Evaluation Metrics for Classification Models

Brooke Anderson

January 26, 2016

```r
knitr::opts_knit$set(fig.path = '../figures/EvalMetrics-',
                     root.dir = '..')
```

Note: I was tired of my Rmd documents, which I put in the sub-directory "Rscripts", running from a different working directory than my R session when I'm working on this project (which was running from "Titanic", the parent directory of that "Rscripts directory"). This was making me have to use one relative pathname when I knitted and a different when I tested code chunks interactively. Therefore, I used the `root.dir` option in `opts_knit` to change the working directory that will be used when I knit to the parent directory of where the Rmd file is saved.

Load required libraries.

```r
library(dplyr)  ## Data wrangling
library(klaR) ## Includes `NaiveBayes` function
library(e1071)
library(caret)
library(pROC)
library(class) ## Includes `knn` function
library(microbenchmark)  ## For measuring computing performance
```

Bring in the data:

```r
train <- read.csv("data/train.csv") %>%
  mutate(Survived = factor(Survived),
         Pclass = factor(Pclass),
         Name = as.character(Name),
         Sex = factor(Sex))
test <- read.csv("data/test.csv") %>%
  mutate(Pclass = factor(Pclass),
         Name = as.character(Name),
         Sex = factor(Sex))
```

For this, I'll do different evaluation metrics for a Naive Bayes with only sex as a predictor. I think that the first of these might be the benchmark models for Kaggle, with the "predict survival if woman, death otherwise" model that some folks in class put in.

This time, I'll try using the `NaiveBayes` function from the `klaR` package. Based on Kuhn and Johnson, in comparing this with the `e1071` package:

> "Both offer Laplace corrections, but the version in the `klaR` package has the option of using conditional density estimates that are more flexible."

```r
nb_sex <- NaiveBayes(Survived ~ Sex, data = train)
```

This function's output provides some of the same values as the other Naive Bayes function I tried, including $P(Y)$ and $P(X|Y)$:

```
nb_sex$apriori
```

```
## grouping
##         0         1
## 0.6161616 0.3838384
```

```
nb_sex$tables
```

```
## $Sex
##         var
## grouping    female      male
##        0 0.1475410 0.8524590
##        1 0.6812865 0.3187135
```

Evidently, you can use `predict` with this function, including with an optional argument specifying `newdata`. To predict with the training data:

```
pred_train <- predict(nb_sex, newdata = train)
```

This prediction includes two elements: the class predictions (0 = died; 1 = survived) and the posterior class probabilities:

```
names(pred_train)
```

```
## [1] "class"     "posterior"
```

```
head(pred_train$class)
```

```
## [1] 0 1 1 1 0 0
## Levels: 0 1
```

```
head(pred_train$posterior)
```

```
##              0         1
## [1,] 0.8110919 0.1889081
## [2,] 0.2579618 0.7420382
## [3,] 0.2579618 0.7420382
## [4,] 0.2579618 0.7420382
## [5,] 0.8110919 0.1889081
## [6,] 0.8110919 0.1889081
```

I checked to see if this always predicts that women survive and men die:

```
table(train$Sex, pred_train$class)
```

```
##
##            0   1
##   female   0 314
##   male   577   0
```

Yep. So, this should give the results from the benchmark “Sex” model in the Kaggle competition.

To create predictions to submit to Kaggle:

```r
pred_test <- predict(nb_sex, newdata = test)
```

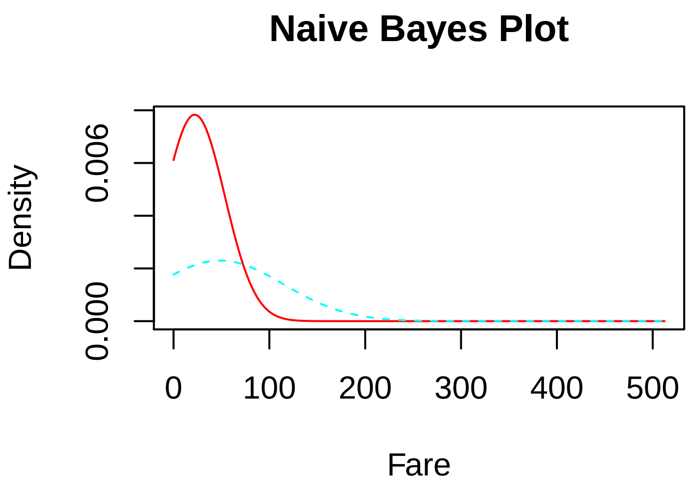
And write them out to a comma-separated file. Note that the `pred_test$class` is saved as a factor variable, so if you don’t convert to a character and then a number using the `as.*` functions, you might get outputs of 1s and 2s instead of 0s and 1s, which would cause problems when you submit to Kaggle.

```r
out <- cbind(test$PassengerId,
             as.numeric(as.character(pred_test$class)))
colnames(out) <- c("PassengerId","Survived")
write.csv(out, file = "predictions/nb_sex.csv", row.names = FALSE)
```

