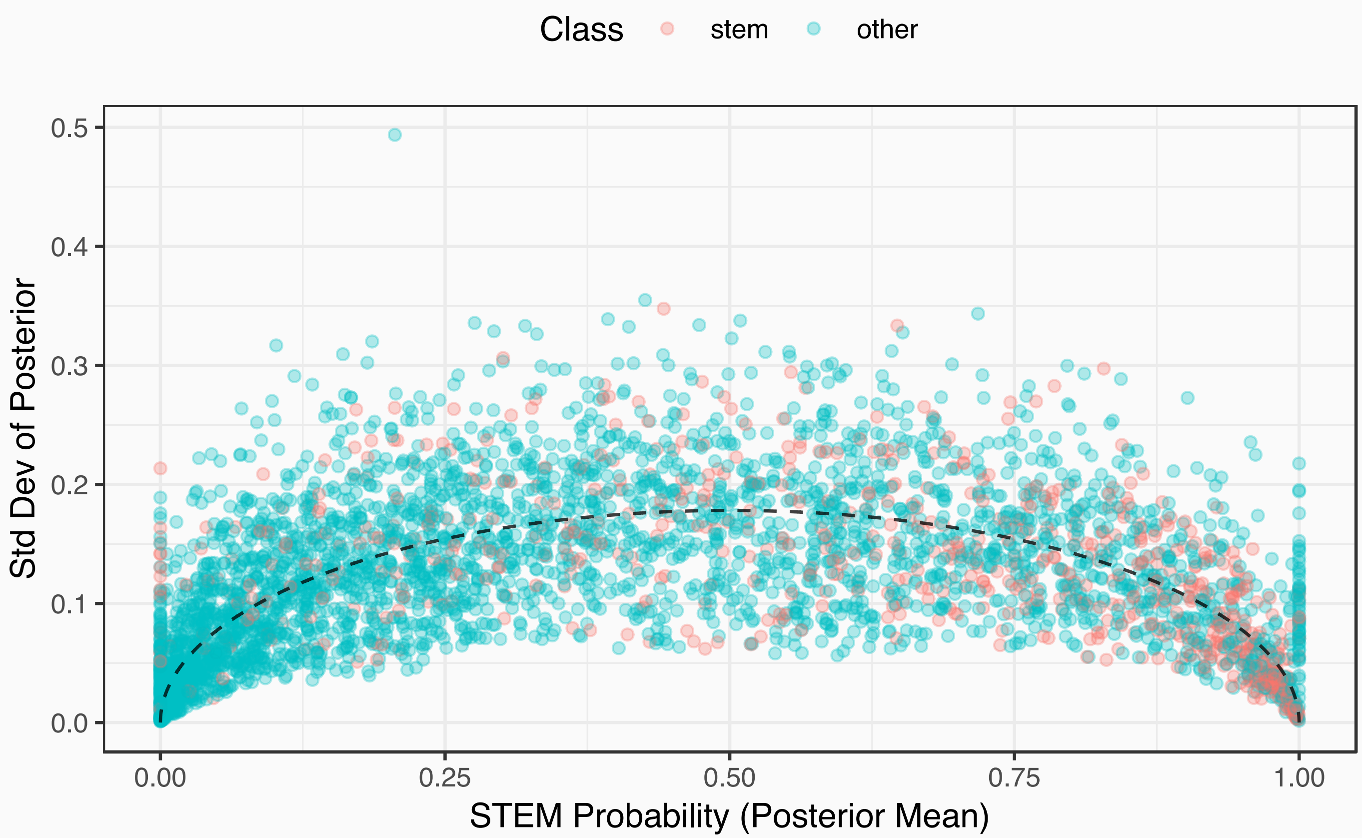
```r
# requires rstanarm package too.
make_fit <- function(recipe, ...) {
  logistic_reg() %>%
    set_engine("stan", chains = 4, cores = 1, QR = TRUE, init = 0, iter = 5000, seed = 25622) %>%
    fit(..., data = juice(recipe))
}

make_preds <- function(splits, recipes, models, ...) {
  # Get the dummy variables
  holdout <- bake(recipes, new_data = assessment(splits))
  holdout %>%
    bind_cols(predict(models, holdout %>% select(-Class), type = "class")) %>%
    bind_cols(predict(models, holdout %>% select(-Class), type = "prob")) %>%
    bind_cols(predict(models, holdout %>% select(-Class), type = "conf_int", std_error = TRUE)) %>%
    dplyr::select(Class, starts_with(".")) %>%
    # Get information about the resample and the original data index
    mutate(
      resample = labels(splits) %>% pull(id),
      .row = as.integer(splits, data = "assessment")
    ) %>%
    as_tibble()
}
```

