Classification with $K > 2$ Classes: Multinomial Logistic Regression and Gradient Tree Boosting

Restatement of logistic regression

Response variable:

$$Y_i = \begin{cases} 1 \text{ if observation i is in class 1} \\ 0 \text{ if observation i is in class 0} \end{cases}$$

Explanatory variables:

$X_i = (X_{i1}, \ldots, X_{ip})$

Model:

$$\begin{aligned} Y_i | X_i = x_i &\sim \text{Bernoulli}(p(x_i)) \\ p(x_i) = P(Y_i = 1 | X_i = x_i) &= \frac{\exp(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip})}{1 + \exp(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip})} \\ P(Y_i = 0 | X_i = x_i) &= 1 - P(Y_i = 1 | X_i = x_i) \\ &= \frac{1}{1 + \exp(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip})} \\ &= \frac{\exp(0 + 0 x_{i1} + \cdots + 0 x_{ip})}{\exp(0 + 0 x_{i1} + \cdots + 0 x_{ip}) + \exp(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip})} \end{aligned}$$

Note:

$Y_i \sim \text{Bernoulli}(p(x_i))$ is equivalent to $Y_i \sim \text{Binomial}(n = 1, p(x_i))$.

It just means that Y_i is either 0 or 1, and $P(Y_i = 1) = p(x_i)$.

Multinomial Distribution

We now have K classes; label them as $1, \ldots, K$ (starting at 1 instead of 0 by convention)

Notation: Class membership indicator vector

We have defined $Y_i = k$ if observation i is in class k.

It will be helpful to have another way of representing this:

$$Y_i^{*(k)} = \begin{cases} 1 \text{ if } Y_i = k \\ 0 \text{ otherwise} \end{cases}$$

For example, if observation i is in class 1, then we can represent this in two ways:

$Y_i = 1$

$(Y_i^{*(1)}, Y_i^{*(2)}, \ldots, Y_i^{*(K)}) = (1, 0, \ldots, 0)$

Multinomial and Categorical Distributions

The multinomial distribution represents the situation that:

- We have n independent trials
- The class membership probabilities are the same for all trials
- We observe how many of the n trials are in each class

We have to keep track of how many observations were in each class, and what all of the class probabilities are:

$(X_1, \ldots, X_K) \sim \text{Multinomial}\,(n, (p_1, \ldots, p_K))$

We have the following connections between the different distributions:

- The categorical distribution is the same as the multinomial, but with $n = 1$.
- The multinomial is similar to the binomial, but for more than 2 classes.
- The categorical is similar to the Bernoulli, but for more than 2 classes.

Multinomial logistic regression

Model:

$$
\begin{aligned}
(Y_i^{*(1)}, \ldots, Y_i^{*(K)}) | X_i = x_i &\sim \text{Categorical}\,((p_1(x_i), \ldots, p_K(x_i))) \\
p_1(x_i) = P(Y_i = 1 | X_i = x_i) &= \frac{\exp(\beta_0^{(1)} + \beta_1^{(1)} x_{i1} + \cdots + \beta_p^{(1)} x_{ip})}{\exp(\beta_0^{(1)} + \beta_1^{(1)} x_{i1} + \cdots + \beta_p^{(1)} x_{ip}) + \cdots + \exp(\beta_0^{(K)} + \beta_1^{(K)} x_{i1} + \cdots + \beta_p^{(K)} x_{ip})} \\
p_2(x_i) = P(Y_i = 2 | X_i = x_i) &= \frac{\exp(\beta_0^{(2)} + \beta_1^{(2)} x_{i1} + \cdots + \beta_p^{(2)} x_{ip})}{\exp(\beta_0^{(1)} + \beta_1^{(1)} x_{i1} + \cdots + \beta_p^{(1)} x_{ip}) + \cdots + \exp(\beta_0^{(K)} + \beta_1^{(K)} x_{i1} + \cdots + \beta_p^{(K)} x_{ip})} \\
&\vdots \\
p_K(x_i) = P(Y_i = K | X_i = x_i) &= \frac{\exp(\beta_0^{(K)} + \beta_1^{(K)} x_{i1} + \cdots + \beta_p^{(K)} x_{ip})}{\exp(\beta_0^{(1)} + \beta_1^{(1)} x_{i1} + \cdots + \beta_p^{(1)} x_{ip}) + \cdots + \exp(\beta_0^{(K)} + \beta_1^{(K)} x_{i1} + \cdots + \beta_p^{(K)} x_{ip})}
\end{aligned}
$$