A few notes on the `klaR` package

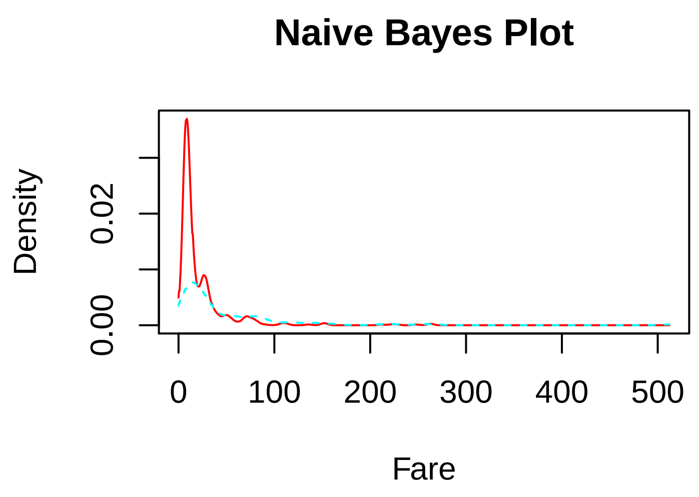
As a note, this particular function for Naive Bayes lets you do a few interesting things. First, it lets you generate a plot or plots to see the conditional probablities for any continuous predictors:

```r
nb_fare <- NaiveBayes(Survived ~ Fare, data = train)
plot(nb_fare, legendplot = FALSE)
```



It also lets you use a kernel density estimate rather than estimating a normal density if you specify `usekernel = TRUE`:

```r
nb_fare <- NaiveBayes(Survived ~ Fare, data = train, usekernel = TRUE)
plot(nb_fare, legendplot = FALSE)
```



This function also allows the user to specify their own prior probabilities.

Evaluation metrics

Accuracy

Kaggle is judging this competition on accuracy, so that's the main metrics we've been using. In the training dataset, the accuracy of this model is:

```
mean(train$Survived == pred_train$class)
```

```
## [1] 0.7867565
```

To get the accuracy on the test data set, I have to submit the file I wrote out, "nb_sex.csv", to Kaggle. I did, and the accuracy based on the test data for the Public Leaderboard was 0.76555. This is indeed the score for the Gender-Based Model benchmark and ties my best score to date.

Sensitivity / specificity

You can get a confusion matrix to use to calculate many of these metrics pretty simply by taking a table of the model predictions versus the observed labels:

```
table(pred_train$class, train$Survived)
```

```
##
##       0   1
##   0 468 109
##   1  81 233
```

You can also use functions from R packages to calculate everything. The `caret` package is a bit of a Swiss Army knife for machine learning. You can use it, among other things, to check sensitivity and specificity, and a variety of other model evaluation metrics using its `confusionMatrix` function:

```
nb_eval <- confusionMatrix(data = pred_train$class,
                           reference = train$Survived,
                           positive = "1")
```

This function outputs an object that includes:

A confusion matrix:

```
nb_eval$table
```

```
##           Reference
## Prediction   0   1
##          0 468 109
##          1  81 233
```

Note: The `confusionMatrix` help file has formulas for many of these metrics, if you want to get those equations. I think that, to make those equations work correctly in comparison to the table printed out with the confusion matrix output, you need to think of this output as being in the order:

–	–
D	C
B	A

In terms of true and false positives and negatives, if you take `Survived = 1` as a positive value (or event, using the wording from the help file), this confusion matrix is in the format:

–	–
True negative (TN)	False negatives (FN)
False positive (FP)	True positives (TP)

Overall accuracy:

```
nb_eval$overall["Accuracy"]
```

```
##  Accuracy
## 0.7867565
```

Linking this measure with the values in the confusion matrix:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

This model correctly classified the survival status of 79% of all passengers (when applied to the training dataset).

The **error rate** is 1 - Accuracy.

Sensitivity and specificity:

Here is the model's **sensitivity** on the training data:

```
nb_eval$byClass["Sensitivity"]
```

```
## Sensitivity
##   0.6812865
```

This metric is calculated:

$$\text{Sensitivity} = \frac{TP}{TP + FN}$$

Therefore, for those passengers that survived, this model correctly predicted the status of 68% of them.

