Applied Machine Learning - Classification

Max Kuhn (RStudio)

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Outline

- Performance Measures
- Amazon Review Data
- Classification Trees
- Boosting

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Load Packages

```r
library(tidymodels)
```

```
## Registered S3 method overwritten by 'xts':
##   method     from
##   as.zoo.xts zoo
```

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

```
## ✔ broom     0.5.1          ✔ purrr     0.3.3
## ✔ dials     0.0.3.9002     ✔ recipes   0.1.7.9001
## ✔ dplyr     0.8.3          ✔ rsample   0.0.5
## ✔ infer     0.4.0          ✔ tibble    2.1.3
## ✔ parsnip   0.0.4          ✔ yardstick 0.0.4
```

```
## ── Conflicts ────────────────────────────────────────────── tidymodels_conflicts() ──
## ✖ purrr::accumulate()    masks foreach::accumulate()
## ✖ purrr::discard()       masks scales::discard()
## ✖ dplyr::filter()        masks stats::filter()
## ✖ recipes::fixed()       masks stringr::fixed()
## ✖ dplyr::group_rows()    masks kableExtra::group_rows()
## ✖ dplyr::lag()           masks stats::lag()
## ✖ purrr::lift()          masks caret::lift()
## ✖ dials::margin()        masks ggplot2::margin()
## ✖ dials::offset()        masks stats::offset()
## ✖ yardstick::precision() masks caret::precision()
## ✖ dials::prune()         masks rpart::prune()
## ✖ dplyr::select()        masks MASS::select()
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Measuring Performance in Classification

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Illustrative Example

`yardstick` contains another test set example in a data frame called `two_class_example`:

```
two_class_example %>% head(4)
```

```
##    truth  Class1 Class2 predicted
## 1 Class2 0.00359  0.996    Class2
## 2 Class1 0.67862  0.321    Class1
## 3 Class2 0.11089  0.889    Class2
## 4 Class1 0.73516  0.265    Class1
```

Both `truth` and `predicted` are factors with the same levels. The other two columns represent *class probabilities*.

This reflects that most classification models can generate "hard" and "soft" predictions for models.

The class predictions are usually created by thresholding some numeric output of the model (e.g. a class probability) or by choosing the largest value.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Class Prediction Metrics

With class predictions, a common summary method is to produce a *confusion matrix* which is a simple cross-tabulation between the observed and predicted classes:

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

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

These can be visualized using mosaic plots.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Accuracy is the most obvious metric for characterizing the performance of models.

```
two_class_example %>%
    accuracy(truth = truth, estimate = predicted)
```

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

However, it suffers when there is a *class imbalance*; suppose 95% of the data have a specific class. 95% accuracy can be achieved by predicting samples to be the majority class.

Two Classes

There are a number of specialized metrics that can be used when there are two classes. Usually, one of these classes can be considered the *event of interest* or the *positive class*.

One common way to think about performance is to consider false negatives and false positives.

- The sensitivity is the *true positive rate* (out of all of the actual positives, how many did you get right?).
- The specificity is the rate of correctly predicted negatives, or 1 - *false positive rate* (out of all the actual negatives, how many did you get right?).

From this, assuming that `Class1` is the event of interest:

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

$\text{sensitivity} = 227/(227 + 31) = 0.88$

$\text{specificity} = 192/(192 + 50) = 0.79$

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Conditional and Unconditional Measures

Sensitivity and specificity can be computed from `sens()` and `spec()`, respectively.

It should be noted that these are *conditional measures* since we need to know the true outcome.

The event rate is the *prevalence* (or the Bayesian *prior*). Sensitivity and specificity are analogous to the *likelihood values*.

There are *unconditional* analogs to the *posterior values* called the positive predictive values and the negative predicted values.

A variety of other measures are available for two class systems, especially for *information retrieval*.

One thing to consider: what happens if our **threshold to call a sample an event is not optimal**?

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Changing the Probability Threshold

For two classes, the 50% cutoff is customary; if the probability of class #1 is >= 50%, they would be labelled as `Class1`.

What happens when you change the cutoff?

- Increasing it makes it harder to be called `Class1` ⇒ fewer predicted events, specificity ↑ , sensitivity ↓
- Decreasing the cutoff makes it easier to be called `Class1` ⇒ more predicted events, specificity ↓ , sensitivity ↑

With two classes, the **Receiver Operating Characteristic (ROC) curve** can be used to estimate performance using a combination of

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

To create the curve, many alternative cutoffs are evaluated.

For each cutoff, we calculate the sensitivity and specificity.

The ROC curve plots the sensitivity (eg. true positive rate) versus 1 - specificity (eg. the false positive rate).

The area under the ROC curve is a common metric of performance.

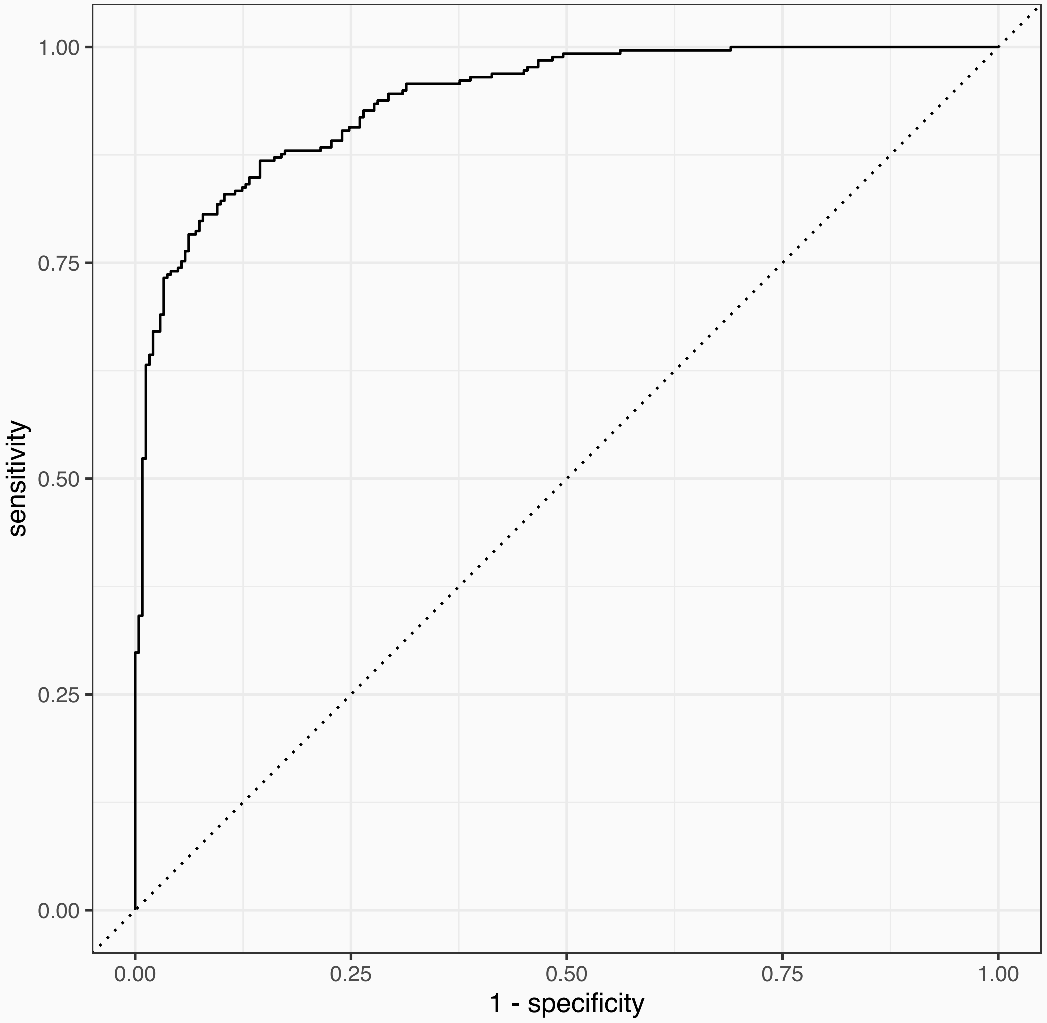
The Receiver Operating Characteristic (ROC) Curve

```
roc_obj <-
  two_class_example %>%
  roc_curve(truth, Class1)
```

```
two_class_example %>% roc_auc(truth, Class1)
```

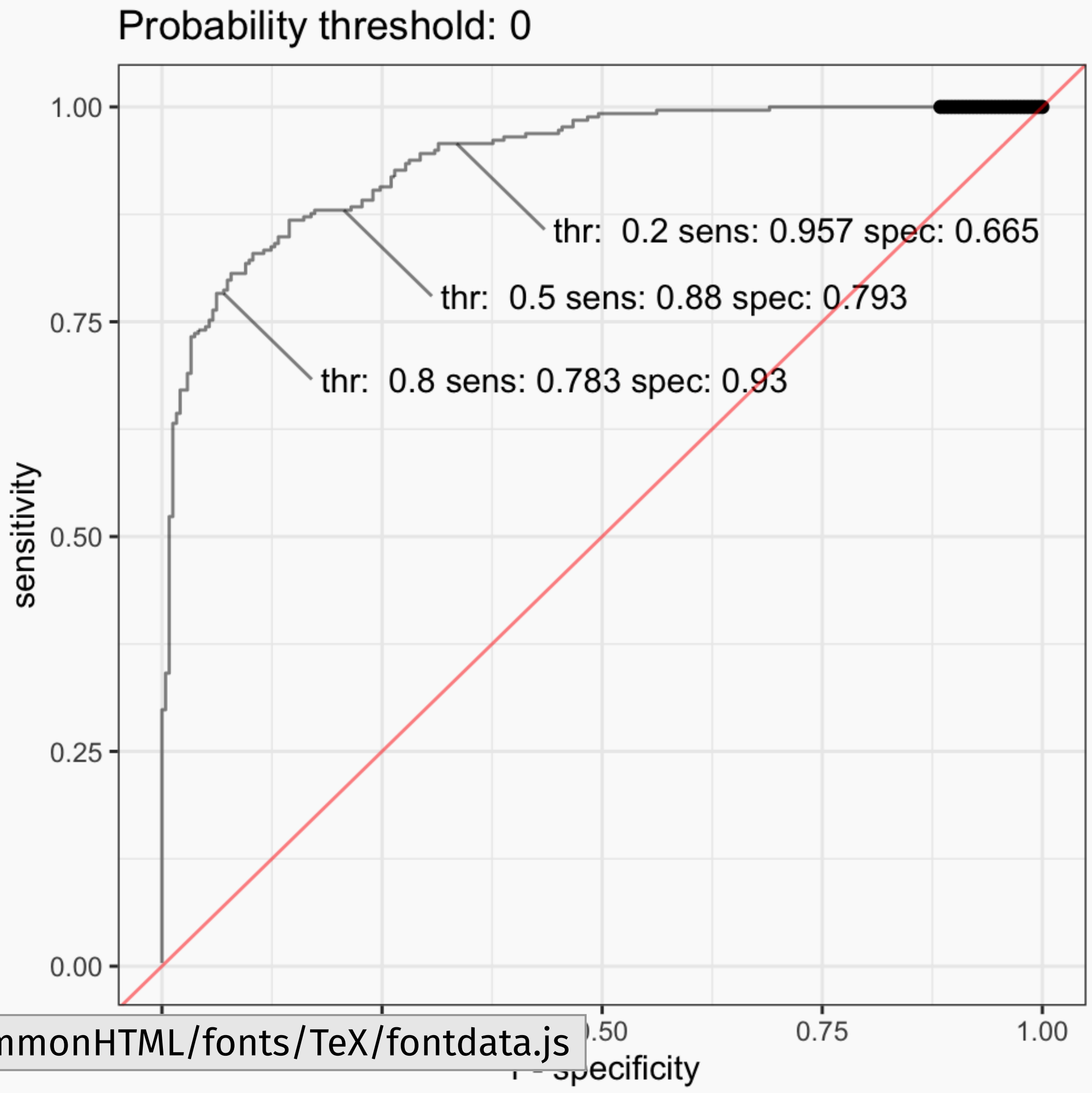
```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc binary         0.939
```