Basically, this is an involved way of saying that the probability that Y_i is in class k is

$$\frac{\exp(\beta_0^{(k)} + \beta_1^{(k)} x_{i1} + \cdots + \beta_p^{(k)} x_{ip})}{\exp(\beta_0^{(1)} + \beta_1^{(1)} x_{i1} + \cdots + \beta_p^{(1)} x_{ip}) + \cdots + \exp(\beta_0^{(K)} + \beta_1^{(K)} x_{i1} + \cdots + \beta_p^{(K)} x_{ip})}$$

Note that the first staterment above is equivalent to

$$(Y_i^{*(1)}, \ldots, Y_i^{*(K)}) | X_i = x_i \sim \text{Multinomial}\,(n = 1, (p_1(x_i), \ldots, p_K(x_i)))$$

Lack of identifiability

- **Definition:** Parameters in a model are *identifiable* if different values of the model parameters yield different distributions for the observable random variables being modelled.
- The parameters of multinomial logistic regression as stated above are not identifiable.
  - For example, we could add 5 to each of $\beta_0^{(1)}, \ldots, \beta_0^{(K)}$ and obtain the same class probabilities.
- Different software packages handle this in different ways. One strategy, used by `nnet::multinom`, is to fix all coefficients for the first class to 0: $\beta_0^{(1)} = \beta_1^{(1)} = \beta_p^{(1)} = 0$
- If a penalty is used on the coefficients (as in lasso or ridge), the lack of identifiability in the model structure is not a problem.

Softmax transformation of linear functions of explanator variables

- **Definition:** The *softmax transformation* is defined by the mapping

$$(a_1, a_2, \ldots, a_K) \mapsto \left(\frac{\exp(a_1)}{\sum_k \exp(a_k)}, \frac{\exp(a_2)}{\sum_k \exp(a_k)}, \ldots, \frac{\exp(a_K)}{\sum_k \exp(a_k)} \right)$$

  - The numbers $a_1, \ldots, a_K$ can be any real number
  - $0 < \frac{\exp(a_j)}{\sum_k \exp(a_k)} < 1$ for each $j = 1, \ldots, K$
  - $\sum_j \frac{\exp(a_j)}{\sum_k \exp(a_k)} = 1$
  - So basically, we took some real numbers $a_1, \ldots, a_K$ (could be negative! could be large!) and turned them into a valid set of probabilities for the K categories.
- One way of viewing multinomial logistic regression is that we pass K linear regressions through the softmax transformation to obtain a set of class membership probabilities that sum to 1.

Gradient Tree Boosting for Multiple Classes

Main Idea: softmax transformation of sums of regression trees

- Last class, our big idea was to take logistic regression and replace the (essentially) linear form in the explanatory variables with a sum of regression trees
- Now we will do the same, but we associate a separate sum of regression trees with each class:

$$\begin{aligned}
P(Y_i = 1|X_i) = p_1(x_i) &= \frac{\exp(\text{sum of regression trees for class 1})}{\exp(\text{sum of regression trees for class 1}) + \cdots + \exp(\text{sum of regression trees for class K})} \\
&= \frac{\exp\left\{a^{(1)}(x_i)\right\}}{\exp\left\{a^{(1)}(x_i)\right\} + \cdots + \exp\left\{a^{(K)}(x_i)\right\}} \\
P(Y_i = 2|X_i) = p_2(x_i) &= \frac{\exp(\text{sum of regression trees for class 2})}{\exp(\text{sum of regression trees for class 1}) + \cdots + \exp(\text{sum of regression trees for class K})} \\
&= \frac{\exp\left\{a^{(2)}(x_i)\right\}}{\exp\left\{a^{(1)}(x_i)\right\} + \cdots + \exp\left\{a^{(K)}(x_i)\right\}} \\
&\vdots \\
P(Y_i = K|X_i) = p_K(x_i) &= \frac{\exp(\text{sum of regression trees for class K})}{\exp(\text{sum of regression trees for class 1}) + \cdots + \exp(\text{sum of regression trees for class K})} \\
&= \frac{\exp\left\{a^{(K)}(x_i)\right\}}{\exp\left\{a^{(1)}(x_i)\right\} + \cdots + \exp\left\{a^{(K)}(x_i)\right\}}
\end{aligned}$$

where $a^{(k)}(x_i)$ is obtained as a sum of B regression trees associated with class k:

$$a^{(k)}(x_i) = \sum_{b=1}^{B} \sum_{m=1}^{|T|^{(b,k)}} \mathbb{I}_{R_m^{(b,k)}}(x_i) \hat{y}_m^{(b,k)}.$$