This measure is also sometimes called **recall**, **true positive rate**, or **power** (see Table 4.7 in James et al.). The **Type II error rate** is 1 - Sensitivity.

Here is the model's **specificity** on the training data:

```
nb_eval$byClass["Specificity"]
```

```
## Specificity
##    0.852459
```

This metric is calculated:

$$\text{Specificity} = \frac{TN}{TN + FP}$$

Therefore, for those passengers that died, this model correctly predicted the status of 85% of them.

The **false positive rate** and **Type 1 error rate** are both 1 - Specificity.

Based on Sensitivity and Specificity, you can calculate **Youden's J Index**:

$$(J) = \text{Sensitivity} + \text{Specificity} - 1$$

Here, **Youden's J Index** is 0.5337456. This provides a single measure of the model's performance on both sensitivity and specificity.

Positive and negative predictive values

The output of the `confusionMatrix` function also provides positive predictive value (PPV) and negative predictive value (NPV).

Here is the model's **positive predictive value** within the training data:

```
nb_eval$byClass["Pos Pred Value"]
```

```
## Pos Pred Value
##      0.7420382
```

Therefore, of passengers who were predicted by the model to surive, 74% actually did survive.

This metric is calculated:

$$\text{PPV} = \frac{\text{Sensitivity} * \text{Prevalence}}{\text{Sensitivity} * \text{Prevalence} + ((1 - \text{Sensitivity}) * (1 - \text{Prevalence}))}$$

If the proportions of the two outcome classes are balanced, then this simplifies to:

$$\text{PPV} = \frac{TP}{TP + FP}$$

This value is also know as the **precision**. The **false discovery proportion** is 1 - PPV.

Here is the model's **negative predictive value** within the training data:

```
nb_eval$byClass["Neg Pred Value"]
```

```
## Neg Pred Value
##      0.8110919
```

Therefore, of passengers who were predicted by the model to die, 81% actually did die.

This metric is calculated:

$$\text{PPV} = \frac{\text{Sensitivity} * (1 - \text{Prevalence})}{\text{Sensitivity} * (1 - \text{Prevalence}) + ((1 - \text{Sensitivity}) * \text{Prevalence})}$$

If the proportions of the two outcome classes are balanced, then this simplifies to:

$$\text{PPV} = \frac{TN}{TN + FN}$$

Kappa

The output of the `confusionMatrix` function also provides a Kappa value, which it gives as an unweighted Kappa statistic (see the `confusionMatrix` help file):

```
nb_eval$overall["Kappa"]
```

```
##    Kappa
## 0.542113
```

The **Kappa* statistic can take values between -1 (completely inaccurate predictions) and 1 (completely accurate predictions). Kappa will equal zero when the model accuracy is equal to the accuracy you would expect if you predicted all observations to have the more common of the two outcome classes (here, that would mean predicting that everyone dies). Any value of Kappa below 0, then, means that your model is essentially worse than just guessing.

This metric is calculated as:

$$\text{Kappa} = \frac{O - E}{1 - E}$$

Where O is the model's observed accuracy (the accuracy calculated based on comparing predictions from the model to true values) and E is the expected accuracy. I had thought that expected accuracy was the **no-information rate**, or accuracy you would get if you predicted that all observations had the most common outcome class. You can get this from the object created by `confusionMatrix`:

```
nb_eval$overall["AccuracyNull"]
```

```
## AccuracyNull
##    0.6161616
```

However, when I plugged this in, I got a different Kappa than that generated by `confusionMatrix`. From the Kuhn and Johnson book:

> "*E* is the expected accuracy based on the marginal totals of the confusion matrix."

Perhaps we can clarify this in class.

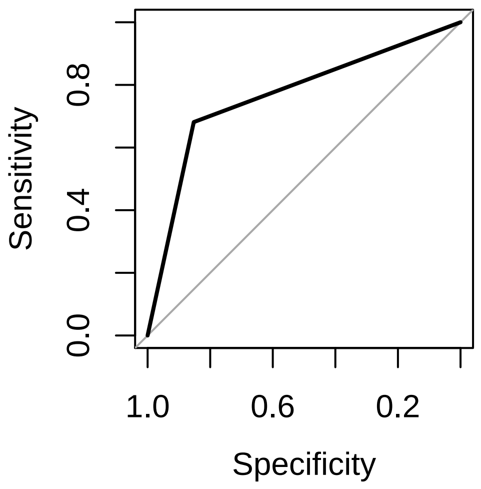
ROC Curve

You can use the `roc` function in the `pROC` package to create Receiver Operating Characteristic (ROC) curves:

```
roc_nb_sex <- roc(response = as.numeric(as.character(train$Survived)),
                  predictor = as.numeric(as.character(pred_train$class)))
```

This can be plotted:

```
plot(roc_nb_sex)
```



(As a reminder, to get the help file for something like this that is a method for a class, use, e.g., `?plot.roc`.)