```
autoplot(roc_obj) + thm
```



Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Changing the Threshold



Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

The Receiver Operating Characteristic (ROC) Curve

The ROC curve has some major advantages:

- It can allow models to be optimized for performance before a definitive cutoff is determined.
- It is *robust* to class imbalances; no matter the event rate, it does a good job at characterizing model performance.
- The ROC curve can be used to pick an optimal cutoff based on the trade-offs between the types of errors that can occur.

When there are two classes, it is advisable to focus on the area under the ROC curve instead of sensitivity and specificity.

Once an acceptable model is determined, a proper cutoff can be determined.

Example Data

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Amazon Review Data

These data are from Amazon, who describe it as

> "This dataset consists of reviews of fine foods from amazon. The data span a period of more than 10 years, including all ~500,000 reviews up to October 2012. Reviews include product and user information, ratings, and a plaintext review."

We will use the text data to predict whether the review have a five-star result or not.

This will involve some natural language processing methods, which we will walk through.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Feature Engineering

For the types of models that we will be using, we'll need to convert data like this:

> "So far, used the rub on pork chops and it turned out great. Can't wait until summer for BBQ ribs."

into data like this:

#Words	#Characters	#Extra Spaces	Hash #1	Hash #47
20	78	1	absent	present

The *Tidy Text Mining in R* book has a lot of good information on the techniques that we will implemented below.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Defining roles

```r
text_rec <-
  recipe(score ~ product + review, data = training_data) %>%
  update_role(product, new_role = "id")
```

`product` is used for data splitting (as we'll see in a bit).

Arguably, it is not a predictor (although some might use it that way).

We update the role so that it is retained in the recipe but not used as a predictor.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Copying a column

```
text_rec <-
  recipe(score ~ product + review, data = training_data) %>%
  update_role(product, new_role = "id") %>%
  step_mutate(review_raw = review)
```

Two of the steps that we'll use will destroy the original predictor.

We'll use a basic `mutate` to make a temporary copy.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Initial feature set

```r
text_rec <-
  recipe(score ~ product + review, data = training_data) %>%
  update_role(product, new_role = "id") %>%
  step_mutate(review_raw = review) %>%
  step_textfeature(review_raw)  %>%
  step_rename_at(
    starts_with("textfeature_"),
    fn = ~ gsub("textfeature_review_raw_", "", .)
  )
```

A set of bumeric predictors are derived from the text.

Most are counts of text elements (e.g. words, punctuation, etc)

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Tokenize

```r
text_rec <-
  recipe(score ~ product + review, data = training_data) %>%
  update_role(product, new_role = "id") %>%
  step_mutate(review_raw = review) %>%
  step_textfeature(review_raw)  %>%
  step_rename_at(
    starts_with("textfeature_"),
    fn = ~ gsub("textfeature_review_raw_", "", .)
  )  %>%
  step_tokenize(review)
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Remove Stop Words

```r
text_rec <-
  recipe(score ~ product + review, data = training_data) %>%
  update_role(product, new_role = "id") %>%
  step_mutate(review_raw = review) %>%
  step_textfeature(review_raw)  %>%
  step_rename_at(
    starts_with("textfeature_"),
    fn = ~ gsub("textfeature_review_raw_", "", .)
  )  %>%
  step_tokenize(review)   %>%
  step_stopwords(review)
```

Stop words are those that occur commonly in text, such as "the", "a", and so on.

Removing them from text *might* be a good idea.

This largely depends on what you are doing with the text.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Word stemming

```
text_rec <-
  recipe(score ~ product + review, data = training_data) %>%
  update_role(product, new_role = "id") %>%
  step_mutate(review_raw = review) %>%
  step_textfeature(review_raw) %>%
  step_rename_at(
    starts_with("textfeature_"),
    fn = ~ gsub("textfeature_review_raw_", "", .)
  )  %>%
  step_tokenize(review) %>%
  step_stopwords(review)  %>%
  step_stem(review)
```

Stemming is a method that uses a common root of a word instead of the original value.

For example, these 7 words are fairly similar: "teach", "teacher", "teachers", "teaches", "teachable", "teaching", "teachings".

Stemming would reduce these to 3 unique values: "teach", "teacher", "teachabl".

Like stop word removal, this may or may not be a good idea.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Feature hashing

```
text_rec <-
  recipe(score ~ product + review, data = training_data) %>%
  update_role(product, new_role = "id") %>%
  step_mutate(review_raw = review) %>%
  step_textfeature(review_raw) %>%
  step_rename_at(
    starts_with("textfeature_"),
    fn = ~ gsub("textfeature_review_raw_", "", .)
  )  %>%
  step_tokenize(review) %>%
  step_stopwords(review)  %>%
  step_stem(review) %>%
  step_texthash(review, signed = FALSE, num_terms = 1024) %>%
  step_rename_at(
    starts_with("review_hash"),
    fn = ~ gsub("review_", "", .)
  )
```

Feature hashing creates numeric terms from words in a sentence (or some other token) similar to dummy variables.

However, there are big differences, including:

- There is no look-up table to consult to make the mapping
- The placement of the non-zero values in meant to emulate randomness.
- The new features are computed on the actual words.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Feature hashing

For string "On Time and product looked like it", a sketch of the calculations to make 8 hashed values:

word	hashed integer value	(integer mod 8) + 1
On	-182693672	4
Time	1593484409	8
and	-1079337235	8
product	-979280496	6
looked	-2120797534	2
like	-592737581	5
it	1278008556	2

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Note that multiple words end up going into the same feature column. This is *aliasing* (statistical term) or a *collision* (comp sci term).

- We wouldn't be able to distinguish the effect of those two words.

We can encode this as a simple zero or, as `textrecipes` does, use the count as the value.

- There are also *signed* hashes that help avoid collisions.

Note that no words were mapped to feature columns three or seven.

Optional step: convert binary to factors

```r
text_rec <-
  recipe(score ~ product + review, data = training_data) %>%
  update_role(product, new_role = "id") %>%
  step_mutate(review_raw = review) %>%
  step_textfeature(review_raw) %>%
  step_rename_at(
    starts_with("textfeature_"),
    fn = ~ gsub("textfeature_review_raw_", "", .)
  )  %>%
  step_tokenize(review) %>%
  step_stopwords(review)  %>%
  step_stem(review) %>%
  step_texthash(review, signed = FALSE, num_terms = 1024) %>%
  step_rename_at(
    starts_with("review_hash"),
    fn = ~ gsub("review_", "", .)
  ) %>%
  step_num2factor(
    starts_with("hash"),
    transform = count_to_binary
  )

# with
count_to_binary <- function(x) {
}
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

The naive Bayes model will be used on these data.

It computes probability values from each predictor.

- If the predictor is numeric, its statistical density is used.
- If categorical, a contingency table is used.

Since the hash values are really about the presence/absence of words, we should convert then to 2-level factor variables to ensure appropriate calculations.

Optional step: remove zero-variance predictors

```r
text_rec <-
  recipe(score ~ product + review, data = training_data) %>%
  update_role(product, new_role = "id") %>%
  step_mutate(review_raw = review) %>%
  step_textfeature(review_raw) %>%
  step_rename_at(
    starts_with("textfeature_"),
    fn = ~ gsub("textfeature_review_raw_", "", .)
  )  %>%
  step_tokenize(review) %>%
  step_stopwords(review)  %>%
  step_stem(review) %>%
  step_texthash(review, signed = FALSE, num_terms = 1024) %>%
  step_rename_at(
    starts_with("review_hash"),
    fn = ~ gsub("review_", "", .)
  ) %>%
  step_num2factor(
    starts_with("hash"),
    transform = count_to_binary
  ) %>%
  step_zv(all_predictors())
```

Removing has features that are all zero in the training set increases computational efficiency and may stop model failures.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Resampling and Analysis Strategy

There are 4000 rows in the training set. If we did 10-fold cross-validation, we would compute the area under the ROC curve using 400 sub-samples.

If these, 400, we would expect about 86.667 reviews to be 5-stars and 313.333 to have less than 5 stars.

That is probably enough data to get reasonable performance estimates.

```r
set.seed(8935)
folds <- vfold_cv(training_data, strata = "score")
```

Classification Trees

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Tree model structure

A classification tree searches through each predictor to find a value of a single variable that best splits the data into two groups.

For the two resulting groups, the process is repeated until a hierarchical structure (a tree) is created.

- In effect, trees partition the X space into rectangular sections that assign a single value to samples within the rectangle.

The final structure in the tree is the *terminal node* and each path through the tree is a *rule*.

```
# Example tree with three terminal nodes
if (x > 1) {
     if (y < 3) {
       class <- "A"
     } else {
       class <- "B"
     }
} else {
  class <- "A"
```

```
# Same tree, stated as rules
if (x >  1 & y <   3) class <- "A"
if (x >  1 & y >=  3) class <- "B"
if (x <= 1)           class <- "A"
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Species of tree-based models

There are a variety of different methods for creating trees that vary over:

- The search method (e.g., greedy or global).
- The splitting criterion.
- The number of splits.
- Handling of missing values.
- Pruning method.

The most popular is the CART methodology, followed by the C5.0 model.

We will focus on CART for single trees.

The CRAN Machine Learning Task View has a good summary of the methods available in R.

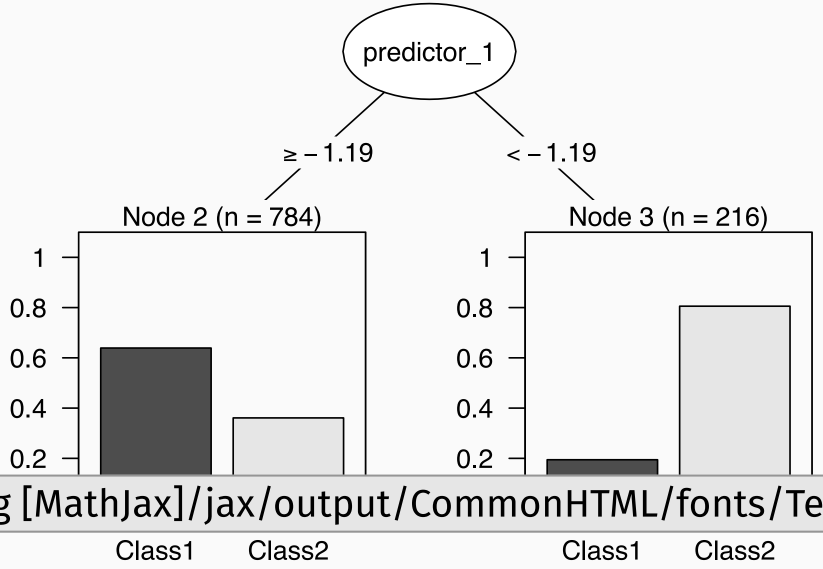
Growing phase

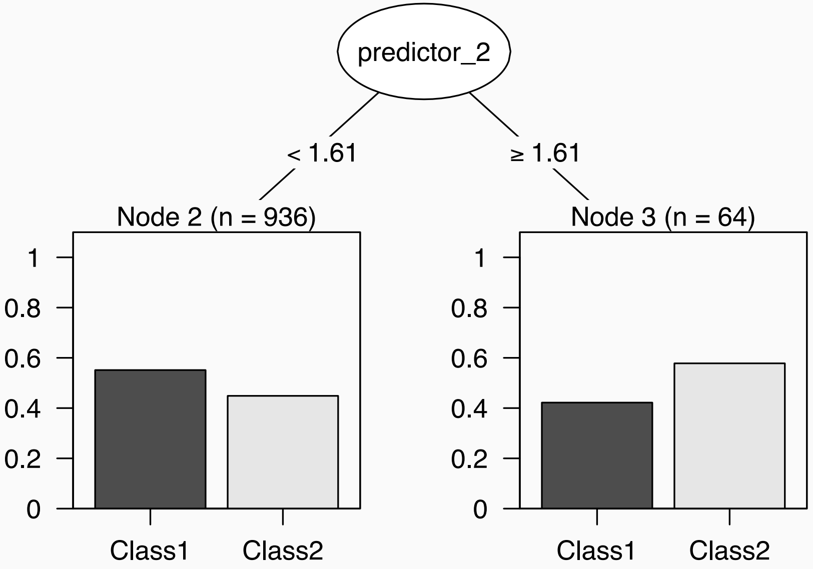
CART starts by *growing* the tree.