Algorithm: Gradient Tree Boosting for Multiple Classes

The gradient tree boosting procedure estimates the regression trees one group at a time, fitting each tree to (two views of the same thing):

- The gradient of the log-likelihood, which is based on the model $Y_i|X_i \sim \text{Multinomial}\left(n = 1, p = (p_1(x_i), \ldots, p_K(x_i))\right)$
- The "residuals" $y_i^{*(k)} - p_k(x_i)$

Notes:

- We could derive these residuals as the derivatives of the log-likelihood
- If we had 2 classes, this is exactly the same as the procedure we talked about for boosting with 2 classes last week
- This is like doing a first-order Taylor series approximation to the log-likelihood; xgboost implements a second-order Taylor series approximation.

Algorithm:

1. Initialize $\hat{a}^{(k,0)}(x) = 0$ for all k. (first index in superscript is class, second is boosting iteration)
2. For $b = 1, \ldots, B$:
   a. Compute the predicted probability of being in class k for each observation i, based on the ensemble that was created after the previous step: $p_k^{(b-1)}(x_i) = \hat{P}^{(b-1)}(Y_i = k|x_i) = \frac{e^{\hat{a}^{(k,b-1)}(x_i)}}{e^{\hat{a}^{(1,b-1)}(x_i)} + \cdots + e^{\hat{a}^{(K,b-1)}(x_i)}}$
   b. For each class $k = 1, \ldots, K$, fit a regression tree $\hat{g}^{(k,b)}(x)$ using the residuals $y_i^{*(k)} - p_k^{(b-1)}(x_i)$ as the response.
   c. Update the ensemble by adding in this new model: $\hat{a}^{(k,b)}(x) = \hat{a}^{(k,b-1)}(x) + \hat{g}^{(k,b)}(x)$

Example: Vertebral Column

Our data example uses "six biomechanical attributes derived from the shape and orientation of the pelvis and lumbar spine" for a patient to classify the patient into one of three groups representing different conditions that may be affecting their spine: "DH" for disk hernia, "SL" for Spondylolisthesis, or "NO" for normal (neither of the other two conditions).

The data are available at https://archive.ics.uci.edu/ml/datasets/Vertebral+Column and were discussed in the following article:

Berthonnaud, E., Dimnet, J., Roussouly, P. & Labelle, H. (2005). 'Analysis of the sagittal balance of the spine and pelvis using shape and orientation parameters', Journal of Spinal Disorders & Techniques, 18(1):40â€“47.

Reading the data in, preprocessing, train/test split, cross-validation splits

```
library(readr)
library(purrr)
library(dplyr)
library(ggplot2)
library(gridExtra)
library(rpart)
library(caret)

vertebral_column <- read_table2("http://www.evanlray.com/data/UCIML/vertebral_column/column_3C.dat",
  col_names = FALSE)
names(vertebral_column) <- c(paste0("X_", 1:6), "type")

vertebral_column <- vertebral_column %>%
  mutate(
    type = factor(type)
  )

set.seed(723)

# Train/test split
tt_inds <- caret::createDataPartition(vertebral_column$type, p = 0.7)
train_set <- vertebral_column %>% slice(tt_inds[[1]])
test_set <- vertebral_column %>% slice(-tt_inds[[1]])
```

```r
# Cross-validation splits
crossval_val_fold_inds <- caret::createFolds(
  y = train_set$type, # response variable as a vector
  k = 10 # number of folds for cross-validation
)

get_complementary_inds <- function(val_inds) {
  return(seq_len(nrow(train_set))[-val_inds])
}

crossval_train_fold_inds <- map(crossval_val_fold_inds, get_complementary_inds)
```


```r
# Cross-validation splits
crossval_val_fold_inds <- caret::createFolds(
  y = train_set$type, # response variable as a vector
  k = 10 # number of folds for cross-validation
)

get_complementary_inds <- function(val_inds) {
  return(seq_len(nrow(train_set))[-val_inds])
}

crossval_train_fold_inds <- map(crossval_val_fold_inds, get_complementary_inds)
```