This can be used to calculate the Area Under the Curve (AUC):

```
auc(roc_nb_sex)
```

```
## Area under the curve: 0.7669
```

I can also try this out for a k-NN model. Just for fun, this time I'll try fitting with the `knn3` function from the `caret` package. This allows you to fit a k-nearest neighbors using the classic formula syntax, although it still seems to have some problems with how to handle missing values when it comes time to predict:

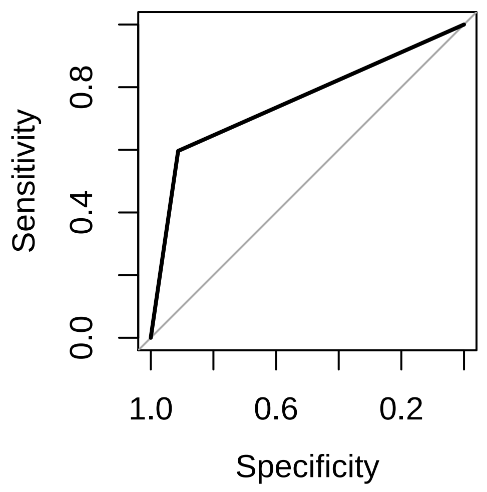
```r
train$scaled_Age <- scale(train$Age)

set.seed(1206)
knn_mod <- knn3(Survived ~ Sex + Pclass + scaled_Age, data = train,
                k = 21)

i_train <- complete.cases(train[ , c("Sex", "Pclass", "scaled_Age")])

train_pred <- predict(knn_mod, newdata = train[i_train, ],
                type = "class")
x_accuracy <- data.frame(Survived = train$Survived,
                           pred = factor("0", levels = c("0", "1")))
x_accuracy$pred[i_train] <- train_pred

roc_knn <- roc(response = as.numeric(as.character(x_accuracy$Survived)),
               predictor = as.numeric(as.character(x_accuracy$pred)))
plot(roc_knn)
```



```r
auc(roc_knn)
```

```
## Area under the curve: 0.7545
```

Splitting the training data to test

You might notice that, for all these metrics, I could only get measures of them when applying the model to the training data that was used to fit the model. Kaggle lets me test its metric (accuracy) through the Leaderboard, but otherwise I can't test metrics on new data (i.e., that I didn't use to train the model) using the methods I've been using.

We'll talk a lot more about this later, but a simple approach is to take the training data I've got and randomly split it into a set of data to use to train the model and a set to use to test the model. An easy way to do that is to sample row numbers from the full dataset and then use those to create `my_train` and `my_test` subsets using indexing. For example, to split into two thirds data to use for training and one third to use for testing:

```r
my_split <- sample(1:nrow(train), size = round(1 / 3 * nrow(train)))
```

```r
my_train <- train[-my_split, ]
my_test <- train[my_split, ]

dim(my_train)
```

```
## [1] 594  13
```

```r
dim(my_test)
```

```
## [1] 297  13
```

Now I can train and test on my local dataset, and maybe come up with a better idea of which models will perform well on the Kaggle leaderboard before I submit answers to that.