- More and more splits are conducted until a pre-specified samples size requirement is exceeded (`min_n`).

The criterion used is the *purity* of the terminal nodes that are created by each split.

For example, for simulated data with a 50% event rate, which one of these splits is better?





Pruning phase

The deepest possible tree has a higher likelihood of overfitting the data.

CART conducts cost-complexity pruning to find the "right sized tree".

It basically penalizes the error rate of the tree by the number of terminal nodes by minimizing

$$Error_{cv} - (c_p \times nodes)$$

Notes:

- The c_p value, usually between 0 and 0.1, controls the depth of the tree.
- CART has an internal 10-fold cross-validation that it uses to estimate the model error.
- If the outcome has a large class imbalance, this method optimizes the tree for the majority class.

For CART, c_p (aka `cost_complexity`) and the minimum splitting size (`min_n`) are the tuning parameters.

Aspects of single trees

- The class percentages in the terminal node are used to make predictions.
- The number of possible class probabilities is typically low.
- Trees are *theoretically* interpretable if the number of terminal nodes is low.
- The training time tends to be very fast.
- Trees are *unstable*; if the data are slightly changed, the entire tree structure can change. These are low-bias/high-variance models.
- Very little, if any, data pre-processing is needed. *Dummy variables are not required*.
- Trees automatically conduct *feature selection*.

Fitting and tuning trees

Like MARS, there are two main ways to tune the CART model:

- Rely on the internal CV procedure to pick the tree depth via purity/error rate:

```
decsion_tree(min_n = tune()) %>% set_engine("rpart") %>% set_mode("classification")
```

- Manually specify c_p values and use external resampling with a metric of your choice:

```
decsion_tree(cost_complexity = tune(), min_n = tune()) %>% set_engine("rpart") %>% set_mode("classification")
```

I prefer the latter approach; I believe that the automated choice tends to pick overly simple models.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

{recipe} and {parsnip} objects

```r
library(textfeatures)
library(textrecipes)

basics <- names(textfeatures:::count_functions)

tree_rec <-
  recipe(score ~ product + review, data = training_data) %>%
  update_role(product, new_role = "id") %>%
  step_mutate(review_raw = review) %>%
  step_textfeature(review_raw) %>%
  step_rename_at(
    starts_with("textfeature_"),
    fn = ~ gsub("textfeature_review_raw_", "", .)
  ) %>%
  step_tokenize(review)  %>%
  step_stopwords(review) %>%
  step_stem(review) %>%
  step_texthash(review, signed = FALSE, num_terms = tune()) %>%
  step_rename_at(starts_with("review_hash"), fn = ~ gsub("review_", "", .))

# and

cart_mod <-
  decision_tree(cost_complexity = tune(), min_n = tune()) %>%
  set_engine("rpart") %>%
  set_mode("classification")

ctrl <- control_grid(save_pred = TRUE)
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Note that:

All of the text processing operations are deterministic and not reliant on any other data in the training set.

We could pre-compute the data prior to the text hashing.

Also, if we were not tuning the number of hashing terms, we could pre-compute the whole feature set with the exception of the zero-variance filter.

Model tuning

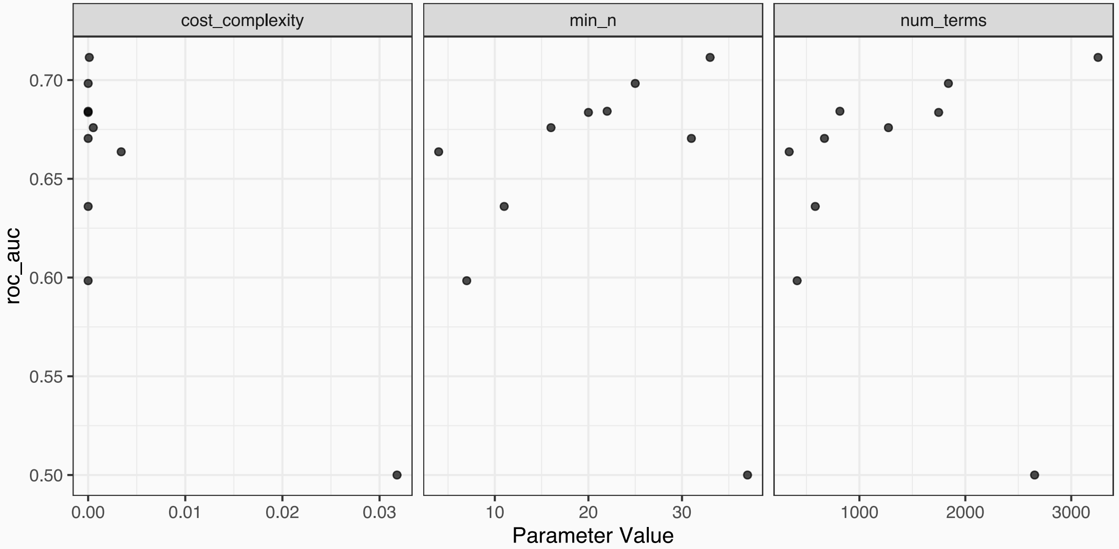
```
set.seed(2553)
cart_tune <-
  tune_grid(
    tree_rec,
    cart_mod,
    folds,
    grid = 10,
    metric = metric_set(roc_auc),
    control = ctrl
  )
show_best(cart_tune)
```

```
## # A tibble: 5 x 8
##   cost_complexity min_n num_terms .metric .estimator  mean     n std_err
##             <dbl> <int>     <int> <chr>   <chr>      <dbl> <int>   <dbl>
## 1  0.000118          33      3253 roc_auc binary     0.711    10 0.00852
## 2  0.00000393        25      1840 roc_auc binary     0.698    10 0.00520
## 3  0.00000000286     22       816 roc_auc binary     0.684    10 0.00601
## 4  0.000000000303    20      1748 roc_auc binary     0.684    10 0.00486
## 5  0.000526          16      1274 roc_auc binary     0.676    10 0.0103
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Parameter profiles

```
autoplot(cart_tune)
```



Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

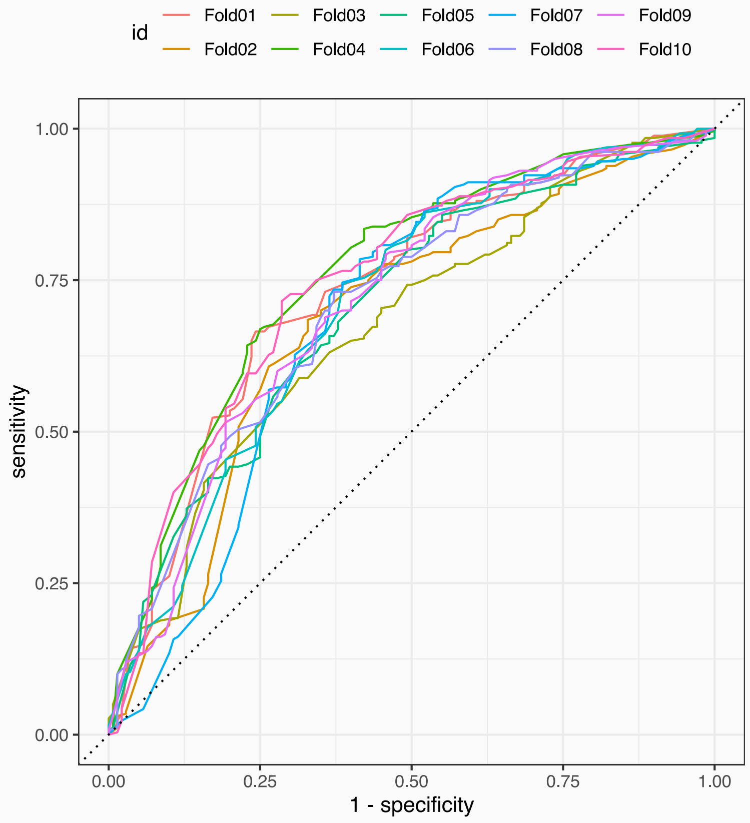
Plotting ROC curves

```
cart_pred <- collect_predictions(cart_tune)
cart_pred %>% slice(1:5)
```

```
## # A tibble: 5 x 8
##   id     .pred_great .pred_other  .row num_terms cost_complexity
##   <chr>        <dbl>       <dbl> <int>     <int>           <dbl>
## 1 Fold01       0           1        24       412     0.000000271
## 2 Fold01       0.939       0.0606   25       412     0.000000271
## 3 Fold01       0.734       0.266    26       412     0.000000271
## 4 Fold01       0.961       0.0390   46       412     0.000000271
## 5 Fold01       0.939       0.0606   48       412     0.000000271
```

```
cart_pred %>%
  inner_join(select_best(cart_tune)) %>%
  group_by(id) %>%
  roc_curve(score, .pred_great) %>%
  autoplot()
```

```
## Joining, by = c("num_terms", "cost_complexity", "min_n")
```



Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

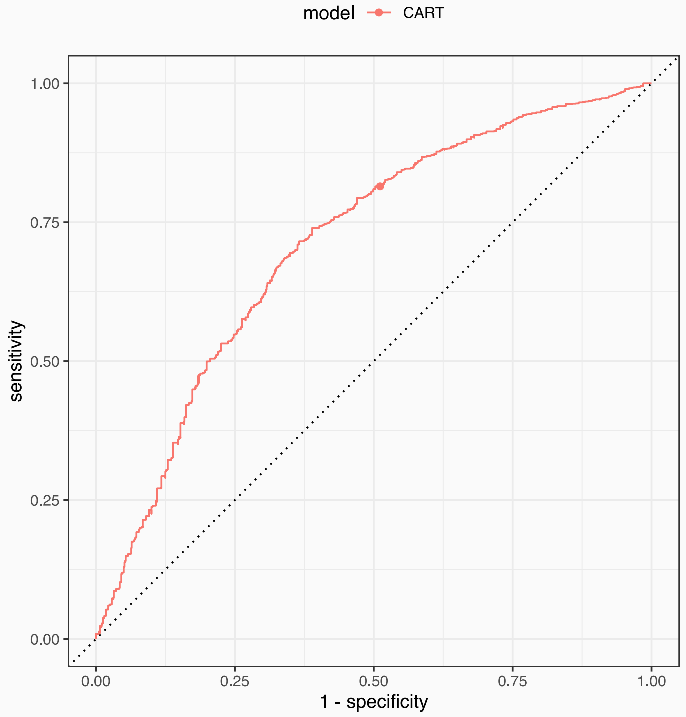
A single (but approximate) ROC curve

Instead of showing all 10 curves, we could pool the 10 fold's worth of data and make a single curve that might approximate the "average" curve.

```r
auc_curve_data <- function(x) {
    collect_predictions(x) %>%
    inner_join(select_best(x, "roc_auc")) %>%
    roc_curve(score, .pred_great)
}
approx_roc_curves <- function(...) {
  curves <- map_dfr(list(...), auc_curve_data, .id = "model")
  default_cut <-
    curves %>%
    group_by(model) %>%
    arrange(abs(.threshold - .5)) %>%
    slice(1)
  ggplot(curves) +
    aes(y = sensitivity, x = 1 - specificity, col = model) +
    geom_abline(lty = 3) +
    geom_step(direction = "vh") +
    geom_point(data = default_cut) +
    coord_equal()
}
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```r
# Use named arguments for better labels
approx_roc_curves(CART = cart_tune)
```



Hands-On: Down-Sampling

Looking at the ROC curve, the default cutoff may not be optimal if FP and FN errors are about equal.