# Modeling Code

```r
load("Data/okc.RData")

keywords <- names(okc_train)[32:91]
okc_lr_train <-
  okc_train %>%
  dplyr::select(Class, where_town, age, male, diet,
                religion, education, !!!keywords)

dummies <-
  recipe(Class ~ ., data = okc_lr_train) %>%
  step_dummy(all_nominal(), -Class) %>%
  step_downsample(Class) %>%
  step_zv(all_predictors())
```

```r
library(furrr)

# or plan("sequential")
plan("multicore") # non-windows implementation

set.seed(9798)
okc_splits <-
  vfold_cv(okc_lr_train) %>%
  mutate(
    recipes =  map(splits, prepper, dummies),
    # The next line takes a long time to execute.
    # It took 77 min using 10 cores for me.
    models = future_map(recipes, make_fit, Class ~ .)
  ) %>%
  mutate(
    preds = pmap(., make_preds)
  )
```

# Predicted Probability versus Uncertainty

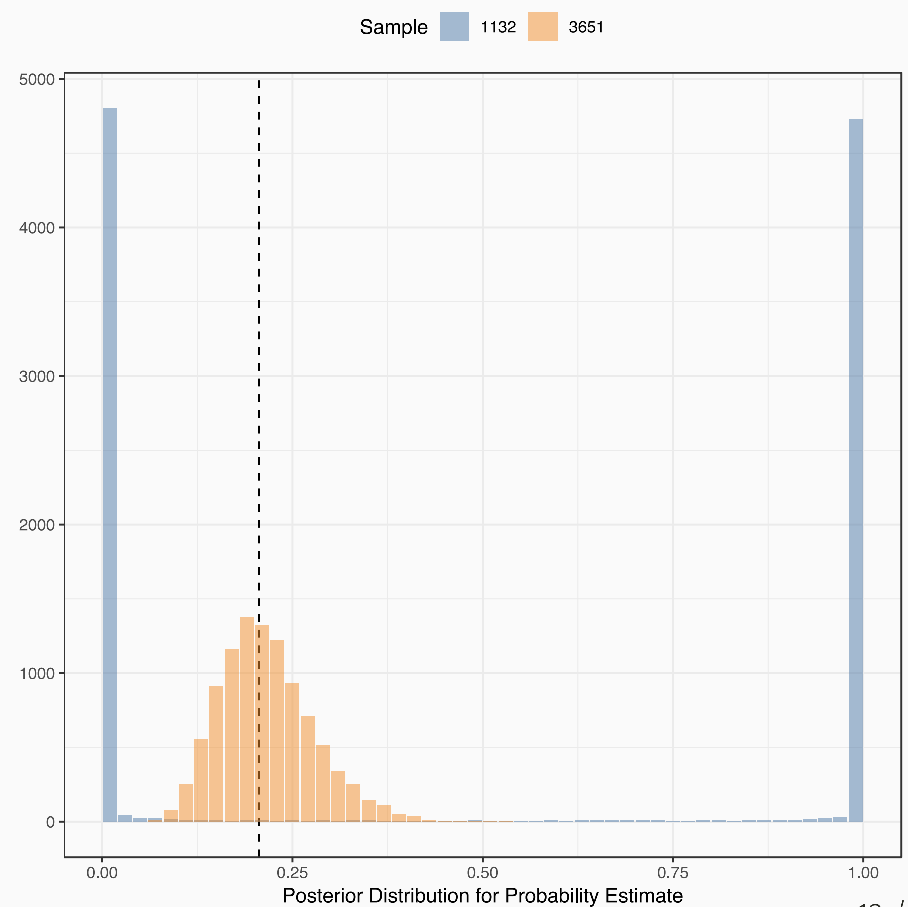


# Two samples with very similar *mean* probabilities

| Class | STEM Prob | Std Err | Fold | Sample # |
|---|---|---|---|---|
| stem | 0.210 | 0.061 | 0.421 | 3651 |
| other | 0.206 | 0.494 | 3.427 | 1132 |

Why might this be? Two of many reasons are:

- It could be due to different models in cross-validation (but it's not here).
- The sample might not be *applicable* if it is very different from the other samples in the training set (i.e. extrapolation).



# Adding a *post hoc* equivocal zone

We'll label samples as equivocal using rules:

- any sample whose standard error is 10-fold above the estimated average standard error **or**
- any sample with a predicted probability between 0.45 and 0.55.