For example:

```r
nb_sex <- NaiveBayes(Survived ~ Sex, data = my_train)

pred_mytrain <- predict(nb_sex, newdata = my_train)
mean(pred_mytrain$class == my_train$Survived)
```

```
## [1] 0.7828283
```

```r
confusionMatrix(pred_mytrain$class,
                my_train$Survived,
                positive = "1")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 314  72
##          1  57 151
##
##                Accuracy : 0.7828
##                  95% CI : (0.7475, 0.8154)
##     No Information Rate : 0.6246
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.5306
##  Mcnemar's Test P-Value : 0.2177
##
##             Sensitivity : 0.6771
##             Specificity : 0.8464
##          Pos Pred Value : 0.7260
##          Neg Pred Value : 0.8135
##              Prevalence : 0.3754
##          Detection Rate : 0.2542
##    Detection Prevalence : 0.3502
##       Balanced Accuracy : 0.7617
##
##        'Positive' Class : 1
##
```

```r
pred_mytest <- predict(nb_sex, newdata = my_test)
mean(pred_mytest$class == my_test$Survived)
```

```
## [1] 0.7946128
```

```r
confusionMatrix(pred_mytest$class,
                my_test$Survived,
                positive = "1")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 154  37
##          1  24  82
##
##                Accuracy : 0.7946
##                  95% CI : (0.7441, 0.8391)
##     No Information Rate : 0.5993
##     P-Value [Acc > NIR] : 5.761e-13
##
##                   Kappa : 0.5645
##  Mcnemar's Test P-Value : 0.1244
##
##             Sensitivity : 0.6891
##             Specificity : 0.8652
##          Pos Pred Value : 0.7736
##          Neg Pred Value : 0.8063
##              Prevalence : 0.4007
##          Detection Rate : 0.2761
##    Detection Prevalence : 0.3569
##       Balanced Accuracy : 0.7771
##
##        'Positive' Class : 1
##
```

Computing time

Sometimes one of your most important considerations might be how long it takes either to fit or predict from the model. The `microbenchmark` package can help you determine if something is taking a very long time. It runs each line of code 100 times and gives, under `mean`, the number of nanoseconds it took to run that line of code.

Here are the required times for fitting Naive Bayes and k-NN models:

```r
i_train <- complete.cases(train[ , c("Sex", "Pclass", "Age")])
train_bench <- train[i_train, c("Survived", "Sex", "Pclass", "Age")]
train_bench$Age <- scale(train_bench$Age)
knn_x <- scale(model.matrix(~ Sex + Pclass + Age, data = train_bench)[ , -1])
knn_y <- train_bench$Survived

microbenchmark(
  nb_mod <- NaiveBayes(Survived ~ Sex + Pclass + Age,
                      data = train_bench),
  nb_mod_2 <- naiveBayes(Survived ~ Sex + Pclass + Age,
                      data = train_bench),
  knn_mod <- knn3(Survived ~ Sex + Pclass + Age,
                 data = train_bench, k = 21)
)
```

```
## Unit: milliseconds
##                                                                              expr
##          nb_mod <- NaiveBayes(Survived ~ Sex + Pclass + Age, data = train_bench)
##        nb_mod_2 <- naiveBayes(Survived ~ Sex + Pclass + Age, data = train_bench)
##  knn_mod <- knn3(Survived ~ Sex + Pclass + Age, data = train_bench,      k = 21)
##       min       lq     mean   median       uq      max neval
##  2.910902 3.008811 3.502130 3.239591 3.584525 7.378191   100
##  2.201333 2.288511 2.747275 2.477569 2.666584 8.071833   100
##  1.943456 2.056245 2.433070 2.183160 2.328566 4.684600   100
```

And here are the times for predicting from the models (notice that the k-nearest neighbors fit using the `knn` function from the `class` package fits and predicts from the model at the same time):

```
microbenchmark(
  nb_preds <- predict(nb_mod, newdata = train_bench),
  nb_pred_2 <- predict(nb_mod_2, newdata = train_bench),
  knn_preds <- predict(knn_mod, newdata = train_bench, type = "class"),
  knn_preds_2 <- knn(knn_x, knn_x, knn_y)
)
```

```
## Unit: milliseconds
##                                                                   expr
##                     nb_preds <- predict(nb_mod, newdata = train_bench)
##                   nb_pred_2 <- predict(nb_mod_2, newdata = train_bench)
##  knn_preds <- predict(knn_mod, newdata = train_bench, type = "class")
##                                knn_preds_2 <- knn(knn_x, knn_x, knn_y)
##         min         lq       mean     median         uq       max neval
##  183.557353 205.577132 216.077378 215.519753 220.397320 334.73141   100
##  109.917544 120.104531 126.184154 125.940313 131.707884 150.33847   100
##   17.836966  18.356263  20.905729  19.871462  22.579650  29.03261   100
##    4.049098   4.135264   4.931068   4.306873   5.173054  10.56777   100
```

For more on profiling R code, see the chapter about it on the online version of Hadley Wickham's *Advanced R* book.

To think about

In the `confusionMatrix` output, what are:

- The No Information Rate
- Prevalence
- Detection Rate
- Detection Prevalence

- Balanced Accuracy

Other things to think about:

- What are some different evaluation metrics used by other Kaggle classification competitions?
- Think of some examples of predictive tasks where Accuracy would be a poor metric to use to evaluate models.
- How do these metrics work when you are classifying into > 2 classes?