We could pick a better cutoff or fit another model using *sub-class sampling*.

The latter approach would balance the data prior to model fitting.

- The most common method would be to *down-sample* the data.
- This is fairly controversial (at least in statistical circles).

Let's take 20m and refit the model code above with a recipe that includes downsampling.

link to recipes documentation

20:00

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Variable importance

Also like MARS, these models judge importance by how much the terminal node purity improved with each split.

These are aggregated over variables.

Note: by default, these measures will contain predictors *not used in the tree* (due to surrogate splits).

You can change this using the `rpart.control()` option.

Unfortunately, for these data, the hashed variables are rather opaque. It takes a lot of work to determine which words map to which features.

Boosting

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Original concept of boosting

The original boosting algorithm was created for two-class problems and was designed to *boost* a weak learner into a strong one.

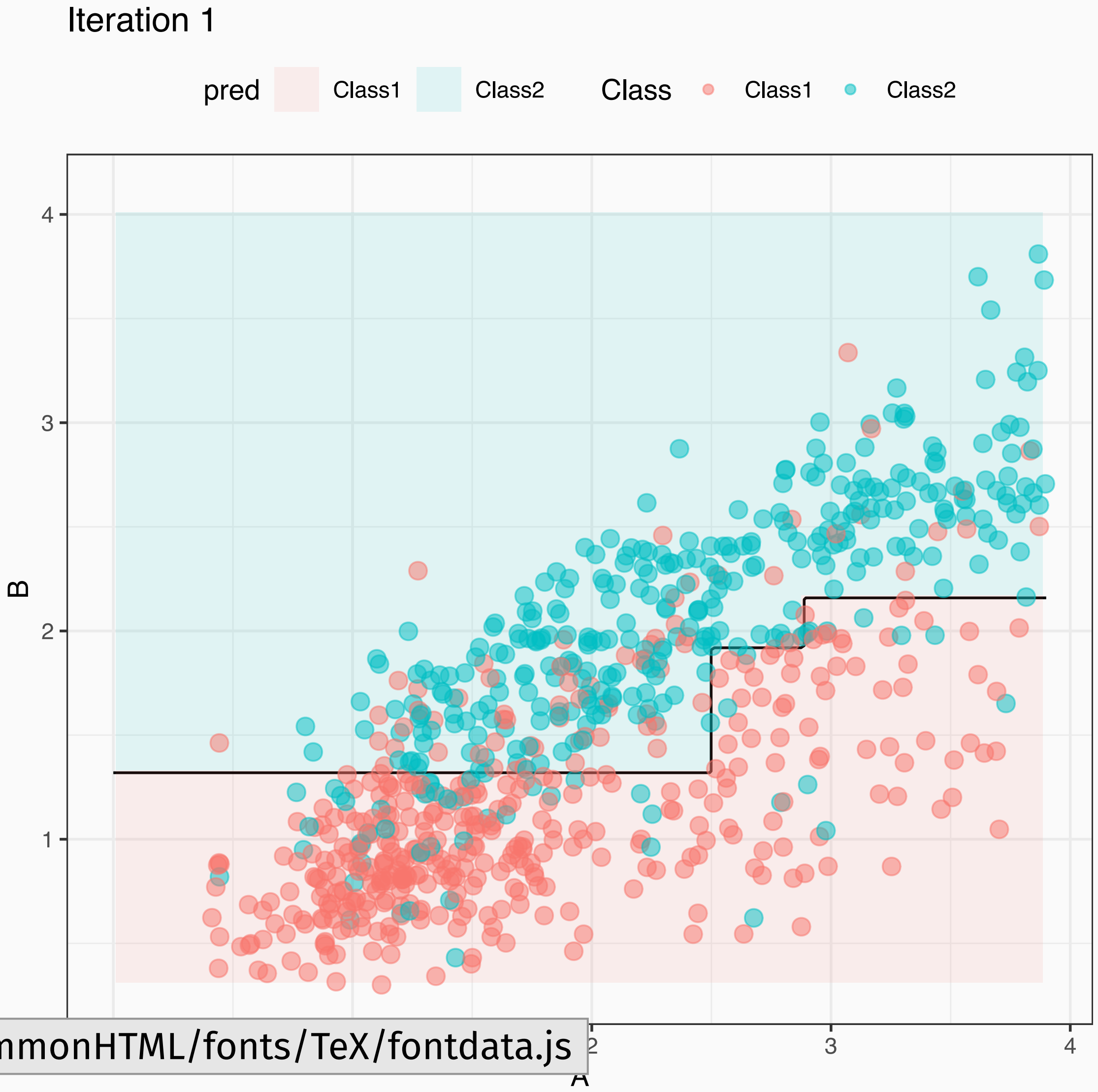
```
Fit an initial tree where all samples are treated equally
for i = 1 to M boosting iterations {
   Samples predicted   correctly have _increased_ weights
   Samples predicted incorrectly have _decreased_ weights

   Fit a new model under the weighting scheme

   Quantify the model fit
}
```

Once this sequence of trees were fit, the final prediction was a weighted average of all of the trees

- These weights were created from the performance estimates for each tree.

This led to a dramatic increase in performance and only works because if the instability of tree-based models.

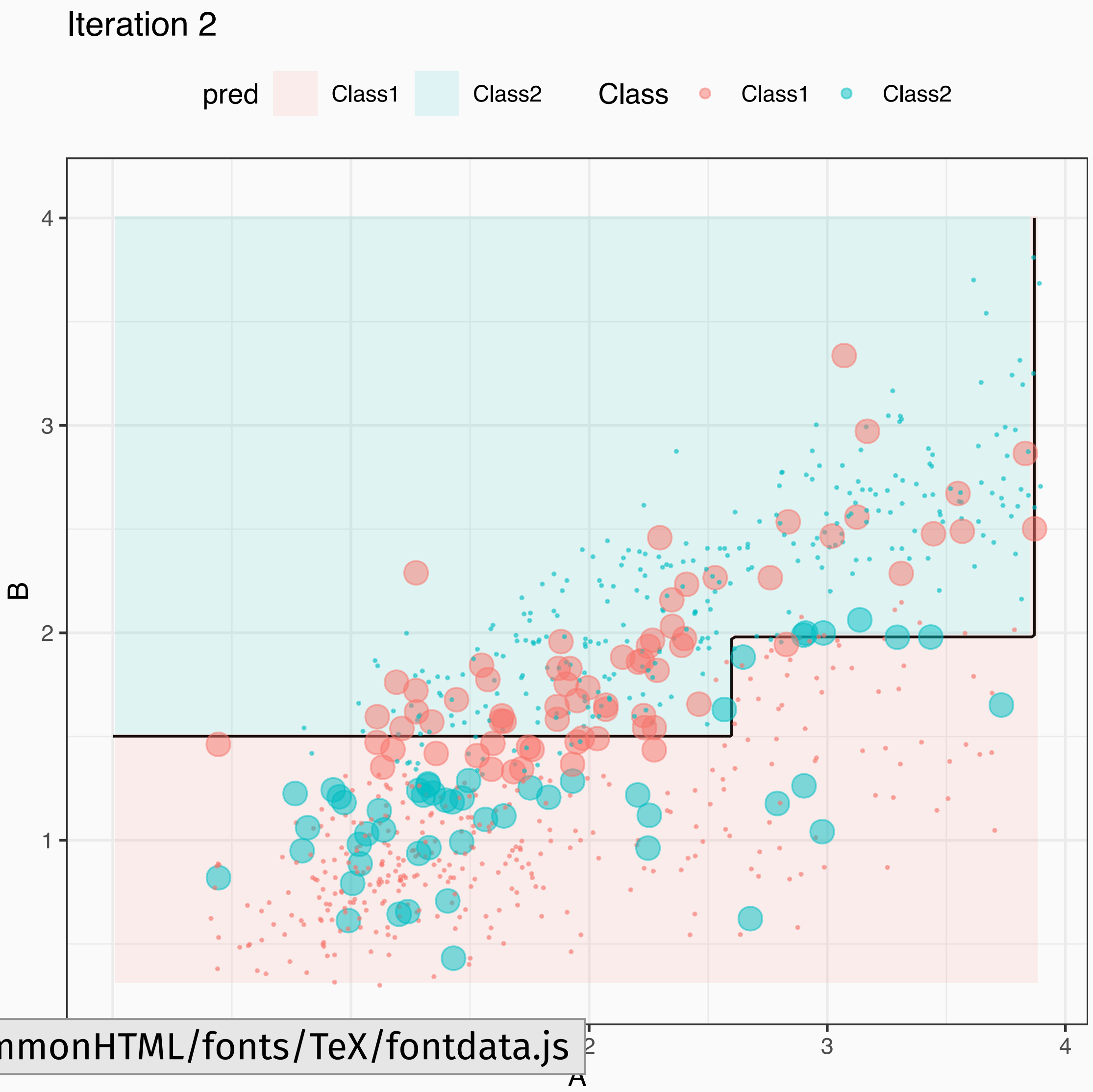
Example tree fit with equal weights



Iteration 1

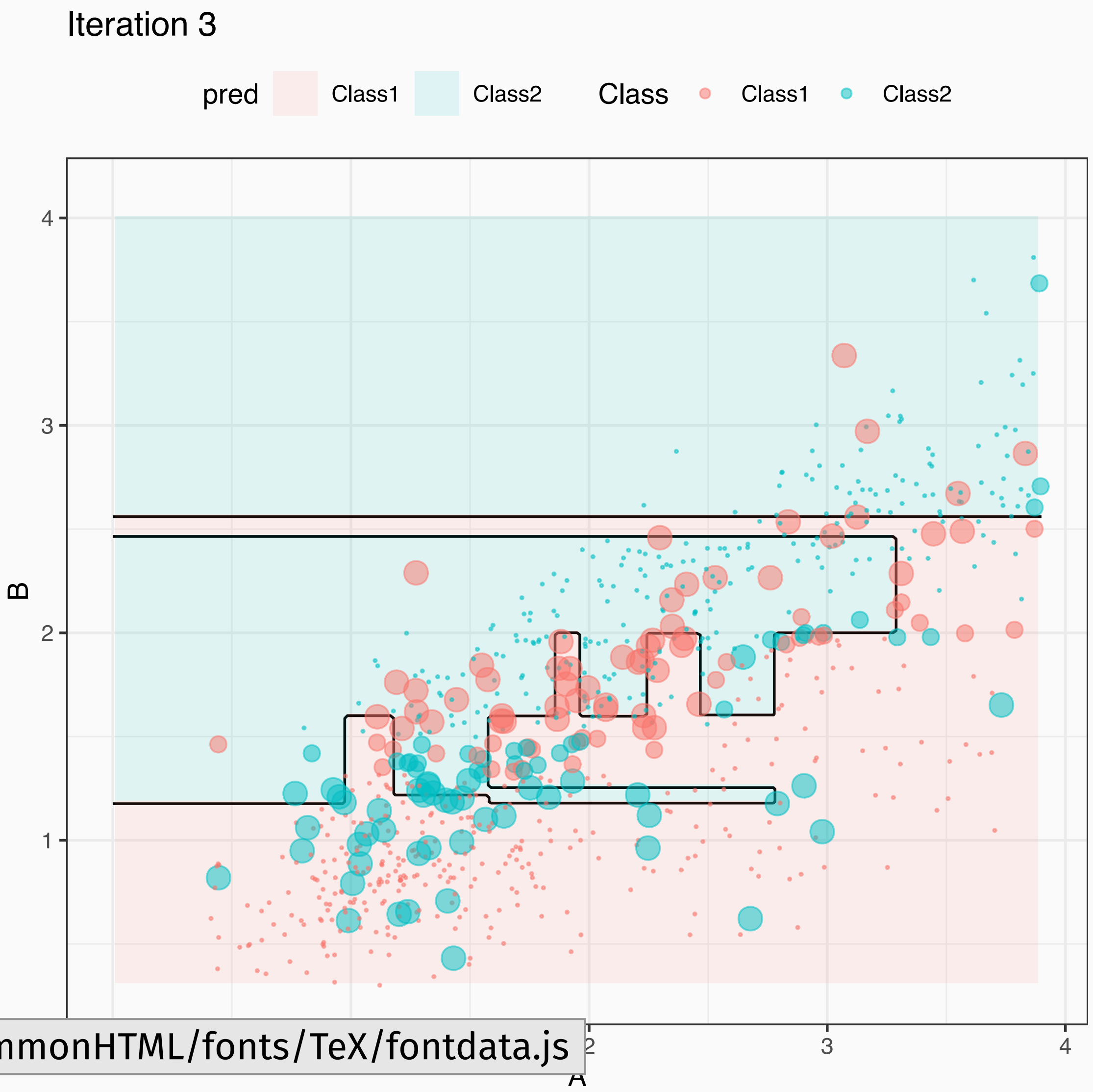
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Boosting iteration 2