To estimate the average standard error, we'll use the standard deviation of the binomial parameter.

```
av_std_error <-
  nls(
    .std_error ~ a *
      sqrt((.pred_stem) *
             (1 - .pred_stem)),
    data = okc_tr_res,
    start = list(a = .5)
  )
```

```
okc_tr_res <-
  okc_tr_res %>%
  mutate(
    exp_std_err = predict(av_std_error, .),
    fold_above = .std_error/exp_std_err
  )
okc_tr_res %>%
  dplyr::select(.pred_stem, .std_error,
                exp_std_err, fold_above) %>%
  slice(1:8)
```

```
## # A tibble: 8 x 4
##   .pred_stem .std_error   exp_std_err  fold_above
##        <dbl>      <dbl>         <dbl>       <dbl>
## 1   2.22e-16     0.0234 0.00000000531 4405600.
## 2   1.86e- 1     0.102  0.139               0.736
## 3   3.77e- 1     0.165  0.173               0.955
## 4   6.03e- 2     0.0942 0.0848              1.11
## 5   2.96e- 2     0.0644 0.0604              1.07
## 6   5.59e- 1     0.0971 0.177               0.549
## 7   1.54e- 1     0.143  0.129               1.11
## 8   9.67e- 1     0.0200 0.0638              0.314
```

# The `probably` Package

`probably` contains methods for post-processing class probability predictions, such as

- calibrating probability estimates (not yet implemented)
- determining appropriate thresholds for two-class data sets
- equivocal predictions

`probably` has a new type of object called `class_pred` that is like a factor but can include which samples should *not* be reported.

The object type builds on Hadley's new `vctrs` package.

# class_pred Objects

```r
library(probably)

okc_tr_res <-
  okc_tr_res %>%
  mutate(
    in_eq_zone =
      fold_above > 10 &
      (.pred_stem > 0.45 | .pred_stem < 0.55),
    new_pred  =
      class_pred(.pred_class, which = which(in_eq_zone))
  )
okc_tr_res %>%
  dplyr::select(.pred_class, new_pred) %>%
  slice(1:5)
```

```
## # A tibble: 5 x 2
##   .pred_class   new_pred
##   <fct>       <clss_prd>
## 1 other             [EQ]
## 2 other            other
## 3 other            other
## 4 other            other
## 5 other            other
```

```r
okc_tr_res %>% pull(new_pred) %>% class()
```

```
## [1] "class_pred" "vctrs_vctr"
```

```r
okc_tr_res %>% pull(new_pred) %>% levels()
```

```
## [1] "stem"  "other"
```

```r
okc_tr_res %>% slice(1:6) %>% pull(new_pred) %>% as.factor()
```

```
## [1] <NA>  other other other other stem
## Levels: stem other
```

# Performance Metrics with Equivocals

Equivocals are not included when performance is calculated (e.g. accuracy) and the *reportable rate* should also be inlcuded.

```
okc_tr_res %>% slice(1:5) %>% pull(new_pred)
```

```
## [1] [EQ]  other other other other
## Levels: stem other
## Reportable: 80.0%
```

```
okc_tr_res %>%
  summarise(reportable = reportable_rate(new_pred))
```

```
## # A tibble: 1 x 1
##   reportable
##        <dbl>
## 1      0.950
```

When converted to a factor, equivocal values are converted to missing.

- The next version of `yardstick` will automatically convert `class_pred` to factor before computing metrics.

```
okc_tr_res %>%
  mutate(new_pred = as.factor(new_pred)) %>%
  kap(Class, new_pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 kap     binary         0.270
```

# Multiclass Metrics

# Multiclass Metrics With yardstick

Multiclass? This just means your outcome has >2 possibilities (Religion: Catholic, Atheist, Buddhist, etc).

Consider binary `precision()`:

$$Pr = \frac{TP}{TP + FP}$$

```
## # A tibble: 5 x 2
##   truth estimate
##   <fct> <fct>
## 1
## 2
## 3
## 4
## 5
```

$$TP = 2$$

$$FP = 1$$

$$Pr = \frac{2}{2+1} = \frac{2}{3}$$

```
precision(prec_example, truth, estimate)
```

```
## # A tibble: 1 x 3
##   .metric   .estimator .estimate
##   <chr>     <chr>          <dbl>
## 1 precision binary         0.667
```

# Macro Averaging

What does this look like in multiclass world?

```
## # A tibble: 5 x 2
##   truth estimate
##   <fct> <fct>
## 1
## 2
## 3
## 4
## 5
```

One technique for dealing with this is *macro averaging*. This reduces the problem to multiple one-vs-all comparisons.

# Macro Averaging

What does this look like in multiclass world?

```
## # A tibble: 5 x 2
##   truth estimate
##   <fct> <fct>
## 1
## 2
## 3
## 4
## 5
```

One technique for dealing with this is *macro averaging*. This reduces the problem to multiple one-vs-all comparisons.