Fit and test set classification error rate via multinomial logistic regression

Fit model, pick tuning parameter values yielding highest cross-validated accuracy.

```r
multilogistic_fit <- train(
  type ~ .,
  data = train_set,
  trace = FALSE,
  method = "multinom",
  trControl = trainControl(
    method = "cv",
    number = 10,
    returnResamp = "all",
    index = crossval_train_fold_inds,
    indexOut = crossval_val_fold_inds,
    savePredictions = TRUE
  ),
  tuneGrid = data.frame(decay = seq(from = 0, to = 0.2, length = 30))
)

multilogistic_fit$results %>% filter(Accuracy == max(Accuracy))
```

```
##   decay  Accuracy     Kappa AccuracySD  KappaSD
## 1     0 0.8626012 0.7802579 0.06603197 0.103936
```

Test set performance: looking for low test set error rate, high test set accuracy

```r
mean(test_set$type != predict(multilogistic_fit, test_set))
```

```
## [1] 0.1290323
```

```r
mean(test_set$type == predict(multilogistic_fit, test_set))
```

```
## [1] 0.8709677
```

Fit and test set classification error rate via gradient tree boosting

Fit model, pick tuning parameter values yielding highest cross-validated accuracy.

```
xgb_fit <- train(
  type ~ .,
  data = train_set,
  method = "xgbTree",
  trControl = trainControl(
    method = "cv",
    number = 10,
    returnResamp = "all",
    index = crossval_train_fold_inds,
    indexOut = crossval_val_fold_inds,
    savePredictions = TRUE
  ),
  tuneGrid = expand.grid(
    nrounds = c(5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100),
    eta = c(0.5, 0.6, 0.7), # learning rate; 0.3 is the default
    gamma = 0, # minimum loss reduction to make a split; 0 is the default
    max_depth = 1:5, # how deep are our trees?
    subsample = c(0.5, 0.9, 1), # proportion of observations to use in growing each tree
    colsample_bytree = 1, # proportion of explanatory variables used in each tree
    min_child_weight = 1 # think of this as how many observations must be in each leaf node
  )
)

xgb_fit$results %>% filter(Accuracy == max(Accuracy))
```

```
##   eta max_depth gamma colsample_bytree min_child_weight subsample nrounds
## 1 0.6         4     0                1                1         1      10
##    Accuracy     Kappa AccuracySD   KappaSD
## 1 0.8620083 0.7787296 0.09117279 0.1439897
```

Test set performance: looking for low test set error rate, high test set accuracy

```
mean(test_set$type != predict(xgb_fit, test_set))
```

```
## [1] 0.172043
```

```
mean(test_set$type == predict(xgb_fit, test_set))
```

```
## [1] 0.827957
```

Fit and test set classification error rate via K nearest neighbors

Fit model, pick tuning parameter values yielding highest cross-validated accuracy.

```
knn_fit <- train(
  type ~ .,
  data = train_set,
  method = "knn",
  trControl = trainControl(
    method = "cv",
    number = 10,
    returnResamp = "all",
    index = crossval_train_fold_inds,
    indexOut = crossval_val_fold_inds,
    savePredictions = TRUE
  ),
  tuneLength = 5
)
```

```
knn_fit$results %>% filter(Accuracy == max(Accuracy))
```

```
##    k  Accuracy     Kappa AccuracySD   KappaSD
## 1 11 0.8346603 0.7340093 0.06958464 0.1090935
```

Test set performance: looking for low test set error rate, high test set accuracy

```
mean(test_set$type != predict(knn_fit, test_set))
```

```
## [1] 0.172043
```

```
mean(test_set$type == predict(knn_fit, test_set))
```

```
## [1] 0.8387097
```

Fit and test set classification error rate via random forest

Fit model, pick tuning parameter values yielding highest cross-validated accuracy.

```
rf_fit <- train(
  type ~ .,
  data = train_set,
  method = "rf",
  trControl = trainControl(
    method = "cv",
    number = 10,
    returnResamp = "all",
    index = crossval_train_fold_inds,
    indexOut = crossval_val_fold_inds,
    savePredictions = TRUE
  ),
  tuneLength = 5
)

rf_fit$results %>% filter(Accuracy == max(Accuracy))
```

```
##   mtry  Accuracy     Kappa AccuracySD   KappaSD
## 1    3 0.8487484 0.7560688 0.06403511 0.1023644
```

Test set performance: looking for low test set error rate, high test set accuracy

```
mean(test_set$type != predict(rf_fit, test_set))
```

```
## [1] 0.1612903
```

```
mean(test_set$type == predict(rf_fit, test_set))
```

```
## [1] 0.8387097
```