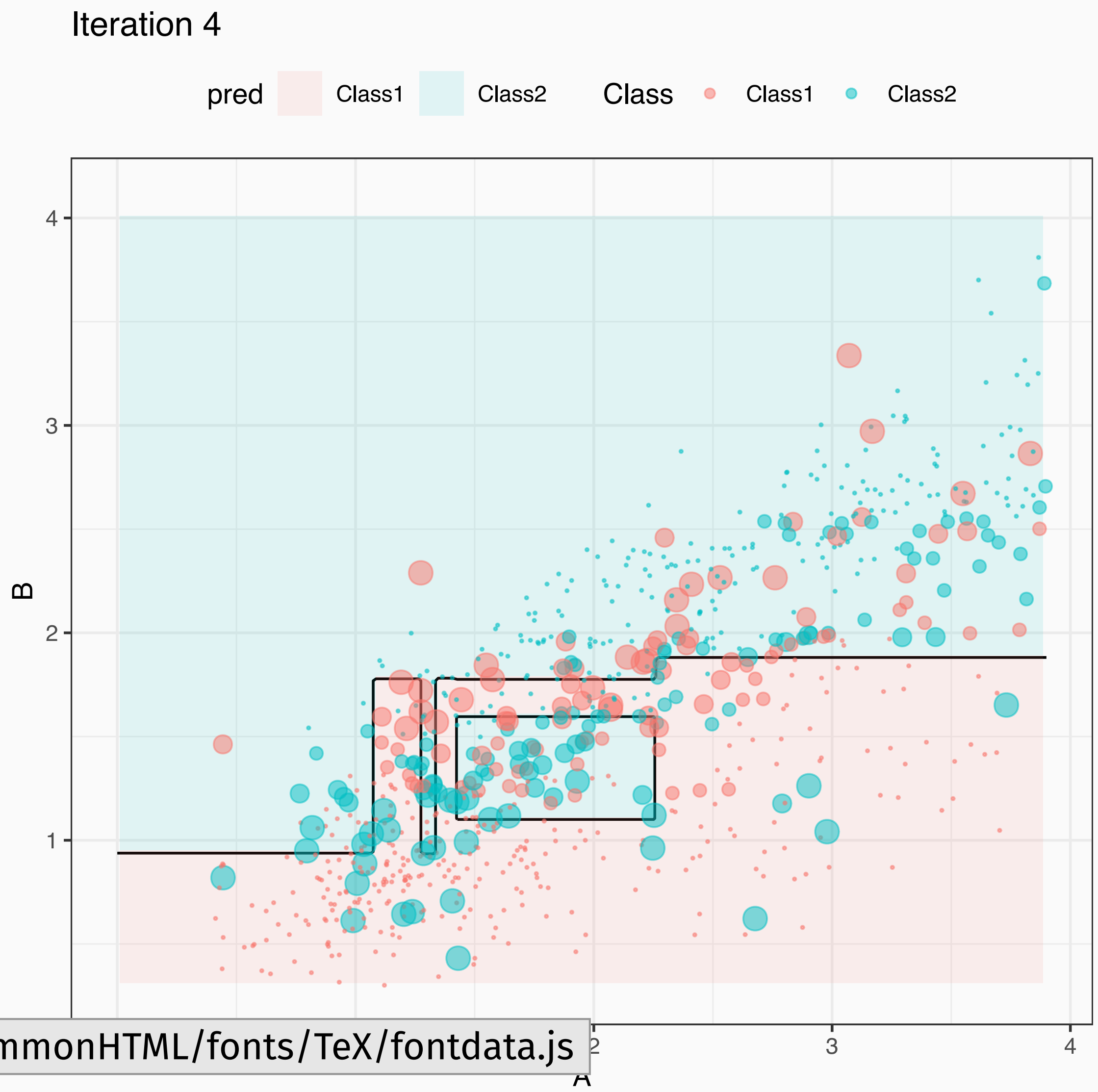
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Boosting iteration 3



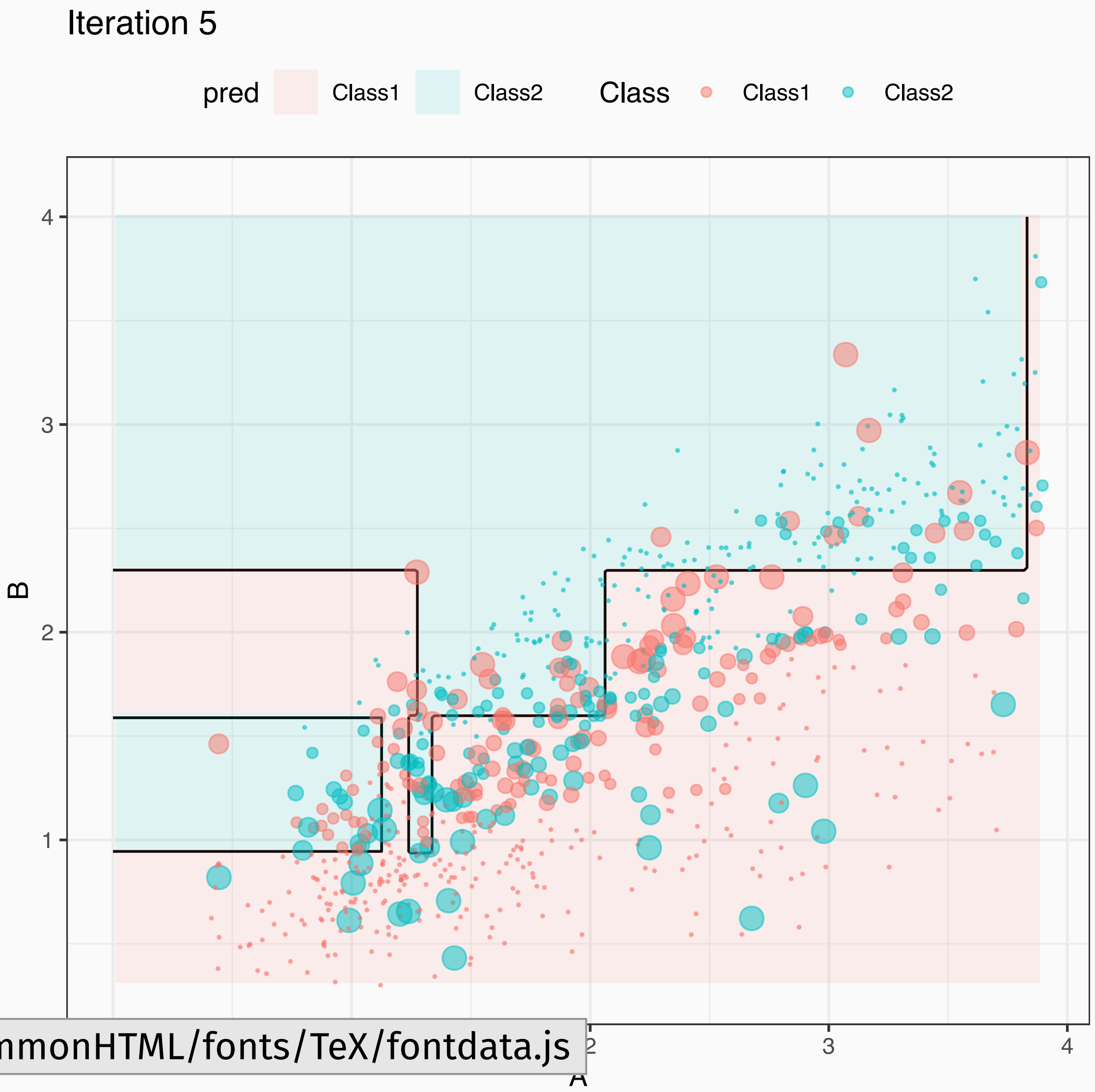
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Boosting iteration 4



Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Boosting iteration 5



Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Initial limitations led to modern boosting

The initial model was only for two-class problems and had some obvious drawbacks.

Once the statisticians took a look at things, they made connections to statistical theory and gradient descent.

- This is why modern boosting is called *stochastic gradient boosting*.

The deep learning people also contributed a lot of interesting work by adding tuning parameters that help optimize these models.

The `xgboost` package is probably the best modern gradient boosting package.

However, we are going to use something less complicated and somewhat old-school but very powerful.

C5.0

A researcher named Ross Quinlan did research on tree- and rule-based models in the same general time frame as the CART folks.

His C4.5 algorithm was somewhat different than CART and, in some ways, much more elegant.

- More than two splits, unbiased selection, better handling of missing values, a different pruning process, etc.

Quinlan's modern models, **C5.0** and **Cubist**, are not as well known mostly since he has not published on them in the literature.

C5.0 takes his C4.5 model and enables a classical boosting approach to building a sequence of trees.

Most improvement comes in the first 50 iterations of boosting and the main two tuning parameters are `min_n` and the number of `trees` in the ensemble.

The `C50` package is based on the original C code for this model. The best references for this are Quinlan's original book and Chapter 14 of *APM*.