1) Convert `truth` / `estimate` to binary with levels: and `other`.

2) Compute `precision()` to get `Pr_1`.

3) Repeat 1) and 2) for each level to get `Pr_1`, `Pr_2`, `Pr_3`.

4) Average the results:

$$Pr_{macro} = \frac{Pr_1 + Pr_2 + Pr_3}{3}$$

# Macro Averaging

```
prec_multi
```

```
## # A tibble: 5 x 2
##   truth estimate
##   <fct> <fct>
## 1
## 2
## 3
## 4
## 5
```

$$Pr_1 = \frac{2}{2+0} = 1$$
$$Pr_2 = \frac{1}{1+1} = 0.5$$
$$Pr_3 = \frac{0}{0+1} = 0$$

$$Pr_{macro} = \frac{1+0.5+0}{3} = 0.5$$

```
precision(prec_multi, truth, estimate)
```

```
## # A tibble: 1 x 3
##   .metric   .estimator .estimate
##   <chr>     <chr>          <dbl>
## 1 precision macro            0.5
```

# Caveats

Macro averaging gives each class *equal weight* to the total precision value (`1/3` here). This may not be realistic when a class imbalance is present.

In that case, you can use a *weighted macro average* which weights by the frequency of that class in the `truth` column.

```
precision(prec_multi, truth, estimate, estimator = "macro_weighted")
```

```
## # A tibble: 1 x 3
##   .metric   .estimator     .estimate
##   <chr>     <chr>              <dbl>
## 1 precision macro_weighted       0.6
```

# Caveats

Macro averaging gives each class *equal weight* to the total precision value (`1/3` here). This may not be realistic when a class imbalance is present.

In that case, you can use a *weighted macro average* which weights by the frequency of that class in the `truth` column.

```
precision(prec_multi, truth, estimate, estimator = "macro_weighted")
```

```
## # A tibble: 1 x 3
##   .metric   .estimator     .estimate
##   <chr>     <chr>              <dbl>
## 1 precision macro_weighted       0.6
```

There is additionally a *micro average* that gives each *observation* equal weight rather than each *class*. This gives classes with more observations more influence.

Find more information at the yardstick vignette.

# tidypredict

# Converting Prediction Equations to SQL

`tidypredict` can convert some R model objects into SQL code that can be used for deployment.

The current set of models are: `lm()`, `glm()`, `randomForest()`, and `ranger()`.

There is work underway for `earth()`, `cubist()`, and tree-based models via `as.party()`.

There are currently some restrictions: no-line functions, non-treatment contrasts, and a few others.

Linear model prediction intervals can be computed though!

# An Example

```r
library(tidymodels)
library(AmesHousing)
library(tidypredict)

ames <- make_ames() %>%
  dplyr::select(-matches("Qu")) %>%
  # Manually log the variables :-(
  mutate(
    Sale_Price = log10(Sale_Price),
    Lot_Area = log10(Lot_Area),
    Gr_Liv_Area = log10(Gr_Liv_Area)
  )

set.seed(4595)
data_split <-
  initial_split(ames, strata = "Sale_Price")

ames_train <- training(data_split)
ames_test  <- testing(data_split)
```

```r
ames_mod <-
  lm(Sale_Price ~ Bldg_Type + Neighborhood +
       Year_Built +  Gr_Liv_Area +
       Full_Bath + Lot_Area +
       Central_Air*Year_Sold,
     data = ames_train)

acceptable_formula(ames_mod) # Silence is golden here

ames_sql <- tidypredict_fit(ames_mod)

# Check against `lm()`:

tidypredict_test(ames_mod, ames_test)
```

```
## tidypredict test results
## Difference threshold: 1e-12
##
##  All results are within the difference threshold
```

]

# Scoring

A taste of the model equation as an R expression:

```
## 18.5002644076146 + (ifelse(Bldg_Type == "TwoFmCon", 1, 0)) *
##     (-0.0164418404830365) + (ifelse(Bldg_Type == "Duplex", 1,
##     0)) * (-0.0790188658258443) + (ifelse(Bldg_Type == "Twnhs",
##     1, 0)) * (-0.0476581323558805) + (ifelse(Bldg_Type == "TwnhsE",
##     1, 0)) * (-0.00400678175266399) + (ifelse(Neighborhood ==
```

or SQL:

```
tidypredict_sql(ames_mod, dbplyr::simulate_mssql()) %>% substr(1, 85)
```

```
## [1] "18.5002644076146 + (CASE WHEN ((`Bldg_Type` = 'TwoFmCon') =  'TRUE') THEN (1.0) WHEN ..."
```

One cool thing about this is that these expressions do not require all of the predictors when the model includes feature selection (*a la* MARS or CART).