C5.0

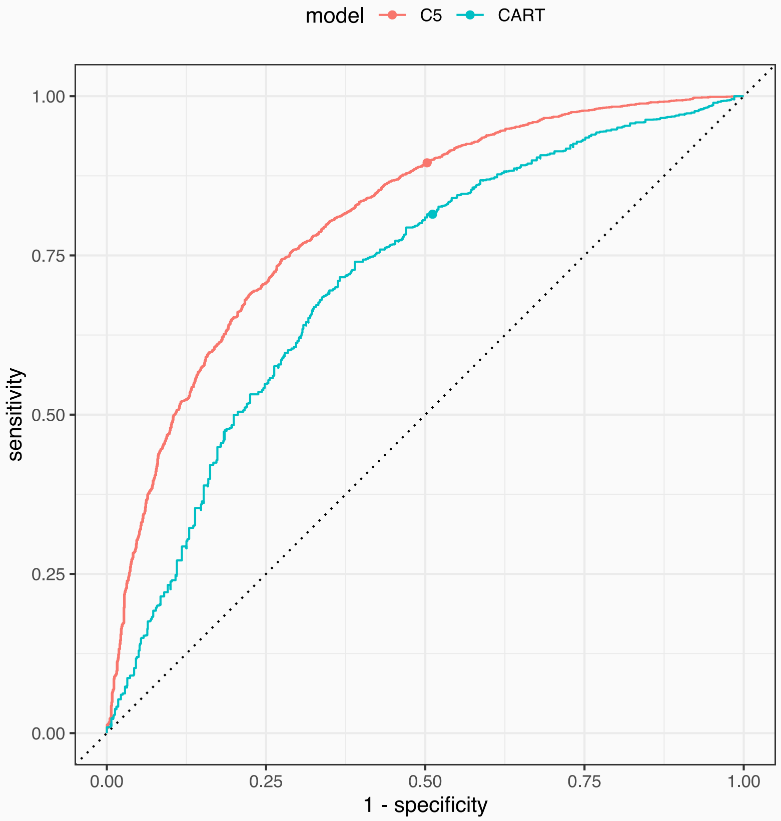
This model is availible as a `parsnip` engine for `decision_tree()` as well as `boosted_tree()`. We will demonstrate with the latter.

```
C5_mod <-
  boost_tree(trees = tune(), min_n = tune()) %>%
  set_engine("C5.0") %>%
  set_mode("classification")

# We will just modify our CART grid and add
# a new parameter:
set.seed(5793)
C5_grid <-
  collect_metrics(cart_tune) %>%
  select(min_n, num_terms) %>%
  mutate(trees = sample(1:100, 10))

C5_tune <-
  tune_grid(
    tree_rec,
    C5_mod,
    folds,
    grid = C5_grid,
    metric = metric_set(roc_auc),
  )
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Comparing models

```
approx_roc_curves(CART = cart_tune, C5 = C5_tune)
```

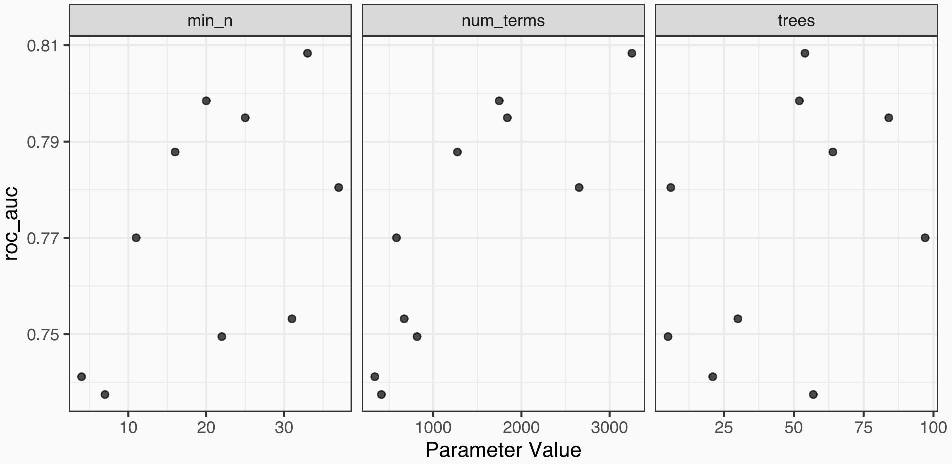


Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
show_best(C5_tune)
```

```
## # A tibble: 5 x 8
##   trees min_n num_terms .metric .estimator  mean     n std_err
##   <int> <int>     <int> <chr>   <chr>      <dbl> <int>   <dbl>
## 1    54    33      3253 roc_auc binary     0.808    10 0.00931
## 2    52    20      1748 roc_auc binary     0.798    10 0.00891
## 3    84    25      1840 roc_auc binary     0.795    10 0.00924
## 4    64    16      1274 roc_auc binary     0.788    10 0.00763
## 5     6    37      2654 roc_auc binary     0.780    10 0.00932
```

```
autoplot(C5_tune)
```



Finalizing the recipe and model

```
best_C5 <- select_best(C5_tune)
best_C5
```

```
## # A tibble: 1 x 3
##   trees min_n num_terms
##   <int> <int>     <int>
## 1    54    33      3253
```

```
final_tree_rec <-
  tree_rec %>%
  finalize_recipe(best_C5) %>%
  prep()

final_C5_mod <-
  C5_mod %>%
  finalize_model(best_C5) %>%
  fit(score ~ ., data = juice(final_tree_rec, -product))
```

```
library(C50) # to get the print method
final_C5_mod
```

```
## parsnip model object
##
## Fit in:  8m 23.9s
## Call:
## C5.0.default(x = x, y = y, trials = 54L, control = C50::C5
##
## Classification Tree
## Number of samples: 4000
## Number of predictors: 3280
##
## Number of boosting iterations: 54 requested;  37 used due
## Average tree size: 21.1
##
## Non-standard options: attempt to group attributes, minimum
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Predicting the test set

```
test_features <-
  bake(final_tree_rec, testing_data, -product, -score)
test_probs <-
  predict(final_C5_mod, test_features, type = "prob") %
  bind_cols(testing_data %>% select(score)) %>%
  bind_cols(predict(final_C5_mod, test_features))

roc_auc(test_probs, score, .pred_great)
```

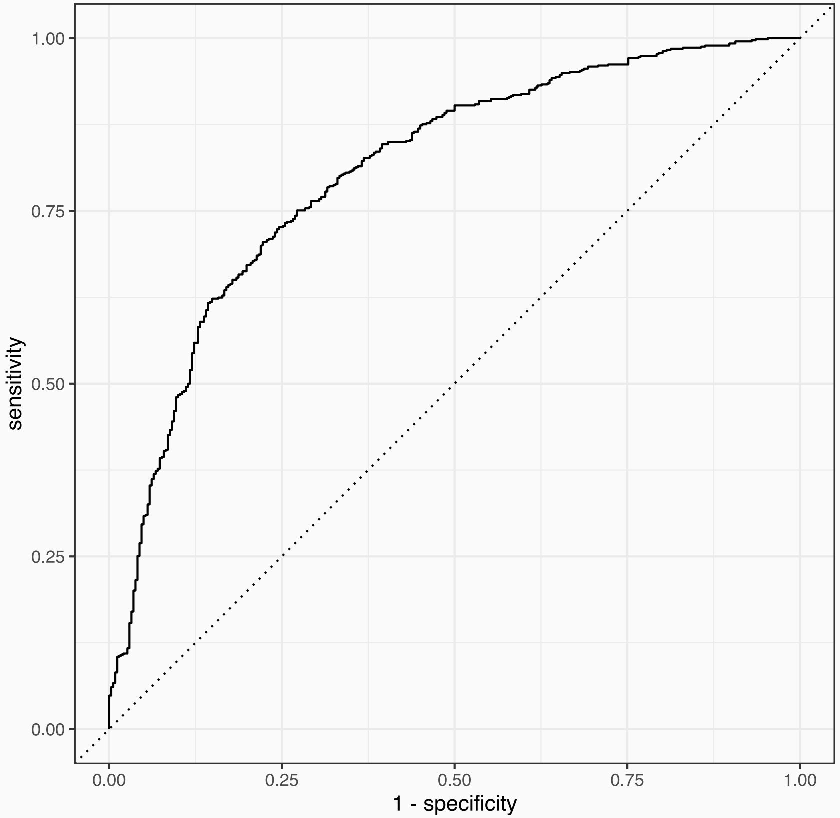
```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc binary         0.807
```

```
conf_mat(test_probs, score, .pred_class)
```

```
##           Truth
## Prediction great other
##      great   580   160
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
roc_values <-
  roc_curve(test_probs, score, .pred_great)
autoplot(roc_values) + thm
```



Extra Slides (as time allows)

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Naive Bayes

Naive Bayes Models

This classification model is motivated directly from statistical theory based on Bayes' Rule:

$$Pr[Class \mid Predictors] = \frac{Pr[Class] \times Pr[Predictors \mid Class]}{Pr[Predictors]} = \frac{Prior \ \times \ Likelihood}{Evidence}$$

In English:

> Given our predictor data, what is the probability of each class?

The *prior* is the prevalence that was mentioned earlier (e.g. the rate of 5-star reviews). This can be estimated or set.

Most of the action is in `Pr[Predictors|Class]`, which is based on the observed training set.

Predictions are based on a blend of the training data and our *prior belief* about the outcome...

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

So Why is it Naive?

Determining $Pr[Predictors | Class]$ can be very difficult without strong assumptions because it measures the *joint probability* of all of the predictors.

- For example, what is the correlation between a person's essay length and their religion?

To resolve this, **naive** Bayes assumes that all of the predictors are *independent* and that their probabilities can be estimated separately.

The joint probability is then the product of all of the individual probabilities (an example follows soon).

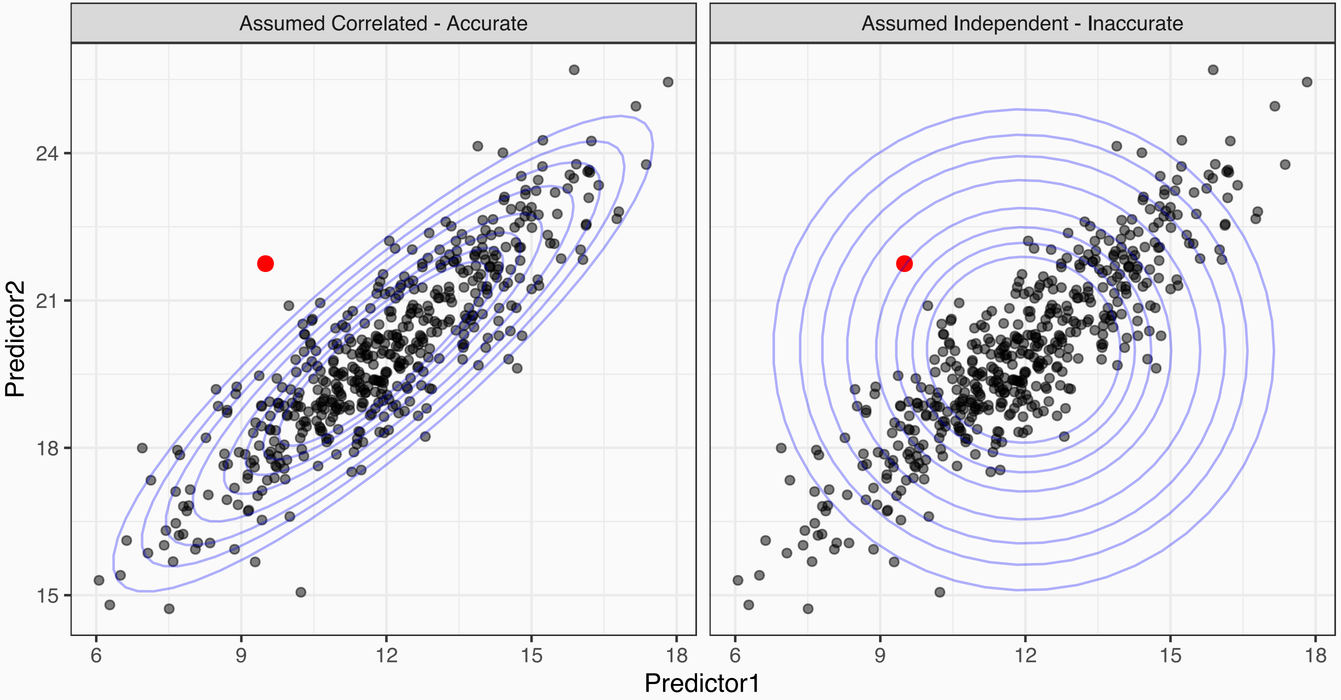
This assumption is almost certainly bogus but the model tends to do well despite this.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

The Effect of Independence

The probability contours assume multivariate normality with different assumptions.

Suppose the red dot is a new sample.

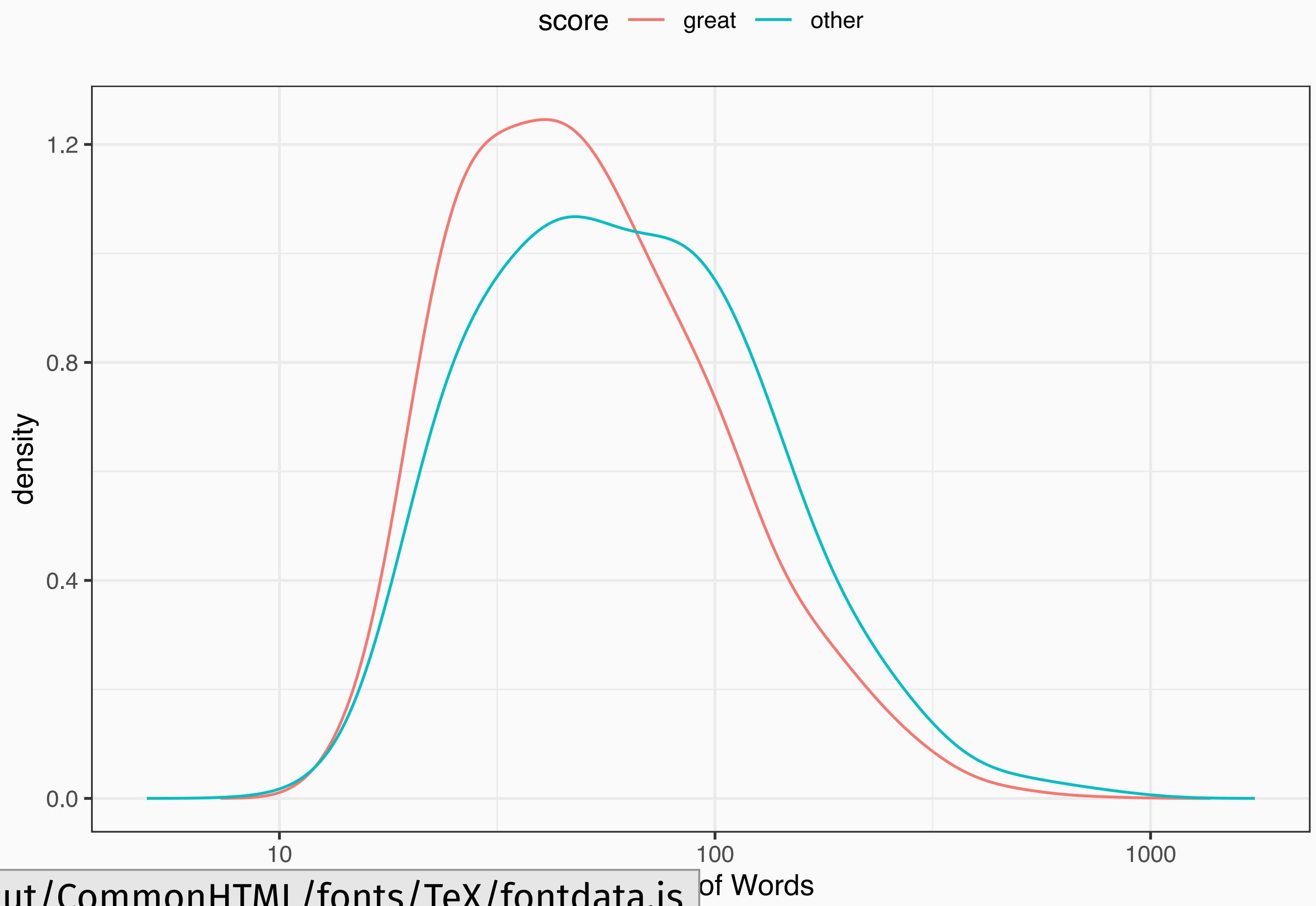


Probability of the red point: 0.0000066 (accurate) and 0.013 (inaccurate).

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Conditional Densities for Each Class

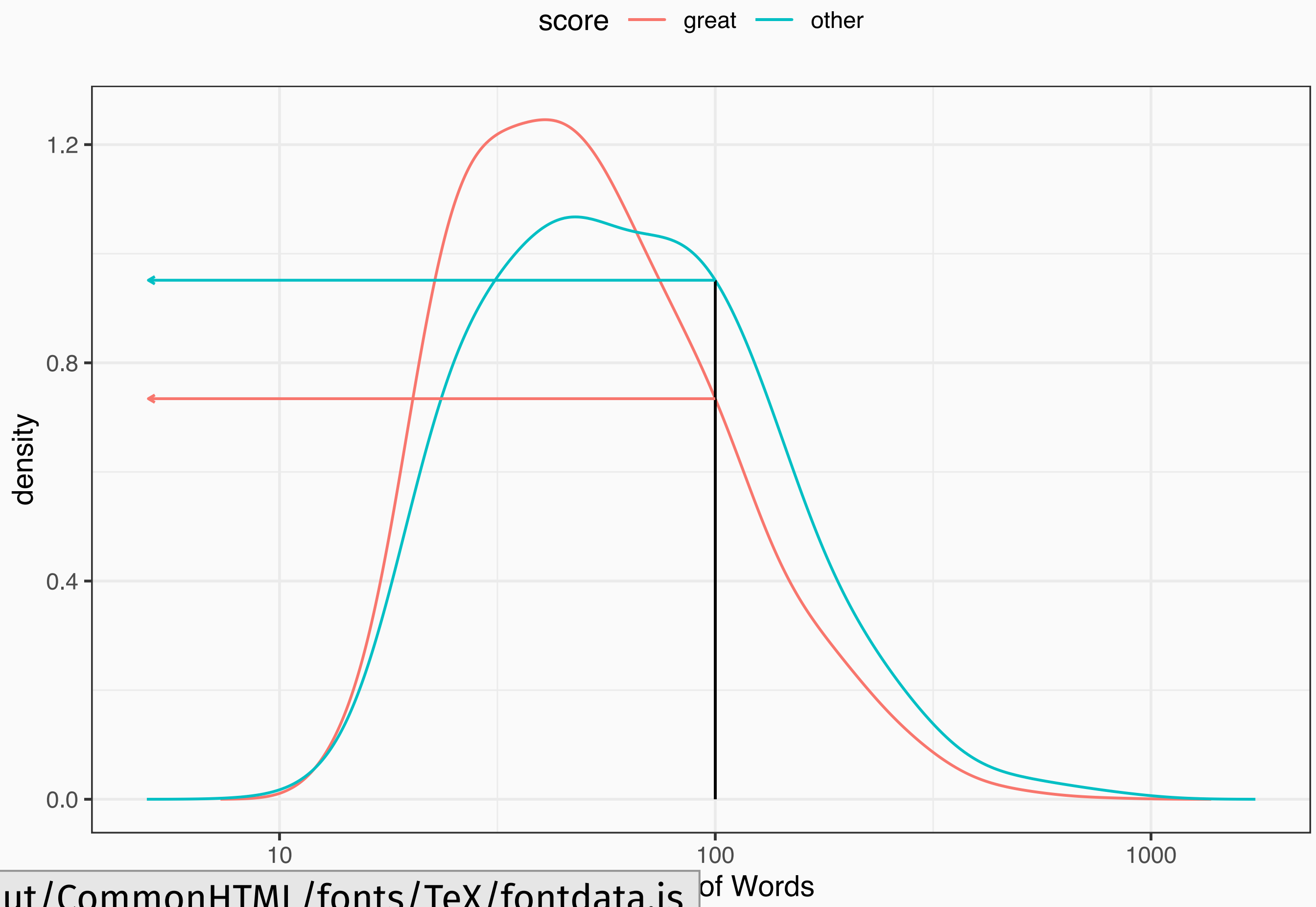
$Pr[Words \mid Class]$



Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Conditional Values for Numeric Predictors

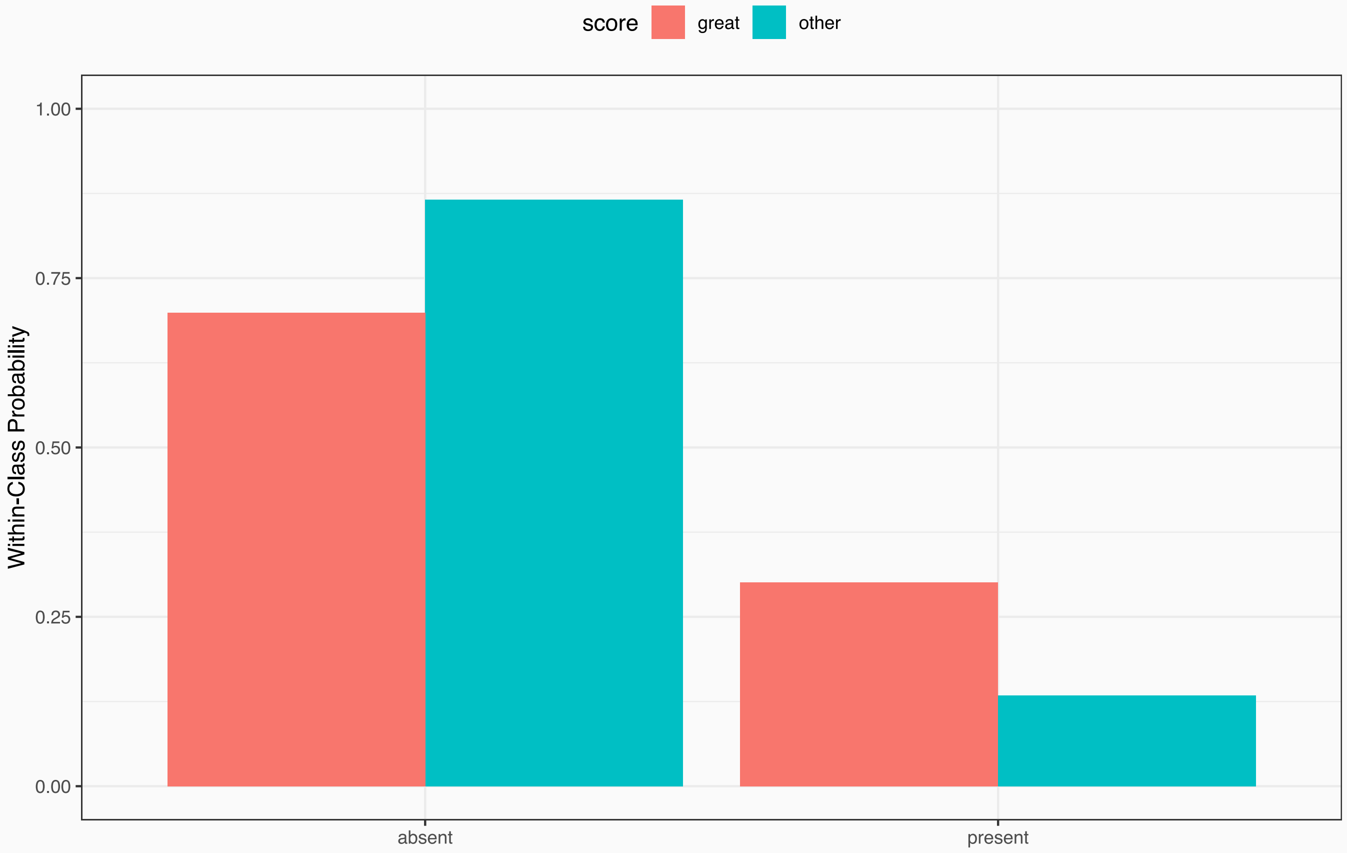
$Pr[Words = 100 \,|\, score]$



Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Conditional Probabilities for Categorical Predictors

$Pr[\text{Has Word } X \,|\, score]$



Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Combining Predictor Scores with the Prior

For a review with 100 words that contained keyword X, their likelihood values were:

- $Pr[Words = 100 | great] \times Pr[\text{Missing Word } X | great]$ = 0.734 x 0.699 = 0.513
- $Pr[Words = 100 | other] \times Pr[\text{Missing Word } X | other]$ =0.951 x 0.866 = 0.824

However, when these are combined with the *prior probability* for each class, the *relative probabilities* show:

- $Pr[Predictors | great] \times Pr[great]$ = 0.513 x 0.65 = 0.334
- $Pr[Predictors | other] \times Pr[other]$ = 0.824 x 0.35 = 0.288

We don't need to compute the evidence; we can just normalize these values to add up to 1.

The results is that the *posterior probability* that this review was 5-star is 53.6%.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Pros and Cons

Good:

- This model can be very quickly trained (and theoretically in parallel).
- Once trained, the prediction is basically a look-up table (i.e. fast).
- Nonlinear class boundaries can be generated.

Bad:

- Linearly diagonal boundaries can be difficult.
- With many predictors, the class probabilities become poorly calibrated and U-shaped with most values near zero or one.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

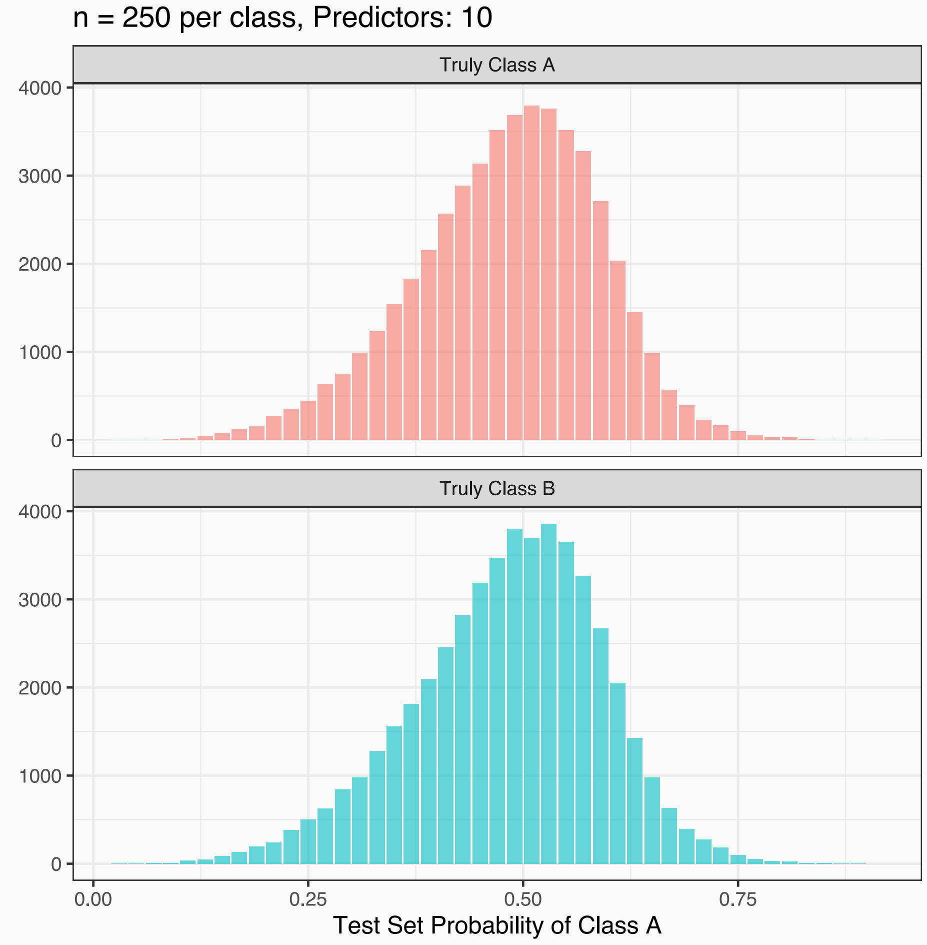
U-Shaped Class Probability Distributions

A completely non-informative data set was simulated using the naive assumption.

The training set has 500 data points over two classes and 450 predictors.

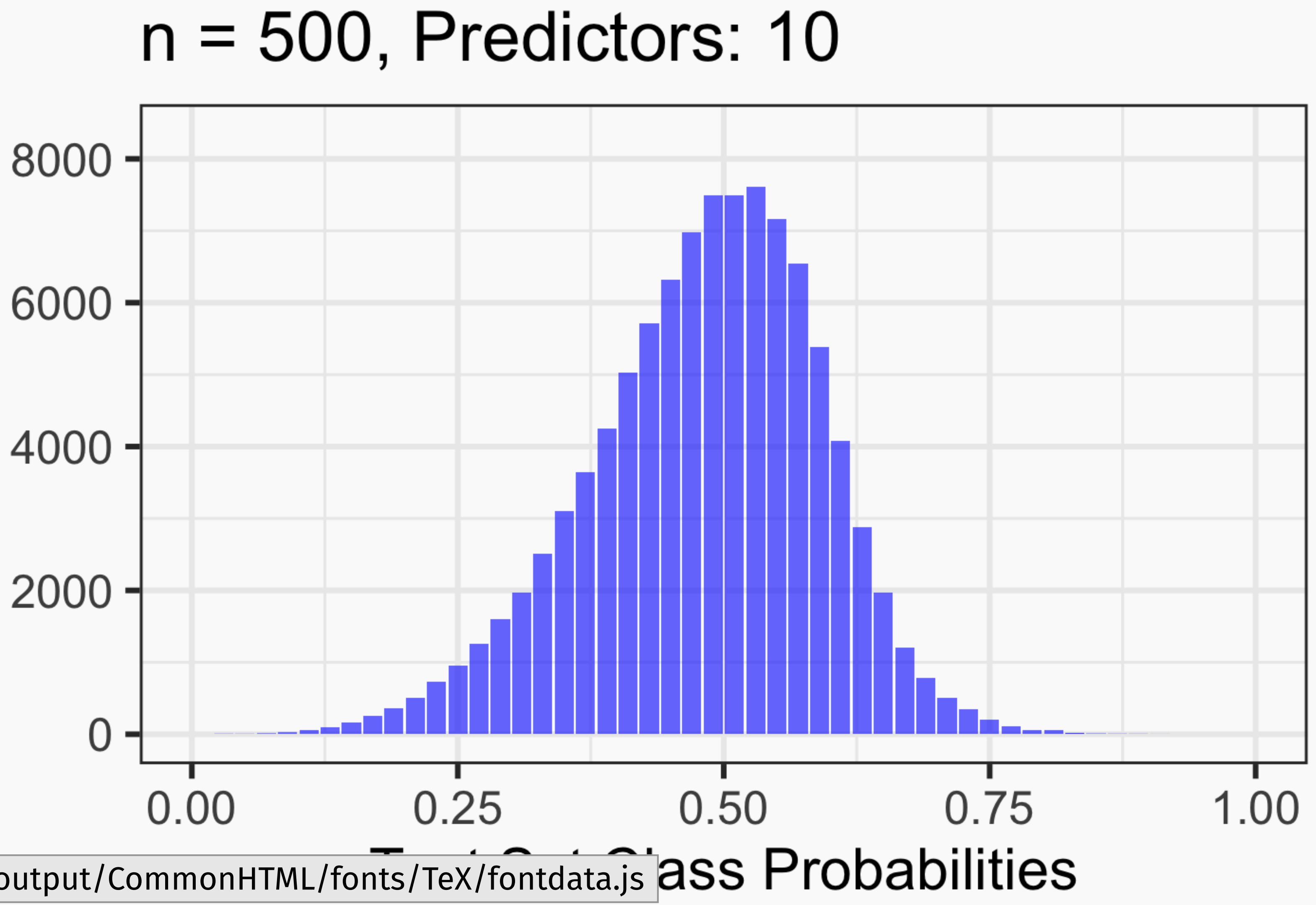
When a model is fit with 10 predictors, the distribution of the class probabilities gives us shapes that we would expect.

What happens when the number of predictors becomes larger?



Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

U-Shaped Class Probability Distributions



Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Naive Bayes recipe and fit

There is a step specifically designed for converting binary dummy variables into factors.

```
count_to_binary <- function(x) {
  ifelse(x != 0, "present", "absent")
}

nb_rec <-
  tree_rec %>%
  step_num2factor(starts_with("hash"), transform = count_to_binary)

library(discrim)

nb_mod <- naive_Bayes() %>% set_engine("klaR")

nb_tune <-
  tune_grid(
    nb_rec,
    nb_mod,
    folds,
    grid = tibble(num_terms = floor(2^seq(8, 12, by = 0.5))),
    metric = metric_set(roc_auc),
    control = ctrl
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Naive Bayes results

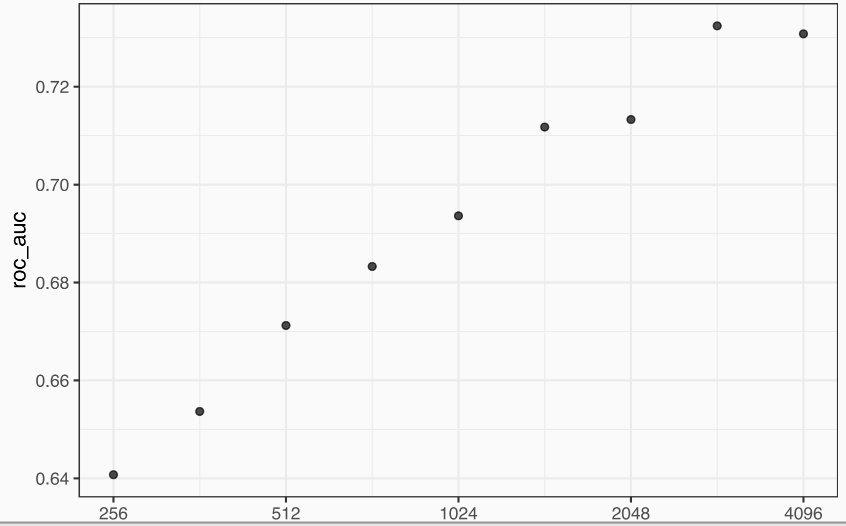
There are a number of warnings that look like:

> `Numerical 0 probability for all classes with observation 1`

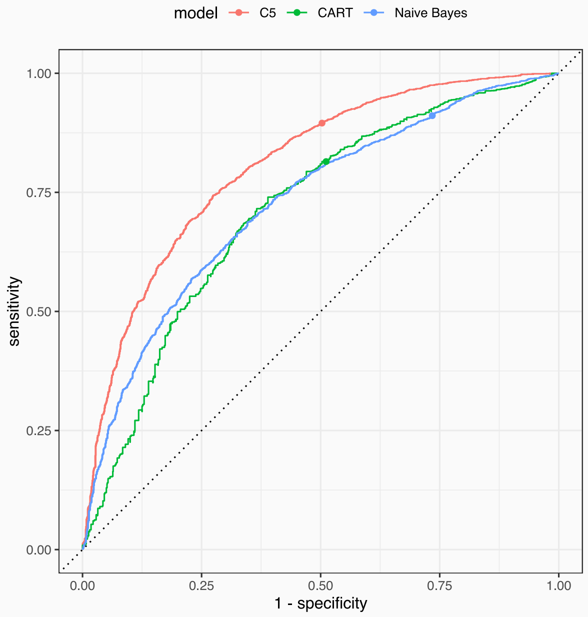
This is due to the poorly calibrated probabilities although the warning is a bit misleading. This issue does not generally affect performance and can be ignored.

```r
autoplot(nb_tune) +
  scale_x_continuous(trans = log2_trans())
```

```r
approx_roc_curves(CART = cart_tune, C5 = C5_tune,
                  "Naive Bayes" = nb_tune)
```



Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js


model
C5
CART
Naive Bayes
sensitivity
1 - specificity